DISEÑO E IMPLEMENTACIÓN DE UN CRIPTO-PROCESADOR ASÍNCRONO DE BAJO CONSUMO BASADO EN EL ALGORITMO DE RIJNDAEL

Tesis de Grado presentada como
Requisito parcial para optar al título de
Doctor en Ingeniería, énfasis en
Ingeniería Electrónica

2009

RUBÉN DARÍO NIETO LONDOÑO

Director
ÁLVARO BERNAL NOREÑA, *Ph.D.*

UNIVERSIDAD DEL VALLE – FACULTAD DE INGENIERÍAS
PROGRAMA DE DOCTORADO EN INGENIERÍA
ESCUELA DE INGENIERÍA ELÉCTRICA Y ELECTRÓNICA
GRUPO DE ARQUITECTURAS DIGITALES Y MICROELECTRÓNICA

A mi hija, Ana Lucía
A mi esposa, Claudia Lucía
A mis padres, María Leonelia
y Noel Antonio

CONTENIDO

Página

Capítulo 1: Introducción

Capítulo 2: Lógica asíncrona

Página

Capítulo 3: Algoritmo de Rijndael

Capítulo 4: Especificación e implementación asíncrona de las Funciones de transformación del algoritmo de Rijndael

LISTA DE FIGURAS

Página

Página

LISTA DE TABLAS

Resumen

Los avances en el desarrollo de herramientas de síntesis automática de circuitos asíncronos han hecho posible el diseño de sistemas con alta escala de integración. Sin embargo, la mayoría de herramientas presentan limitaciones a nivel de simulación y también a nivel de capacidades de depuración, lo cual plantea distintos desafíos y oportunidades para incursionar en la línea de investigacion de circuitos asíncronos.

De otro lado, diversas investigaciones han demostrado que los circuitos criptográficos CMOS implementados con el estilo de diseño síncrono son vulnerables ante los llamados ataques de *Análisis de Potencia Diferencial (DPA: Differential Power Analysis)*, los cuales se basan en debilidades en la implementación hardware que dan la posibilidad de realizar mediciones para filtrar ruido y explotar la relación entre el dato procesado y las corrientes de fuga asociadas. El diseño asíncrono es una línea de investigación que está siendo utilizada en la implementación de circuitos digitales de alta seguridad; una de sus cualidades es que permite mejorar la resistencia a los ataques DPA mediante la implementación de circuitos balanceados que minimizan la fuga de corriente.

Son pocos los reportes de resultados de implementaciones asíncronas del Estándar de Encriptación Avanzada (*AES: Advanced Encryption Standard*), también conocido algoritmo de Rijndael. Estos reportes sólo mencionan resultados de desempeño de implementaciones sobre *ASICs* (*Application Specific Integrated Circuits*). Hasta el momento no se han encontrado reportes de implementaciones asíncronas sobre *FPGAs* (*Field Programmable Gate Arrays*) del algoritmo AES completo, ni de bloques individuales del mismo.

Esta tesis está dirigida a lograr la síntesis e implementación asíncrona del algoritmo de Rijndael, usando para ello la integración entre la herramienta de especificación y síntesis asíncrona llamada *Balsa* y la plataforma de desarrollo para FPGAs *Xilinx ISE 9.2i*. El algoritmo está implementado con base en bloques funcionales que han utilizado arquitecturas optimizadas a nivel de circuito lógico. Este trabajo también hace una evaluación de la integración entre las herramientas de diseño con base en el análisis de los resultados de síntesis de especificaciones asíncronas sobre FPGAs.

Abstract

Advances in the development of tools for automatic synthesis of asynchronous circuits have made possible the design of systems with high integration level. Nevertheless, most tools have weaknesses at simulation level and also at debugging capabilities level, which raises different challenges and opportunities to break into the investigation line of asynchronous circuits.

Moreover, several investigations have shown that the cryptographic CMOS circuits implemented with synchronous design styles are vulnerable to attacks by the so-called Differential Power Analysis (DPA), which are based on weaknesses in the hardware implementation give the possibility to perform measurements to filter noise and exploit the relationship between data processing and associated currents. The asynchronous design is a research line that is being used to implement digital circuits of high security, one of his qualities is that it improves resistance to DPA attacks through the implementation of balanced circuits which minimize leakage current.

The reports of results of asynchronous implementations of the Advanced Encryption Standard (AES), also known Rijndael algorithm, are few. These reports only mention the results of performance implementations on ASICs (Application Specific Integrated Circuits). So far there have been no reports of asynchronous implementations on FPGAs (Field Programmable Gate Arrays) complete AES algorithm, or individual blocks of the same.

This thesis is aimed at synthesis and asynchronous implementation of the Rijndael algorithm, using the integration between the asynchronous tool for specification and synthesis call Balsa and development platform for FPGAs Xilinx ISE 9.2i. The algorithm is implemented based on functional blocks that have used architectures optimized at logic circuit level. This work also makes an assessment of the integration between design tools based on an analysis of the results of synthesis of asynchronous specifications on FPGAs.

Reconocimientos

Este trabajo de investigación doctoral ha sido dirigido por el Doctor *Alvaro Bernal Noreña*, de la *Escuela de Ingeniería Eléctrica y Electrónica de la Universidad del Valle*. Es a él, en primer lugar, a quien deseo expresarle mis más sinceros agradecimientos por su estímulo, sugerencias y comprensión en momentos de tensión.

Gracias también a los evaluadores de este trabajo de tesis doctoral: *Doctores Edward David Moreno Ordóñez*, de la Universidad del Estado de Amazonas; Brasil; *Freddy Alexander Rivera Vélez*, del Departamento de Arquitectura de Computadores y Automática de la Universidad Complutense de Madrid, España, y *José Édinson Aedo Cobo* de la Universidad de Antioquia, Colombia.

Agradezco desde lo más profundo de mi corazón a mi amigo y colega *Angel García* por su apoyo y confianza. También por la ayuda que me brindó en la revisión y corrección del código de descripción del procesador criptográfico.

Tambien agradezco a mis amigos *Luis Garreta, Carlos Acosta, Gustavo Cerezo, Adolfo Jaramillo, Rodrigo Llorente y Mauricio Muñoz*, por asesorarme en la instalación del software de descripción de hardware asíncrono sobre Linux y también por la información que me facilitaron; a *Blanca Estella, a Santiago Isaías y a Luis Javier* por su solidaridad y buenos deseos.

Por último, un agradecimiento especial a mis colegas de la EIEE: Profesor *Hernando Vásquez*, por estar pendiente, no solo de mis avances sino también de mi bienestar laboral; a mis profesores *Oscar Polanco*, *Carlos Rafael Pinedo*, *Mario Andres Llano*; a mis profesores de Inglés *Moises y Eva*.

Este reconocimiento no estaría completo sin un saludo al papel desempeñado por mi familia para escribir este informe de tesis. El emocional e incomparable apoyo, el aliento sin límites, el amor y el afecto incondicional de mi esposa, la ternura de mi hija, ..., la lista sigue.

Rubén Darío Nieto Londoño
Septiembre 6 de 2009

Capítulo 1: Introducción

1.1. Perspectiva de la tesis

En octubre del año 2000 el Instituto Nacional de Estándares y Tecnología de Estados Unidos, *NIST* (*National Institute of Standards and Technology*), seleccionó el algoritmo de Rijndael como el nuevo estándar de encriptación avanzada, *AES* (*Advanced Encryption Standard*) [1]. El *NIST* registró el algoritmo con el estándar *FIPS-197* (*Federal Information Processing Standard- document 197*) [2].

La mayoría de implementaciones del algoritmo de Rijndael han sido realizadas en software o en *FPGAs (Field Programmable Gate Arrays*); la implementación en software permite flexibilidad y bajo costo mientras que la implementación en hardware proporciona mayor desempeño [3],[5]. El algoritmo también se ha implementado en Circuitos Integrados de Aplicación Específica (*ASICs: Application Specific Integrated Circuits*) [6]-[9], los cuales soportan altas velocidades de encriptación (del orden de decenas de Gbits/s) que los hace muy útiles para encriptación de bloques de datos en redes ópticas.

Uno de los principales problemas de seguridad que debe enfrentar la criptografía moderna son los llamados ataques de Análisis de Potencia Diferencial (*DPA: Differential Power Analysis*), sin embargo, son muy pocas las propuestas de arquitecturas orientadas a mejorar la resistencia a tales ataques. Otro problema a resolver es que el algoritmo, en muchas ocasiones, requiere ser implementado sobre plataformas de sistemas embebidos en los que el consumo de potencia debe ser disminuido al máximo. La metodología de diseño asíncrono se constituye en una alternativa importante para disminuir el consumo de potencia dinámica (el cual es proporcional a la frecuencia de reloj en sistemas síncronos) además de mejorar la seguridad del algoritmo contra los ataques *DPA*, esto último debido, principalmente, a la codificación *dual-rail* balanceada [10]-[14].

Este trabajo de investigación ha tenido como objetivo principal diseñar e implementar, en hardware reprogramable (FPGA), el algoritmo de Rijndael de acuerdo con el estándar FIPS-197 utilizando la metodología de diseño asíncrono. Para lograr este objetivo se han utilizado dos herramientas de software: la herramienta de descripción, síntesis y simulación de sistemas asíncronos llamado Balsa y la plataforma de desarrollo e implementación de circuitos digitales en FPGAs llamada Xilinx ISE 9.2i.

Para obtener mejor rendimiento y disminuir el número de circuitos utilizados en las transformaciones que conforman el algoritmo de Rijndael, se han implementado optimizaciones con base en propuestas arquitecturales de distintos autores [2], [15]-[23]. Al final se integran las mejores opciones para la implementación del algoritmo.

1.2. Contribución de la Investigación

Este trabajo se inició con la idea de implementar el procesador criptográfico de Rijndael

en hardware usando técnicas asíncronas para obtener bajo consumo de potencia y además, de acuerdo con los reportes de distintos autores, para incrementar la seguridad ante ataques del lado de canal como el DPA. Aunque esta idea no constituye por si misma una contribución, el seguimiento de la misma permitió obtener algunas conclusiones que si constituyen aportes relevantes del trabajo:

- Se logró especificar el algoritmo de Rijndael para los procesos de encriptación y desencriptación de manera completamente asíncrona utilizando para ello una herramienta software de síntesis y simulación de libre distribución.

- Los resultados obtenidos mediante las simulaciones realizadas con la herramienta de diseño asíncrono Balsa, permitieron determinar que de las funciones de transformación del algoritmo, la transformación de sustitución de byte es la que determina en mayor porcentaje el rendimiento general del circuito para los procesos de encriptación y desencriptación.

- A pesar de verificarse la correcta integración entre la herramienta de especificación asíncrona Balsa y la plataforma de desarrollo *Xilinx ISE 9.2i,* se ha comprobado que existen factores limitantes, especialmente relacionados con los recursos de interconexión de las FPGAs y la manera en que estos se adecúan a las necesidades de conexión de los componentes asíncronos definidos por Balsa. Esto significa que la síntesis automática para diseño asíncrono, al depender aún de plataformas para sistemas síncronos, requiere de un extenso trabajo de investigación y desarrollo por parte de la comunidad científica.

1.3. Estructura del documento

La secuencia de capítulos de este informe de investigación es la siguiente:

Capítulo 2: Lógica asíncrona

Este capítulo hace una introducción al estado del arte del diseño de circuitos asíncronos. Se enfoca en los conceptos básicos, en los modelos de tiempo asociados con las metodologías de diseño y en el estado reciente del diseño de herramientas y procesadores asíncronos. También se presenta el ambiente de diseño de circuitos asíncronos conocido como Balsa, siendo esta la herramienta utilizada para el diseño del Algoritmo de Rijndael.

Capítulo 3: Algoritmo de Rijndael

En este capítulo se presenta una introducción del estándar AES (algoritmo de Rijndael), así como las tendencias arquitecturales para su implementación. Se hace una revisión de la literatura relacionada con diferentes propuestas de implementación del algoritmo tanto en hardware reconfigurable (*FPGAs*) como en *ASICs*. Finalmente se muestran reportes de implementaciones asíncronas publicados hasta el momento.

Capítulo 4: Especificación e implementación asíncrona de las funciones de transformación del algoritmo de Rijndael

Este capítulo presenta las descripciones de las transformaciones que conforman el algoritmo de Rijndael y la manera como cada una de ellas puede ser optimizada para conformar la implementación asíncrona del algoritmo usando claves de 128 bits (AES-128) sobre una FPGA de Xilinx.

Capítulo 5: Implementación asíncrona del algoritmo de Rijndael

En este capítulo se describe la arquitectura utilizada en la implementación asíncrona del algoritmo de Rijndael para el estándar AES-128 (*Advanced Encryption Standard* – clave de 128 bits). Se muestran los resultados de simulación en Balsa para los procesos de encriptación y desencriptación con base en las especificaciones del estándar *FIPS-197* del *NIST* de Estados Unidos.

Capítulo 6: Conclusiones y trabajo futuro

1.4. Publicaciones

Nieto, R., Bernal, A. "Implementación asíncrona de las funciones *MixColumn* e *InvMixColumn* del algoritmo de Rijndael".. Sometido a la revista Dyna, Universidad Nacional de Colombia en octubre de 2008, revisado en agosto de 2009.

Nieto, R., Bernal, A. *"An architectural approach for asynchronous implementation of the Rijndael Algorithm"*. Submitted to IEEE Transaction on communications, octubre de 2008.

Nieto, R., Bernal, A. *"Arquitectura para la implementación asíncrona de la función de sustitución de byte del algoritmo de Rijndael"*. Séptima Conferencia iberoamericana en Sistemas, Cibernética e Informática, CISCI 2008, junio 29 a julio 2. Memorias, ISBN-10:1-934272-41-8 (Volumen III). P-p 74-79. Orlando, Florida, EEUU. 2008.

Nieto, R., Bernal, A. *"Metodologías para la Implementación asíncrona del Algoritmo de Rijndael"*. V Congreso Internacional de Electrónica y tecnologías de Avanzada, Universidad de Pamplona, Colombia, 27-29 septiembre 2006.

Nieto, R., Bernal, A. *"Estado del arte en las metodologías de diseño de circuitos digitales asíncronos"*. Revista Ingeniería y Competitividad, vol. 7 N° 2, p.p 71-83. Universidad del Valle, Cali, Colombia. 2005.

Referencias

[1] **NIST**, ***National Institute of Standards and Technology***. 2000. Disponible en: http://csrc.nist.gov/CryptoToolkit/aes/rijndael/

[2] **FIPS**, **Federal Information Processing Standards.** "S*pecification for the ADVANCED ENCRYPTION STANDARD (AES)"*. Publication 197, November 26. 2001a Disponible en: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

[3] **Weaver, N., Wawrzynek, J.** *¨High Performance, Compact AES Implementations in Xilinx FPGAs¨*. U. C. Berkeley BRASS group. 2002.

[4] **Lu, J., Lockwood, J.** *"IPSec Implementation on Xilinx Virtex-II Pro FPGA and Its Application".* Reconfigurable Network Group, Applied Research laboratory, Washington University, St. Luis, IEEE. 2005.

[5] **Mroczkowski, P.** *"Implementation of the block cipher Rijndael using Altera FPGA".* Military University of Technology, Warsaw, Poland. 2000

[6] **Panato, A., Barcelos M., & Reis, R**. *"A Low Device Occupation IP to Implement Rijndael Algorithm"*. Universidade federal do Rio Grande do Sul, PPGC – Instituto de Informática, Porto Alegre- RS- Brasil. 2003.

[7] **Regeb, J., Ramaswamy, V., & Ghadiri K.** *"Hardware Implementation of the Algorithm for High-Speed Networks"*. Electrical Engineering Dept, San Jose State University, San Jose, CA -95172, USA. 2003.

[8] **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.

[9] **Mukhopadhyay, D., Roychowdhury, D.** *"An Efficient End to End Design of Rijndael CryptoSystem in 0.18 µm CMOS".* Proceedings of the 18th International Conference on VLSI Design; 4th International Conference on Embedded Systems Design (VLSID´05). IEEE. 2005.

[10] **KULRD and SCARD Consortium.**. *"Intermediate report, Side Channel Analysis Resistant Design Flow"*. SCARD, Information Society, Project Number: IST-2002-507270, 15 January 2005.

[11] **Shang, D., Burns, F., Bystrov, A., Koelmans, A., Sokolov, D., Yakovlev, A.** *"A Low and Balanced Power Implementation of the AES Security Mechanism Using Self-Timed Circuits"*. E. Macii et al. (Eds.): PATMOS 2004, LNCS 3254, pp. 471–480, 2004.c. Springer-Verlag Berlin Heidelberg 2004.

[12] Shang, D., Burns, F., Bystrov, A., Koelmans, A., Sokolov, D., Yakovlev, A. *"High-security asynchronous circuit implementation of AES"*. IEE Proceedings online no. 20050088, doi:10.1049/ip-cdt:20050088. IEE Proc.-Comput. Digit. Tech., Vol. 153, No. 2, March 2006.

[13] Bouesse, F., Renaudin, M., Germain, F. *"Asynchronous AES Crypto-Processor Including Secured and Optimized Blocks"*. TIMA Laboratory, Grenoble, France; SGDN/DCISS, Paris, France. Journal of Integrated Circuits and Systems, Vol 1, No 1, pp 5-13. March 2004.

[14] Bouesse, F.; Renaudin, M.; Witon, A.; Germain, F. *"A clock-less low-voltage AES crypto-processor"***.** *Solid-State Circuits Conference*, ESSCIRC.20005. 2005. Proceedings of the 31st European. Volume , Issue , 12-16 Sept. 2005. P.p. 403–406. Digital Object Identifier 10.1109/ESSCIR.2005.1541645.

[15] Morioka, S., Satoh, A. *"An Optimized S-Box Circuit Architecture for Low Power AES Design"*. Cryptographic Hardware and Embedded Systems (CHES 2002), LNCS 2523, pp.172-186, Springer-Verlag Berlin Heidelberg, 2003.

[16] Huang, Y.H., Lin, Y.S., Hung, K.Y., Lin, K.C. "Efficient Implementation of AES IP". IEEE, Asia Pacific, Conference on Circuits and Systems APCCAS-2006, pp 1418-1421. Dec 2006.

[17] Li, H., Friggstad, Z. *"And Efficient Architecture for the AES MixColumns Operation"*. IEEE International Symposium on Circuits and Systems, 2005, ISCAS. Vol. 5, pp. 4637-4640. May 2005.

[18] Cordero, V., Silva, C. *"Design of an ASIC CORE for data encryption and decryption Using the NIST Advanced Encryption standard"*. Microelectronics Group – Pontificia Universidad Católica del Perú. IX Workshop de Iberchip, La Habana, Cuba. 2003.

[19] Fischer, V., Drutarovský, M., Chodowiec, P., Gramain, F. *"InvMixColumn Descomposition and Multilevel Resource Sharing in AES Implementations"*. *IEEE Transactions* on Very Large Scale Integration (VLSI) Systems, Vol. 13, No 8, August 2005.

[20] Hsiao, S.F., Chen, M.C., Tu, C.H. "Memory-Free Low-Cost Designs of Advanced Encryption Standard Using Common Subexpression Elimination for Subfunctions in Transformations". IEEE Trans. on circuits and systems-I: Regular papers, vol. 53, No. 3, pp. 615-626, March 2006.

[21] Lu, C.C., Tseng, S.Y. "Integrated Designs of AES (Advanced Encryption Standard) Encrypter and Decrypter". IEEE Proceedings on Application-Specific Systems, Architectures, and Processors (ASAP'02), Jul 2002.

[22] Wolkerstorfer, J. "An ASIC implementation of the AES MixColumn operation". Proceedings Austrochip 2001, pp. 129-132. Vienna, Austria, Oct. 12, 2001.

[23] Chodowiec, P., Gaj, K. *"Very Compact FPGA Implementation of the AES Algorithm".* CHES 2003, LCNS 2779, pp. 319-333, *Springer-Verlag* Berlin Heidelberg 2003.

Capítulo 2: Lógica asíncrona

Introducción

Casi todos los sistemas digitales de hoy se basan en aproximaciones síncronas. Por lo general, un sistema se compone de uno o más subsistemas, y cada uno de ellos es una máquina de estados finitos que usa una señal de reloj para manejar los cambios de estado. El correcto funcionamiento de tales subsistemas depende del manejo de los retardos de la lógica combinatoria tal que los tiempos de establecimiento (*set up*) y sostenimiento (*hold*) del conjunto de *flip-flops* se mantengan en unos límites mínimos y máximos previamente determinados por el diseñador.

El diseño asíncrono no sigue esta metodología debido a que, en general, no existe un reloj que gobierne la temporización de los cambios de estado. Los subsistemas intercambian información en instantes de tiempo mutuamente negociados, sin depender de temporización externa.

En el contexto de la criptografía, la mayoría de propuestas arquitecturales para la implementación del algoritmo de Rijndael (*AES: Advanced Encryption Standard*) se basan en la metodología de diseño síncrono, ya sea usando librerías de Circuitos Integrados de Aplicación Específica (*ASICs: Application Specific Integrated Circuits*) ó *FPGAs* (*Field Programmable Gate Arrays*). Son muy pocas las propuestas de arquitecturas AES que utilizan el estilo de diseño asíncrono, a pesar que esta metodología está orientada a disminuir el consumo de potencia en circuitos digitales, y en el caso particular de la criptografía, está orientada también a mejorar la resistencia contra los ataques de Análisis de Potencia Diferencial (*DPA: Differential Power Analysis*). En la literatura no se han encontrado reportes de resultados de implementaciones asíncronas del estándar AES sobre *FPGAs*. Hasta el momento sólo se han reportado dos implementaciones asíncronas del algoritmo basadas en *ASICs* las cuales solo reportan resultados para el proceso de encriptación: en el TIMA Laboratory, de Grenoble (Francia) se reporta una arquitectura QDI balanceada con codificación *1-de-N* y un retardo máximo de 850ns; en la EECE School de la Newcastle University (Inglaterra) se reporta una arquitectura *pipeline* con codificación *dual-rail* con un retardo máximo de 800ns. Tales implementaciones se describen en detalle en el capítulo 3.

Este capítulo brinda una introducción al estado del arte del diseño de circuitos asíncronos. Se centraliza en los conceptos fundamentales, en los modelos de tiempo asociados con las metodologías de diseño y en el estado reciente del diseño de herramientas y procesadores asíncronos. Al final del capitulo se presenta el ambiente de diseño conocido como Balsa, siendo esta la herramienta utilizada para el diseño del Algoritmo de Rijndael, el cual se describe en extenso en los capítulos 3 y 4.

2.1. Diseño asíncrono

El diseño asíncrono ha sido un área activa de investigación desde mediados de los años 50, cuando el foco principal eran los circuitos de relés mecánicos. En 1956, D. E. Muller

y W. S. Bartky [1], estudiaron en detalle una serie de aspectos teóricos[1] relacionados con circuitos asíncronos; desde entonces, este campo ha tenido un número de ciclos de alto interés. En años recientes ha habido un interés sin precedentes tanto en el entorno académico como en el industrial, lo cual se ha visto reflejado en el desarrollo de herramientas de especificación, síntesis, simulación y verificación y también en el desarrollo de circuitos integrados con una amplia variedad de aplicaciones.

2.1.1. Ventajas de la lógica asíncrona

De acuerdo con [2], el diseño lógico de hoy está basado en dos aspectos: *todas las señales son binarias y el tiempo es discreto*. Asumir valores binarios sobre las señales permite usar lógica booleana para describir y manipular construcciones lógicas. Asumir que el tiempo es discreto permite ignorar los riesgos (*glitches*) y los retardos de la realimentación asociados con los circuitos síncronos

Los circuitos asíncronos mantienen el supuesto que las señales son binarias pero remueven el supuesto que el tiempo es discreto. Lo anterior tiene varios posibles beneficios [3],[4]:

- ***No hay retrasos de reloj (clock Skew)***: ya que los circuitos asíncronos no tienen señal de reloj.
- ***Bajo Consumo de Potencia*****:** debido a que no es necesario conmutar líneas de reloj y precargar y descargar circuitos que no serán usados en alguna ruta de cálculos.
- ***Desempeño del caso promedio en lugar del peor caso:*** los circuitos síncronos deben esperar a que todos los posibles cálculos se hayan llevado a cabo antes de capturar el resultado manteniendo el peor caso. Muchos circuitos asíncronos censan cuándo ha finalizado el cálculo manteniendo el caso promedio.
- ***Facilidad de manejo de la temporización global:*** en microprocesadores síncronos, el reloj del sistema y por tanto el desempeño del sistema está determinado por la ruta más lenta (crítica) por lo que ésta se debe optimizar para obtener la más alta tasa de reloj, esto incluye optimizar porciones poco usadas del circuito. Los circuitos asíncronos operan a la velocidad de la ruta concreta de operación, raramente se pueden optimizar porciones de circuito sin que esto afecte en forma adversa el resto del sistema.
- ***Mejor Potencial de Migración Tecnológica:*** muchos circuitos son implementados en diferentes tecnologías durante su tiempo de vida. Para que un sistema síncrono pueda obtener mejor desempeño, todas sus partes deben migrar de tecnología ya que el desempeño está ligado con la ruta crítica. En sistemas asíncronos sólo el componente mas crítico influye sobre el desempeño promedio global ya que el desempeño solo depende de la ruta activa en el instante.
- ***Adaptación Automática a propiedades físicas:*** el retardo a través de un circuito puede cambiar con variaciones en la fabricación, temperatura y voltaje de la fuente de potencia. Los circuitos síncronos deben asumir que se presenta la peor combinación posible de factores para determinar el reloj del sistema. En la mayoría de circuitos

[1] Estos estudios se publicaron como ***"A theory of asynchronous circuits"*** **Tomos I y II**, para el Digital *Computer Laboratory, University of Illinois*, en noviembre de 1956 y marzo de 1957, respectivamente.

asíncronos se censa la finalización del cálculo y estos correrán tan rápidamente como las propiedades físicas lo permitan.

- ***Exclusión Mutua Robusta y Manejo de entradas externas:*** los elementos que garantizan la exclusión mutua de señales y sincronización de señales externas para un reloj están sujetas a metaestabilidad, por lo que los circuitos síncronos pueden fallar. La mayoría de sistemas asíncronos pueden esperar un largo tiempo arbitrario y como no necesitan un reloj con el cual las señales se sincronicen, pueden acomodar sus entradas desde el exterior.

2.1.2. Desventajas de la lógica asíncrona

Hoy día los sistemas síncronos mantienen el predominio a pesar de las ventajas potenciales de los circuitos asíncronos. La razón es que el diseño asíncrono presenta desventajas respecto del diseño síncrono, las cuales influyen en la aparente falta de voluntad de la industria para adoptar nuevas técnicas, según [3],[4], las principales desventajas son las siguientes:

- ***Complejidad:*** el paradigma del diseño con reloj tiene una única regla fundamental: "*cada etapa de procesamiento debe completar sus actividades antes que termine el periodo de reloj".* El diseño asíncrono requiere hardware extra para permitir que cada bloque realice la sincronización local para el paso de datos a otros bloques. Esta complejidad adicional resulta en circuitos más grandes y procesos de diseño más difíciles.
- ***Punto muerto (deadlock):*** una complicación adicional que aparece con el diseño asíncrono es la posibilidad de introducir un punto muerto en el sistema. Este es un estado en el cual un sistema no puede progresar hacia un objetivo requerido. Se puede producir un punto muerto al perderse un evento o si éste se introduce, por ejemplo, como resultado de un ruido o radiación iónica.
- ***Difícil verificación:*** para un diseño asíncrono, la verificación es difícil debido al comportamiento no determinista de los elementos de arbitración, y no es fácil detectar el punto muerto sin una exploración exhaustiva del espacio de estados.
- ***Difícil testibilidad:*** las pruebas para las fallas de fabricación en sistemas asíncronos es el principal obstáculo debido al comportamiento no determinista de los elementos de arbitración.
- ***Falta de familiaridad de los diseñadores con las técnica de diseño asíncrono:*** debido a que son las técnicas de diseño síncronas, principalmente, las que han sido usadas y enseñadas ampliamente en las universidades por mas de dos décadas.

2.2. Protocolos de *handshaking* [5]

2.2.1. Protocolos de dato acotado (*bundled-data*)

El término 'dato acotado' se refiere a una situación donde las señales de dato usan niveles booleanos normales para codificar información, y donde las líneas separadas *request* (solicitud) y *acknowledge* (reconocimiento) están acotadas por las señales de dato tal como se aprecia en la figura 2.1.

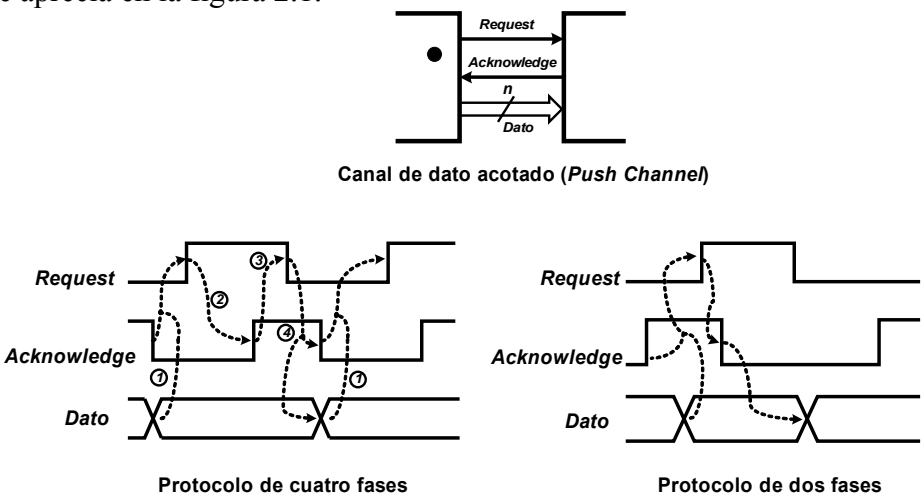


Figura 2.1: Protocolos de dato acotado

En el protocolo de cuatro fases las señales *request* y *acknowledge* también utilizan niveles booleanos normales para codificar información, el término 'cuatro fases' se refiere al número de acciones de comunicación:

(1) el transmisor emite un dato y activa una señal de solicitud, *request*, en alto
(2) el receptor absorbe el dato y activa un reconocimiento, *acknowledge*, en alto
(3) el transmisor responde desactivando *request* (en este instante no se garantiza la validez del dato)
(4) el receptor reconoce este evento haciendo que *acknowledge* se ponga en nivel bajo. El transmisor puede iniciar el siguiente ciclo de comunicación.

El protocolo de *handshaking* de cuatro fases es familiar para la mayoría de diseñadores de circuitos digitales pero tiene la desventaja de la innecesaria transición de retorno a cero que es costosa en tiempo y energía. El protocolo de dato acotado de dos fases evita esta situación.

En el protocolo de dos fases las señales de *request* y *acknowledge* se codifican como transiciones de señal y no hay diferencia entre $1 \rightarrow 0$ y $0 \rightarrow 1$, ambas representan un evento de señal. Idealmente, el protocolo de dato acotado de dos fases puede permitir circuitos más rápidos que el de cuatro fases, pero generalmente la implementación de circuitos que responden a eventos es compleja y no hay una respuesta generalizada acerca de qué protocolo es mejor.

Resulta apropiado discutir la terminología. En lugar del término 'dato acotado' (*bundled data*) usado en algunos textos, algunos utilizan el término *'single-rail'* (único riel). El término *'bundled data'* da a entender una relación de temporización entre las señales de dato y las señales de *handshake*, mientras que el término *'single-rail'* da a entender el uso de una línea para llevar un bit de dato, lo que también es consistente con la representación de *'dual-rail'* que se discute mas adelante.

En lugar del término '*handshaking* de cuatro fases', algunos textos usan el término *"Return-To-Zero (RTZ)",* y en lugar del término *'handshaking de dos fases'* usan el término *'Non-Return-To-Zero (NRZ)'*. Por lo tanto, un protocolo de pista única de retorno a cero es lo mismo que un protocolo de dato acotado de cuatro fases, etc.

Los protocolos mencionados asumen que el transmisor es la parte activa que inicia la transferencia de datos sobre el canal y se conoce como *'Push Channel'*. Lo opuesto, es decir el receptor preguntando por un nuevo dato, también es posible y se conoce como '*Pull Channel*'. En este caso, las direcciones de las señales de *request* y *acknowledge* son inversas, y la validez del dato está indicada por la señal *acknowledge* yendo del transmisor al receptor.

2.2.2. Protocolo *dual-rail de cuatro fases*

El protocolo *dual-rail* de cuatro fases codifica la señal de *request* entre las señales de dato usando dos líneas por bit de información comunicada. En esencia, se trata de un protocolo de cuatro fases que utiliza dos líneas de *request* por cada bit de información ***d***; una línea ***d.t*** se utiliza para indicar un 1 lógico (o *true*), y la otra línea ***d.f*** se usa para señalar un 0 lógico (o *false*). Cuando se observa un canal de un bit se verá una secuencia de *handshake* de cuatro fases donde la participación de la señal de *request* en cualquier ciclo de *handshake* puede ser ***d.t*** o ***d.f.*** Este protocolo es muy robusto [5]; dos circuitos se pueden comunicar de manera confiable a pesar de los retardos en las líneas que conectan las partes (el protocolo es insensible a los retardos). La figura 2.2 describe las características del protocolo.
Una visión más abstracta del protocolo *dual-rail* de cuatro fases es la siguiente [5]:
(1) el transmisor emite una palabra de código válido
(2) el receptor absorbe la palabra de código y activa la señal de *acknowledge*
(3) el transmisor responde mediante la emisión de una palabra de código vacío
(4) el receptor reconoce el evento haciendo que *acknowledge* se ponga en bajo. En este instante el transmisor puede iniciar el siguiente ciclo de comunicación.

La aproximación se puede extender a canales de bits paralelos. Un canal de datos de N bits esta formado por N pares de líneas concatenadas. Cada una utiliza la codificación descrita anteriormente. Un receptor tiene siempre la habilidad de detectar el instante en que todos los bits de datos son validos para lo cual responde con un nivel alto de la señal de *acknowledge*, cuando todos los bits son vacíos responde con una señal de *acknowledge* en bajo.

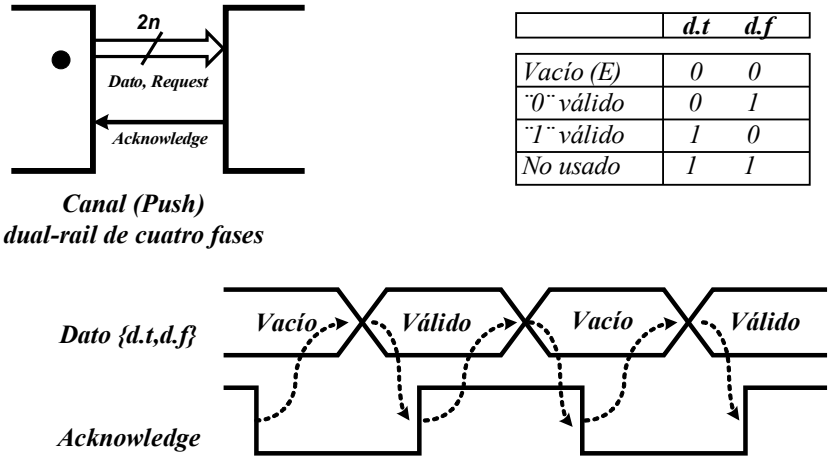


Figura 2.2: Protocolo de *handshaking dual-rail* de cuatro fases

2.2.3. Protocolo *dual-rail de dos fases*

Este protocolo también usa dos líneas por bit {***d.t,d.f***}, pero la información se codifica como transiciones (eventos). Sobre un canal de N bits se recibe un nuevo código cuando exactamente una línea ha hecho una transición en cada uno de los N pares de líneas. No existe el código vacío; un mensaje válido es reconocido y seguido por otro mensaje que también es reconocido. La figura 2.3 muestra las formas de honda de señal sobre un canal de dos bits usando el protocolo *dual-rail* de dos fases.

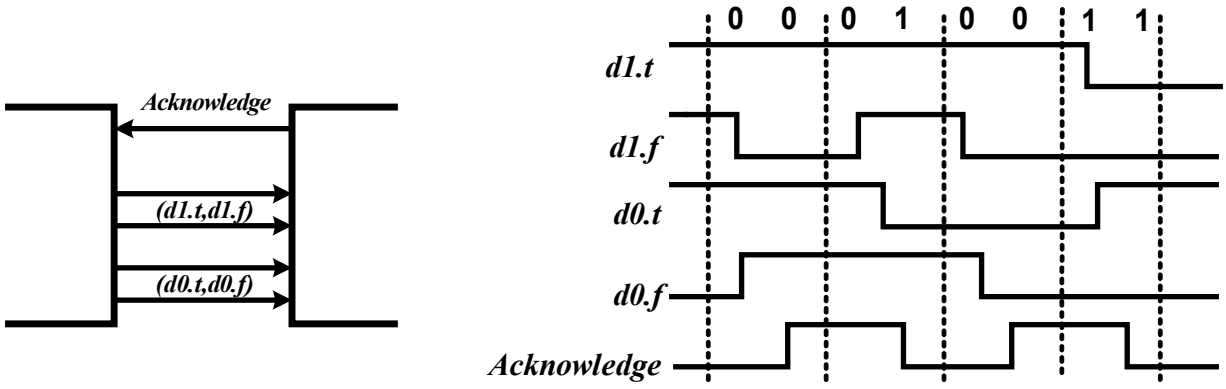


Figura 2.3: Protocolo de *handshaking dual-rail* de dos fases

2.2.4. Otros protocolos

Hasta el momento se han especificado cuatro de los protocolos mas comunes de canal: canal *push* de dato acotado de cuatro fases, canal *push* de dato acotado de dos fases, canal push *dual-rail* de cuatro fases, canal *push dual-rail* de dos fases; pero hay muchas otras posibilidades. Las dos líneas por bit usadas en los protocolos *dual-raíl* se pueden interpretar como una codificación *one-hot* y a menudo se puede extender a codificaciones

1-de-n en lógica de control y en codificaciones de datos de bases más grandes. Si se focaliza en comunicaciones, más que en computación, la codificación *m-de-n* puede ser de relevancia. El espacio de solución se puede expresar como el producto cruzado de un número de opciones que incluyen:

{2 fases,4 fases}x{dato acotado, dual-rail,1-de-n }x{push,pull}

2.3. Metodologías de diseño asíncrono

Una forma de distinguir el amplio espectro de diseños asíncronos es entendiendo los diferentes modelos de retardo y operación. Cada circuito físico tiene un retardo inherente, sin embargo, ya que los circuitos síncronos procesan entradas en cada pulso de reloj, ellos pueden ser considerados como operadores instantáneos que calculan un nuevo resultado en cada ciclo de reloj. Por otro lado, ya que los circuitos asíncronos no tienen reloj, estos se pueden considerar como elementos de cálculo dinámico a través del tiempo, por lo tanto, un modelo de retardo no define de forma eficiente el comportamiento dinámico de un circuito asíncrono. Hay dos modelos fundamentales de retardo, el *modelo de retardo puro* y el *modelo de retardo inercial* [1]. Un retardo puro puede retardar la propagación de una forma de onda pero no la altera. Un modelo inercial puede alterar la forma de la onda atenuando los *glitches* cortos, es decir, un retardo inercial tiene un periodo de umbral, δ, y los pulsos que tengan una duración menor que δ, son filtrados.

Los retardos también se caracterizan por sus modelos de tiempo [1]: en un *modelo de retardo fijo* se asume que el retardo es fijo; en un *modelo de retardo acotado*, un retardo puede tener cualquier valor en un intervalo de tiempo dado. En un modelo de *retardo no acotado*, un retardo puede tener cualquier valor finito.

El comportamiento de un circuito entero se puede modelar sobre la base de los modelos de sus componentes. En un modelo a nivel de compuertas cada compuerta y cada componente en el circuito tienen su correspondiente retardo. En un modelo de compuertas complejo, una sub-red de compuertas se modela por medio de un retardo único, esto es, la red se asume como un operador único sin retardos internos. Las líneas entre compuertas también son modeladas por retardos. Un modelo de circuito se define en términos de modelos de retardos para las líneas individuales y los componentes. Típicamente, la funcionalidad de una compuerta esta modelada por un operador instantáneo con un retardo asociado. Dado un modelo de circuito, es importante caracterizar la interacción del circuito con su ambiente. El circuito y el ambiente forman un sistema cerrado llamado *Circuito Completo*. Si al ambiente se le permite responder a las salidas del circuito sin restricciones de tiempo, los dos interactúan en un modo de entrada/salida. El ejemplo mas común es el *modo fundamental*, donde el ambiente debe esperar para que un circuito se estabilice antes de responder a las salidas del circuito. Tal requerimiento se puede ver como el tiempo de sostenimiento (*hold time*) para un *latch* o un *flip-flop*.

Dados los modelos para un circuito y su ambiente, los circuitos asíncronos se pueden clasificar jerárquicamente, tal como se describe en [6] y se resume en la figura 2.4.

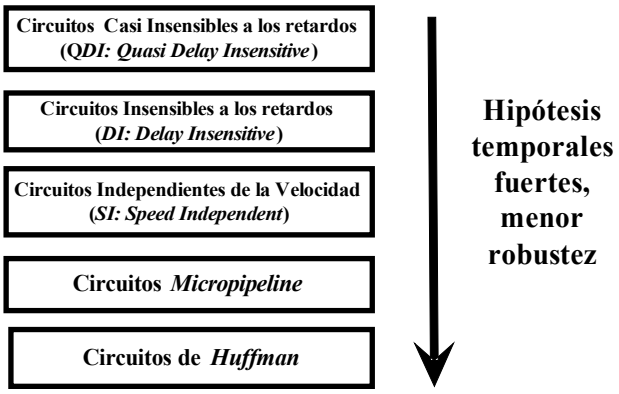


Figura 2.4: Jerarquía de circuitos asíncronos de acuerdo con robustez y complejidad [6]

2.3.1. Modelos de Huffman o de retardo acotado

Tradicionalmente han existido dos modelos importantes en el mundo de los circuitos asíncronos [7], [8]:

- **El modelo de Huffman**, en el cual el circuito se descompone en un circuito combinacional y un conjunto de hilos de realimentación. Además, se asocia un elemento de retardo acotado a cada interconexión entre puertas (modelo de retardo de hilo acotado).
- **El modelo de Muller**, el circuito se descompone en un conjunto de puertas arbitrariamente interconectadas y se asocia un elemento de retardo no acotado pero finito a cada salida de puerta (modelo de retardo de puerta no acotado).

2.3.1.1. Circuitos de Huffman en Modo Fundamental

La metodología de diseño basada en el modelo de Huffman (en honor de su inventor David A. Huffman) clásicamente produce ecuaciones lógicas que implementan la llamada *Tabla de Flujo,* que es una especificación similar a las Máquinas de Estados. El resultado obtenido está libre de riesgos únicamente bajo el *Modo Fundamental*, en este, el diseñador debe asegurarse de que las entradas del circuito sólo pueden cambiar cuando el circuito se halle en un estado estable y preparado para aceptarlas. Este modelo es altamente costoso de analizar y menos fiable que otros ante variaciones en el proceso de fabricación [7]. El modelo básico de Huffman, según [9],[10] se resume a continuación:

- ***Retardos de Realimentación***: en cada línea de realimentación se coloca un elemento de retardo, cuya salida define una variable de entrada secundaria; se supone que todas las compuertas y las demás líneas tienen un retardo de propagación igual a cero.
- ***Incertidumbre de retardos***: en un cambio de estado en el que toman parte dos o más variables secundarias, estas pueden cambiar de valor en cualquier orden arbitrario.

- ***Modo Fundamental:*** una señal de entrada primaria puede cambiar de valor entre los niveles 0 y 1 únicamente después que el circuito haya alcanzado un estado estable.
- ***Cambios en una sola entrada***: se supone que las señales de entrada primarias cambian de valor de una en una.

2.3.1.2. Extensiones de Circuitos de Huffman a Modo No Fundamental

Existen también técnicas para extender el modelo de Huffman a circuitos no fundamentales que permiten nuevas transiciones de las señales antes de lo que permite el *Modo Fundamental.* Una alternativa dentro del modelo de retardos acotados que combina aspectos del mundo síncrono (simplicidad y eficiencia) con aspectos del mundo asíncrono (velocidad y modularidad), son los llamados circuitos *auto-sincronizados* en los cuales se genera un reloj para cada máquina de estados que es independiente del resto. Existe una lógica global de control de reloj que produce pulsos siempre y cuando el estado de alguna salida deba cambiar y que controla los elementos de memoria implicados en la realimentación.

Para satisfacer una implementación libre de riesgos basta con que esta lógica global de control cumpla ciertas condiciones. Tal técnica permite además el uso de métodos de codificación típicos de circuitos síncronos. En este sentido se han presentado diversos trabajos como los de Nowick, Yun y Dill [1],[12] que usan una forma restringida de tabla de flujo llamada máquina de estados *'Burst Mode'*. El estilo de diseño '*Burst Mode*' se desarrolló con base en los trabajos realizados previamente en los laboratorios Hewlett Packard y la Universidad de Stanford por Davis, Stevens y Coates [1],[13]. Ladd y Birmingham [14], utilizan un sistema de comunicación en el que la validez de un dato se define en referencia a una señal de reloj local libre de riesgos, además, tanto las entradas como las salidas se mantienen con el uso de registros. Rosenberger y otros [15],[16], usan un tipo especial de *flip-flop* con señal de terminación llamado *Q-flop* que asegura que el circuito se ha estabilizado cuando se produce el pulso de reloj.

2.3.2. Modelo de Muller

El trabajo de Muller [17], introdujo el primer modelo formal de circuito asíncrono basándose en el concepto de *Diagramas de Transiciones de Estados (STD).* Su modelo de circuito consta de un conjunto de puertas lógicas con retardos no acotados; cada puerta tiene asociada una expresión que describe una función lógica completamente especificada [18]. Muller mostró que el comportamiento del circuito se puede describir de forma equivalente con un tipo de autómata finito llamado Diagrama de Transiciones de Estados. Los circuitos diseñados con base al modelo de Muller se denominan *independientes de la Velocidad* (ver sección 2.3.5.)

Se han propuesto distintos métodos para la síntesis de circuitos con retardos no acotados de puerta basándose en los trabajos de Muller: Armstrong y otros [19] proponen el uso de un código auto-sincronizado para la codificación de los estados de entrada y salida del circuito y la utilización de un protocolo de petición/confirmación para la sincronización

entre circuitos que cooperen. Un ejemplo de este tipo de código es el de *dual-rail* implementado con dos señales.

Las metodologías mas extendidas en el campo de la síntesis asíncrona son las basadas en la utilización de Redes de Petri [20] como herramientas de descripción del comportamiento de un circuito. Se destacan los Grafos de Señales [21] y los Grafos de Transición de Señales STGs [22] que son redes de Petri interpretadas en las que cada transición representa un cambio de valor en una de las señales del circuito. Así mismo destacan los Diagramas de Cambios (CDs) propuestos en [23], que son una forma similar a los STGs. Las técnicas basadas en STGs tienen actualmente gran número de seguidores. De ellos, se pueden comentar trabajos como lo de Lavagno [24], Moon y otros [25].

2.3.3. Circuitos Insensibles a los Retardos (*DI: Delay Insensitive*)

Un circuito *Delay Insensitive* (*DI*) es aquel que está diseñado para operar correctamente, sean cual sean los retardos en sus compuertas y líneas de interconexión. Este tipo de circuitos asume el modelo de retardo de línea y de compuerta no acotados. El concepto de circuito insensible a los retardos fue sugerido por Clark y Molnar en el trabajo *Macromodules* [26]. Los sistemas DI han sido formalizados por Udding [27] y Dill [28]. Las clases de circuitos DI que han sido construido es muy limitado, de hecho, se ha probado que la mayoría de circuitos DI útiles están restringidos a una clase de compuertas y operadores muy sencillos, el elemento C es un ejemplo [29].

2.3.4. Circuitos Casi Insensibles a los Retardos (*QDI: Quasi-Delay Insensitive*)

Un circuito casi insensible a retardos, (*quasi-DI*) o *QDI*, es un circuito insensible a retardos excepto que requiere bifurcaciones isocrónicas (la diferencia del retardo entre las ramas de la bifurcación es inferior al mínimo retardo de puerta). En contraste, en un circuito *DI* los retardos en las diferentes ramas bifurcadas es completamente independiente y puede variar considerablemente. La motivación de los circuitos *QDI* es que ellos constituyen el más delicado compromiso para la insensibilidad al retardo puro requerido para construir circuitos prácticos usando compuertas simples. Los elementos C son algo problemáticos como operadores ya que inherentemente contienen una salida que es realimentada internamente al elemento, este caso representa la peor forma de bifurcación isocrónica ya que una de las bifurcaciones esta contenida dentro del módulo de circuito del elemento C mientras la otra es llevada a los módulos externos [29].

En diferentes documentos publicados por Martin y otros [30]-[32] se minimiza la hipótesis de que todos los hilos del circuito tengan retardos no acotados y se realiza el supuesto de que ciertas bifurcaciones en los hilos son isocrónicas. Por esta razón tales circuitos reciben el nombre de *casi insensibles a retardos*. Los circuitos QDI no asumen, o reconocen, retardos en los operadores o en los alambres, excepto para bifurcaciones isocrónicas, para las cuales el diseñador asume que tienen retardos similares sobre las diferentes ramas. Los circuitos asíncronos QDI son los más conservativos en términos de

retardos, debido a eso, su dependencia de los retardos es muy pequeña, siendo los más robustos contra las variaciones de parámetros físicos; esta robustez permite el intercambio de energía y ajustes de voltaje en forma sencilla [33].

2.3.5. Circuitos Independientes de la Velocidad (*SI: Speed Independent*)

Los circuitos SI fueron introducidos por David Muller en la década de 1950. Los circuitos SI asumen que los retardos de las interconexiones son insignificantes, solamente las compuertas del circuito exhiben retardos apreciables. Las bifurcaciones se asumen como isocrónicas [34]. Según [35], si en un circuito los retardos en las líneas se asumen como cero (o en la práctica, menores que el mínimo retardo de compuerta), y el circuito exhibe una correcta operación, sean cual sean los retardos en cualquiera de los elementos del circuito, entonces se dice que el circuito es independiente de la velocidad. El supuesto de que el retardo de las líneas es cero es valido para circuitos pequeños. En general, los circuito SI y QDI son considerados como equivalentes desde el punto de vista de diseño.

2.3.6. Circuitos *Scalable Delay-Insensitive* (*SDI*)

El diseño basado en modelos *DI* o *QDI* se utiliza cuando se desea garantizar una operación uniforme en presencia de una distribución de retardo que es improbable que ocurra en la práctica (modelo pesimista), como consecuencia, el volumen de hardware resultante tiende a ser mas grande, y el desempeño de velocidad, basado en la tecnología actual, tiende a ser más bajo que un diseño equivalente sobre un modelo de retardo más optimista [36].

Una metodología de diseño prometedora consiste en dividir el sistema completo en partes de razonable tamaño, diseñar cada parte basados en un modelo de retardo más optimista, y diseñar las partes de interconexión o las partes globales basados en el modelo *DI*. Esta metodología está propuesta en el modelo *Scalable-Delay-Insensitive* (*SDI*). El modelo *SDI* es muy realista y ofrece una base amigable para el diseño de procesadores asíncronos de alto desempeño razonablemente confiable [36].

El modelo *SDI* es un modelo de retardo no acotado que no asume un límite superior sobre los retardos de compuertas y líneas de interconexión. Diferente a los modelos *DI* o *QDI*, el modelo *SDI* asume que existe una relación de retardo relativa y acotada entre cualquiera de dos componentes. El modelo *SDI* se define como sigue [37],[38]:

> ***Modelo Scalable Delay Insensitive***:
> Un componente se refiere a una compuerta o a una línea de interconexión entre dos compuertas. Sean ***d1*** y ***d2*** los retardos para los componentes ***C1*** y ***C2***. El retardo relativo ***D*** de ***C1*** a ***C2*** se expresa como ***D=d1/d2***. Sea ***De*** el retardo relativo estimado por el diseñador, y sea ***Da*** el retardo relativo observado a través del tiempo de vida del sistema. La relación de variación relativa ***R*** se expresa como ***R=Da/De***. Luego, el modelo SDI asume que ***R*** está acotada para cualquiera de dos componentes, es decir, $1/K < R < K$, donde **K** es una constante y se llama relación de variación máxima.

La figura 2.5 ilustra el orden de cambio de señales garantizado en los modelos *QDI* y *SDI*. Lo que el modelo *QDI* garantiza es que el cambio de la señal ***t*** precede el cambio en las señales ***t1*** y ***t2***. De otro lado, el modelo *SDI* supone que los retardos estimados para las rutas ***CC1*** y ***CC2*** son ***d1*** y ***d2***, respectivamente. Entonces, se garantiza que el cambio de señal en ***t1*** precede a ***t2*** si ***d2 > K*d1***.

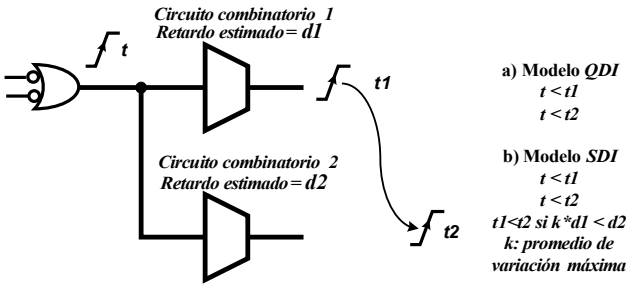


Figura 2.5: Orden de señal garantizada (a) Modelo *QDI*, (b) Modelo *SDI* [36]

2.3.7. *Micropipelines*

La metodología de *micropipelines* [39], propuesta por Ivan Sutherland, pretendía ser en principio una alternativa asíncrona a los *pipelines* elásticos síncronos (donde la cantidad de datos dentro de ellos puede variar). Posteriormente se observó que también podían ser utilizados de manera eficiente en aplicaciones que requirieran cálculos segmentados. A grandes rasgos un *micropipeline* se compone de un *datapath* con retardos acotados en cada etapa y gobernado por un circuito de control insensible a retardos [7].

El nombre de *micropipeline* se ha asignado a una forma particularmente simple de *pipeline elástico* manejado por eventos, con o sin procesamiento interno. El nombre '*micro'* se debe a que los *micropipelines* contienen circuitería muy simple, a que son útiles en longitudes muy cortas y también a que son adecuados para *layouts* en microelectrónica.

El procesador *pipeline* es un paradigma común para máquinas de computación de muy alta velocidad. Los procesadores *pipeline* deben su alta velocidad al hecho de que sus etapas separadas operan concurrentemente. Estos procesan y almacenan datos alternando, a lo largo del circuito, los elementos de memoria y los elementos de procesamiento. Los *pipelines* pueden ser sincronizados por reloj o manejados por eventos. Algunos *pipelines* son *inelásticos* (donde la cantidad de datos en ellos es fija). Las tazas de velocidad de entrada y de salida de un pipeline inelástico son idénticas. Otros *pipelines* son *elásticos* (la cantidad de datos en ellos puede variar); la taza de velocidad de entrada y la de salida pueden diferir momentáneamente debido al almacenamiento interno (*buffering*). Un *pipeline* elástico se asemeja a una memoria *FIFO* (*first-in-first-out*). Las *FIFOs* pueden ser manejadas por reloj o por eventos, su propiedad importante es que ellos son elásticas.

Los *micropipelines* están conformados por distintos circuitos para llevar a cabo transiciones de señales, estos incorporan módulos básicos que funcionan de acuerdo con

distintas combinaciones lógicos de eventos. Aquí hay algunos ejemplos:

- El circuito ***OR exclusivo XOR***, actúa como el elemento ***OR*** para eventos. Cuando una de ambas entradas de un circuito XOR cambia de estado, su salida también cambia. Es decir, un evento recibido sobre la primera entrada OR la segunda entrada de la XOR, producirá un evento de salida.
- El elemento ***C de Muller***, actúa como el elemento ***AND*** para eventos. Cuando ambas entradas de un elemento C de Muller están en el mismo estado lógico, el estado del elemento C de Muller y su salida son copias de ese estado. Cuando las dos entradas difieren, el elemento C de Muller usa almacenamiento interno para retener su estado previo y mantener su salida invariante. Así, solo después que un evento tome lugar sobre sus entradas, el elemento C de Muller producirá un evento en su salida. Se utiliza el símbolo lógico estándar AND con una 'C' dentro de este para representar el elemento C de Muller que implementa una AND lógica para eventos de transición. La figura 2.6 muestra el símbolo de circuito y la implementación a nivel de transistores para dos entradas.

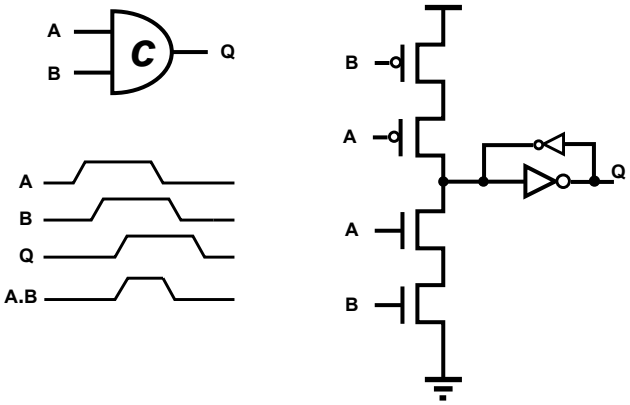


Figura 2.6: Elemento C de Muller

La figura 2.7 muestra otros símbolos para circuitos controladores de eventos que implementan operaciones elementales:

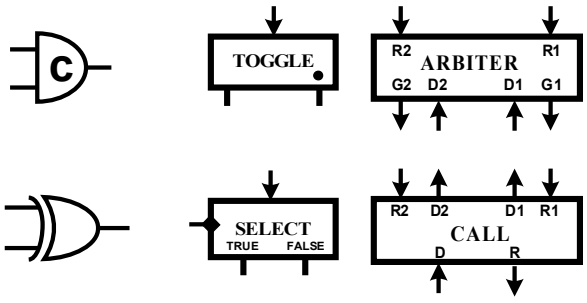


Figura 2.7: Módulos lógicos para control de eventos [39]

- El elemento ***TOGGLE*** dirige eventos a la salida de manera alterna, comenzando con la salida marcada con punto
- El elemento ***SELECT*** dirige eventos a la salida de acuerdo con el valor booleana presente en la entrada marcada con rombo.

- El elemento **CALL** recuerda cual cliente, R1 ó R2, llamó al procedimiento R, y luego de ser atendido ese procedimiento se retorna un evento D sobre D1 ó D2.
- El elemento **ARBITER** garantiza un servicio, G1 ó G2, a una sola solicitud de entrada, R1 ó R2 que se dan casi al mismo tiempo, retardando subsiguientes servicios hasta que se haya completado el servicio indicando un evento sobre D1 ó D2.

Sutherland también describe varios diseños como elementos de memoria llamados *registros controlados de eventos*, los cuales responden simétricamente a transiciones de subida y de bajada sobre las entradas. Tales *pipelines* han sido utilizados por muchos investigadores en el diseño de microprocesadores asíncronos. Sutherland, Sproull, Molnar y otros en los laboratorios de Sun Microsystems, diseñaron el *'Counterflow microprocessor'* basado en *micropipelines* [40]. Los *micropipelines* también forman la base del microprocesador *Manchester ARM* (desarrollado por Furber y el grupo AMULET) [41],[42].

Control para *Micropipelines*

La figura 2.8 muestra una cadena de elementos *C de Muller* con inversores interpuestos. Esta es la lógica requerida para controlar el *micropipeline* descrito. Según la figura, una solicitud y una señal de reconocimiento pasan entre etapas adyacentes de este control. Las líneas de datos también pasan entre etapas. Para cada interfaz entre etapas las señales de solicitud y reconocimiento y los datos siguen exactamente un protocolo de dos fases. Cada etapa en la figura 8 sigue la siguiente regla:

> ***IF*** *el predecesor y el sucesor difieren de estado*
> ***THEN*** *copiar el estado del predecesor*
> ***ELSE*** *mantenga el estado presente*

Esta regla de estado de etapa hace estable al sistema de control cuando todas las etapas están en el mismo estado y cuando las etapas alternas están en estados opuestos. La condición en la cual todos los elementos de control están en el mismo estado corresponde a un *pipeline* vacío, y la condición en la cual etapas alternas están en estados opuestos corresponde a un *pipeline* lleno. Para inicializar el *micropipeline* a la condición de vacío, sus elementos *C de Muller* deben ponerse todos en el mismo estado por medio de una señal de *clear* maestra (los circuitos de *reset* de las celdas C se han omitido).

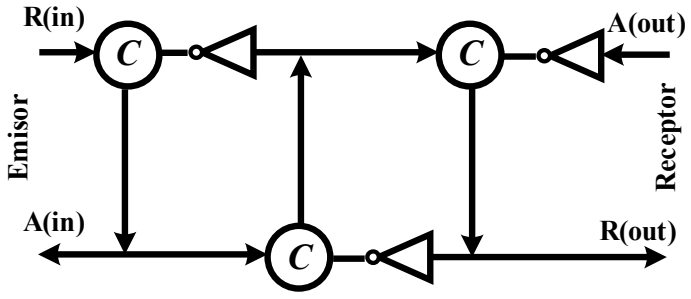


Figura 2.8: Circuito de control para *Micropipeline* [7],[39]

Hay dos tipos de circuitos *micropipelines*: *sin procesamiento* y *con procesamiento*. Los primeros cuentan con registros controlados por eventos para ruta de datos y elementos C de Muller para control. La figura 2.9 muestra un *micropipeline* de cuatro etapas con procesamiento. Entre etapas adyacentes se conforma un protocolo de dos fases de datos acotados. Los *micropipelines* sin procesamiento omiten la lógica combinatoria entre los registros controlados por eventos.

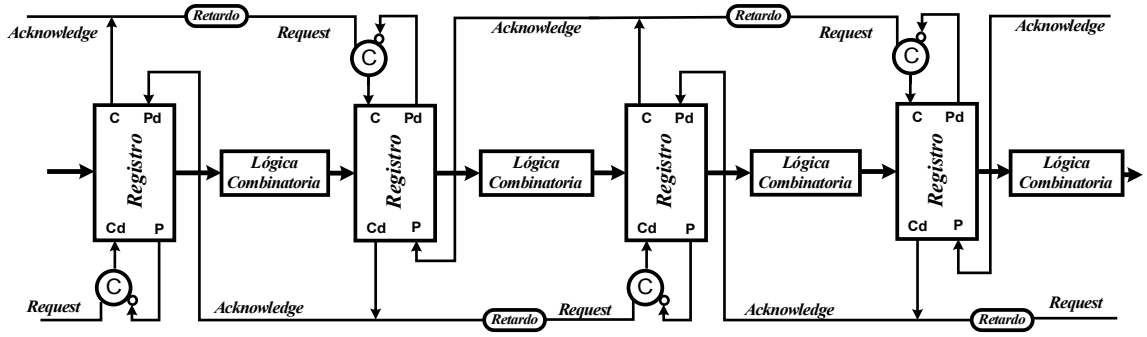


Figura 2.9: *Micropipeline* con procesamiento [7],[39]

Las investigaciones sobre *micropipelines* asíncronos y '*datapaths*' están avanzando en muchas direcciones. Se ha propuesto nuevos esquemas de *pipelines* asíncronos, algunos enfatizan la baja potencia [43]-[45] mientras que otros enfatizan el alto desempeño [44]-[47]. También se han propuesto algunas generalizaciones a las estructuras de *pipelines* asíncronos: *rings* (anillos) [48],[49], *multi-rings* [50] y *2-dimensional micropipelines* [51]. Se han establecido estructuras *micropipeline* de baja potencia usando escalamiento adaptativo de fuentes de voltaje [52].

2.3.8. Método de de-sincronización

Se trata de una metodología alterna para derivar circuitos asíncronos a partir de circuitos síncronos optimizados mediante el reemplazo del árbol de distribución de reloj por una red de *handshake*. La novedad de esta metodología es que no se requiere de un conocimiento del diseño asíncrono, en particular, esta metodología parte de una especificación HDL sintetizable o una lista de nodos a nivel de compuertas, para conseguir las ventajas claves de la asincronicidad. La idea esencial de la aproximación por de-sincronización es comenzar desde un circuito síncrono completamente sintetizado (o diseñado manualmente), y luego reemplazar directamente la red de reloj global por un conjunto de circuitos de *handshake* locales (figura 2.10).

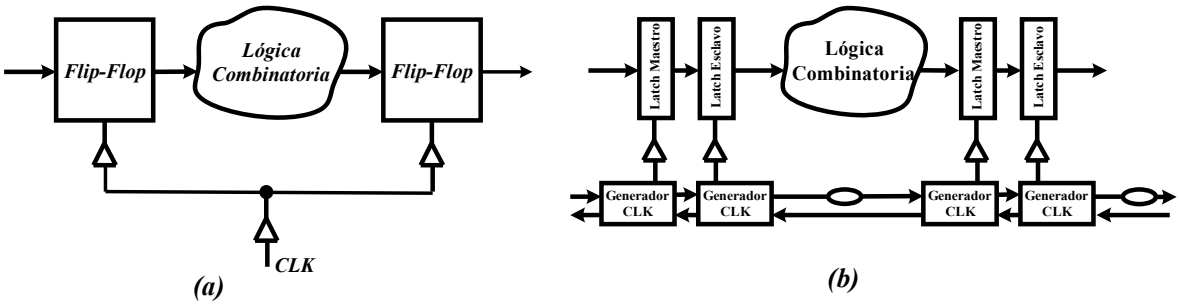


Figura 2.10: Circuito síncrono (*a*) y de-sincronizado (*b*) [35]

Datos de diseños reales muestran que en circuitos *ASIC* de alto desempeño la red de reloj disipa entre el 30 y el 50% de la potencia dinámica total. El método reduce en forma considerable este consumo ya que los relojes se generan localmente sin los nocivos efectos de los retrasos de reloj (*clock skew*). Más aún, la emisión electromagnética (EMI) también se reduce drásticamente ya que los elementos de memoria (*latches* o *flip-flops*) no conmutan en forma simultánea [53]-[55].

2.4. Procesadores Asíncronos

En 1988 se diseñó el primer microprocesador completamente asíncrono del mundo, el *CAM* (*Caltech Asynchronous Microprocessor*) es un microprocesador RISC (*Reduced Instruction Set Computing*) de 16 bits con 23000 transistores; una década después se diseñó el *MiniMIPS*, una versión asíncrona del microprocesador MIPS R3000 de 32 bits [33]. En 2003 fue diseñado el procesador asíncrono *Lutonium* que es compatible con el microcontrolador 8051 asíncrono orientado al bajo consumo de energía [56],[57]. La tabla 2.1 permite comparar los resultados de desempeño de varios procesadores síncronos con su contraparte síncrona el *Lutonium* [58]. En la tabla 2.2 se observa el desempeño de algunos procesadores comerciales similares al procesador *MiniMIPS* [32].

Tabla 2.1: Comparación de desempeño de algunos procesadores síncronos con el microprocesador asíncrono *Lutonium* [58]

Microprocesador	*Tecnología del Proceso*	*Frecuencia*	*Consumo de Potencia*	*MIPS/mW*	*Año de Fabricación*
Pentium Pro (síncrono)	0.60 µm	150 MHz	23 W	0.0065	1995
Celeron (síncrono)	0.25 µm	400 MHz	23.4 W	0.017	1999
Athlon (K7) (síncrono)	0.25 µm	700 MHz	45 W	0.019	1999
DEC21364	0.18 µm	1 GHz	100 W	0.047	2000
SH7708	0.50 µm	60 MHz	0.6 W	0.1	1994
Lutonium Asíncrono. 1.8 V	0.18 µm	200 MHz	0.10 W	1.8	2004
Lutonium Asíncrono 0.5 V.	0.18 µm	4 MHz	0.17 mW	23	2004

Tabla 2.2: Comparación de desempeño del Procesador MiniMIPS con otros Procesadores Comerciales [33]

Microprocesador	*Tamaño de Word*	*Tecnología [/µm]*	*Frecuencia [/MHz]*	*Potencia por bit*	*Energía [/10^{-10}J]*	*Et^2 * [/10^{-26} Js^2]*
MiniMIPS (Simulado)	32	0.60	280	0.2190	7.8	1.0
MiniMIPS (fabricado)	32	0.60	180	0.1250	7.0	2.1
R3000 (cpu síncrona)	32	1.20	25			
R3000A (cpu síncrona)	32	1.00	33			
VR3600 (cpu+fpu)	32	0.80	40			
R4600 (síncrono)	64	0.64	150	0.0719	4.8	2.1
21064 (síncrono)	64	0.60	200	0.4690	23.5	2.1
R4400(síncrono)	64	0.60	150	0.2340	15.6	7.0
SH7708 (síncrono)	16/32	0.50	60	0.0180	3.0	8.3
P6 (síncrono)	32	0.60	150	1.8000	120.0	52.0

La figura 2.11, ofrece una comparación: la línea continua corresponde a la relación $\mathbf{Et^2}$ del *MiniMIPS* en 0.6µm; la línea punteada muestra el ajuste para la tecnología de 0.35µm. La métrica Et^2 relaciona la energía promedio por instrucción y el tiempo de ciclo, y permite estimar, de forma alterna, el consumo de potencia dinámica ya que tal relación es independiente de la fuente de voltaje [59]. En la figura, los puntos representan cinco procesadores diferentes: el *MiniMIPS,* diseñado en tecnología de 0.6µm, se compara con el procesador síncrono *DEC Alpha 21064* [60] y dos diseños asíncronos en 0.5µm, el *Titac2* [33],[34] (microprocesador MIPS diseñado en la Universidad de Tokio) y el *AMULET2e* (microprocesador ARM diseñado en la Universidad de Manchester). En tecnología de 0.35µm, las comparaciones son con el *DEC Alpha 21164* [60] y el *StrongARM SA110* [61].

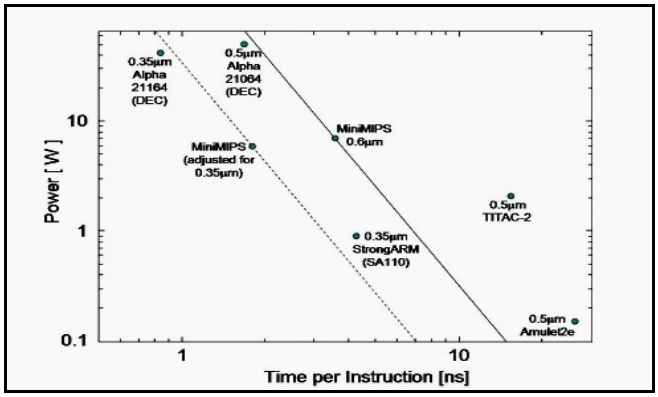


Figura 2.11: Desempeño de otros procesadores relativo a las líneas iso-Et^2 del MIPS en tecnologías de 0.6 y 0.35µm [33]

Cuando el MiniMIPS opera a su voltaje de operación nominal, se demuestra la eficiencia de energía de los circuitos QDI y su gran robustez a las variaciones de parámetros físicos (voltaje de fuente y temperatura). La habilidad de los circuitos integrados QDI para operar a muy bajos voltajes [59], aún ligeramente debajo del umbral de voltaje del

transistor, permite la gestión de energía por instrucción, ***E***, en relación al tiempo de ciclo, ***t***, a través del ajuste de la fuente de voltaje. Cuando la energía y el tiempo deban ser optimizados, surge la pregunta sobre qué exactamente optimizar, y qué tan importante es la energía comparada con el tiempo y viceversa. Estos dos factores se combinan en la métrica Et^2, la cual es independiente del voltaje para circuitos CMOS. Debido a esta independencia de la fuente de voltaje, se puede escoger entre diseños de circuitos conociendo el menor Et^2 a un voltaje dado dentro de condiciones normales de operación; esto significa que si la fuente de voltaje puede ser ajustada, el circuito con menor Et^2 puede obtener un tiempo de ciclo mas bajo que uno con mayor Et^2 con el mismo consumo de energía, o en forma equivalente, una menor energía en el mismo tiempo de ciclo. En la tabla 2.3 se realiza una comparación entre el desempeño del Lutonium y otros diseños de la familia 8051 fabricados en tecnología CMOS de 0.5µm.

Tabla 2.3: Comparación del Lutonium en 0.18µm con otros diseños 8051 (Todos en tecnología CMOS 0.5µm) [33]

Microprocesador	***MIPS***	***MIPS/W***	Et^2 [$/10^{-25}$ Js^2]
Lutonium	100	600	1.7
Philips 8051 (asíncrono)	4	444	6300
Philips 8051 (síncrono)	4	100	1400
Dallas DS89C420 (Síncrono, *'Ultra-high-speed')*	50	100	40

Los resultados de la tabla 2.3 muestran que el procesador Lutonium realiza un mayor número de instrucciones por Wattio que los demás procesadores y a la vez tiene una menor relación Et^2 (superior en un factor de 24 al Dallas DS89C420).

2.5. Estado del Diseño Asíncrono en la Industria

De acuerdo con [62] en reportes sucesivos de la *IST (Information Society Technologies)* sobre el "Estado del Diseño Asíncrono en la Industria" se hace un reconocimiento de las compañías comprometidas, o que muestran interés en la tecnología asíncrona. Las compañías involucradas se dividen en:

- aquellas con modelos de negocios primarios basados en tecnología asíncrona
- aquellos que venden productos que incorporan tecnología asíncrona
- aquellos con una significativa presencia de investigación en circuitos asíncronos
- aquellas afiliadas al *ACiD-WG* (*Working Group on Asynchronous Circuit Design*) y quienes siguen la pista de la tecnología pero no tienen un esfuerzo visible en investigación
- aquellas que advierten un interés en la tecnología asíncrona, pero que no tienen un perfil significativo en la comunidad asíncrona.

En [62] se describe el estado de los métodos y herramientas para el diseño de sistemas VLSI digitales asíncronos hasta el año 2004. El reporte fue pensado para que fuese utilizado por compañías miembros y no miembros del *ACiD-WG,* quienes han sido conscientes de los beneficios potenciales de la tecnología de circuitos asíncronos y que

requerían mayores conocimientos acerca de los métodos de diseño y herramientas antes de comprometer recursos. El reporte resaltó deficiencias en las aproximaciones existentes para motivar el posterior desarrollo de herramientas. El reporte también contiene auto-evaluaciones recibidas de varios desarrolladores de herramientas. Según este informe, *"las metodologías de prueba para circuitos asíncronos están todavía en su etapa de infancia"*. El documento presenta un anexo que identifica metodologías y herramientas de diseño. Se listan un total de 22 herramientas/metodologías en orden alfabético. Cuatro de ellas (ACK, Balsa, Tangram y TAST) están principalmente ocupados con la compilación de silicio, y dos (LARD y VHDL++) con simulación. Otros dos (BUTLER y NCL *technologies*) explotan librerías de componentes de propósito especial. Cinco mas (3D, ATACS, MINIMALIST, Petrify y SIS) realizan síntesis lógica, tres mas (di2pn, Pipefitter and VSTGL) se usan como parte de Petrify. Finalmente, seis (CADP, CCS, Firemaps, Transyst, Veraci y XDI) soportan verificación formal.

En 2007, el *APT Group* (*The Advanced Processor Technologies Group*) perteneciente a la Escuela de Ciencias de la Computación de la Universidad de Manchester (Inglaterra), publicó una nueva lista de herramientas para diseño asíncrono en su portal de Internet (http://intranet.cs.man.ac.uk/apt/async/tools/). Varias de las herramientas mencionadas en 2004 ya no están disponibles o simplemente su vinculo URL (*Universal Resource Locator*) ha cambiado. La tabla 2.4 muestra la lista de herramientas de diseño asíncrono y la plataforma de sistema operativo sobre la cual corre.

Se puede observar en la tabla que la mayoría de las herramientas trabajan bajo plataformas compatibles con Unix (Linux, Solaris, MacOSX).

Tabla 2.4: Herramientas de diseño asíncrono [63]

Herramienta	***Plataforma***
ATACS	*Linux, MacOS, Windows, Sun Solaris*
ATN_OPT	*Linux*
Balsa	*Unix*
CaSCADE tools	*Linux*
DES Analyzer	*Linux*
DGC	*Windows, UNIX, MacOSX*
DI2PN	*Unix*
Diana	*Unix*
LARD	*Unix*
MINIMALIST	*Linux*
MVSIS 2.0	*Source*
PCHB/STFB	*MOSIS*
SPHINX	*Solaris*
Syndi	*Unix*
VerilogCSP	*Verilog*

La comisión Europea fundó el Grupo de Diseño de Circuitos Asíncronos *ACiD-WG* (*Working Group on Asynchronous Circuit Design*), el cual tuvo una vigencia de cinco

años (entre septiembre del 2000 y enero del 2005). Este grupo fué financiado bajo el programa FP5 (*Fifth Framework Programme*) *Microelectronics* [64] de la IST (*Information Society Technologies*).

Los objetivos principales del programa FP5 mediante el *ACiD-WG* apuntaron a mejorar el intercambio de información y la creación de enlaces entre equipos y promover actividades alrededor del tema del diseño de circuitos asíncronos. Los objetivos fueron los siguientes:

1. Estimular la excelencia en la investigación en ciencia y tecnología pertenecientes a los circuitos y sistemas asíncronos
2. Facilitar el desarrollo de métodos y herramientas que sean usadas por ingenieros para el diseño de sistemas VLSI asíncronos.
3. Promover la adopción del diseño de circuitos asíncronos en la industria.

Los resultados que evidenciaron el éxito del grupo *ACID-WG* en los objetivos propuestos fueron publicados en el reporte "*Design, Automation and Test for Asynchronous Circuits and Systems"* [64], el cual tiene un valor significativo para usuarios potenciales de la tecnología de circuitos asíncronos y desarrolladores de herramientas. La tabla 2.5 enumera las instituciones miembro del *ACID-WG*, las cuales participaron en el desarrollo de los objetivos planteados.

Tabla 2.5: Instituciones Miembro de *ACID-WG* [64]

Número Participante	***Nombre Corto***	***Código País***	***Tipo de Organización***	***Fecha Vinculación***	***Fecha Terminación***	***Representante ante comité***
1	**LSBU**	UK	U	1/9/00		Mark Josephs (Chair)
2	**Philips Research**	NL	I	1/9/00		Ad Peeters
3	**Infineon**	D	I	1/9/00		Christoph Heer
4	**ST**	F	I	1/9/00		Jean-Pierre Schoellkopf
5	**CSEM**	S	R	1/9/00		Christian Piguet
6	**IHP**	D	R	1/9/00		Eckhard Grass
7	**MBDA**	UK	I	19/3/02		Eric Campbell
8	**UoM**	UK	U	1/9/00		Doug Edwards
9	**UNew**	UK	U	1/9/00		Alex Yakovlev
10	**UCam-CLab**	UK	U	1/9/00		Simon Moore
11	**UPC**	E	U	1/9/00		Jordi Cortadella
12	**UDI**	I	U	1/9/00	2/5/02	Luciano Lavagno
13	**TUE**	NL	U	1/9/00	5/4/03	Tom Verhoeff
14	**INPG-TIMA**	F	U	1/9/00		Marc Renaudin
15	**DTU**	DK	U	1/9/00		Jens Sparsø
16	**Technion**	IL	U	1/9/00		Ran Ginosar
17	**Åbo Akademi**	FIN	U	1/9/00		Kaisa Sere
18	**AT&T**	UK	I	5/6/02	24/4/02	Phil Endecott
19	**NNT**	IL	I	2/5/02	2/1/05	Victor Varshavsky
20	**PoliTo**	I	U	2/5/02		Luciano Lavagno
21	**ICS-FORTH**	EL	U			Christos Sotiriou

U = Universidad; R= Research (Instituto de investigación); I= Organización Industrial

Las siguientes compañías son afiliadas industriales del Grupo de Trabajo:

• Accentime Inc, CA
• ARM Limited, UK
• Cadence Design Systems, US
• ExMark (Technical Marketing Services), UK
• Intel Corporation (Strategic CAD Labs), US
• Kramer-Consulting, DE
• Philips Semiconductors (Automotive), NL
• Sun Microsystems Laboratories (Asynchronous Design Group), US
• Theseus Logic, US

En la actualidad, el evento más importante a nivel mundial es el *IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC).* En 2009 este evento se realizó en *Chapel Hill, North Carolina*, E.U con el nombre de *ASYNC-2009*. En mayo de 2010 se realizará el *ASYNC-2010* en Grenoble, France. El evento reúne los científicos más destacados en el área de la investigación y el desarrollo de circuitos asíncronos, entre los principales tópicos o sub-áreas tratados se pueden mencionar los siguientes:

- Arquitecturas Mixtas síncronas/asíncronas, interfaces y circuitos.
- Técnicas de diseño, síntesis y verificación para sistemas GALS.
- Interacción síncrona-asíncrona a diferentes niveles.
- Lógica asíncrona de alta-velocidad/baja-potencia para memorias e interconexiones.
- Diseño de alto-nivel y síntesis de circuitos auto-temporizados.
- Diseño físico de lógica sin reloj y *pipelines*.
- Métodos Formales para desempeño y análisis de diseños asíncronos.
- Test, confiabilidad, seguridad y tolerancia a la radiación.
- CAD para diseño y validación asíncrona.
- *Asynchronous System-on-Chip (SoC), System-in-Package (SiP).*
- *Networks-on-Chip (NoC).*
- Nuevas Arquitecturas asíncronas.
- Asincronismo y tolerancia de latencia en diseño a nivel de sistema.
- Motivaciones de estudios de caso, comparaciones y aplicaciones.
- Diseño de sistemas embebidos con arquitecturas/implementaciones asíncronas.
- Diseño asíncrono para manufactura.

Cada uno de los sub-temas de este listado se puede considerar como un importante desafío a fin y se puede considerar como punto de partida hacia un trabajo futuro a nivel de formación de posgrado en ingeniería.

2.6. Sistema Balsa

Para realizar la implementación asíncrona del algoritmo de Rijndael, en este trabajo de tesis de doctorado, se utilizó el programa *Balsa* [34],[65]. Esta herramienta se seleccionó teniendo en cuenta las siguientes consideraciones:

- Actualmente es la herramienta de libre distribución mas avanzada para la descripción de sistemas asíncronos.
- Utiliza un lenguaje de alto nivel (también llamado Balsa).
- Integra ambientes para especificación, síntesis, y simulación de circuitos asíncronos, además de herramientas que permiten generar archivos de implementación sobre hardware real.

El sistema Balsa fué desarrollado en el *APT Group* (*The Advanced Processor Technologies Group*) de la *School of Computer Science, University of Manchester* (Inglaterra). *Balsa* es el nombre de un conjunto de herramientas de síntesis de sistemas de hardware asíncrono y también el nombre del lenguaje para describir esos sistemas [65]. Las herramientas pueden correr sobre cualquier ambiente POSIX[2] con X11 que manejen enteros al menos de 32 bits (*Linux*, *FreeBSD*, *MacOS X*, *Solaris*). Sin embargo, para producir una implementación en hardware real, ya sea en Silicio o en *FPGA*, se requiere el complemento de herramientas comerciales específicas: por ejemplo el software de diseño de *Xilinx*, o el ambiente de diseño de *Cadence* con una apropiada tecnología de librería de celdas.

Balsa utiliza una metodología de compilación dirigida a sintaxis entre componentes de *handshake,* muy parecida a la que usa el sistema *Tamgram* de Philips [34], [65]. La ventaja de esta metodología es que la compilación es transparente: hay un mapeo uno-a-uno entre las construcciones del lenguaje en la especificación y los circuitos intermedios de *handshake* que se generan. Es relativamente fácil para un usuario experimentado imaginar la arquitectura del circuito que resulta producto de la descripción original. Los cambios hechos a nivel de lenguaje resultan en cambios predecibles a nivel de implementación de circuito. Lo anterior es importante ya que las optimizaciones y las mejoras de diseño son fáciles a nivel de código fuente, lo cual contrasta con la descripción en VHDL, en la que pequeños cambios en la especificación pueden producir radicales alteraciones en el circuito resultante [34].

Un circuito descrito en lenguaje Balsa se compila en una red de comunicación que se compone de un pequeño conjunto de componentes de *handshake* (alrededor de 45, los cuales se listan en el anexo 2.A como '*Componentes Breeze*')**.** Los componentes están conectados mediante canales sobre los cuales toman lugar *comunicaciones* o *handshakes.* Los canales pueden tener rutas de datos asociados (en cuyo caso un *handshake* involucra una transferencia de dato), o pueden ser solo de control (en cuyo caso el *handshake* actúa como un punto de sincronización). Cada canal conecta un puerto pasivo de componente

[2] POSIX (IEEE,1996) es el estándar de interfaz de sistemas operativos portables de IEEE basado en el sistema operativo UNIX. POSIX se ha normalizado dentro de IEEE con la referencia 1003 y está siendo desarrollado como estándar internacional con la referencia ISO/IEC 9945. POSIX también constituye una familia de estándares en evolución, cada uno de los cuales cubre diferentes aspectos de los sistemas operativos [66].

de *handshake* a un puerto activo de otro componente de *handshake*. Un puerto activo es un puerto que inicia una comunicación. Un puerto pasivo responde (cuando está listo) a una solicitud (*request*) hecha desde un puerto activo mediante una señal de *acknowledge.*

2.6.1 Conjunto de herramientas y flujo de diseño de Balsa

El flujo de diseño de Balsa, mostrado en la figura 2.12, se desarrolla gracias al conjunto de herramientas que se listan a continuación:

• ***balsa-c:*** es el compilador para el lenguaje Balsa. La salida del compilador es un lenguaje intermedio llamado *breeze*.

• ***balsa-netlist:*** esta herramienta produce un archivo tipo '*netlist'* (lista de nodos) compatible con la apropiada tecnología/herramienta CAD (*Computer Aided Design*) desde una descripción de *breeze*.
• ***breeze2ps***: es una herramienta que produce un archivo *postscript* correspondiente a un grafo de circuito de *handshake*.

• ***breeze-cost:*** es una herramienta que brinda el costo estimado de área del circuito

• ***balsa-md:*** herramienta para la generación de '*makefiles'*.
• ***balsa-mgr:*** brinda un *front-end*[3] gráfico para *balsa-md* con facilidades para el manejo de proyectos.

• ***balsa-make-test:*** permite generar un archivo de prueba ('*test harness'*) para una descripción de Balsa.

• ***breeze-sim:*** es el simulador utilizado para trabajar a nivel de componentes de *handshake*.

• ***breeze-sim-control:*** *front-end* gráfico para el ambiente de simulación y visualización.
Se compone de dos paquetes separados:

- ***gtkwave:*** un visor de formas de onda.
- ***balsa-verilog-sim:*** es un paquete que realiza simulación *Verilog* de descripciones Balsa. Este utiliza funciones escritas por el usuario las cuales pueden ser llamadas desde Balsa.

[3] ***Front-End:*** este término, al igual ***back-end,*** no aparecen definidos en diccionarios impresos de idioma inglés. Se trata de neologismos asociados con la terminología general de computadores y aparecen definidos en 'diccionarios digitales' de Internet. Según [67], la definición es la siguiente: **"***Front-end* y *back-end* son términos usados para caracterizar interfases de programa y servicios relativos al usuario inicial de estos. El 'usuario' puede ser una persona o un programa. Una aplicación *'front-end'* es aquella con la que el usuario interactúa en forma directa. Una aplicación o programa *'back-end'* sirve indirectamente de soporte para los servicios de *front-end*.**"**

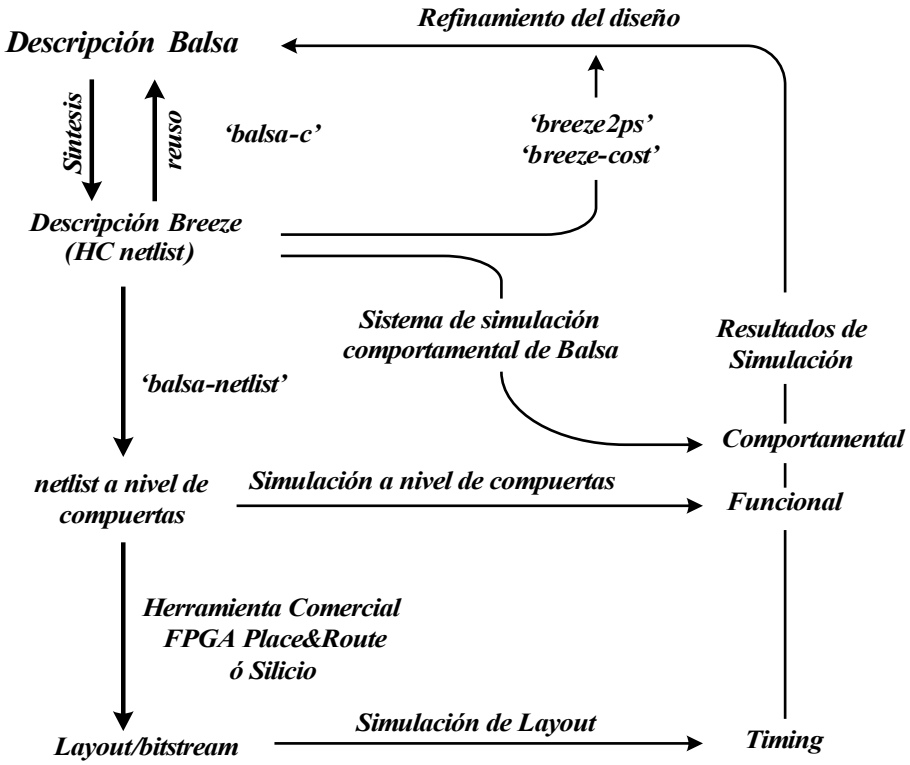


Figura 2.12: Flujo de diseño de Balsa [65]

Un circuito descrito en Balsa se compila usando *balsa-c* para obtener una descripción *breeze* intermedia. La mayoría de las herramientas de Balsa están relacionadas con la manipulación de archivos intermedios de *handshake* de *breeze*.

La simulación comportamental es provista por *breeze-sim*. Este simulador, a pesar que está aun bajo desarrollo activo, permite depuración a nivel de código fuente, además de visualización de la actividad de canal a nivel de circuito de *handshake* produciendo trazos de forma de onda convencionales que se pueden observar por medio del visor de ondas *gtkwave*. También se puede utilizar la herramienta previamente seleccionada de CAD para realizar simulaciones mas precisas y así validar el diseño.

Balsa es primordialmente un sistema de síntesis, por lo que para producir una implementación en silicio o en arreglos de compuertas real se requiere acceder a herramientas CAD comerciales. Para este propósito, Balsa únicamente produce un archivo del tipo '*netlist*' en formato apropiado para un sistema CAD que soporte la tecnología. Cuando se crea una implementación, los usuarios pueden escoger una tecnología particular, diferentes estilos dentro de esa tecnología y para cada estilo una variedad de opciones. Un circuito descrito en Balsa se puede compilar en una red de comunicación compuesta de un conjunto de componentes de *handshake*.

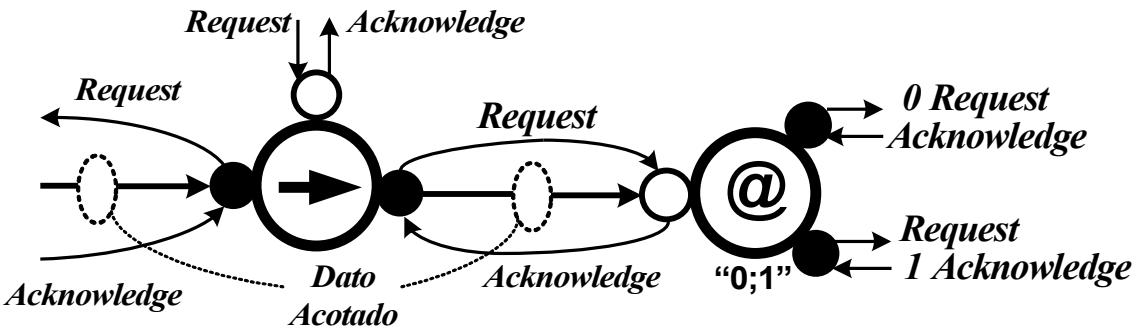


Figura 2.13: Conexión de componentes de *handshake* [65]

La figura 2.13 muestra un ejemplo de un circuito de *handshake* en el que un componente *fetch* (u operador de transferencia) denotado por "→" y un componente *case* denotado por "@" están conectados por medio de un canal orientado a datos. La acción del circuito es activada por un evento de solicitud (*request*) al componente *fetch* el cual a su vez establece un *request* a su entorno (al lado izquierdo del diagrama). El entorno provee el dato solicitado e indica su validez con una señal de reconocimiento (*acknowledge*). El componente *fetch* presenta una señal de *request* y el dato al componente *case* usando un puerto activo (mostrado como un circulo negro), estos son recibidos sobre el puerto pasivo de *case* (mostrado como círculo en blanco). Dependiendo del valor del dato, el componente *case* establece un *handshake* en su entorno sobre el puerto inferior y superior del lado derecho. Finalmente, la señal *acknowledge* es recibida por el componente *case*, se retorna otra señal *acknowledge* a lo largo del canal original terminando así el *handshake*. El circuito está listo para operar de nuevo.

En este ejemplo, el dato sigue la dirección de *request* y el reconocimiento (*acknowledge*) para ese *request* fluye en dirección opuesta. En la figura, el *request* físico, las líneas de *acknowledge* y de dato se muestran en forma explícita. El dato es llevado sobre líneas separadas por medio del protocolo de señales (este está 'acotado' con el control), aunque esto no es necesario con otros esquemas de codificación de datos/protocolos. El esquema de 'dato acotado' ilustrado en la figura 2.13 no es la única implementación posible. Existen otras metodologías para implementar conexiones de canal con protocolos insensibles a retardo donde las relaciones de temporización entre líneas individuales de un canal no afectan la funcionalidad del circuito [65].

2.6.2. Tecnologías

Actualmente Balsa, mediante *balsa-mgr,* soporta y controla un rango amplio de estilos y tecnologías. Los usuarios pueden seleccionar entre *single-rail* (dato acotado), *dual-rail* y *1-de-4*. Cada tecnología tiene su propia librería de celdas, restricciones de *fan-in* de compuertas y convenciones de mapeo de pines. Cada tecnología puede también usar varios formatos de lista de nodos (*netlist*) diferentes. Las tecnologías deben ser descargadas e instaladas como paquetes separados, estas son:

- ***balsa-tech-ams:*** esta tecnología soporta el kit de diseño AMS 350ηm y produce

una lista de nodos de *Verilog* (*Verilog netlist*).

- ***balsa-tech-amulet:*** esta tecnología contiene un conjunto de celdas diseñadas por el grupo Balsa. Está basada en la biblioteca SGS-ST 180ηm y produce una lista de nodos de *Verilog*.

- ***balsa-tech-sths018:*** esta tecnología contiene únicamente celdas estándar de la biblioteca SGS-ST 180ηm.

- ***balsa-tech-example:*** esta tecnología produce una descripción *Verilog* basada en ejemplos de celdas y esta orientada con perfiles para usuarios que deseen crear sus propios *back-ends*.

- ***balsa-tech-xilinx:*** esta tecnología produce una lista de nodos EDIF apropiada para arreglos de compuertas de Xilinx.

2.6.3. Estilos

La actual versión de Balsa (3.5.1) soporta los siguientes protocolos de *back-end* para usar con cada tecnología:

- ***four_b_rb:*** un esquema de dato acotado usando un protocolo de señales *four-phase-broad/reduced-broad*.

- ***dual_b:*** una codificación *dual-rail delay-insensitive*.

- **one_of_2_4:** una codificación *1-de-4 delay-insensitive*.

El estilo de dato acotado debería ser el más rápido y más pequeño, pero necesita una validación de temporización *post-layout* mas completa. Los dos esquemas *delay-insensitive* son más grandes y más lentos y podrían ser mas robustos ante variaciones de layout. Las opciones de cada estilo se pueden consultar en el anexo 2.B.

Se debe resaltar que Balsa permite activar la opción de estilo de *lógica balanceada* la cual permite crear circuitos balanceados en los que los retardos de ruta a través de los circuitos asíncronos están orientados a evitar los ataques tipo *DPA (Differential power Analysis)* especialmente en aplicaciones criptográficas.

2.7. Conclusiones del capítulo

2.7.1. El estado del arte indica que el diseño asíncrono ha sido una línea creciente de investigación que ya tiene un impacto importante en el desarrollo de herramientas de diseño como también de circuitos integrados comerciales.

2.7.2. Hay signos significativos que llevan a pensar que los circuitos asíncronos continuarán siendo una disciplina de primer orden en el inmediato futuro. Hay serios esfuerzos orientados al apoyo de la investigación por parte, especialmente, de la Comunidad Económica Europea. Esto se ve reflejado en la aparición de nuevas herramientas de diseño, como también la vinculación al desarrollo de objetivos concretos, por parte de universidades, institutos de investigación y organizaciones industriales

2.7.3. El incipiente desarrollo de herramientas para *test* y verificación formal de los circuitos asíncronos constituyen una de las principales desventajas respecto del estilo de diseño síncrono. Sin embargo, existen notables esfuerzos por parte de la comunidad académica e industrial en desarrollar herramientas de diseño como lo evidencian los resultados publicados en los *journals* de IEEE de Estados Unidos y en la IEE europea.

2.7.4. Gracias a que el sistema Balsa permite activar la opción de estilo de *lógica balanceada*, esta herramienta es ideal para la descripción asíncrona del algoritmo de Rijndael ya que tal opción incluye características apropiadas para fortalecer la implementación contra ataques a la seguridad tipo *DPA (Differential Power Analysis)*.

Referencias

[1] Davis, A., Nowick, S.M. *"An Introduction to Asynchronous Circuit Design"*. Computer Science University of Utah, Salt Lake City, UT 84112; Computer Science, Columbia University, New York, NY 10027. September 19, 1997.

[2] Hauk, S. *"Asynchronous Design Metodologies: An Overview"*. Department of Sciencie and Engineering, University of Washington, Seattle, WA 98195. Proceedings of IEEE, Vol 83, No. 1, pp. 69-93, January 1995.

[3] Bainbridge, W.J. Thesis: *"Asynchronous System-on-Chip Interconnect"* University of Manchester, Faculty of Science & Engineering. Department of Computer Science. March 2000.

[4] Ozdag, R. Ph.D. Thesis *"Template Based Asynchronous Design"*. Faculty of the graduate School University of Southern California, November 2003.

[5] Sparsø, J., Furver, S. *"Principles of Asynchronous Circuits Design- A System Perspective"*. Kluwer Academics Publishers. 2001**, ISBN 0-7923-7613-7.** Disponible en:
http://intranet.cs.man.ac.uk/apt/async/pubwork/index.html

[6] Vivet, P. *"Une Methodologie de Conception de circuits Integres Quasi-Insensibles aux delais: application a l´etude et a la realization d´un processeur RISC 16-bit asynchrone"***.** Institut National Polytechnique de Grenoble. France, Juin 2001.

[7] Peña, B., M. A. Tesina: *"Síntesis de circuitos asíncronos mediante la traducción de sincronización a redes de Petri"*. Universitate Politécnica de Catalunya. Departament d´Arquitectura de Computadores, Barcelona, marzo 1995.

[8] Liu, J. Ph.D. Thesis: *"Arithmetic and Control Components for an Asynchronous System"*. University of Manchester, Faculty of Science and Engineering, Department of Computer Science, 1997.

[9] Nelson, V.; Nagle, T.; Irwin, D. *"Análisis y Diseño de Circuitos Lógicos Digitales"*. Primera Edición, Ed. Prentice-Hall.1996. ISBN: 968-880-706-0.

[10] Tinder, R.F. "*Digital Engineering Design, A Modern Approach"*. Ed. Prentice-Hall. 1.994. ISBN: 0-13-2117077-X.

[11] Nowick S.M., Dill D.L. *"Automatic Synthesis of Locally-Clocked Asynchronous State Machines".* IEEE International Conference on Computer-Aided Design (ICCAD), Santa Clara, CA 1991.

[12] Yun K.Y., Dill D.L., Nowick, S.M. *"Synthesis of 3D Asynchronous State Machines"'*. IEEE International Conference on Computer Design (ICCD), Cambridge, MA. 1992.

[13] Davis, A., Coates, B., Stevens, K. *"The Post Office Experience: Designing a Large Asynchronous Chip"*. Proceedings of the 26th Annual Hawaii International Conference on Systems Sciences, Vol. I, pp. 409-418, 1993.

[14] Ladd, M.; Birminghan, W.P. *"Synthesis of multiple-input change asynchronous finite state machines"*. *Proceedings. ACM/IEEE Design Automation Conference*, pp. 309–314, 1991.

[15] Rosenberger, F.U., Molnar, C.E.; Fang, T. P.. *"Synthesis of delay-insensitive modules"*. Chapel Hill Conference on VLSI. pags 67-86, May 1985.

[16] Rosenberger, F.U., Molnar, C.E.; Chaney, T.J., Fang, T.P."Q-modules Internally clocked delay-insensitive modules". IEEE Transactions on Computers Volume 37, IEEE Computer Society, Washington, DC, USA Issue 9 September, 1988. Pages: 1005–1018, ISSN:0018-9340.

[17] Muller, D.E., Bartky, W.C. *"A theory of asynchronous circuits"*. Annals of Computing Laboratory of Harvard University, pp. 204-243, 1959.

[18] Miller, R.E. "*Sequential Circuits and Machines: Volume 2 of Switching Theory*". Wiley, 1965

[19] Armstrong, D.B., Friedman, A.D, Menon, P.R *"Design of Asynchronous Circuits Assuming Unbounded Gate Delays"*. IEEE Transactions on Computers, C-18, 12, pp.1110-1120, December, 1969.

[20] Murata, T. "*Petri Nets: Properties, Analysis and Applications*", Proceedings of the IEEE, vol. 77(4), , pp. 541-580, April, 1989.

[21] Rosenblum, L.Y, Yakovlev, A.V. "*Signal Graphs: from self-timed to timed ones"*. Proceedings. of the Int. Workshop on Timed Petri nets, 199-207, IEEE Computer Society, 1985.

[22] Chu, T.A. Ph.d. thesis: "Synthesis of Self-timed VLSI Circuits from Graph-Theoretic Specifications", M.I.T. Tech. Rep. MIT/LCS/TR-393, June 1987.

[23] Kishinevsky, M.A, Kondratyev, A.Y., Taubin, A.R., Varshavsky, V. *"On Self-timed Behavior Verification"*. Proceedings of TAU´92, March 1992.

[24] Lavagno, L.. PhD thesis*: "Synthesis and Testing of Bounded Wire Delay Asynchronous Circuits from Signal Transition Graphs"*.U.C. Berkeley, 1992.

[25] Moon, C.W., Stephan, R., Brayton, R.K. "Synthesis of hazard-free asynchronous circuits from graphical specifications". IEEE International Conference on Computer-Aided Design, pages 322-325, November 1991.

[26] Clark, W.A. *"Macromodular computer systems"*. Proceedings of the Spring Joint Computer Conference, AFIPS, April 1967.

[27] Udding, J.T. *"A formal model for defining and clasifying delay-insensitive circuits and systems"* . Distributed Computing, pp 1 (4): 197-204, 1986.

[28] Dill, D.L. *"Trace Theory for Automatic Hierarchical Veri_cation of Speed_Independent Circuits"* MIT Press, Cambridge, MA,

[29] Davis, A., Nowick, S.M. "An Introduction to Asynchronous Circuit Design". UUCS,Computer Science. University of Utah, Columbia University, Salt Lake City, U.T, New York, NY 10027. September 19, 1997.

[30] Burns, S., Martin, A.J. "A synthesis method for self-timed VLSI circuits". Proceedings of the International Conference on Computer Design, 1987.

[31] Martin, A.J. *"Formal program transformations for VLSI synthesis"*. E.W. Dijkstra, editor. Formal Development of Programs and Proofs UT Year of Programming Series. Addison-Wesley, 1990.

[32] Martin, A.J. *"Programming in VLSI From communicating processes to delay-insensitive circuits"*. C.A.R Hoare, editor. Developments in Concurrency and Communications, págs 1-64. UT Year of Programming Series, Addison-Wesley, 1990.

[33] Martin, A.J., Nystrom, M., Wong, C.G. *"Three Generations of Asynchronous Microprocessors"***.** California, Institute of Technology. IEEE Design and Test of Computers, vol. 20, no. 6, pp. 9-17, Nov./Dec. 2003.

[34] Bardsley, A. Thesis: "*Implementing Balsa Handshake Circuits"*. Department of Computer Science, Faculty of Sciencie & Engineering, University of Manchester, 2000.

[35] Janin, L. Thesis: *"Simulation and Visualisation for Debugging Large Scale Asynchronous Handshake Circuits"* pp 23, 24. University of Manchester, Faculty of Science & Engineering. Department of Computer Science, 2004.

[36] Yamazaki, A., Ryu, H., Yoneda, T. *"Verification of Scalable-Delay-Insensitive asynchronous circuits"***.** IEICE TRANS. INF. & SYST., VOL. E00–D, NO. 1 January, 1995.

[37] Takamura, A., Kuwako, M., Imai, M., Fujii, T. *"TITAC-2: An asynchronous 32-bit microprocessor based on Scalable-Delay Insensitive"*. Graduate School of Information Science and Engineering, Tokyo Institute of Technology. Research Center

for advanced Science and Technology, University of Tokyo. Japan. 1997

[38] Nanya, T., Takamura, A., Kuwako, M., Imai, M., Ozawa, M., Ozcan, M., Morizawa, R., Nakamura, H. *"Scalable-Delay Insensitive Design: A high-Performance approach to dependable asynchronous Systems"*. Research Center for advanced Science and Technology, University of Tokyo. Japan. 1998

[39] Sutherland, I.E. *"Micropipelines"* Association for Computing Machinery, Inc. (ACM) Volume 32, number 6, june 1989.

[40] Molnar, C.E., Sproull, R.F., Sutherland, I.E. *"The counterfow pipeline processor architecture"*. IEEE Design, Test of Computers, 11(3): pp. 48-59. Sun Microsystems Laboratories, Inc. and Institute for Biomedical Computing, Washington University. SMLI TR-94-25. April, 1994

[41] Furber, S.B, Day, P., Garside, J.D., Paver, N.C., Wood, J.V. *"A micropipelined ARM"* Proceedings of the IFIP TC10/WG 10.5 International Conference on Very Large Scale Integration, p.p. 211-220, September 07-10, 1993.

[42] Furber, S.B, Day, P., Garside, J.D., Paver, N.C., Wood, J.V. *"The Design and Evaluation of an Asynchronous Microprocessor"* Proceedings of the1994 IEEE International Conference on Computer Design: VLSI in Computer & Processors, pp.217-220, October 10-12, 1994.

[43] Farnsworth, C., Eduards, D.A., Sikand, S.S. *"Utilizing dynamic logic for low power consumption in asynchronous circuits"*. Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems (Async 94), pages 186-194. IEEE Computer Society Press, November 1994.

[44] Amulet Group. "*Power Reduction for System Technology"*. PREST Project Deliverable D1.2. Report on Asynchronous Design Techniques/Arithmetic Styles Department of Computer Science, University of Manchester Oxford Rd. Manchester, M13 9PL.

[45] Furber, S.B., Liu, J. *"Dynamic logic in four_phase micropipelines"*. Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems (Async96), pages 11-16. IEEE Computer Society Press, November 1996.

[46] Day, P., Woods, J.V. *"Investigation into micropipeline latch design styles"*. IEEE Transactions on VLSI Systems, 3(2): 264-272. June 1995.

[47] Yun, K.Y., Beerel, P.A., Arceo, J. *"High_performance asynchronous pipeline circuits"*. Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems (Async96), pages 17-28. IEEE Computer Society Press, November 1996.

[48] Molnar, Ch., Jones, I.W., Coates, B., Lexau, J. *"A fifo ring oscillator performance experiment"*. Proceedings of the International Symposium on Advanced Research

in Asynchronous Circuits and Systems (Async97), IEEE Computer Society Press, April 1997.

[49] Williams, T.E. Ph.D Thesis *"Self-timed rings and their application to division"*. Technical Report CSL-TR-91-482, Computer Systems Laboratory, Stanford University, 1991.

[50] Sparso, J., Staunstrup, J. *"Design and performance analysis of delay insensitive multi-ring structures"*. Proceedings of the Twenty-Sixth Annual Hawaii International Conference on System Sciences, volume I, pages 349-358. IEEE Computer Society Press, January 1993.

[51] Gopalakrishnan, G. "*Micropipeline wavefront arbiters using lockable C-elements"*. IEEE Design and Test, 1(4):55-64, Winter 1994.

[52] Nielsen, L.S., Niessen, C., Sparso, J., Van Berkel, K. *"Low-Power Operation Using Self-Timed Circuits and Adaptive Scaling of the Supply Voltage"*. IEEE Transactions on VLSI, 2(4):7, 1994.

[53] Cortadella, J., Kondratyev, A., Lavagno, L., Sotiriou, C. *"A concurrent model for de-synchronization"*. Univ. Politécnica de catalunya, Barcelona, Spain; Cadence Design Systems, San Jose; USA; Politenico di Torino, Italy; FORTH, Crete, Greece. 2003

[54] ASPIDA, IST-2002-37796. *"Asynchronous open-Source Processor Ip of the DLX Architecture"*. August 6, 2003.

[55] Sotiriou, Ch.P., Lavagno, L. *"De-Synchronization: Asynchronous Circuits from Synchronous Specifications"* Foundation for Research and Technology – Ellas (FORTH), Heraklion, Crete, Greece; Politeccnico di Torino, Torino, Italy. 2003

[56] Martin, A.J., Nystrom, M., Papadontonakis, K., Penzes, P.I., Prakash P., Wong, C.G., Chang, J., Ko, K.S., Lee, B., Ou, E., Pugh, J., Talvala, E.V., Tong, J.T., Tura, A. *"The Lutonium: A Sub-Nanojoule Asynchronous 8051 Microcontroller"*. *9th IEEE International Symposium on Asynchronous Systems & Circuits*, 2003.

[57] Talvala, E.V. Thesis: "*Designing the Port Interface Unit for the Lutonium Asynchronous Microcontroller"* California Institute of Technology, Pasadena, California, June 2003.

[58] Siemers, C. *"Configurable Computing"*. V1.01, SS 2005. Institut für Informatik der Technischen Universität Clausthal.

[59] Martin, A. J, Nystrom, M., Papadantonakis, K., Penzes, P. I., Prakash, P., Wong, C. G., Chang, J., Ko, K. S., Lee, B., Ou, E., Pugh, J., Talvala, E-V., Tong, J. T., Tura, A. *"The Lutonium: A Sub-Nanojoule Asynchronous 8051 Microcontroller"*.

Ninth IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC'03) . May 2003.

[60] **Alpha Processors.com**: http://www.alphaprocessors.com/21064.html http://www.alphaprocessors.com/21164.html

[61] **Intel**. *"SA-110 Microprocessor Technical Reference Manual"*. December 2000**.** Disponible en**:** http://www.cpu-world.com/CPUs/SA-110/Intel-StrongARM SA-110 2021281DB.html

[62] **Edwards, D.A., Toms, W.B.** *"Design, Automation and Test for Asynchronous Circuits and Systems"* Information Society Technologies (IST). Programme Concerted Action Thematic Network Contract IST-1999-29119, 3rd Edition June 2004.

[63] ***The Asynchronous Logic Home Page*****.** Actualizada en noviembre 29 de 2007. (http://intranet.cs.man.ac.uk/apt/async/tools/).

[64] ***ACiD-WG (Working Group on Asynchronous Circuit Design). "****IST (Information Society Technologies Programme)"*. Contract no.: IST-1999-29119. Period: 1 September 2000 - 31 January 2005. Disponible en: http://www.bcim.lsbu.ac.uk/ccsv/ACiD-WG/FinalReportFP5.pdf

[65] **Edwards, D., Bardsley, L., Janin, L., Toms, W.** *"Balsa: a Tutorial Guide"*. Version 3.5. 2006. Disponible en: http://intranet.cs.man.ac.uk/apt/projects/tools/balsa/

[66] **Carretero, J., Anasagasti, P., García, F., Pérez., F.** "Sistemas Operativos, una visión aplicada". Ed. McGraw Hill. P-p 59-60. Madrid, España. ISBN: 84-481-3001-4. 2001

[67] Cooperative Educational Service Agency (CESA 8). Digital Dictionary. 2008. **http://www.cesa8.k12.wi.us/media/digital_dictionary.htm**

Anexo 2.A: Componentes *breeze* [65]

Los componentes de *handshake* listados a continuación, en orden alfabético, se describen de acuerdo con una notación inventada por Bardsley en [34].

De acuerdo con su uso, los componentes *breeze* se clasifican en:

Componentes de control de activación: Loop, SequenceOptimised, Concur, Fork, WireFork

Componentes de terminación de canal: Continue, ContinuePush, Halt, HaltPush

Control para Componentes de interfaz de 'datapath' : While, Bar, Fetch, FalseVariable, Case, NullAdapth, Encode.

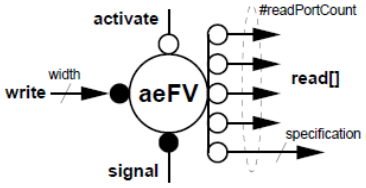
Componentes 'Pull datapath' : Adapt, slice, Constant, Combine, CombineEqual, CaseFetch, UnaryFunc, BinaryFunc, BinaryFuncConstR.

Componentes de conexión: ForkPush, Call, CallMux, CallDemux, passivator, passivatorPush, Synch, SynchPull, DecisionWait, Split, Arbiter, Variable.

Componentes 'non-delay-insensitive`: **CallActive, CallDemuxPull.**

Componentes de sólo simulación: ***BuiltinVariable.***

Anexo 2.A: Componentes *brezze*.



ActiveEagerFalseVariable

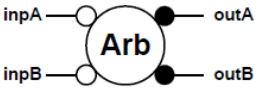
(**parameter** width : cardinal;
parameter readPortCount : cardinal;
parameter specification : string;
passive sync activate;
active input write : width **bits**;
active sync signal;
array readPortCount **of passive output** read)

#[activate: write ?° [signal ; v := *write*]] || #[$read_0$!° v] || ... || #[$read_{readPortCount\text{-}1}$!° v]

Adapt

(**parameter** outputWidth : cardinal;
parameter inputWidth : cardinal;
parameter outputIsSigned, inputIsSigned : boolean;
passive output out : outputWidth **bits**;
active input inp : inputWidth **bits**)

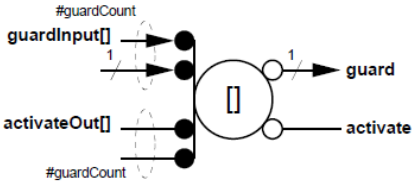
#[out !° inp ?• *adapt*(*outputWidth*, *inputWidth*, *outputIsSigned*, *inputIsSigned*, *inp*)]



Arbiter

(**sync** inpA, inpB, outA, outB)

#[[inpA : inpB | inpB : outB]]

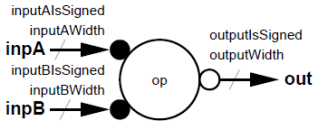


Bar

(**parameter** guardCount : cardinal;
passive output guard : 1 **bits**;
passive sync activate;
array guardCount **of active input** guardInput : 1 **bits**;
array guardCount **of active sync** activateOut)

#[guard !° (c := [$guardInput_0$?• ... ?• $guardInput_{guardCount\text{-}1}$?•
choose($guardInput_0$, ..., $guardInput_{guardCount\text{-}1}$)]) != -1)] ||
#[activate : $activateOut_c$]

Anexo 2.A (Continuación)

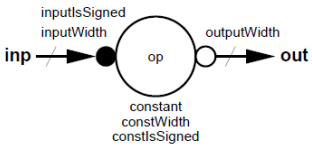


BinaryFunc

(**parameter** outputWidth : cardinal;
parameter inputAWidth : cardinal;
parameter inputBWidth : cardinal;
parameter op : BinaryOperator;
parameter outputIsSigned : boolean;
parameter inputAIsSigned : boolean;
parameter inputBIsSigned : boolean;
passive output out : outputWidth **bits**;
active input inpA : inputAWidth **bits**;
active input inpB : inputBWidth **bits**)

type BinaryOperator is enumeration (op symbol between brackets)
Add (+), Subtract (-), ReverseSubtract (\\-), Equals (==), NotEquals (!=), LessThan (<), GreaterThan (>), LessOrEquals (<=), GreaterOrEquals (>=), And (&), Or (|)
end

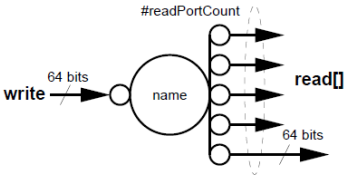
#[out !° inpA ?• inpB ?• *op*(*outputWidth*, *outputIsSigned*, *inputAIsSigned*, *inputBIsSigned*, *op*, *inpA*, *inpB*)]



BinaryFuncConstR

(**parameter** outputWidth : cardinal;
parameter inputWidth : cardinal;
parameter constWidth : cardinal;
parameter op : BinaryOperator;
parameter outputIsSigned : boolean;
parameter inputIsSigned : boolean;
parameter constIsSigned : boolean;
parameter constant : constWidth **bits**;
passive output out : outputWidth **bits**;
active input inp : inputWidth **bits**)

#[out !° inp ?• *op*(*outputWidth*, *outputIsSigned*, *inputIsSigned*, *constIsSigned*, *op*, *constant, inp*)]

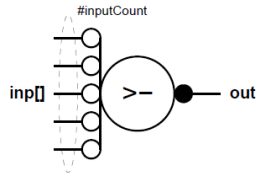


BuiltinVariable

(**parameter** readPortCount : cardinal;
parameter name : string;
passive input write : 64 **bits**;
array readPortCount **of output read** : 64 **bits**)

#[write ?° *v* := *write*] ||
#[$read_0$! *v*] || ... || #[$read_{readPortCount-1}$!° *v*]

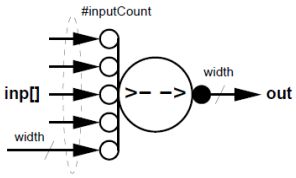
Anexo 2.A (Continuación)



Call

(**parameter** inputCount : cardinal;
array inputCount **of passive sync** inp;
active sync out)

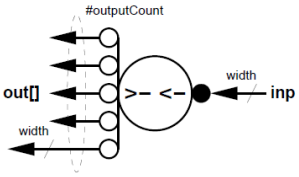
#[[inp_0 : out | ... | $inp_{inputCount-1}$: out]]



CallMux

(**parameter** width : cardinal;
parameter inputCount : cardinal;
array inputCount **of passive input** inp : width **bits**;
active output out : width **bits**)

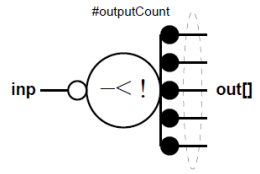
#[[out $!^{\bullet}$ inp_0 | ... | out $!^{\bullet}$ $inp_{inputCount-1}$]]



CallDemux

(**parameter** width : cardinal;
parameter outputCount : cardinal;
array outputCount **of passive output** out : width **bits**;
active input inp : width **bits**)

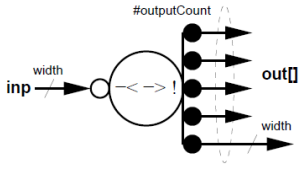
#[[out_0 $!^{\circ}$ inp $?^{\bullet}$ *inp* | ... | $out_{outputCount-1}$ $!^{\circ}$ inp $?^{\bullet}$ *inp*]]



CallActive

(**parameter** outputCount : cardinal;
passive sync inp;
array outputCount **of active sync** out)

#[inp: [out_0 , ... , $out_{outputCount-1}$]] but non-DI: RTZ to all outputs when first ack received.

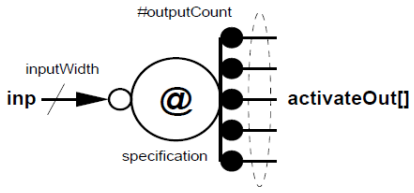


CallDemuxPush

(**parameter** width, outputCount : cardinal;
passive input inp : width **bits**;
array outputCount **of active output** out : width **bits**)

#[inp: [out_0 , ... , $out_{outputCount-1}$]] but non-DI: RTZ to all outputs when first ack received.

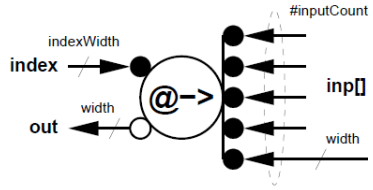
Anexo 2.A (Continuación)



Case

(**parameter** inputWidth : cardinal;
parameter outputCount : cardinal;
parameter specification : string;
passive input inp : inputWidth **bits**;
array outputCount **of active sync** activateOut
)

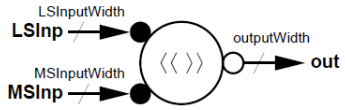
#[inp ?° [*decode*(*outputCount*, *specification*, *inp*) != - 1 ->
activateOut$_{decode(outputCount,\ specification,\ inp)\ !=\ -\ 1}$]]



CaseFetch

(**parameter** width, indexWidth : cardinal;
parameter inputCount : cardinal;
parameter specification : string;
passive output out : width **bits**;
active input index : indexWidth **bits**;
array inputCount **of active input** inp : width **bits**)

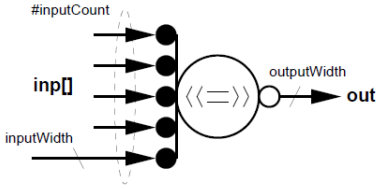
#[out !° index ?• inp_{index} ?• inp_{index}]



Combine

(**parameter** outputWidth : cardinal;
parameter LSInputWidth : cardinal;
parameter MSInputWidth : cardinal;
passive output out : outputWidth **bits**;
active input LSInp : LSInputWidth **bits**;
active input MSInp : MSInputWidth **bits**)

#[out !° LSInp ?• MSInp ?• *combine*(*LSInp*, *MSInp*)]

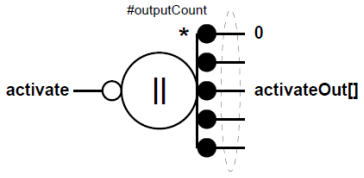


CombineEqual

(**parameter** outputWidth : cardinal;
parameter inputWidth : cardinal;
parameter inputCount : cardinal;
passive output out : outputWidth **bits**;
array inputCount **of active input** inp : inputWidth
bits)

#[out !° inp_0 ?• ... ?• $inp_{inputCount-1}$?• *combineEqual*(inp_0, ..., $inp_{inputCount-1}$)]

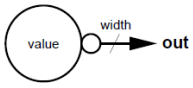
Anexo 2.A (Continuación)



Concur

(**parameter** outputCount : cardinal;
passive sync activate;
array outputCount **of active sync** activateOut
)

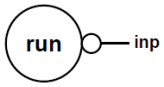
[activate : [$activateOut_0$ || ... || $activateOut_{outputCount\text{-}1}$]]



Constant

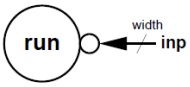
(**parameter** width : cardinal;
parameter value : width **bits**;
passive output out : width **bits**)

#[out !° *value*]



Continue

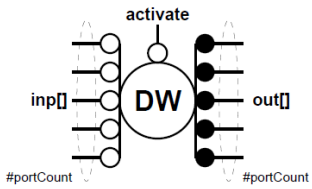
(**passive sync** inp)



ContinuePush

(**parameter** width cardinal;
passive input inp : width **bits**)

#[inp]

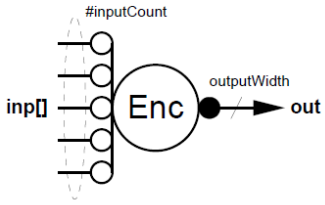


DecisionWait

(**parameter** portCount : cardinal;
passive sync activate;
array portCount **of passive sync** inp;
array portCount **of active sync** out)

#[activate : [inp_0 : out_0 | ... | $inp_{portCount\text{-}1}$: $out_{portCount\text{-}1}$]]

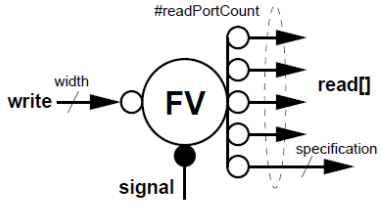
Anexo 2.A (Continuación)



Encode

(**parameter** outputWidth : cardinal;
parameter inputCount : cardinal;
parameter specification : string;
array inputCount **of passive sync** inp;
active output out : outputWidth **bits**)

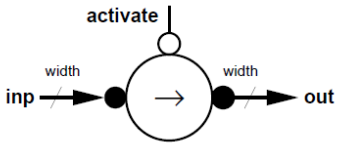
#[[inp_0 : out !• *encode*(*outputWidth*, *inputCount*, *specification*, *0*) | ... | $inp_{inputCount-1}$: out !• *encode*(*outputWidth*, *inputCount*, *specification*, *inputCount*)]]



FalseVariable

(**parameter** width : cardinal;
parameter readPortCount : cardinal;
parameter specification : string;
passive input write : width **bits**;
active sync signal;
array readPortCount **of passive output** read)

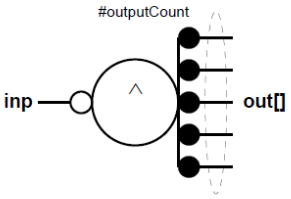
#[write ?° [*v* := *write* ; signal]] || #[$read_0$!° *v*] || ... || #[$read_{readPortCount-1}$!° *v*]



Fetch

(**parameter** width : cardinal;
parameter outBroad: boolean;
passive sync activate;
active input inp : width **bits**;
active output out : width **bits**)

#[activate : out !• inp ?• *inp*]

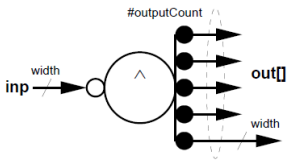


Fork

(**parameter** outputCount : cardinal;
passive sync inp;
array outputCount **of active sync** out)

#[activate : [out_0 , ... , $out_{outputCount-1}$]]

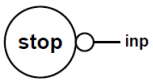
Anexo 2.A (Continuación)



ForkPush

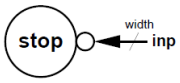
(**parameter** width, outputCount : cardinal;
passive input inp : width **bits**;
array outputCount **of active output** out : width **bits**)

#[inp ?° [out0 !• *inp* , ... , $out_{outputCount-1}$!• *inp*]]



Halt

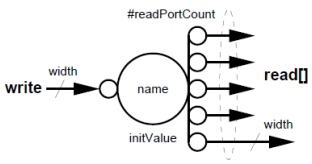
(**passive sync** inp)



HaltPush

(**parameter** width cardinal;
passive input inp : width **bits**)

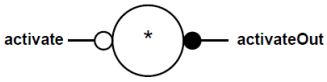
stop



InitVariable

(**parameter** width, readPortCount : cardinal;
parameter name : string;
parameter initValue: width **bits**;
passive input write : width **bits**;
array readPortCount **of output read** : width **bits**)

v := *initValue* ;
#[write ?° *v* := *write*] ||
#[$read_0$! *v*] || ... || #[$read_{readPortCount-1}$!° *v*]

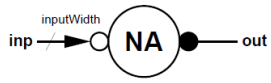


Loop

(**passive sync** activate;
active sync activateOut)

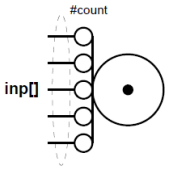
activate : #[activateOut]

Anexo 2.A (Continuación)



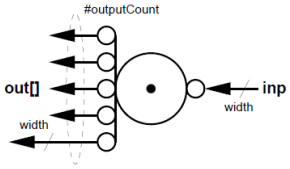
NullAdapt
(**parameter** inputWidth : cardinal;
passive sync out;
active input inp : inputWidth **bits**)

#[inp : out]



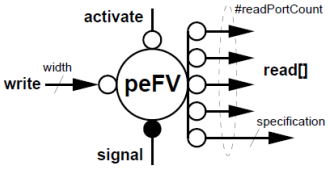
Passivator
(**parameter** count : cardinal;
array count **of passive sync** inp)

#[inp0 : ... : $inp_{count\text{-}1}$]



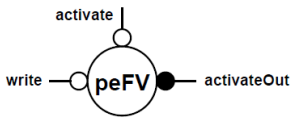
PassivatorPush
(**parameter** width, outputCount : cardinal;
array outputCount **of passive output** out : width **bits**;
passive input inp : width **bits**)

#[out0 !° ... !° $out_{outputCount\text{-}1}$!° inp ?° *inp*]



PassiveEagerFalseVariable
(**parameter** width : cardinal;
parameter readPortCount : cardinal;
parameter specification : string;
passive sync activate;
passive input write : width **bits**;
active sync signal;
array readPortCount **of passive output** read)

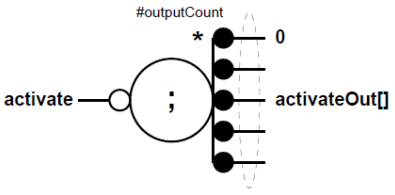
#[activate: write ?° [signal ; *v* := *write*]] || #[$read_0$!° *v*] || ... || #[$read_{readPortCount\text{-}1}$!° *v*]



PassiveSyncEagerFalseVariable
passive sync activate;
passive sync write;
active sync activateOut;

#[activate: (activateOut ; write)]

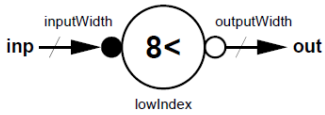
Anexo 2.A (Continuación)



SequenceOptimised

(**parameter** outputCount : cardinal;
parameter specification : string;
passive sync activate;
array outputCount **of active sync** activateOut
)

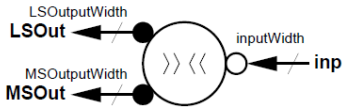
[activate : [$activateOut_0$; ... ; $activateOut_{outputCount\text{-}1}$]]



Slice

(**parameter** outputWidth : cardinal;
parameter inputWidth : cardinal;
parameter lowIndex : cardinal;
passive output out : outputWidth **bits**;
active input inp : inputWidth **bits**)

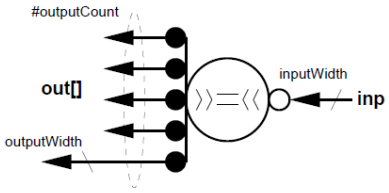
#[out !° inp ?• *slice*(*outputWidth*, *lowIndex*, *inp*)]



Split

(**parameter** inputWidth : cardinal;
parameter LSOutputWidth : cardinal;
parameter MSOutputWidth : cardinal;
passive input inp : inputWidth **bits**;
active output LSOut : LSOutputWidth **bits**;
active output MSOut : MSOutputWidth **bits**)

#[inp ?° [LSOut !• *bitfield*(0, *LSOutputWidth*-1, *inp*) ||
MSOut !• *bitfield*(*LSOutputWidth*, *inputWidth*-1, *inp*)]]

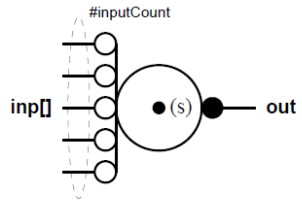


SplitEqual

(**parameter** inputWidth : cardinal;
parameter outputWidth : cardinal;
parameter outputCount : cardinal;
passive input inp : inputWidth **bits**;
array outputCount **of active output** out :
outputWidth **bits**)

#[inp ?° [out_0 !• *bitfield*(0, *outputWidth-1*, *inp*) || ... ||
$out_{outputCount\text{-}1}$!• *bitfield*(*inputWidth-outputWidth*, *inputWidth*-1, *inp*)]]

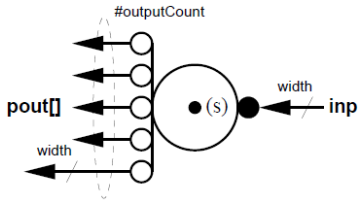
Anexo 2.A (Continuación)



Synch

(**parameter** inputCount : cardinal;
array inputCount **of passive sync** inp;
active sync out)

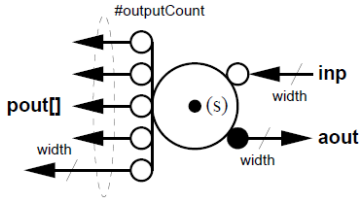
#[inp_0 : ... : $inp_{inputCount-1}$: out]



SynchPull

(**parameter** width, outputCount : cardinal;
array outputCount **of passive output** pout : width **bits**;
active input inp : width **bits**)

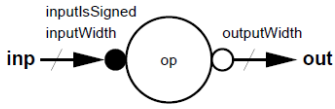
#[$pout_0$!° ... !° $pout_{outputCount-1}$!° inp ?• *inp*]



SynchPush

(**parameter** width, outputCount : cardinal;
passive input inp : width **bits**;
array outputCount **of passive output** pout : width **bits**;
active output aout : width **bits**)

#[$pout_0$!° ... !° $pout_{outputCount-1}$!° inp ?° aout !• *inp*]



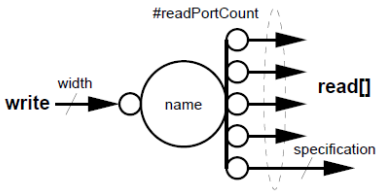
UnaryFunc

(**parameter** outputWidth : cardinal;
parameter inputWidth : cardinal;
parameter op : UnaryOperator;
parameter inputIsSigned : boolean;
passive output out : outputWidth **bits**;
active input inp : inputWidth **bits**)

type UnaryOperator is enumeration (op symbol between brackets)
Negate (~), Invert (-)
end

#[out !° inp ?• *op*(*outputWidth*, *inputIsSigned*, *op*, *inp*)]

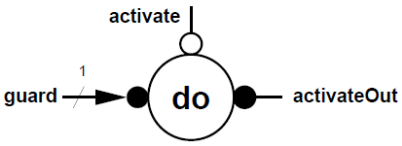
Anexo 2.A (Continuación)



Variable

(**parameter** width, readPortCount : cardinal;
parameter name : string;
parameter specification : string;
passive input write : width **bits**;
array readPortCount **of output read** : width **bits**)

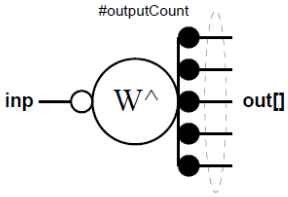
#[write ?° *v* := *write*] ||
#[$read_0$! *v*] || ... || #[$read_{readPortCount-1}$!° *v*]



While

(**passive sync** activate;
active input guard : 1 **bits**;
active sync activateOut)

#[activate : [guard ?• *g* ; [*g* -> activateOut]]]



WireFork

(**parameter** outputCount : cardinal;
passive sync activate;
array outputCount **of active sync** out)

#[activate : [out_0 , ... , $out_{outputCount-1}$]]

Anexo 2.B: Opciones de estilo de Balsa [62]

Las opciones de estilo listadas en este anexo se acivan/desactivan por medio de la interfaz de usuario *balsa-mgr* del sistema Balsa. Las opciones están acompañadas del tipo de tecnología y permanecen en el archivo '*netlist'* generado, cuando este es importado por la herramienta de diseño CAD para la síntesis final y su implementación en hardware.

Option	Values	Notes
suggest-buffer	set/unset	Handshake circuit descriptions allow for nodes in the circuit to be identified as being points at which buffers may be inserted because the node may be heavily loaded. Setting this option will cause the buffers to be instantiated.
cad	cadence	This option only makes sense for Xilinx technology. Makes balsa-netlist produce verilog netlists compatible with Cadence implementations of Xilinx libraries.
	ise	This option only makes sense for Xilinx technology. Makes balsa-netlist produce verilog netlists compatible with Xilinx ISE implementations of Xilinx libraries.
logic	DIMS	Implements "helper" cells – those cells composed from the basic cell library – in a DIMS style
	NCL	Implements "helper" cells – those cells composed from the basic cell library – using NCL style gates. In many circumstances smaller implementations result.
	Balanced	Creates balanced circuits where the notional path delays through the DIMS circuts are matched in an attenpt to defeat Differential Power Analysis attacks in security applications such as smartcards
variable	SR	Variables stored in "standard" SR latches
	Spacer	Each variable latch is reset to a NULL state before a writie operation in an attenpt to defeat Differential Power Analysis attacks in security applications such as smartcards
	NCL	Variables are stored in pipleline style latches. More efficient for 1-of-4 codes.
n-of-m mapping	set/unset	Enables general n-of-m mapping strategy for dual rail (**dual_b)** styles. General users should accept the default option – the option is included for historical reasons.
sim	icarus vxl ncv vcs modelsim cver	These option are only available for the *example* technology. They are various Verilog simulators known to work with the Balsa system. Note that `balsa-sim-verilog` must be configured to locate a particular simulator (see the installation notes). `icarus` and `cver` are publicly available simulators.

Anexo 2.B: Opciones de estilo de Balsa [62] (Continuación)

FV	conv	Use conventional implementation of the FalseVariable component with full enclosure of activity on the read channels within the handshake upon the write channel.
	ovlp	Introduce concurrency within the passive enclosure of FalseVariable components by overlapping the return-to-zero phases of the "write" port and the "signal" port which triggers activity on the readports. This is the default option.
PAR	conv	Use conventional implementation of the Concur component with full enclosure of activity on the output channels within the handshake of the activate channels.
	ovlp	Introduce concurrency between the return-to-zero phases of the activation and output channels of Concur components. This is the default option.
SEQ	conv	Use conventional implementation of the Sequence component, a full handshake is completed on each output channel before initiating activity upon the next channel in the sequence.
	ovlp	Introduce concurrency, where safe, between the return-to-zero phase of the an output channel and the processing phase of the next channel in the sequence. The balsa-c compiler determines within which components this optimisation can be safely implemented. This is the default option.
PP	conv	Use the conventional implementation of PassivatorPush Components. To maintain the data-validity protocols across the broad/reduced-broad interface in single-rail implementations, this option implements enclosure of output channel activity within input channel activity in Fetch components.
	broad	Use a "broad" implementation of single-rail PassivatorPush component. The broad implementation latches the data within PassivatorPush components, solving the data-validity problems caused by broad/reduced-broad protocol interfaces in Fetch components. This is the default option.

Capítulo 3: Algoritmo de Rijndael

Introducción

El número de usuarios de Internet y de comunicaciones inalámbricas aumenta en forma constante y acelerada, al mismo tiempo aumenta la demanda por fortalecer las medidas de seguridad y adquirir dispositivos para proteger los datos de usuario transmitidos por canales abiertos. Se pueden utilizar dos clases de sistemas criptográficos para este propósito: los cripto-sistemas de clave asimétrica y los de clave simétrica. La criptografía de clave asimétrica (tal como RSA y los Algoritmos de Curvas Elípticas) utilizan claves distintas para encriptación y desencriptación. La criptografía de clave simétrica (tal como DES, TDES y AES) utiliza la misma clave para encriptar y desencriptar el mensaje. En octubre del 2000, el Instituto Nacional de Estándares y Tecnología de Estados Unidos, NIST (*National Institute of Standards and Technology*) [1], seleccionó el algoritmo de Rijndael como el nuevo estándar AES (*Advanced Encryption Standard*) del *FIPS* (*Federal Information Processing Standard*) [2], para reemplazar el antiguo estándar de encriptación *DES* (*Data Encryption Standard*), pero fue solo hasta el mes de mayo de 2002 que DES fue finalmente reemplazado por AES.

Se han reportado numerosas implementaciones del algoritmo de Rijndael, la mayoría de esas implementaciones han sido hechas en software o en *FPGAs (Field Programmable Gate Arrays*). Las ventajas de la implementación en software incluye flexibilidad y bajo costo, mientras que la implementación en FPGA proporciona mayor velocidad y mejor desempeño y generalmente el diseño se realiza utilizando plataformas de Xilinx como en [3]-[5] o Altera en [6],[7]. Adicionalmente, se han desarrollado trabajos de implementaciones de Rijndael mediante Circuitos Integrados de Aplicación Específica (*ASICs: Application Specific Integrated Circuits*). En la actualidad los *ASICs* soportan velocidades de encriptación en el orden de decenas de Gbits/s, lo cual los hace muy útiles para encriptación de bloques de datos en redes ópticas donde los usuarios están conectados a enlaces con velocidades de transferencia de datos también del orden de decenas de Gbits/s [8]-[12].

Este capítulo presenta una introducción al algoritmo y las tendencias arquitecturales para la implementación del estándar AES. Se hace un resumen de la revisión de la literatura relacionada con diferentes propuestas de implementacion en hardware del algoritmo de Rijndael. Inicialmente se recopilan los resultados de las arquitecturas síncronas segmentadas y no segmentadas, implementadas tanto en hardware reconfigurable (*FPGAs*) como en *ASICs* con *Standard-Cell* CMOS. Finalmente se muestran los resultados de implementaciones asíncronas desarrollados hasta el momento. Los detalles referentes a los módulos que conforman el algoritmo, tanto para encriptación como para desencriptación, se desarrollarán en el capítulo 4.

3.1. Esquema general del algoritmo para encriptación y desencriptación

Para el algoritmo AES, la longitud de los bloques de entrada, de salida y el estado intermedio es de 128 bits. Estos se representan con *Nb*=4 (número de columnas en el

estado), equivalente al total de 4 palabras de 32 bits. La longitud de la clave de encriptación, *K*, es de 128, 192, o 256 bits, y está representada por *Nk*=4, 6 u 8, (número de columnas), que refleja el número de palabras de 32 bits en la clave de encriptación.

En el algoritmo AES, el número de rondas a realizarse durante la ejecución del algoritmo, representado por *Nr,* depende del tamaño de la clave. La siguiente tabla muestra la relación existente entre la longitud de la clave, el tamaño del bloque y el número de rondas.

Tabla 3.1.: Relación entre longitud de clave, tamaño de bloque y número de rondas en AES de acuerdo con el estándar FIPS-197 [2].

	Longitud de Clave (Nk palabras)	***Tamaño de bloque (Nb palabras)***	***Número de Rondas (Nr)***
AES-128	**4**	**4**	**10**
AES-192	**6**	**4**	**12**
AES-256	**8**	**4**	**14**

El algoritmo de encriptación AES incluye cuatro funciones de transformación llamadas: *ByteSub, ShiftRow, MixColumn y AddRoundKey*. El algoritmo opera sobre una matriz de bytes de datos de cuatro filas y cuatro columnas (128 bits), conocida como *matriz de estado,* los bytes representan elementos de un campo finito llamado *Campo de Galois* $GF(2^8)$, *(GF: Galois Field)***.** Los resultados intermedios de las transformaciones que realiza el algoritmo se almacenan en la matriz de estado.

El algoritmo, mostrado en la figura 3.1, consiste en una serie de *Nr* rondas. En cada ronda de encriptado se ejecutan cuatro transformaciones sobre la matriz de estado, además del proceso de extensión de la clave.

- La función *ByteSub* es una transformación no lineal que utiliza tablas de sustitución de 16 bytes (128 bits) llamadas Sbox.
- La función *ShiftRow* es una operación de desplazamiento cíclico en cada fila de 4 bytes del bloque de datos, a la primera fila no se le aplica desplazamiento, mientras que a las filas 2, 3 y 4 se les aplica un desplazamiento de 1, 2 y 3 bytes a la izquierda respectivamente.
- La función *MixColumn* trata los 4 bytes de cada columna del bloque de datos como coeficientes de un polinomio de 4 términos, cada dato se multiplica con un polinomio fijo y al resultado se le aplica el módulo x^4+1.
- La función *AddRoundKey* consiste en una operación XOR bit a bit entre la clave de ronda y el dato.

Una descripción detallada de estas transformaciones y del algoritmo en general se da en [2],[5],[10],[13],[14].

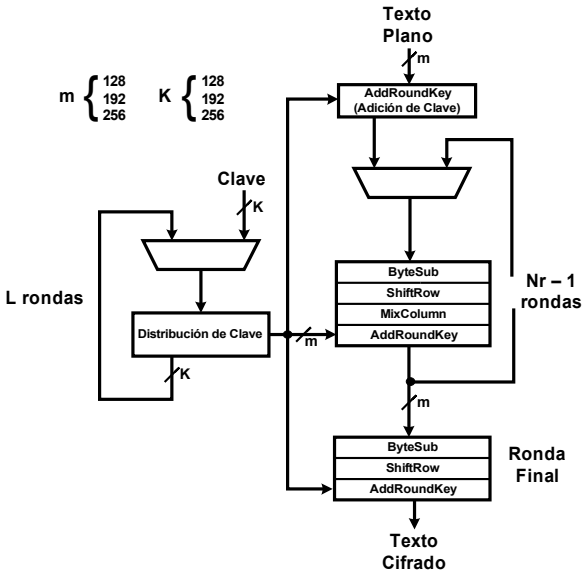


Figura 3.1: Diagrama de bloques del Algoritmo de Rijndael (AES) para encriptación [9]

El proceso de encriptación inversa (o desencriptación) consiste en sustituir las transformaciones utilizadas en la encriptación por sus inversas e invertir el orden de aplicación de dichas transformaciones o funciones matemáticas, en este caso las funciones inversas se denominan *InvByteSub, InvShiftRow, InvMixColumn*, la función *AddRoundkey* es su propia inversa. El proceso de encriptación y el de desencriptación para cada transformación se describe en detalle en el capítulo 4. La implementación de los circuitos de distribución de clave y el algoritmo se describen en el capítulo 5.

3.2. Modos de configuración

Para mejorar la seguridad o el desempeño del algoritmo de Rijndael, y en general de los cifradores de bloque simétrico, se emplean diversos modos de operación especificados en un conjunto de recomendaciones del NIST consignadas en tres documentos publicados en su página de Internet (http://csrc.nist.gov).

La primera recomendación [15], publicada en diciembre del 2001, define cinco modos de confidencialidad de operación para usar con algoritmos de cifrado de bloque de clave simétrica aprobados por el FIPS: *Electronic Code Book (ECB), Cipher Block Chaining (CBC), Cipher Feedback (CFB), Output Feedback (OFB),* y *Counter (CTR).*

La segunda recomendación [16], publicada en mayo del 2004, define un modo adicional de confidencialidad llamado *CCM* para ser utilizado con algoritmos de cifrado de clave simétrica. El modo CCM es una combinación de las técnicas usadas en el modo CTR y el algoritmo *CBC-MAC (Cipher Block Chaining-Message Authentication Code)* [17,18]. Un MAC provee una mejor seguridad de autenticidad que un código "*checksum*" o un código de detección de error, los cuales están diseñados solamente para detectar modificaciones accidentales de datos, mientras que la verificación de un MAC, como ocurre con CCM, está diseñada para detectar modificaciones intencionales no autorizadas del dato, así como también modificaciones accidentales del mismo [16]. El modo CCM

se diseñó para garantizar la confidencialidad y la autenticidad de los datos al mismo tiempo.

La tercera recomendación [19], publicada en mayo del 2005, especifica un algoritmo MAC basado en un cifrado de clave simétrica llamado *CMAC*. Este algoritmo se puede utilizar para brindar seguridad de autenticidad y, por lo tanto, la integridad del dato binario.

De manera general, algunos modos de configuración previenen pérdida de información asegurando que entradas idénticas creen salidas distintas; otros modos aseguran que cualquier cambio en el dato encriptado sea reconocible.

Los modos de operación de los cifradores de clave simétrica se pueden dividir en dos categorías [20]:

- ***Modos no realimentados***, como *ECB y CTR.*
- ***Modos realimentados***, como *CBC, CFB, OFB*.

La figura 3.2 ilustra el tipo de realimentación en los modos de operación mencionados. En los modos no realimentados, la encriptación de cada bloque de datos subsiguiente se puede realizar independientemente del procesamiento de los otros bloques. En particular, todos los bloques se pueden encriptar en paralelo. En los modos realimentados solo se puede encriptar el siguiente bloque de datos cuando se haya terminado de encriptar el bloque previo. De acuerdo con lo expresado anteriormente, es claro que los modos de operación sin realimentación pueden ser implementados mediante arquitecturas *pipeline***,** mientras que los modos con realimentación están soportados por arquitecturas *non-pipeline.*

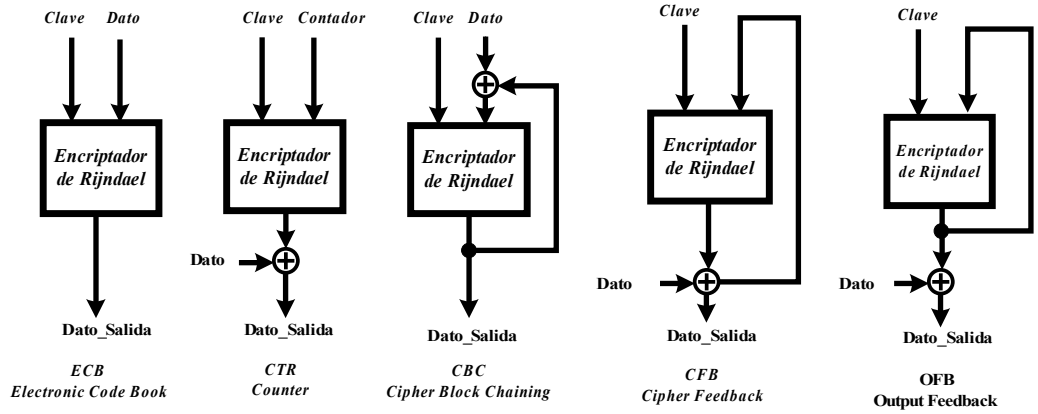


Figura 3.2: Modos de operación realimentados y no realimentados [20], [21]

En [21] se hace una descripción de hardware a alto nivel del algoritmo de Rijndael (figura 3.3) la cual indica la posición general definida para los modos de configuración. Esta consta de un camino de datos (*datapath)* de encriptación y un *datapath* para la distribución de la clave. Alrededor de ellos se encuentra el *datapath* utilizado para implementar los modos de operación. La interfaz de entrada y salida consiste en la

entrada de lectura de datos y la escritura de la salida encriptada, el cripto-controlador interpreta las instrucciones leídas y controla otras unidades.

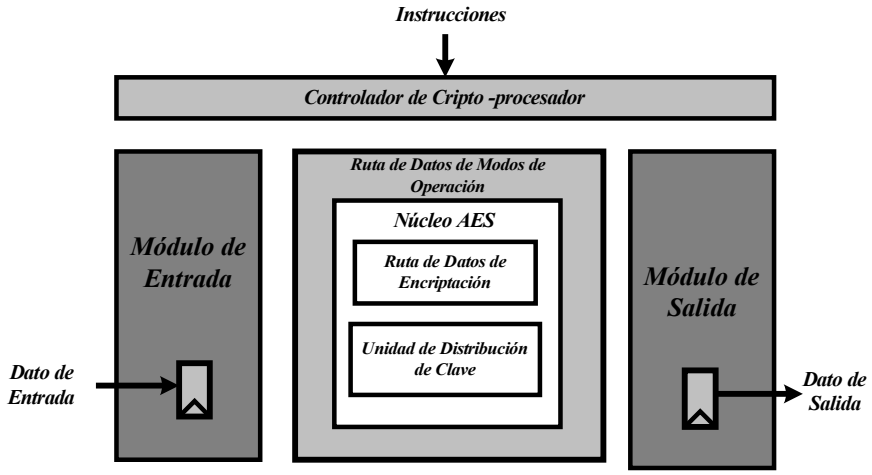


Figura 3.3: Diagrama de bloques del cripto-procesador de Rijndael [21]

3.3. Implementaciones síncronas del algoritmo en hardware

Existen diversas arquitecturas para realizar el algoritmo de Rijndael, sin embargo, la mayoría de ellas se derivan de dos arquitecturas básicas [22]:

- **Arquitectura iterativa** (o *"rolling architecture"*): utiliza una estructura de realimentación donde los datos son transformados de forma iterativa por las funciones de ronda. Se requiere poca área pero el rendimiento que se obtiene es bajo.
- **Arquitectura no iterativa** (o *"unrolling architecture"*): en esta arquitectura se insertan registros *pipeline* entre las rondas por lo que se obtiene un alto rendimiento, pero el área requerida para su implementación se incrementa significativamente.

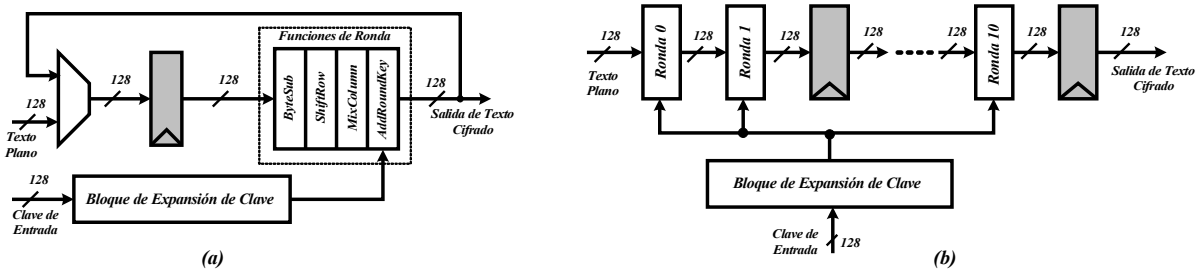


Figura 3.4: Arquitecturas de Rijndael de 128 bits: (a) *Rolling*; (b) *Unrolling*; [22]

La técnica *pipeline*, al insertar registros entre rondas de transformación (dibujados en color gris), acorta la ruta crítica, permite frecuencias de reloj más altas y reduce la actividad de señal; el sistema experimenta un incremento en su desempeño ya que es posible procesar múltiples bloques de datos en paralelo. Además, las arquitecturas con

pipeline permiten modos no realimentados como ECB o CTR. De acuerdo con [23], el rendimiento obtenido con arquitecturas *pipeline* usando tecnología CMOS para implementaciones de Rijndael va desde 150 Mbits/s para arquitecturas de 32 bits hasta decenas de Gbits/s para arquitecturas de 128 bits. Estas últimas se describen a continuación.

3.3.1. Implementaciones síncronas segmentadas (*pipeline*)

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

En esta arquitectura (conocida también como *sub-pipeline*), mostrada en la figura 3.5, se insertan etapas *pipeline* dentro de cada ronda de transformación y en las salidas de las funciones que conforman la unidad de encriptación y la unidad de claves. Esto decrementa los retardos entre etapas *pipeline* pero incrementa el número de ciclos de reloj que requiere la encriptación. Según [24], cada una de las 10 rondas se divide en cuatro etapas *pipeline*: *ByteSub, ShiftRow, MixColumn y AddRoundKey*, como se ilustra en la figura 3.5.

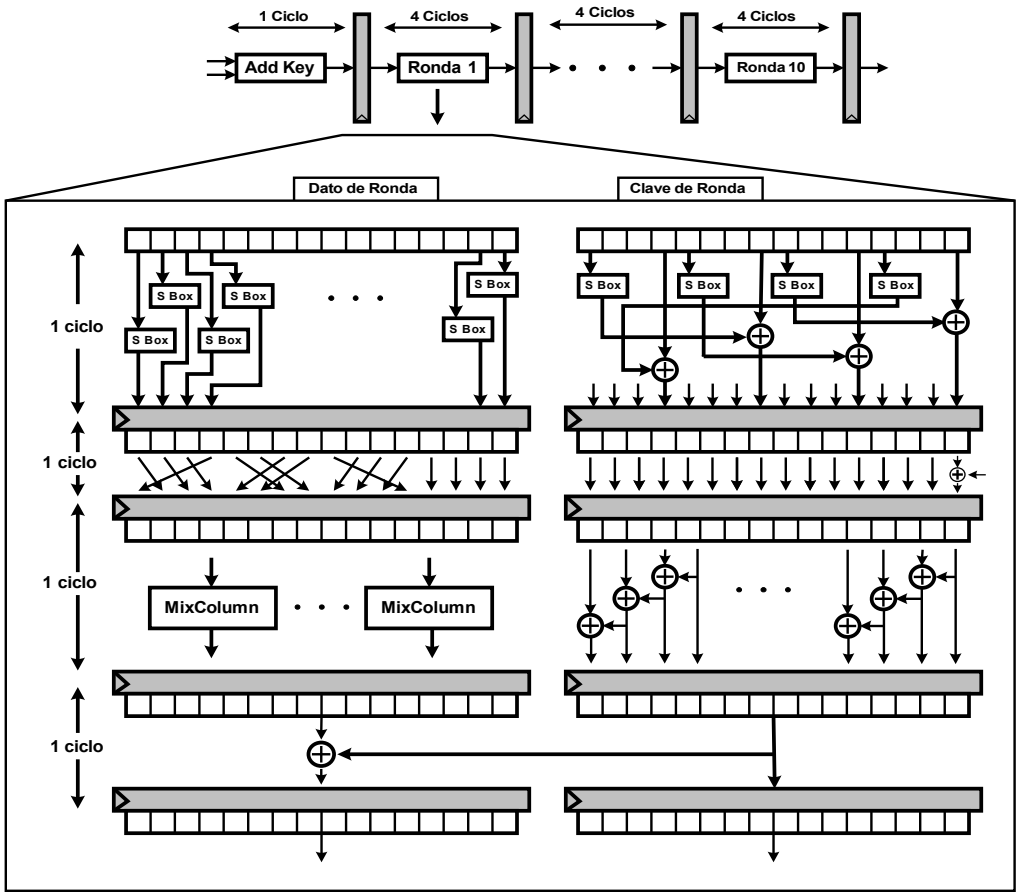


Figura 3.5: Arquitectura *pipeline* de ronda interna y ronda externa o *sub-pipeline* [9]

Hay un total de 41 etapas *pipeline*, incluyendo la primera etapa (*AddRoundKey*). En la implementación de cada ronda se utilizan 20 bloques Sbox (16 para la función de sustitución de byte y 4 en la unidad de claves). La arquitectura es completamente paralela en el procesamiento de los datos de cada ronda y la utilización de registros a la salida de cada función de transformación permite aumentar el desempeño.Según [9], la arquitectura ha alcanzado un máximo rendimiento de 77.6 Gbits/s utilizando tecnología CMOS de 0.18µm. Se presentaron reportes de la implementación en dispositivos FPGA Virtex de Xilinx en [24], obteniéndose un rendimiento de 12.16 Gbits/s y en [25] con un rendimiento de 17.8 Gbits/s. Tanto el tamaño de clave como el número de rondas se fijaron, en todos los casos, a 128 bits y 10 rondas respectivamente. El máximo rendimiento se obtuvo tomando una salida por ciclo de reloj, lo que solo es posible cuando se tiene la arquitectura *unrolled* y se ha aplicado *pipeline* entre etapas. La principal desventaja de esta arquitectura la constituye el alto costo en área.

3.3.1.2. Arquitectura *pipeline* de ronda externa

Debido al elevado costo en área de la arquitectura con *pipeline* interno y externo, se decidió optimizar el diseño mediante la eliminación de los registros *pipeline* internos. En esta arquitectura, hay un total de 11 etapas *pipeline* y cada ronda se realiza en un ciclo de reloj. También las claves de ronda se calculan en cada ciclo de reloj. La ruta crítica incluye todas las fases de cada ronda (*ByteSub, ShiftRow, MixColumn, AddRoundKey*), y la transformación *ByteSub* constituye, además, la fase crítica. El máximo rendimiento, reportado en [9], es de 48.2 Gbits/s y se logró utilizando tecnología CMOS de 0.18 µm. El ahorro de área respecto de la arquitectura *sub-pipeline* llega hasta un 32.5%. La figura 3.6 ilustra esta arquitectura.

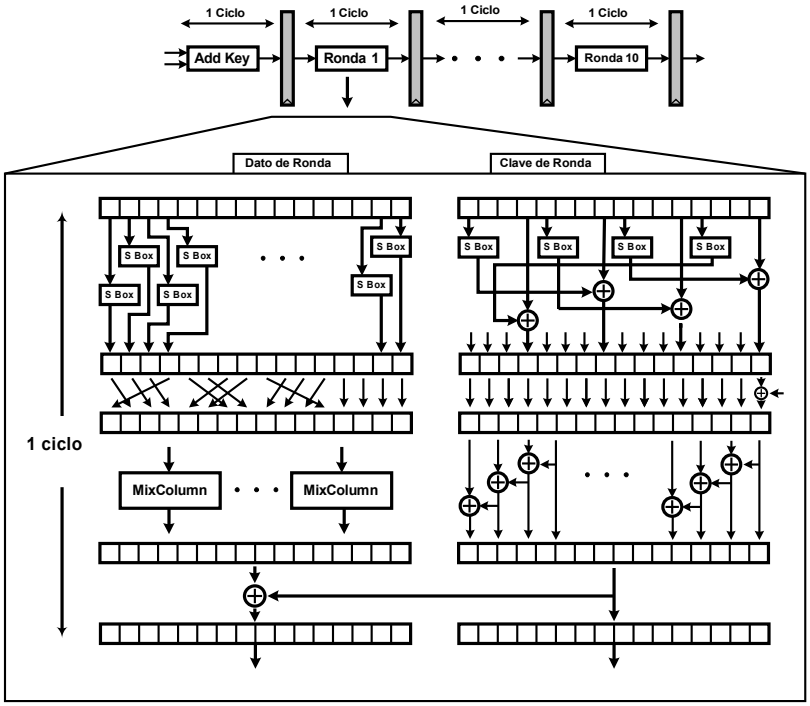


Figura 3.6: Arquitectura *pipeline* de ronda externa [9]

3.3.1.3. Arquitectura *pipeline* multi-ronda

Esta arquitectura se diseñó 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. De esta manera, se obtiene un rendimiento de 15.7 a 23.1 Gbits/s con un costo de área mucho menor que los dos casos anteriores; el ahorro de área respecto de la arquitectura de ronda externa alcanza el 60.4% [9]; sin embargo el rendimiento disminuye considerablemente.

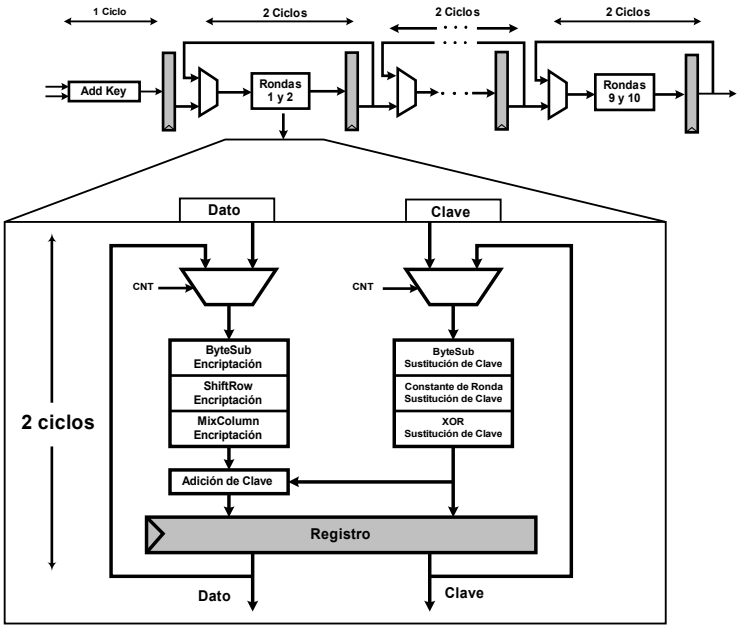


Figura 3.7: Arquitectura pipeline Multi-Ronda [9]

3.3.1.4. Arquitectura *pipeline* de campo compuesto

En [26],[27] 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 3.8, se implementa con base en el uso de un número óptimo de registros *pipeline* en la fase de Sustitución de Byte y explota, además, la implementación hardware de las Sbox propuesta por Rijmen [11].

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) [28]. 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 $GF(2^4)$.

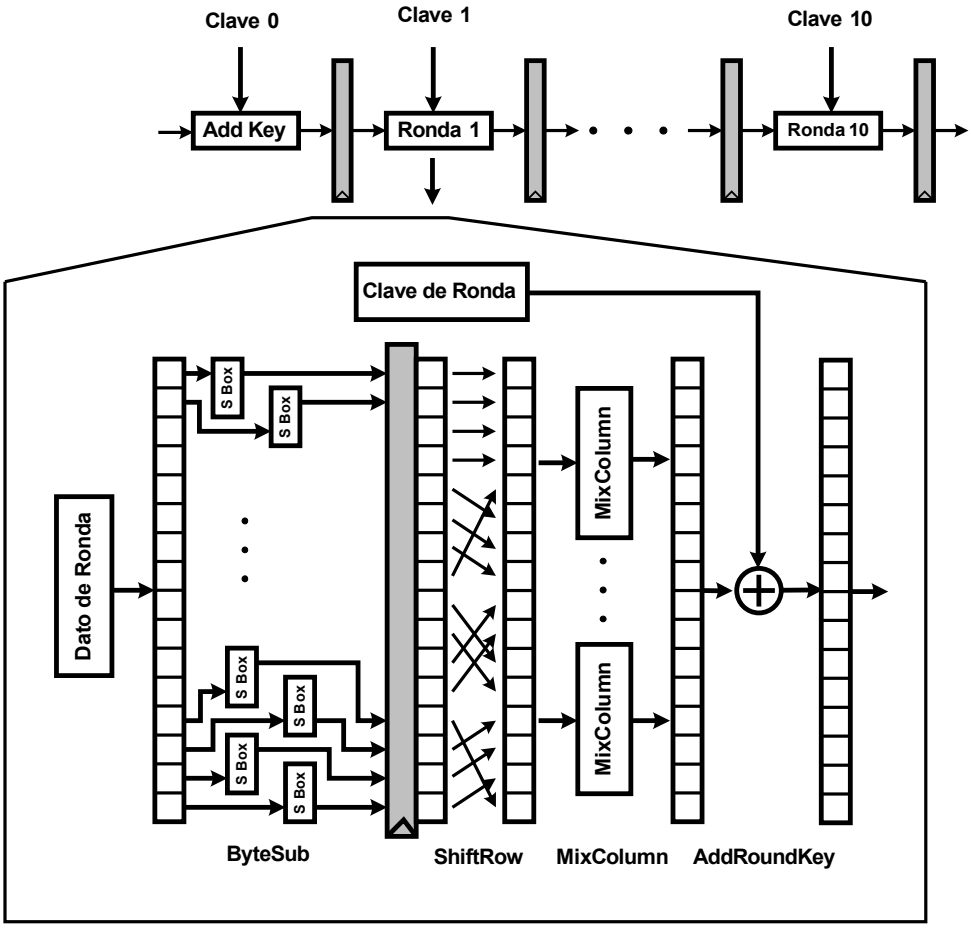


Figura 3.8: Arquitectura *pipeline* de campo compuesto [26], [27]

La figura 3.9 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.

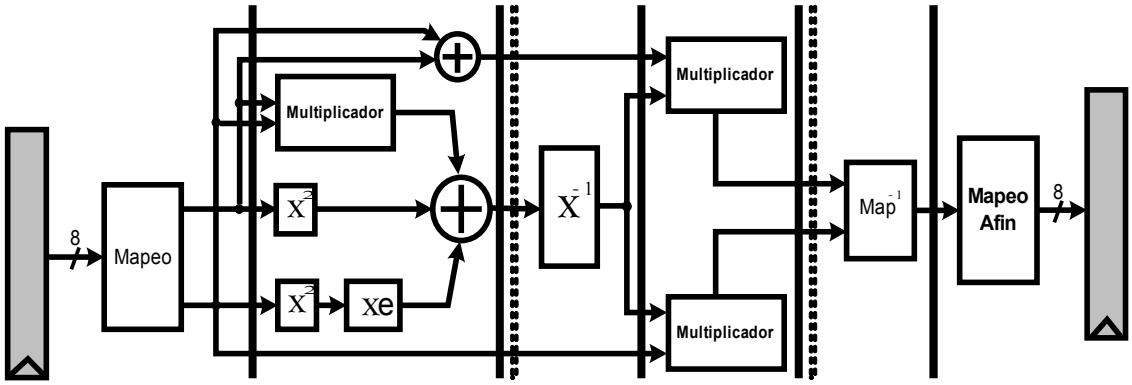


Figura 3.9: *Pipeline* de fase de sustitución de bytes de Campo Compuesto [27]

Esta arquitectura se implementó utilizando tecnología CMOS de 0.18μm, y operando en el modo CTR [26] lográndose un desempeño entre 30 y 70 Gbits/s en la fase de sustitución de byte para distintas etapas *pipeline*. Adicionalmente se implementó la unidad de distribución de clave "fuera de línea". Es decir, fuera del *datapath* de ronda AES, por lo que las subclaves se calculan por anticipado y se almacenan en los respectivos registros de ronda de clave. Luego, en la ronda de adición de clave, el *datapath* de encriptación utiliza las claves de ronda almacenadas en los registros. Lo anterior se traduce en un ahorro significativo de área ya que solo se necesita implementar una ronda de la unidad de distribución de clave. Con esta modificación, la arquitectura logra una reducción hasta del 28% respecto de la arquitectura que calcula cada clave de ronda por ciclo de reloj. En [27] se reporta un máximo desempeño de 21.54 Gbits/s al implementar la arquitectura de campo compuesto en una FPGA *VirtexII-Pro* de *Xilinx*.

3.3.1.5. Arquitectura *pipeline* con modo de operación CCM

Como se sabe, el modo de operación *CCM* combina las técnicas usadas en el modo *CTR* y el algoritmo *CBC-MAC (Cipher Block Chaining-Message Authentication Code*), por lo que el algoritmo AES no puede utilizar una arquitectura *pipeline* con este modo. Sin embargo, en [29] se ha presentado una arquitectura para alto desempeño que utiliza una técnica *pipeline* de bloques en la que se puede aplicar diseño de control jerárquico y una interfaz de *handshaking* entre todos los módulos.

La figura 3.10 muestra un diagrama de bloques de un cripto-procesador en el que las tramas de datos y control están separadas. El flujo de datos se hace por medio del módulo de entrada y luego pasa por el módulo de encriptación para luego dirigirse al módulo de salida, mientras tanto el controlador principal se encarga de la lectura de instrucciones. Los módulos de entrada y salida realizan su tarea sin que el controlador principal interfiera. Esta arquitectura, al presentar un diseño con múltiples controladores, facilita la comunicación mediante protocolos de *handshake.* También se facilita el diseño jerárquico del control lo cual redunda en una simplificación de las comunicaciones de los controladores y además permite combinar alto desempeño y programabilidad [29]. La figura también muestra cuatro bloques de control que son controlados a su vez por la

unidad de control principal, estos son: *FSM* de entrada, *CBC FSM*, *CNT FSM* y *FSM* de salida. Las FSMs de entrada y salida realizan las secuencias de *handshaking* para lectura y escritura de los bloques de datos. La *CBC FSM* controla la secuencia para generar el modo de operación *CBC-MAC*. La *CNT FSM* controla la secuencia de encriptación para el modo de operación *CTR*.

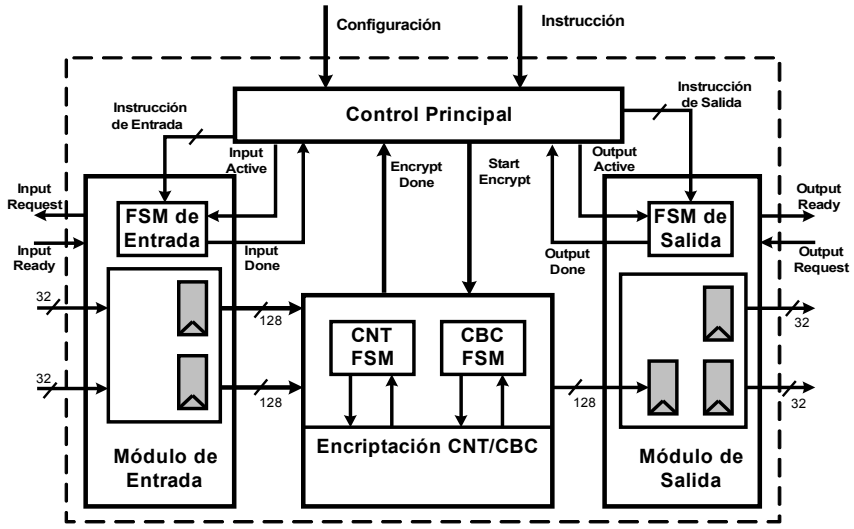


Figura 3.10: Diagrama de bloques del cripto-procesador con arquitectura *CCM* [29]

3.3.1.6. Reportes de Implementaciones *pipeline* de Rijndael

La tabla 3.2 muestra un listado de publicaciones sobre implementaciones *pipeline* en ASICs con tecnología de 0.18µm. Se resaltan características significativas de la implementación del algoritmo. La tabla 3.3 muestra un listado de publicaciones con reportes de implementaciones *pipeline* del algoritmo de Rijndael en FPGAs.

Tabla 3.2: Implementaciones síncronas *ASIC pipeline* del algoritmo de Rijndael

Referencia	***Núm. de Comp. (10^3)***	***FREC. (MHz)***	***Throughput (Gbits/s)***	***Comentarios especiales***
[9]	***473***	***606***	***77.6***	- *Pipeline* de ronda interna y ronda externa - Clave de ronda generada en paralelo
[9]	***372***	***377***	***48.2***	- *Pipeline* de ronda externa - Clave de ronda generada en paralelo
[9]	***225***	***362***	***23.1***	- *Pipeline* multi- ronda - Clave de ronda generada en paralelo
[26]	***250***	***362***	***70.0***	- *Pipeline* multi- ronda - Clave de ronda generada en paralelo
[29]		***295***	***3.43***	- Arquitectura CCM - Modelo de Pipeline de Bloques
[12]	***1600***	***250***	***8.0***	- *Pipeline* de ronda externa - Clave de ronda generada en paralelo - Se implementa encriptación y desencriptación
[8]		***200***	***25.6***	- Implementación con modo de operación CBC - Pipeline de ronda externa

Tabla 3.3: Implementaciones síncronas *pipeline* en *FPGA* del algoritmo de Rijndael

Referencia	*CLBs*	*BRAMs*	*LUTs*	*Frec. (MHz)*	*Rendimiento (Gbits/s)*	*Comentarios Especiales*
[3]		***10***	***780***	***115***	***1.3***	- S-box en LUT -Arquitectura Pipeline de ronda Interna - Subclaves generadas en Paralelo - Diseño con Spartan II-5
[30]		***10***	***770***	***155***	***1.75***	- S-box en LUT - Arquitectura Pipeline de ronda Interna - Subclaves generadas en paraleo - Diseño con Virtex E-8
[31]	***12600***				***12.2***	- Estudio comparativo con otras implementaciones AES - Arquitectura pipeline de ronda interna y ronda externa -. Diseño en Virtex XCV-1000
[32]	***2507***			***32***	***0.414***	-Arquitectura "*Full Unrolled*" - Diseño con Virtex
[32]	***2057***		***8***	***99***	***1.265***	- Utiliza una ronda con pipeline - Diseño con Virtex
[32]	***12600***	***80***		***95***	***12.16***	- Arquitectura "*unrolled*" con *Pipeline* - Diseño con Virtex
[25]	***10750***			***139.1***	***17.8***	- Sbox implementada en forma Combinatoria - Arquitectura "Full Pipeline": ronda interna y ronda externa - Diseño con Virtex-II, XC2V2000-5
[25]	***11719***			***129.2***	***16.54***	- Sbox implementada en forma Combinatoria - Arquitectura "Full Pipeline": - Diseño con Virtex-II, XCV1000-8
[33]	***2222***	***100***		***54.35***	***7***	- Una ronda por ciclo de reloj - Arq. Pipeline -Hardware para 10 rondas de encriptación -S-box en ROM (LUT)
[34]		***20***		***2.5***	***7.6***	- S.box en ROM -Arquitectura pipeline - Hardware para 10 rondas de encriptación - Diseño Virtex-II, XC2V1500
[35]	***4325***	***1***		***75***	***0.739***	- Arquitectura pipeline de ronda externa - Claves generadas y almacenadas en memoria antes de encriptación - Diseño en Virtex-II XC2V1000-4
[22]			***12847 (*4 LUT)***	***82***	***10.49***	- Utiliza una arquitectura PPR *(Pipelined Partial Rolling)* - Diseño con Altera Stratix 1S20C5
[36]	***2052***			***135.7***	***1.57***	- Arquitectura Sub-pipeline. - Diseño con Virtex II
[37]	***8141***			***77.69***	***4.73***	- Arquitectura Sub-pipeline. - Subclaves generadas en paralelo - Diseño con Virtex-II

Referencia	*CLBs*	*BRAMs*	*LUTs*	*Frec. (MHz)*	*Rendimiento (Gbits/s)*	*Comentarios Especiales*
[27]	***5177***	***84***	***8285***	***168.3***	***21.54***	- Arquitectura pipeline de ronda interna y ronda externa - Diseño con VirtexII-Pro
[38]	***163***	***3***		***71.5***	***0.208***	- Diseño con Spartan-3 XCV3S50-4
[38]	***146***	***3***		***123***	***0.358***	- Diseño con Virtex-II XC2V40

3.3.2. Implementaciones síncronas no segmentadas (*non-pipeline)*

Dadas las necesidades de poca área, las arquitecturas no segmentadas son útiles para llevar a cabo la implementación, en un mismo circuito, de la encriptación y la desencriptación. Este tipo de arquitecturas se basa en la implementación de la ruta de datos de una sola ronda de transformación, la cual se utiliza de forma iterativa en las 10 rondas del algoritmo AES-128.

En [39], se propuso una arquitectura para el cripto-procesador de Rijndael en la que se implementó el encriptador y el desencriptador de datos, ambos comparten el mismo circuito de distribución de clave. Este último calcula de forma iterativa las subclaves de ronda para cada ciclo de reloj a partir de la clave inicial. Cada ronda se realiza en un ciclo de reloj, excepto la primera ronda (ronda cero) que únicamente lleva a cabo la fase de adición de clave, las otras diez rondas realizan los cuatro pasos del algoritmo AES: *ByteSub, ShiftRow, MixColumn* (excepto la última ronda), y *AddRoundKey*. Por lo tanto, se requieren 11 ciclos para encriptar un bloque de datos de 128 bits. La figura 3.11 muestra el diagrama de bloques de esta arquitectura.

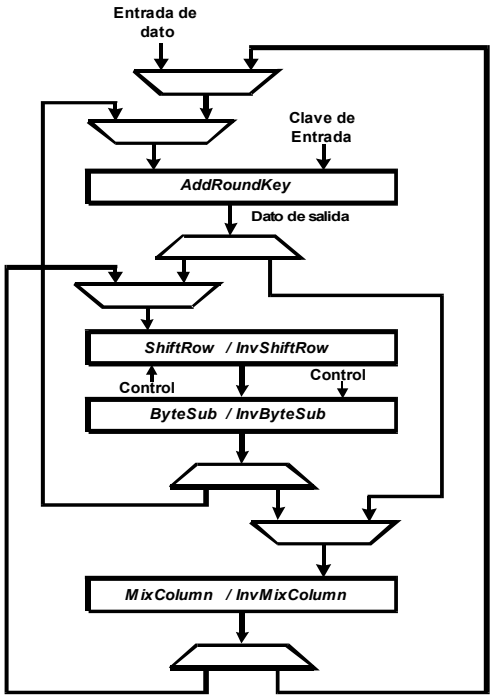


Figura 3.11: Arquitectura no segmentada para encriptación/desencriptación [39]

En este tipo de arquitectura las unidades de encriptación y desencriptación pueden coexistir en el mismo sistema en un único circuito integrado. Si la concurrencia entre encriptación y desencriptación no es tan importante, entonces se puede usar un núcleo combinado entre el encriptador y desencriptador para obtener un circuito más pequeño [39]. Ya que el encriptador y el desencriptador poseen algunos submódulos comunes se puede realizar el diseño de tal manera que estos puedan ser compartidos. La figura 3.12 ilustra esta idea.

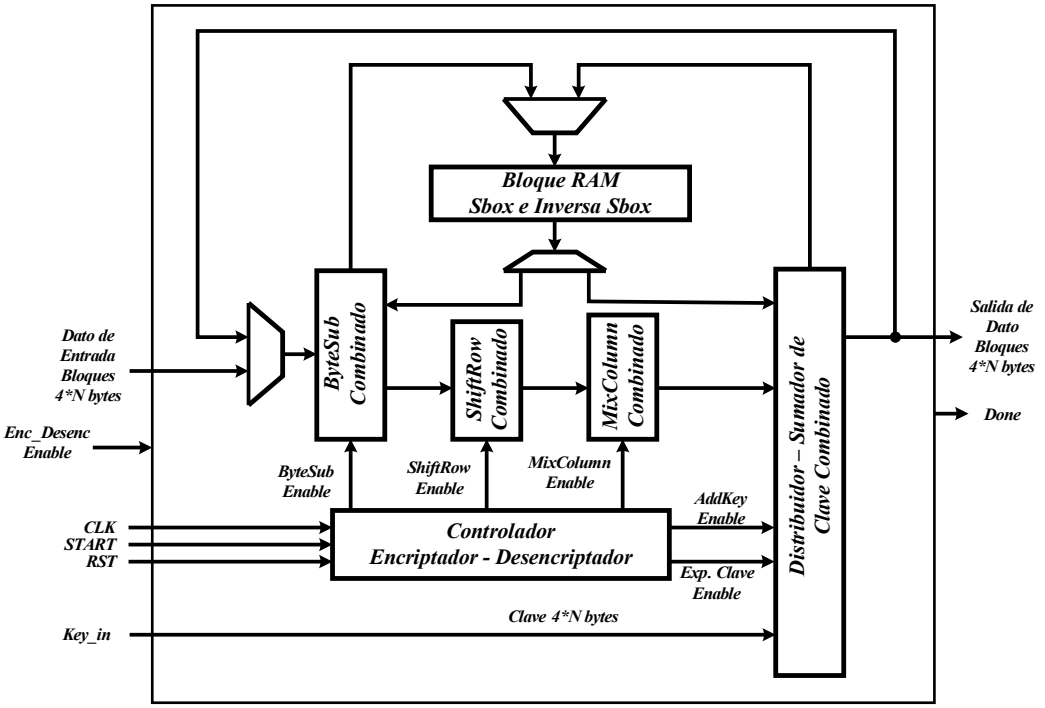


Figura 3.12: Diagrama no segmentado de encriptador/desencriptador con módulos compartidos [39]

3.4. Implementaciones asíncronas

La mayoría de las implementaciones hardware del algoritmo de Rijndael se basan en plataformas síncronas que a su vez están soportadas por herramientas de síntesis, simulación, prueba y verificación de sistemas síncronos. Esta es una de las razones por las cuales el diseño asíncrono ha sido poco explorado en el contexto del desarrollo de cripto-procesadores. Existen pocos antecedentes que muestren el desarrollo de procesadores criptográficos asíncronos; en [40] se presentan resultados sobre el desarrollo de un *Criptoprocesador de Curvas Elípticas* de bajo consumo de potencia en la Universidad de Hong Kong en China. En [41],[42] se presenta un sistema de hardware reconfigurable llamado *Async-WASMII*, el cual permite emular sistemas asíncronos mediante el uso de un dispositivo reconfigurable asíncrono llamado *PCA* (*Plastic Cell Array*). Los resultados de la implementación de un criptoprocesador *DES* asíncrono, fabricado en octubre de 2004, se muestran en [43], el circuito cumple con el estándar *NIST* para *DES*: bloques de datos y claves de 64 bits.

En [44] se resumen las características del diseño de un sistema multiprocesador de memoria compartida para encriptación/desencriptación de grandes volúmenes de datos, tal sistema utiliza un sistema de bus asíncrono. Se trata de una metodología de bajo costo y bajo esfuerzo de diseño que implementa el algoritmo AES mediante el uso de *GALS* (*Global Asynchronous local Synchronous Systems*). Según [44], el uso de las GAL reduce el riesgo de criptoanálisis basado en la medición de la potencia y la radiación electromagnética, esto debido a que los ataques de potencia diferencial se basan en el conocimiento previo del diseño del circuito integrado junto con amplitudes relativas de picos de radiación o de potencia, por lo que remover esos picos hace que este tipo de ataques sea mucho mas difícil.

El primer reporte sobre el estudio de una arquitectura AES asíncrona compatible con el estándar NIST se hizo en [45],[46]. Esta arquitectura (mostrada en la figura 3.13) explota las propiedades fundamentales del modelo de retardo *Quasi Delay Insensitive* (QDI).

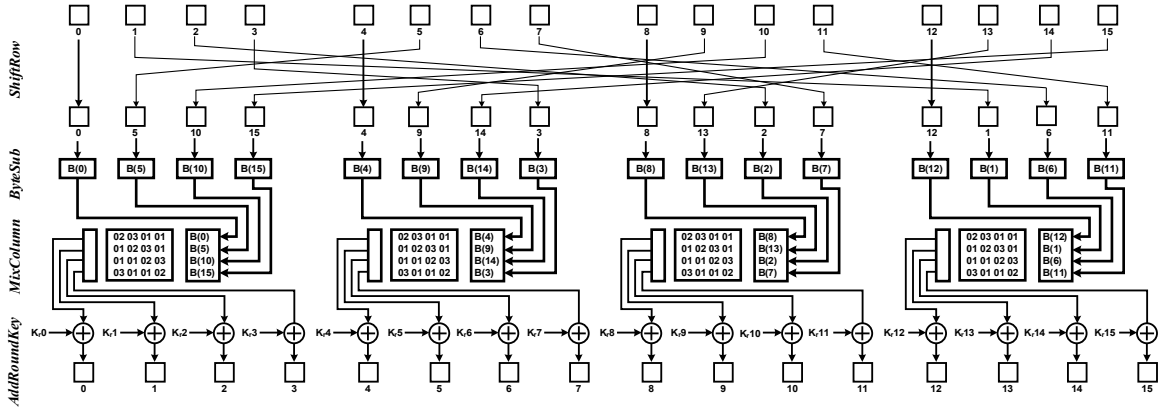


Figura 3.13: Ronda de operación asíncrona de Rijndael [36]

El método *QDI* permite realizar arquitecturas balanceadas, particularmente útiles para mejorar la resistencia a *Ataques de Potencia Diferencial* (DPA)[4][47]. El análisis de la variación del consumo de potencia revela información confidencial procesada por un circuito debido a que el consumo de potencia depende de variables como el dato procesado, las instrucciones ejecutadas, y el tipo de memoria accesada (RAM, EEPROM, etc.).

Resultados publicados en [48],[49] muestran una arquitectura asíncrona de bajo consumo de potencia que implementa el algoritmo AES. Tal diseño reduce la fuga de informacion mediante el consumo balanceado de potencia. Se utiliza la codificación de *dual-rail* y el protocolo de retorno a cero para evitar riesgos que generen glitches y para que las actividades de suicheo sean independientes de los datos [48]. La implementación usa una arquitectura compleja basada en etapas *pipeline* y con controladores implementados como cadenas de celdas de David (un tipo especial de latches que realiza un *reset* previo al almacenamiento de un nuevo dato) [49].

[4] Hay dos tipos de análisis de consumo de potencia: *SPA (Single Power Analysis)* que consiste en un análisis directo del consumo de potencia; y *DPA (Differential Power Analysis*).que utiliza análisis estadístico y correlación en el estudio del consumo de potencia [38].

La figura 3.14 muestra la arquitectura AES asíncrona de bajo costo propuesta por [48], en la que solo se usan cuatro tablas Sbox (representadas en las funciones *ByteSub*). Para implementar el algoritmo, las Sbox se reutilizan dentro de la ruta de datos y a la vez se comparten con la unidad de distribución de claves. Dentro de la unidad de datos, esta arquitectura tiene alguna desventaja en desempeño debido a que las cuatro Sbox no pueden procesar todos los 128 bits simultáneamente. En la unidad de claves no existen desventajas de desempeño debido a que las tablas Sbox se utilizan en instantes de tiempo diferentes. Los números de 1 a 5 encerrados en circulo en la figura 3.14 se utilizan en el diagrama de la figura 3.15 para ilustrar la naturaleza segmentada del algoritmo en la operación de encriptación que procede de arriba hacia abajo.

Debido a que todas las etapas del algoritmo son independientes, se puede incrementar el desempeño usando la técnica de *pipeline* insertando registros entre etapas, lo cual permite realizar cálculos simultáneamente.

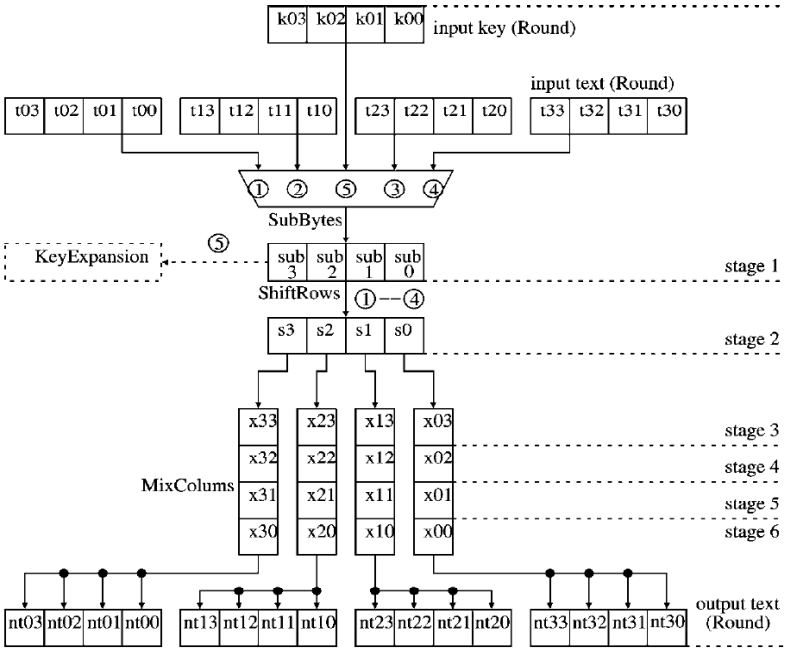


Figura 3.14: Arquitectura AES asíncrona de bajo costo [48]

En la distribución *pipeline* mostrada en la figura 3.15, los números de 1 a 4 indican el dato intermedio usado durante el procesamiento de la información, estos pasan a través de todas las etapas. El número 5 indica el dato para la generación de claves, el cual pasa sólamente a través de las etapas *ByteSub* y *ShiftRow*. El *pipeline* tiene una entrada de 32 bits y una salida de 128 bits, esto, de acuerdo con el algoritmo AES, se debe a que la etapa *MixColumn* usa todos los datos intermedios (4 x 32 bits) obtenidos de las etapas previas para producir el dato de 128 bits, el cual se utiliza en la etapa *AddRoundKey*. Así, en la figura 3.16, la etapa *MixColumn* se utiliza cuatro veces para generar una palabra de dato intermedio de 128 bits en cada ronda. Para optimizar el desempeño, la ruta de datos entre la salida de *MixColumn* y la etapa *AddRoundKey* tiene 128 bits. El circuito de control para este *pipeline* usa un modelo de Red de Petri, como se observa en la figura

3.16 [48]; en este se modelan los controladores para las entradas y la funciones *ByteSub, ShiftRow y MixColum* (no se incluyen las partes dibujadas con líneas punteadas).

	stage 1	***stage 2***	***stage 3***	***stage 4***	***stage 5***	***stage 6***
	SubBytes	***ShiftRows***	***MixCol1***	***MixCol2***	***MixCol3***	***MixCol4***
time 0	**1**					
time 1	**2**	**1**				
time 2	**3**	**2**	**1**			
time 3	**4**	**3**	**2**	**1**		
time 4	**5**	**4**	**3**	**2**	**1**	
time 5		**5**	**4**	**3**	**2**	**1**

Figura 3.15: Diagrama del *pipeline* usado en AES [48]

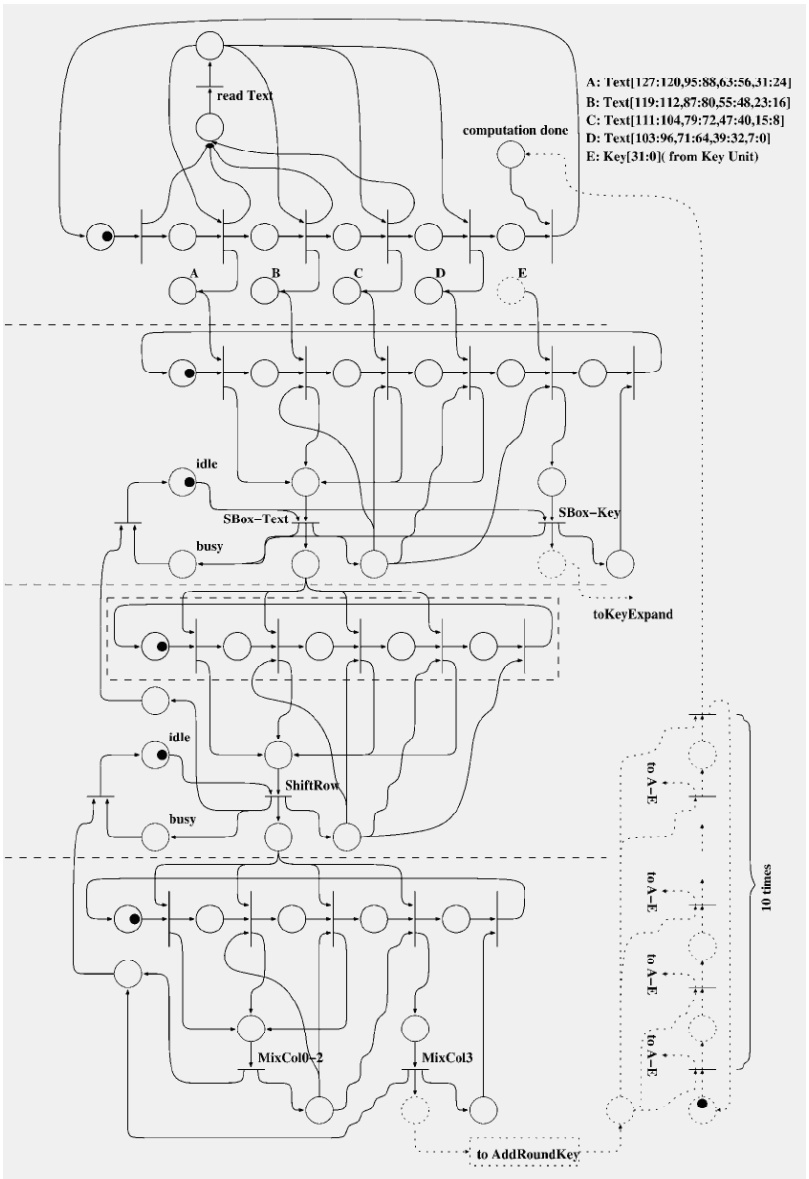


Figura 3.16: Modelo de control de Red de Petri [48],[49]

El modelo de red de Petri trabaja de la siguiente manera: luego de leer un texto en la primera etapa (entradas), se pasa un token a la segunda etapa (*ByteSub*), las entradas quedan listas para leer el segundo, tercero y cuarto texto y pasar tokens a las Sbox; las Sbox son luego disparadas por los *tokens*, estas leen el token secuencialmente y realizan los calculos para luego generar un dato intermedio (*token*) para la tercera etapa (*ShiftRow*); la etapa *ShiftRow* trabaja de forma similar a la etapa *ByteSub* pero pasa tokens a la etapa *MixColumn*; esta lee el dato (datos de 32 bits cuatro veces) y finalmente genera un dato de 128 bits para la etapa *AddRounKey*. Luego la red de Petri se mapea a los circuitos hardware.

La tabla 3.4 resume algunas características relevantes de las implementaciones asíncronas reportadas. Estas implementaciones han sido hechas sobre *ASICs* y los reportes sobre ellas solamente incluyen resultados sobre procesos de encriptación.

Tabla 3.4: Implementaciones asíncronas del Algoritmo de Rijndael

Referencia	***Desempeño***	***Tamaño (mm^2)***	***Comentarios especiales***
[45],[46]	215 Mbits/seg	0.088 (clave 128 bits)	V_{dd} = 1.2v Tecnología CMOS 0.13µm Codificación 1 de N Modelo de circuito *QDI* Arquitectura Balanceada Solo se implementa el proceso de encriptación
	721 Mbits/seg	0.192 (clave 192 bits)	
	892 Mbits/seg	0.427 (clave 256 bits)	
[48],[49]	800 ns	1.897 (clave 128 bits)	Tecnología CMOS 0.35µm Modelo de circuito *SI* Codificación *dual-rail* Arquitectura Balanceada Solo se presentan resultados de simulación para encriptación

3.4 Conclusiones del capítulo

3.4.1 El análisis de los reportes sobre hardware AES muestra consideraciones para implementaciones que optimizan area y para implementaciones de alto desempeño, en ambos casos se apunta a referenciar el estado del arte en lo relacionado con implementaciones segmentadas.

3.4.2. Se han recopilado los resultados de un conjunto de diferentes implementaciones segmentadas del algoritmo de Rijndael (estándar AES). La mayoría de ellos utilizan hardware reconfigurable (*FPGAs*) como tecnología de implementación. Solamente unos pocos trabajos presentan reportes de implementaciones sobre *ASICs* con *Standard-Cell CMOS*.

3.4.3. Desafortunadamente, no todas las implementaciones presentadas en los reportes se pueden utilizar en todas las aplicaciones reales. Por ejemplo, las arquitecturas segmentadas de ronda interna y ronda externa (*sub pipeline*), no se pueden utilizar en modos de operación realimentados.

3.4.4. Las arquitecturas no segmentadas son útiles para llevar a cabo la implementación, en un mismo circuito, de la encriptación y la desencriptación dadas las necesidades de poca área. La desventaja principal la constituye la considerable disminución del desempeño respecto de las arquitecturas segmentadas.

3.4.5. Los resultados de los trabajos publicados en [44]-[49] coinciden en que para reducir la fuga de información y mejorar la resistencia al análisis de potencia diferencial se requiere diseñar circuitos cuyo número de transiciones lógicas sea independiente del valor del dato procesado. Además, se evidencia la necesidad de evitar la probabilidad de que se presenten riesgos dinámicos con el fin de disminuir el consumo de potencia, esto se logra diseñando circuitos balanceados. Estas consideraciones sugieren que el diseño asíncrono es la mejor opción para implementar el algoritmo criptográfico de Rijndael con las mejores carateristicas de seguridad contra ataques de hardware no invasivos.

Referencias

[1] **NIST**, ***National Institute of Standards and Technology***. 2000. Disponible en: http://csrc.nist.gov/CryptoToolkit/aes/rijndael/

[2] **FIPS**, **Federal Information Processing Standards.** "S*pecification for the ADVANCED ENCRYPTION STANDARD (AES)"*. Publication 197, November 26. 2001a Disponible en: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

[3] **Weaver, N., Wawrzynek, J.** *¨High Performance, Compact AES Implementations in Xilinx FPGAs¨*. U. C. Berkeley BRASS group. 2002.

[4] **Lu, J., Lockwood, J.** *"IPSec Implementation on Xilinx Virtex-II Pro FPGA and Its Application"*. Reconfigurable Network Group, Applied Research laboratory, Washington University, St. Luis, IEEE. 2005

[5] **Segredo, A., Zabala, E., & Bellora, G.** *"Diseño de un Procesador Criptográfico Rijndael en FPGA"*. Facultad de Ingeniería, Bernard Wand-Polak. Ingeniería en Telecomunicaciones. Universidad ORT. Uruguay. 2003

[6] **Mroczkowski, P.** *"Implementation of the block cipher Rijndael using Altera FPGA"*. Military University of Technology, Warsaw, Poland. 2000

[7] **Panato, A., Barcelos M., & Reis, R**. *"A Low Device Occupation IP to Implement Rijndael Algorithm"*. Universidade federal do Rio Grande do Sul, PPGC – Instituto de Informática, Porto Alegre- RS- Brasil. 2003.

[8] **Regeb, J., Ramaswamy, V., & Ghadiri K.** *"Hardware Implementation of the Algorithm for High-Speed Networks"*. Electrical Engineering Dept, San Jose State University, San Jose, CA -95172, USA. 2003

[9] **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.

[10] **Muñoz, A.** *"Seguridad Europea para EEUU, Algoritmo de Rijndael"*. Madrid, España. 2004

[11] **Rijmen, V.** *¨Efficient Implementation of the Rijndael S-box¨*. Katholieke Universiteit Leuven, Dept. ESAT, Heverlee, Belgium, 2000. Disponible en: http://www.iaik.tu-graz.ac.at/research/krypto/AES/old/~rijmen/rijndael/sbox.pdf

[12] **Mukhopadhyay, D., Roychowdhury, D.** *"An Efficient End to End Design of Rijndael CryptoSystem in 0.18 µm CMOS"*. Proceedings of the 18th International Conference on VLSI Design; 4th International Conference on Embedded Systems Design (VLSID´05). IEEE. 2005

[13] Daemen, J., & Rijmen V. *"AES Proposal: Rijndael"*. Proton Worl Int. I, Zweefvliegtuigstraat, Brussel, Belgium; Katholieke Universiteit Leuven, ESAT-COSIC, Heverlee, Belgium. Document V. 2. 1999

[14] Angel, J.J. *"AES – Advanced Encryption Standard"*. P-p 78-81 2005. Disponible en: http://anubis.server01.org/storage/docs/cyberpunk/AES_v2005_jjaa.pdf

[15] Deworkin, M. *"Recommendation for Block Cipher Modes of Operation: Methods and techniques"*. National Institute of Standards and Technology, (NIST). *Special Publication 800-38A* 66 pages, Washington D.C. 2001. Disponible en: http://csrc.nist.gov/groups/ST/toolkit/BCM/index.html

[16] Deworkin, M. *"Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality"*. National Institute of Standards and Technology, (NIST). *Special Publication 800-38C* 25 pages, Washington D.C. 2004. Disponible en: http://csrc.nist.gov/publications/nistpubs/800-38C/SP800-38C.pdf

[17] Whiting, D., Housley, R., Ferguson, N. *¨Counter with CBC-MAC (CCM), AES Mode of Operation¨*. RSA Laboratories. Disponible en: http://csrc.nist.gov/CryptoToolkit/modes/proposedmodes/ccm/ccm.pdf

[18] Black, J., Rogaway P. *"Comments to NIST concerning AES Modes of Operations: A Suggestion for Handling Arbitrary-Length Messages with the CBC MAC"*. University of Nevada, Reno, USA; University of California at Davis, USA. 2003. Disponible en: http://csrc.nist.gov/CryptoToolkit/modes/proposedmodes/xcbc-mac/xcbc-mac-spec.pdf

[19] Deworkin, M. *"Recommendation for Block Cipher Modes of Operation: The CMAC Mode for Authentication"*. National Institute of Standards and Technology, (NIST). *Special Publication 800-38B,* 23 pages, Washington D.C. 2005. Disponible en: http://csrc.nist.gov/CryptoToolkit/modes/800-38_Series_Publications/SP800-38B.pdf

[20] Hodjat, A., Hwang, D., Lai, B., Tiri, K., Verbauwhede I. *"A 3.84 Gbits/s AES Crypto Coprocessor with Modes of Operation in a 0.18-um CMOS Technology"*. University of California, Los Angeles and Katholieke Universiteit Leuven. GLSVLSI`05, Chicago, Illinois, USA. 2005

[21] Lipmaa, H., Rogaway, P. *"Comments to NIST concerning AES Modes of Operations: CTR-Mode Encryption"*. Helsinki University of Technology, Finland; University of California at Davis. USA. Disponible en: http://www-08.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/ctr/ctr-spec.pdf

[22] Qin, H., Sasao, T., Iguchi, Y. *"An FPGA Design of AES Encryption Circuit with 128-bit Keys"*. GLSVLSI`05, Chicago, Illinois, USA. 2005

[23] ECRYPT, **Information Society**. *"State of Art in Hardware Architectures"*. *IST-2002-507932*. European Network of Excellence in Cryptology. Revision 1.0. 2005

[24] Chodowiec, P., Khuon, P., & Gaj, K. "*Fast Implementations of secret-key block ciphers using mixed inner-and outer-round pipelining"*. Proceedings ACM/SIGDA, International Simposium on Field Programmable Gate Arrays, FPGA`01, pages 94-102. Monterey, CA, USA. 2001

[25] Jarvinen, K., Tommiska, M., & Skitta, J. *"A fully Pipelined Memoryless 17.8 Gbits/s AES-128 Encrypto"*. FPGA´03, ACM 1-58113-651-x/03/0002. Monterey California, USA. 2003.

[26] 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

[27] Hodjat, A., Verbauwhede, I. *"A 21.54 Gbits/s Fully Pipelined AES Processor on FPGA"*. Electrical Engineering Department, California University, Los Angeles. 2004b

[28] FIPS, *Federal Information Processing Standards*. (2001b). "*Specification for the Advanced Encryption Standard (AES)"*. Disponible en:
http://www.ipsec.pl/aes/dfips-AES.pdf

[29] Hodjat, A., Schaumont, Patrick, Verbauwhede, I. *"Architectural Design Features of a Programmable High Throughput AES Coprocessor"*. Electrical Engineering Department, University of California, Los Angeles. 2004

[30] Weaver, N., Wawrzynek, 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.

[31] 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.

[32] Chodowiec, P., Khuon, P., Gaj, K. *"Fast Implementations of secret-key block ciphers using mixed inner-and outer-round pipelining". Proceedings ACM/SIGDA, International Simposium on Field Programmable Gate Arrays, FPGA`01*, pages 94-102. Monterey, CA, USA. 2001.

[33] 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.

[34] Alam, M., Badawy W., Jullien G. *"A novel pipelined threads architecture for AES encryption algorithm"*. In M. Schulte, S. Bhattacharyya, N. Burgess, and R.

Schreiber, editors, *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures, and Processors (ASAP),* pages 296–302, San Jose, CA, USA. IEEE Computer Society Press. 2002.

[35] Chitu, C., Chien, D., Chien, C., Verbauwhede, 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.

[36] 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.

[37] 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.

[38] 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 Microélectronique, Université Catholique de Louvain. *Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC´04). IEEE.* 2004.

[39] Ozpinar, A., Yurdakul, A. *"Configurable Design and Implementation of the Rijndael Algorithm-AES"*. *Proceedings of WIP-Euromicro2003,* Sept. 1-6, 2003, Antalya, Turkey. Disponible en: http://www.cmpe.boun.edu.tr/~yurdakul/papers/OzpinarDSD03.pdf

[40] Leung, P-K., Choy, C-S., Chang, C-F., Pun, K-P. *"A Low Power Asynchronous GF(2exp173) ALU for Elliptic Curve Crypto-processor"*. Department of Electronic Engineering, University of Hong Kong, Shatin, Hong Kong. IEEE 0-7803-7761-3/03. 2003.

[41] Adashi, Y., Ishikawa, K., Tsutsumi, S., Amano, H. *"An Implementation of the Rijndael on Async-WASMII"*. Proceedings of IEEE, pp 44-53. December 2003.

[42] Shibata, Y., Miyaski, H., Ling, S-P., Amano, H. *"HOSMII: A Virtual Hardware Integrated with DRAM" Dept. of Computer Science, Keio University and Dept. of Information and Computer Science, Kanagawa Institute of Technology.* Disponible en: http://ipdps.cc.gatech.edu/1998/raw/shibata.pdf

[43] Monnet, Y., Bouesse, F., Renaudin, M., Dumont, S., Ninon, N. *"An asynchronous DES crypto-processor secure against fault attacks"*. Concurrent Integrated Systems (C.I.S). TIMA laboratory France Grenoble Cedex ,France Circuit Exhibition Booklet Edited by Laurent Fesquet Gilles Sicard Marc Renaudin. Async March 2006.

[44] Smith, S. "*The Advanced Encryption Standard on an Asynchronous Shared-memory Multiprocessor*". Department of Electrical and Computer Engineering, Boise

State University, Boise, ID, U.S.A. International Conference on VLSI (VLSI ’03). 2003.

[45] Bouesse, F., Renaudin, M., Germain, F. *“Asynchronous AES Crypto-Processor Including Secured and Optimized Blocks”.* TIMA Laboratory, Grenoble, France; SGDN/DCISS, Paris, France. Journal of Integrated Circuits and Systems, Vol 1, No 1, pp 5-13. March 2004.

[46] Bouesse, F.; Renaudin, M.; Witon, A.; Germain, F. *“A clock-less low-voltage AES crypto-processor”***.** *Solid-State Circuits Conference*, ESSCIRC.20005. 2005. Proceedings of the 31st European. Volume , Issue , 12-16 Sept. 2005. P.p. 403–406. Digital Object Identifier 10.1109/ESSCIR.2005.1541645.

[47] KULRD and SCARD Consortium.. *“Intermediate report, Side Channel Analysis Resistant Design Flow”.* SCARD, Information Society, Project Number: IST-2002-507270, 15 January 2005.

[48] Shang, D., Burns, F., Bystrov, A., Koelmans, A., Sokolov, D., Yakovlev, A. *“A Low and Balanced Power Implementation of the AES Security Mechanism Using Self-Timed Circuits”.* E. Macii et al. (Eds.): PATMOS 2004, LNCS 3254, pp. 471–480, 2004.c. Springer-Verlag Berlin Heidelberg 2004.

[49] Shang, D., Burns, F., Bystrov, A., Koelmans, A., Sokolov, D., yakovlev, A. ***“****High-security asynchronous circuit implementation of AES”.* IEE Proceedings online no. 20050088, IEE Proc.-Comput. Digit. Tech., Vol. 153, No. 2, March 2006.

Capítulo 4: Especificación e implementación asíncrona de las funciones de transformación del algoritmo de Rijndael

4.1. Perspectiva general para la implementación de las funciones

Este capítulo presenta la descripción de las transformaciones que conforman el algoritmo de Rijndael (AES-128) para los procesos de encriptación y desencriptación y la manera en que cada una de ellas puede ser optimizada con el fin de conformar la implementación asíncrona total optimizada del algoritmo sobre una FPGA de Xilinx.

Todas las funciones de transformación se han implementado y probado de acuerdo con las normas establecidas por el NIST (*National Institute of Standards and Technology*) [1] y según el estándar *FIPS-197* [2]. Adicionalmente, con el objetivo de obtener mejor desempeño y disminuir el número de circuitos utilizados en cada transformación, se han implementado optimizaciones con base en estudios de distintos autores. Al final de cada sección se consignan los resultados obtenidos en la simulación asíncrona y la implementación en FPGA de cada función de transformación. La figura 4.1 es una representación en diagrama de bloques del algoritmo de Rijndael para los procesos de encriptación y desencriptación.

Para la especificación asíncrona y la simulación de las funciones que conforman el algoritmo de Rijndael se utilizó el sistema Balsa (descrito en el capítulo 2). Esta herramienta genera un archivo de descripción asíncrona en formato *Verilog* a alto nivel el cual es compatible con el sistema *ISE 9.2i* de *Xilinx*. El archivo en *Verilog* es sintetizado por *ISE* con el fin de lograr una implementación en hardware sobre FPGA.

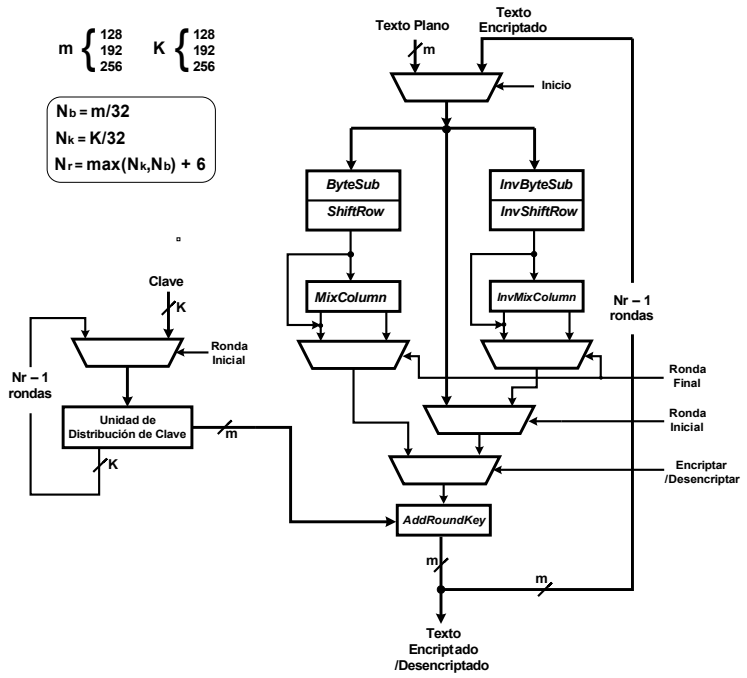


Figura 4.1: Diagrama de bloques del algoritmo AES para encriptación /desencriptación

4.2. Funciones de Transformación del Algoritmo de Rijndael

4.2.1. Funciones de Sustitución de Byte: *ByteSub e InvByteSub*

Según el estándar *FIPS-197* [2], la transformación *ByteSub* consiste en una sustitución no lineal de byte que opera independientemente sobre cada byte de la matriz de estado usando una tabla de sustitución llamada *Sbox*. La *Sbox* es una tabla invertible que se construye mediante una composición de dos transformaciones:

- Primero, cada byte se reemplaza por su inverso multiplicativo (o recíproco) en $GF(2^8)$, excepto el elemento {00}, que no tiene recíproco y se reemplaza por si mismo.
- Luego, se aplica la transformación afín sobre GF(2), descrita por la ecuación (1)

$$b'_i = b_i \oplus b_{(i+4) \bmod 8} \oplus b_{(i+5) \bmod 8} \oplus b_{(i+6) \bmod 8} \oplus b_{(i+7) \bmod 8} \oplus c_i \quad (1)$$

Para $0 \le i < 8$, donde b_i es el *i-ésimo* bit del byte, y c_i es el *i-ésimo* bit de un byte $\boldsymbol{c}$ con valor {63h} o $\{01100011_2\}$. En forma matricial, el elemento de transformación afín se puede expresar como:

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

La función *InvByteSub* es la inversa de la transformación de sustitución de byte, en la cual la *Sbox* inversa se aplica a cada byte de la matriz de estado. Ésta se obtiene aplicando el inverso de la transformación afín y luego tomando el inverso multiplicativo en $GF(2^8)$. La ecuación 2 describe la transformación afín inversa.

$$b'_i = b_{(i+2) \bmod 8} \oplus b_{(i+5) \bmod 8} \oplus b_{(i+7) \bmod 8} \oplus d_i \quad \textbf{(2)}$$

para $0 \le i < 8$, donde b_i es el *i-ésimo* bit del byte, y d_i es el *i-ésimo* bit de un byte d con valor {05h} o $\{00000101_2\}$. En forma matricial, el elemento de transformación inversa afín se expresa como:

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

La operación de mayor costo en la Sbox la constituye el cálculo de la función inverso

multiplicativo, pero gracias a que las transformaciones usadas para encriptación y desencriptación son ligeramente diferentes, la implementación de la función inverso multiplicativo se puede utilizar para ambos procesos. Ello tiene un impacto importante sobre el diseño completo en términos de eficiencia de área y potencia [7]. La figura 4.2 ilustra una arquitectura generalizada para la implementación de la función de sustitución de byte tanto para encriptación como para desencriptación.

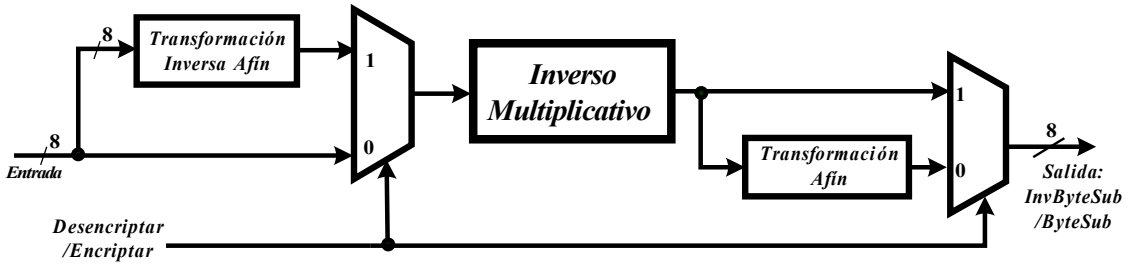


Figura 4.2: Circuito general para implementación de función *ByteSub* e *InvBytesub* de AES

De acuerdo con investigaciones reportadas en [8], se ha encontrado que las Sbox consumen mucha de la potencia total del circuito AES; en una arquitectura paralela que realiza una ronda por ciclo de reloj las Sbox consumen aproximadamente el 75% de la potencia total del circuito, de allí que gran parte de los esfuerzos de los diseñadores se ha focalizado en obtener arquitecturas que atacan el problema desde distintas ópticas. En la literatura se encuentran dos aproximaciones para diseñar circuitos Sbox:

La primera aproximación consiste en construir un único circuito cuya relación entrada/salida sea equivalente a la Sbox. Con este método se logran implementaciones rápidas. El circuito *Sbox* se puede obtener de su tabla de verdad usando niveles lógicos tales como *SOP* (*Sum of Products*), *POS* (*Product Of Sums*), *PPRM* (*Positive Polarity Reed Muller*) [8], o usando diagramas de decisión binaria como *BDD* (*Binary Decision Diagram*) [9], o *twisted-BDD* [10].

El segundo método consiste en construir un circuito que implemente la función inverso multiplicativo y otro circuito para la transformación afín, y luego conectar esos dos circuitos en serie. Esta metodología permite utilizar teoremas matemáticos sobre GF para lograr reducciones en el área del circuito [11]. Algunos de los métodos estudiados para obtener circuitos compactos de la función inverso multiplicativo sobre GF, se basan en el llamado *pequeño teorema de Fermat* [12], el algoritmo de baja latencia de Itoh-Tsujii [13],[14], y el algoritmo extendido de Euclides [15]. En particular, la inversión de campo compuesto propuesta en [16] es efectiva sobre $GF(2^8)$ como se demuestra en [17],[18].

Muchas de las implementaciones eficientes en área están basadas en la idea de transformar el campo original $GF(2^8)$ en un campo compuesto de campos más pequeños $GF((2^4)^2)$ como en [8], [17], [19]-[24].

Un estudio del consumo de potencia de los componentes AES implementados con librerías de *Standard Cells* de 0.13μm para campos compuestos se muestra en la tabla 4.1 [8]. Aquí se observa claramente que la operación *ByteSub* (para 16 *Sbox*) consume mucha más potencia que los otros componentes.

Tabla 4.1: Consumo para componentes AES [8]
(0.13 μm, 10 MHz, 1.5 V, Standard Cell CMOS, 1 compuerta = NAND de 2 entradas)

Arquitectura	*Potencia máxima observada (μW)*	*Potencia promedio (%)*
ByteSub (Sbox SOP x16)	*1940*	*75*
MixColumn	*262*	*10*
AddRoundkey	*>10*	*>1*
Multiplexores	*>10*	*>1*
Flip-Flops+Drivers de Reloj	*400*	*15*

También en [8] se hizo un estudio comparativo del desempeño de varias *Sbox* implementadas en librerías *ASIC* de *Standard Cells* de 0.13 μm, el cual concluyó que el consumo de potencia en las Sbox está fuertemente influenciado por la cantidad de riesgos dinámicos presentes en las distintas implementaciones. Dos de las principales razones por las que se pueden crear o propagar riesgos dinámicos son:

- *Diferencia de tiempos de arribo de señal a cada compuerta* debido a ramificaciones y/o cruces de rutas de señal.
- *Probabilidad de propagación de transiciones de señal* relacionada con el tipo de puerta lógica, particularmente el uso de puertas XOR.

Ya que la principal razón para el alto consumo de potencia de la Sbox de campo compuesto son las complicadas rutas de señal, algunas partes del circuito se pueden simplificar convirtiéndolas a dos niveles lógicos. Esta idea se materializó con la implementación de una arquitectura llamada *PPRM Multi-etapa*. La arquitectura (figura 4.3) consiste en convertir a dos niveles lógicos cada uno de los tres subcomponentes de la Sbox de campo compuesto: *sección de pre-inversión, sección de inversión y sección de post-inversión*. En la figura 4.3 se observan dos rutas de señal externas a la sección de inversión, y conectadas a rutas con tiempo de retardo cercano al de la inversión. Se obtiene así una arquitectura de tres etapas cuyos reportes de implementación se muestran en la tabla 4.2.

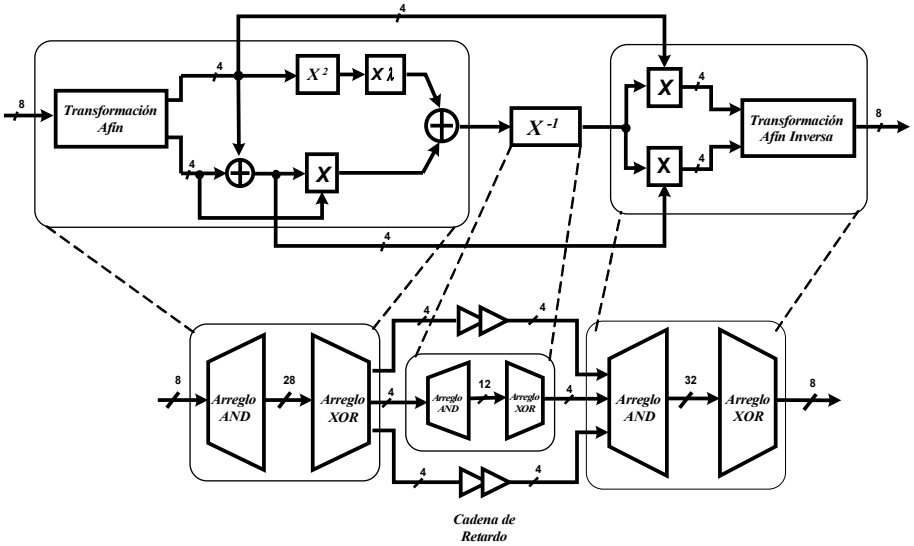


Figura 4.3: Circuito PPRM multi-etapa [8]

Tabla 4.2: Comparación de varias arquitecturas de Sbox
(0.13 μm, 10 MHz, 1.5 V, Standard Cell CMOS, 1 compuerta = NAND de 2 entradas)

ARQUITECTURA	*Tamaño (Número compuertas)*	*Potencia promedio Sbox (μW) a 10MHz)*
PPRM 3 etapas	*712*	*29*
(PPRM 5 etapas) Campo Compuesto	*354*	*136*
SOP 3 etapas	*6114*	*697*

4.2.1.1. Implementación asíncrona de los módulos Sbox

Esta sección muestra los diseños de las transformaciones *ByteSub*, *InvByteSub*, y *EDByteSub* (para encriptación/desencriptación), su especificación en lenguaje Balsa y los resultados de la implementación asíncrona en una FPGA de Xilinx.

La implementación de la función de sustitución de byte en hardware utilizando el estilo de diseño asíncrono permite evitar los problemas de los riesgos dinámicos presentados en la implementación dentro de un sistema síncrono. Esto se debe a que el estilo asíncrono se basa en modelos insensibles a los retardos, es decir, el correcto funcionamiento de un circuito no depende de los retardos asociados a sus componentes.

Tomando como base la arquitectura de Sbox PPRM desarrollada en [8] y el circuito de la figura 4.2, se implementó la arquitectura de Sbox para encriptación (función *ByteSub*) y desencriptación (función *InvByteSub*) mostrada en la figura 4.4. Se resalta el hecho que la arquitectura PPRM, en este caso implementada de manera asíncrona, no requiere de rutas de retardo externas a la sección de inversión como sí se señala en la figura 4.3.

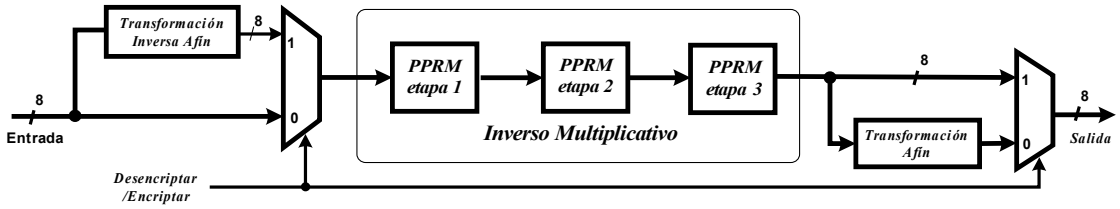


Figura 4.4: Sbox para encriptación y desencriptación.

La figura 4.5 ilustra un diagrama de bloques en el que se indica la comunicación asíncrona por medio de protocolos de *handshake* entre los módulos que conforman la Sbox. El protocolo de *handshake* puede ser de dos o cuatro fases mientras que la codificación puede ser del tipo *'single-rail'* (único-riel), *'dual-rail'* (doble-riel) ó 1-de-4, con tecnología de circuito QDI.

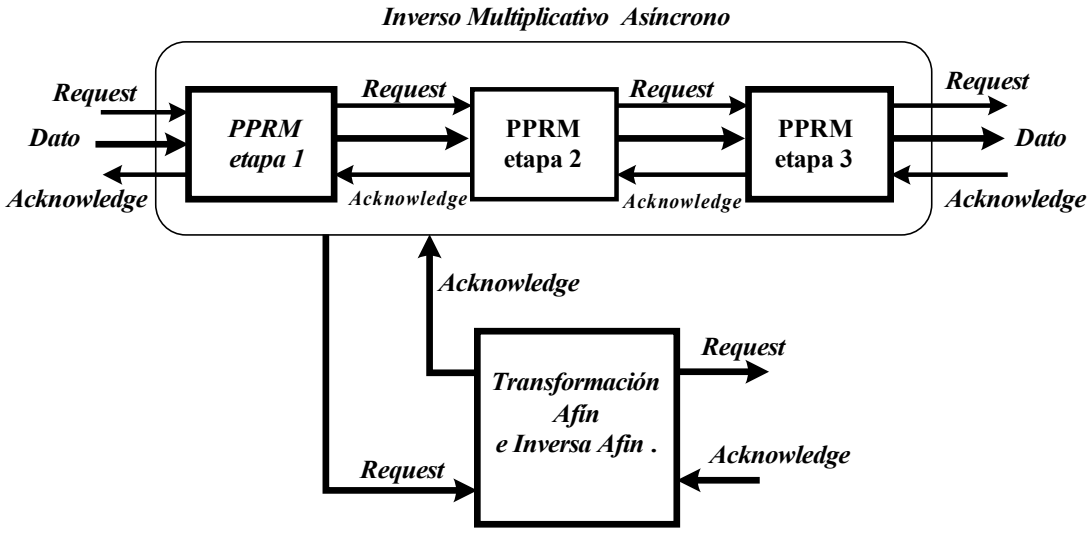


Figura 4.5: Comunicación asíncrona entre módulos de Sbox

4.2.1.2. Especificación asíncrona para las funciones *ByteSub* e *InvByteSub*

4.2.1.2.1. Módulo Inverso Multiplicativo

Para la especificación del circuito que permite calcular el inverso multiplicativo en la transformación *ByteSub*, se utilizó el sistema Balsa [25] (descrito en el capítulo 2). La descripción se hizo siguiendo parte de la especificación de la arquitectura Sbox *PPRM* desarrollada en [8] para circuitos síncronos. Esta codificación se muestra en la figura 4.6.

El diagrama de *handshake* que representa el funcionamiento asíncrono del circuito que calcula el inverso multiplicativo se hace muy denso debido a la representación funcional de circuito combinatorio que realiza, por tal razón, el diagrama no se incluye.

```
-- Función Inverso Multiplicativo
-- Implementado por medio de tres etapas combinatorias XOR-AND de acuerdo con [Satoh-Morioka]

import [balsa.types.basic]
procedure invmult1(input  ent : byte; output out : byte) is
        variable u : byte
        variable x : array 0 .. 7 of bit
        variable y : array 0 .. 7 of bit
        variable a : array 0 .. 3 of bit
        variable b : array 0 .. 3 of bit
        variable c : array 0 .. 3 of bit
        variable d : array 0 .. 3 of bit
begin
ent -> u;
x := #u; -- para convertir la entrada 'ent' en un arreglo de bits
-- Etapa 1
a[3] := x[7] xor x[5] ||
a[2] := x[7] xor x[6] xor x[4] xor x[3] xor x[2] xor x[1] ||
a[1] := x[7] xor x[5] xor x[3] xor x[2] ||
a[0] := x[7] xor x[5] xor x[3] xor x[2] xor x[1] ||
b[3] := x[6] xor x[5] xor x[2] xor x[1] ||
b[2] := x[6] ||
b[1] := x[7] xor x[6] xor x[5] xor x[4] xor x[3] xor x[2] xor x[1] ||
b[0] := x[7] xor x[6] xor x[5] xor x[3] xor x[2] xor x[0] ||
c[3] :=  (x[5] and x[1]) xor (x[7] and x[1]) xor (x[5] and x[2]) xor (x[6] and x[5]) xor (x[7] and x[5]) xor (x[5] and x[4]) xor (x[7] and x[4])
xor (x[5] and x[0]) xor (x[7] and x[0]) xor (x[3] and x[1]) xor (x[4] and x[1]) xor (x[3] and x[2]) xor (x[4] and x[2]) xor (x[6] and x[4]) xor
(x[2] and x[1]) xor (x[6] and x[2]) xor (x[6] and x[1]) ||
c[2] := (x[6] and x[1]) xor (x[6] and x[2]) xor (x[6] and x[3]) xor (x[7] and x[6]) xor (x[1] and x[0]) xor (x[2] and x[0]) xor (x[3] and x[0])
xor (x[4] and x[0]) xor (x[6] and x[0]) xor (x[7] and x[0]) xor (x[5] and x[2]) xor (x[5] and x[3]) xor (x[4] and x[2]) xor (x[4] and x[3]) xor
(x[7] and x[5]) xor (x[7] and x[2]) xor (x[6] and x[5]) xor (x[3] and x[2])xor (x[7] and x[3])  ||
c[1] := (x[2] and x[1]) xor (x[4] and x[2]) xor (x[5] and x[4]) xor (x[6] and x[3]) xor (x[6] and x[5]) xor (x[2] and x[0]) xor (x[3] and x[0])
xor (x[5] and x[0]) xor (x[7] and x[0]) xor (x[5] and x[2]) xor (x[7] and x[2]) xor (x[5] and x[3]) xor (x[7] and x[5]) xor (x[3] and x[2]) xor
x[7] xor x[5] xor x[4] xor x[2] xor x[1] ||
c[0] :=  (x[1] and x[0]) xor (x[2] and x[0]) xor (x[3] and x[0]) xor (x[5] and x[0]) xor (x[7] and x[0]) xor (x[3] and x[1]) xor (x[6] and x[1])
xor (x[6] and x[3]) xor (x[6] and x[5]) xor (x[7] and x[6]) xor (x[4] and x[3]) xor (x[7] and x[4]) xor (x[5] and x[3]) xor (x[4] and x[1]) xor
(x[3] and x[2]) xor (x[6] and x[4]) xor x[6] xor x[5] xor x[3]  xor x[2] xor x[0] ;
-- Etapa 2
d[3]:= (c[3] and c[2] and c[1]) xor (c[3] and c[0]) xor (c[3] xor c[2]) ||
d[2]:= (c[3] and c[2] and c[0]) xor (c[3] and c[0]) xor (c[3] and c[2] and c[1]) xor (c[2] and c[1]) xor c[2] ||
d[1]:= (c[3] and c[2] and c[1]) xor (c[3] and c[1] and c[0]) xor (c[2] and c[0]) xor c[3] xor c[2] xor c[1] ||
d[0]:= (c[3] and c[2] and c[0]) xor (c[3] and c[1] and c[0]) xor (c[3] and c[2] and c[1]) xor (c[3] and c[1]) xor (c[3] and c[0]) xor (c[2] and
c[1] and c[0]) xor (c[2] and c[1]) xor c[2] xor c[1] xor c[0] ;
-- Etapa 3
y[7]:= (d[3] and a[0]) xor (d[2] and a[1]) xor (d[1] and a[2]) xor (d[0] and a[3]) xor (d[2] and a[0]) xor (d[0] and a[2]) xor (d[1] and a[1]) xor
(d[1] and a[0]) xor (d[0] and a[1]) xor (d[1] and b[1]) xor (d[1] and b[0]) xor (d[0] and b[1]) xor (d[3] and b[2]) xor (d[2] and b[3]) xor (d[2]
and b[2])   ||
y[6]:= (d[2] and a[2]) xor (d[2] and a[0]) xor (d[0] and a[2]) xor (d[3] and a[3]) xor (d[3] and a[1]) xor (d[1] and a[3]) xor (d[2] and b[2])
xor (d[2] and b[0]) xor (d[0] and b[2]) xor (d[3] and b[3]) xor (d[3] and b[1]) xor (d[1] and b[3])   ||
y[5]:= (d[2] and a[0]) xor (d[0] and a[2]) xor (d[3] and a[3]) xor (d[3] and a[1]) xor (d[1] and a[3]) xor (d[1] and a[1]) xor (d[1] and a[0]) xor
(d[0] and a[1]) xor (d[3] and a[2]) xor (d[2] and a[3]) xor (d[1] and b[1]) xor (d[1] and b[0]) xor (d[0] and b[1]) xor (d[3] and b[2]) xor (d[2]
and b[3]) xor (d[2] and b[2])   ||
y[4]:= (d[2] and a[0]) xor (d[0] and a[2]) xor (d[3] and a[1]) xor (d[1] and a[3]) xor (d[1] and a[0]) xor (d[0] and a[1]) xor (d[0] and a[0]) xor
(d[2] and b[0]) xor (d[0] and b[2]) xor (d[3] and b[3]) xor (d[3] and b[1]) xor (d[1] and b[3]) xor (d[1] and b[1]) xor (d[1] and b[0]) xor (d[0]
and b[1]) xor (d[3] and b[2]) xor (d[2] and b[3])  ||
y[3]:= (d[1] and a[0]) xor (d[0] and a[1]) xor (d[2] and a[2]) xor (d[0] and a[0]) xor (d[3] and a[3]) xor (d[3] and b[0]) xor (d[0] and b[3])
xor (d[2] and b[1]) xor (d[1] and b[2]) xor (d[2] and b[0]) xor (d[0] and b[2]) xor (d[1] and b[1]) xor (d[1] and b[0]) xor (d[0] and b[1])  ||
y[2]:= (d[3] and a[1]) xor (d[3] and a[0]) xor (d[2] and a[1]) xor (d[1] and a[3]) xor (d[1] and a[2]) xor (d[0] and a[3]) xor (d[0] and a[0]) xor
(d[1] and a[1]) xor (d[3] and b[0]) xor (d[2] and b[1]) xor (d[1] and b[2]) xor (d[0] and b[3]) xor (d[2] and b[0]) xor (d[0] and b[2]) xor (d[1]
and b[1]) xor (d[1] and b[0]) xor (d[0] and b[1])  ||
y[1]:= (d[0] and a[1]) xor (d[1] and a[0]) xor (d[2] and a[2]) xor (d[0] and a[0]) xor (d[3] and a[3]) ||
y[0]:= (d[2] and a[0]) xor (d[0] and a[2]) xor (d[3] and a[1]) xor (d[1] and a[3]) xor (d[1] and a[0]) xor (d[0] and a[1]) xor (d[0] and a[0]) xor
(d[2] and b[2]) xor (d[2] and b[0]) xor (d[0] and b[2]) xor (d[3] and b[1]) xor (d[1] and b[3]) xor (d[0] and b[0]) xor (d[1] and b[1]) xor (d[3]
and b[2]) xor (d[2] and b[3]) ;

out <- (y as byte)
end
```

Figura 4.6: Código Balsa de arquitectura *PPRM* para el circuito que calcula el inverso Multiplicativo de un byte de la función *ByteSub.*

4.2.1.2.2. Módulo de Transformación afín

Este módulo (figura 4.7) se especifica con base en la expansión obtenida de la ecuación (1), el diagrama de *handshake* correspondiente se aprecia en la figura 4.8.

```
procedure afin1 (input a : byte ; output b: byte) is
        variable u    : byte
        variable y,taf   : array 0 .. 7 of bit
begin
        a ->  u ;
        y :=  #u ;
-- Transformación afín
taf[7] :=  y[3] xor y[4] xor y[5] xor y[6] xor y[7] ||
taf[6] :=  y[2] xor y[3] xor y[4] xor y[5] xor y[6] xor 0b_1 ||
taf[5] :=  y[1] xor y[2] xor y[3] xor y[4] xor y[5] xor 0b_1 ||
taf[4] :=  y[0] xor y[1] xor y[2] xor y[3] xor y[4] ||
taf[3] :=  y[0] xor y[1] xor y[2] xor y[3] xor y[7] ||
taf[2] :=  y[0] xor y[1] xor y[2] xor y[6] xor y[7] ||
taf[1] :=  y[0] xor y[1] xor y[5] xor y[6] xor y[7] xor 0b_1  ||
taf[0] :=  y[0] xor y[4] xor y[5] xor y[6] xor y[7] xor 0b_1;
                        b <- (taf as byte) -- Resultado final
end
```

Figura 4.7: Especificación para transformación afín

Para producir la gráfica de circuito de *handshake,* Balsa usa la utilidad *breeze2ps* [25], la cual genera archivo *postscript* que puede ser convertido a un formato visible mediante un editor de Adobe.

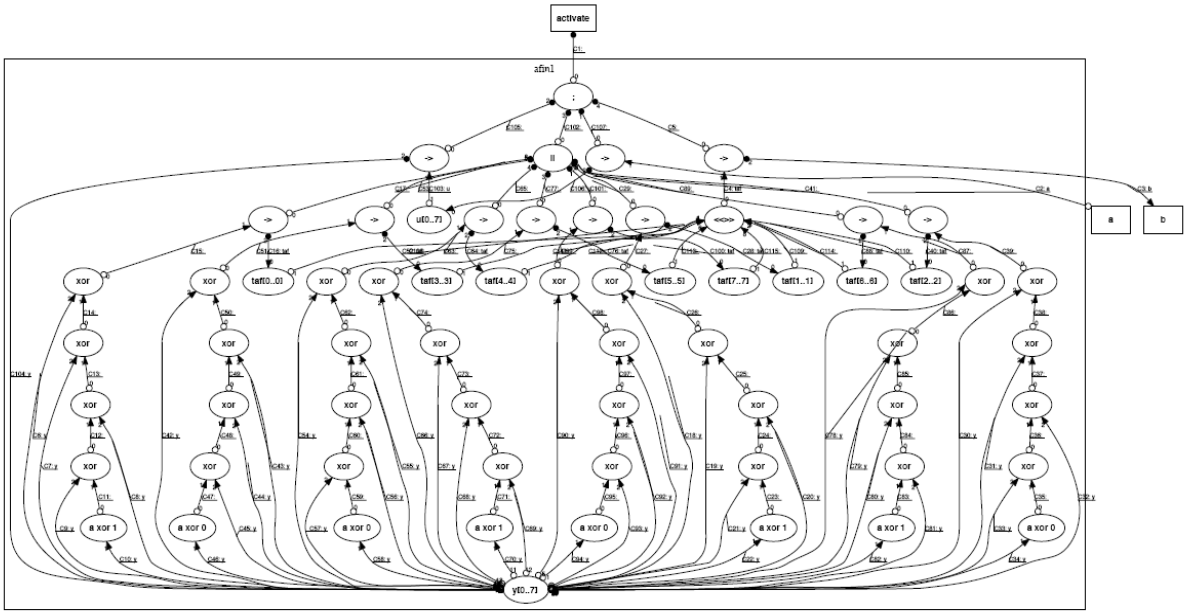


Figura 4.8: Diagrama de *handshake* para transformación afín de *ByteSub*

4.2.1.2.3. Módulo de Transformación afín inversa

De manera similar a la transformación afín, este módulo se especifica con base en la expansión obtenida de la ecuación (2). El código se muestra en la figura 4.9 y el diagrama de *handshake* correspondiente en la figura 4.10.

```
procedure inv_afin1 (input a : byte ; output b: byte) is
variable w  : byte
variable y,tiaf  : array 0 .. 7 of bit
-- Transformación inversa afín
begin
a ->  w  ;
y :=  #w ;
tiaf[7] :=  y[1] xor y[4] xor y[6] ||
tiaf[6] :=  y[0] xor y[3] xor y[5] ||
tiaf[5] :=  y[2] xor y[4] xor y[7] ||
tiaf[4] :=  xor y[1] xor y[3] xor y[6] ||
tiaf[3] :=  y[0] xor y[2] xor y[5] ||
tiaf[2] :=  y[1] xor y[4] xor y[7] xor 0b_1 ||
tiaf[1] :=  y[0] xor y[3] xor y[6] ||
tiaf[0] :=  y[2] xor y[5] xor y[7] xor 0b_1 ;
          b <- (tiaf as byte)   -- Resultado final
end
```

Figura 4.9: Especificación de transformación inversa afín

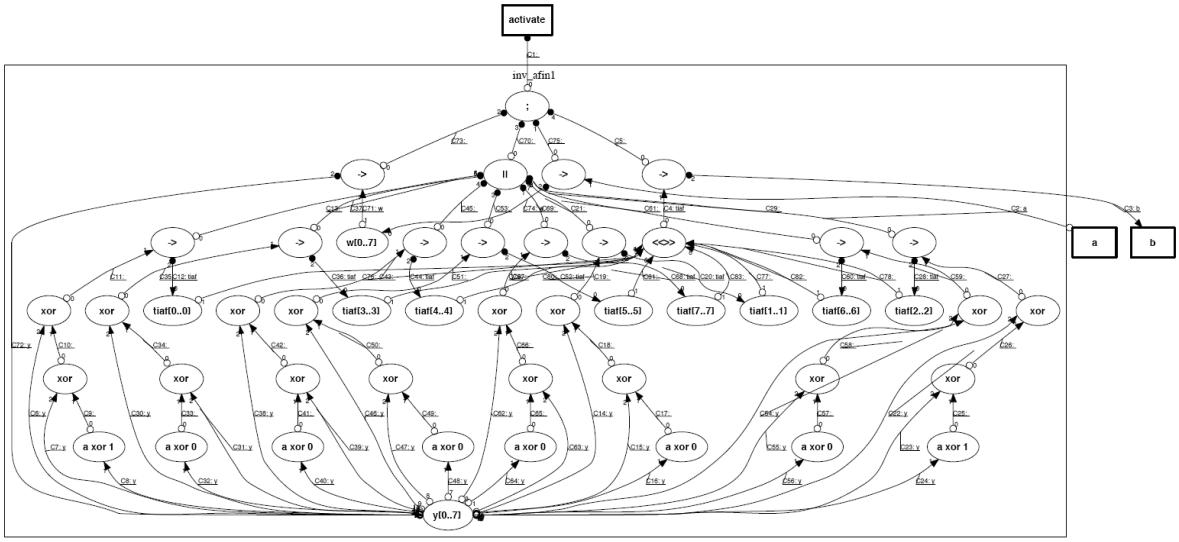
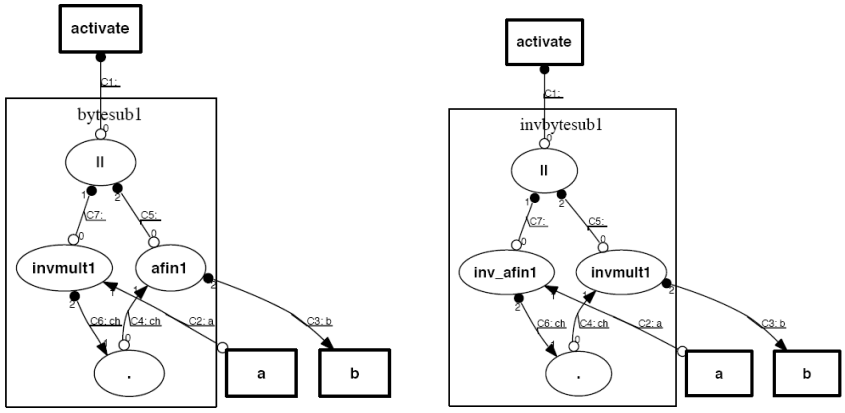


Figura 4.10: Diagrama de *handshake* para transformación inversa afín

Las figuras 4.11(a) y 4.11(b) presentan los diagramas de *handshake* obtenidos en *Balsa* al compilar las funciones *ByteSub* e *InvByteSub* por separado. Como se observa, los diagramas de *handshake* se diferencian en el tipo transformación afín involucrada y el orden en que se ejecutan los procedimientos asociados en cada función.



(a) **(b)**

Figura 4.11: Diagramas de *handshake* para (a) *ByteSub* y (b) *InvByteSub*

Como se conoce, todos los programas compilados de Balsa contienen un puerto de activación. En la figura 4.11(a), correspondiente a la función *ByteSub*, el puerto de activación se encarga de iniciar la operación del circuito mediante un *handshake* con el operador '||', este último indica que las unidades conectadas a él, *invmult1* (inverso multiplicativo) y *afin1* (transformación afín), deben operar en paralelo pero, en este caso, tales unidades no pueden operar totalmente independientes debido a que la salida de un procedimiento (*invmult1*) escribe en la entrada de otro procedimiento (*afin1*) creando un punto de sincronización. El operador '||' primero establece un *handshake* al lado izquierdo entre el componente *invmult1* y la entrada "*a*" de dato causando que se genere un resultado que se almacena internamente en una variable temporal. Luego de esto, se establece un *handshake* sobre el lado derecho entre el componente *afin1* y la salida en el que el dato temporal se procesa y envía luego a la variable '*b'* de salida. El diagrama de *handshake* de la figura 4.11(b) representa un comportamiento similar al *handshake* de la figura 4.11(a). Aquí, la entrada se recibe sobre el procedimiento *invafin1* y la salida se ejecuta con *invmult1*.

```
procedure edbytesub1 (input i: bit; input a : byte ; output b: byte) is
        channel ch     : byte
        variable modo : bit
begin
        i -> modo;
        if modo = 0b_0 then          --Encriptar
        invmult1 (a,ch)   ||
        afin1    (ch,b)
        else                         --Desencriptar
        inv_afin1 (a,ch)  ||
        invmult1  (ch,b)
        end
end
```

Figura 4.12: Procedimiento *Balsa EDByteSub* para encriptación/desencriptación

El procedimiento *EDByteSub* (figura 4.12) permite seleccionar, mediante el bit de modo de entrada *i,* el proceso de sustitución de byte para encriptación ó desencriptación. El diagrama de *handshake* resultante se ilustra en la figura 4.13. En este, el *handshake* se inicia con el operador de secuencia ';' el cual establece un *handshake* al componente de fetch '->' del lado derecho causando que el dato de entrada '*i*' se almacene en la variable '*modo*'. De acuerdo con el valor recibido en esta variable, el dato es procesado por los elementos de encriptación (*invmult1* y *afin1*) o desencriptación (*invafin1* e *invmult1*).

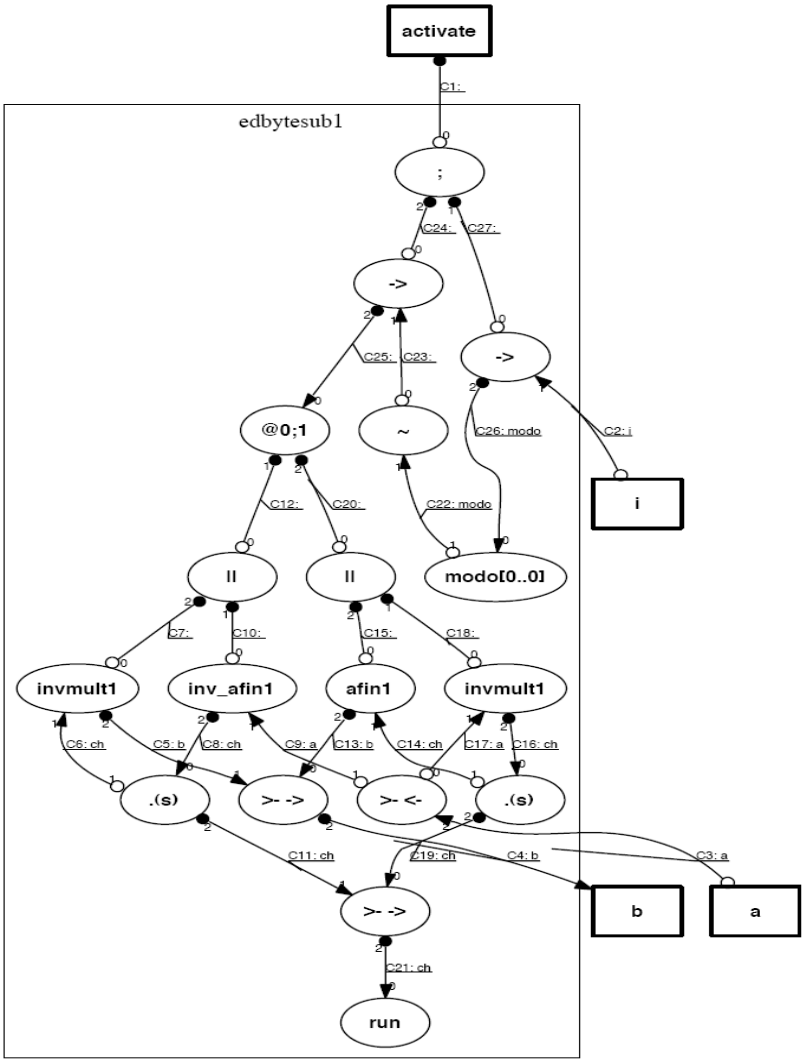


Figura 4.13: *Handshake* para *EDByteSub*

4.2.1.2.4. Implementación asíncrona de *Sbox* e *InvSbox* mediante tablas

Como se mencionó en la sección 4.2.1, la función de sustitución de byte se puede implementar de forma directa por medio de un circuito cuya relación entrada/salida sea equivalente a la Sbox de acuerdo con la tabla mostrada en la figura 4.14. Por ejemplo, la

aplicación de la tabla Sbox al byte de $9d_{16}$ dá como resultado el byte $5e_{16}$. Este valor se obtiene de la intersección de la fila 9 y la columna *d*.

		y														
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	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 4.14: Tabla *Sbox* AES [2]

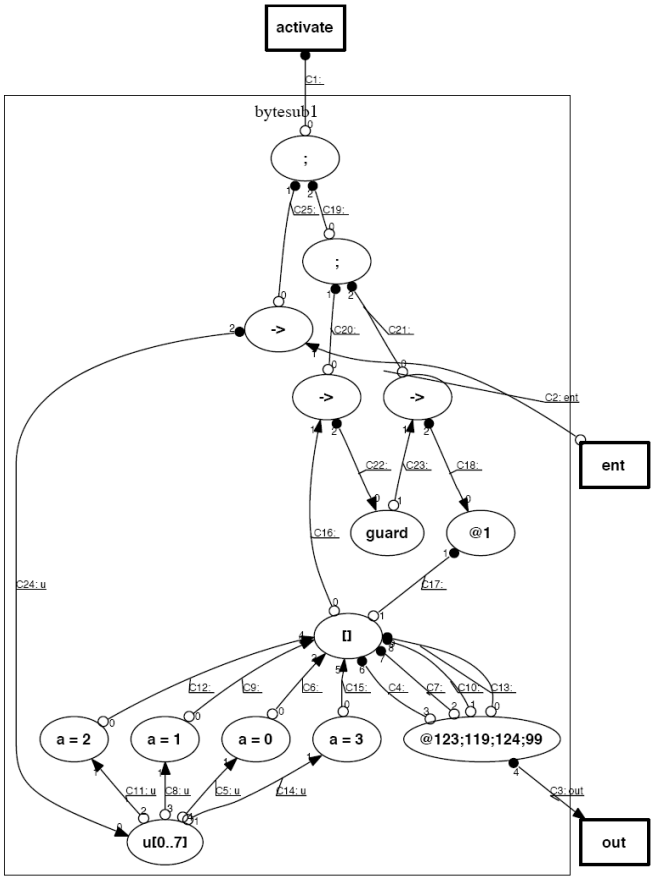


Figura 4.15: *Handshake* para tabla Sbox de cuatro bytes

La representación en diagrama de *handshake* para las 256 combinaciones de la tabla Sbox resulta demasiado densa para ser mostrada. Por tal motivo, para ilustrar su comportamiento asíncrono, se implementó una Sbox de solo cuatro bytes cuyo diagrama se observa en la figura 4.15.

La entrada del circuito está definida con cuatro posibles valores para $a=00_{16}$, $a=01_{16}$, $a=02_{16}$ o $a=03_{16}$, (en la gráfica se muestran codificados en decimal al igual que las salidas). Una vez activada la acción del circuito, se hace la solicitud de dato mediante el componente *fetch* '->', el ambiente suple la demanda del dato el cual se almacena en la variable *u*. Luego de esto se inicia un ciclo de *handshake* para la búsqueda del byte sustituto en la tabla Sbox. Para esto, el componente case '@' determina la correspondencia de la entrada almacenada en la variable con el byte de la tabla que es enviado a la salida. Se puede verificar que cuando la variable de entrada toma los valores 00_{16}, 01_{16}, 02_{16} , 03_{16}, las salidas respectivas son 66_{16}(99), $7c_{16}$(124), 77_{16} (119), $7b_{16}$ (123).

La figura 4.16 presenta la tabla para la Sbox inversa y la figura 4.17 el diagrama de *handshake* de circuito definido en Balsa (también para cuatro bytes). El comportamiento del circuito es idéntico al de la implementación asíncrona de la tabla Sbox. Adicionalmente, para encriptación/desencriptación, se implementó la función *EDByteSubTabla* cuyo diagrama de *handshake* se ilustra en la figura 4.18.

Los anexos 4.A y 4.B, al final del capítulo, presentan la codificación en Balsa para las Tablas Sbox (*ByteSubTabla*) y Sbox inversa (*InvByteSubTabla*).

		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 4.16: Tabla *Sbox* AES inversa [2]

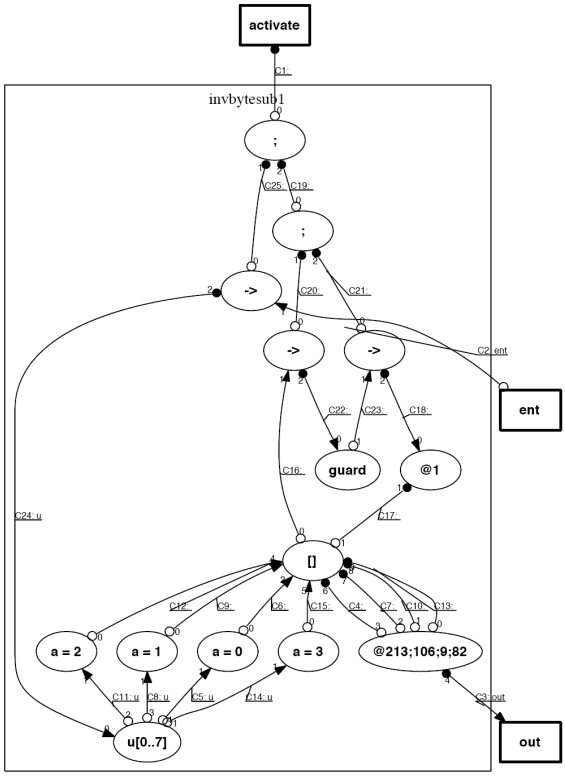


Figura 4.17: *Handshake* para tabla *Sbox* inversa de cuatro bytes

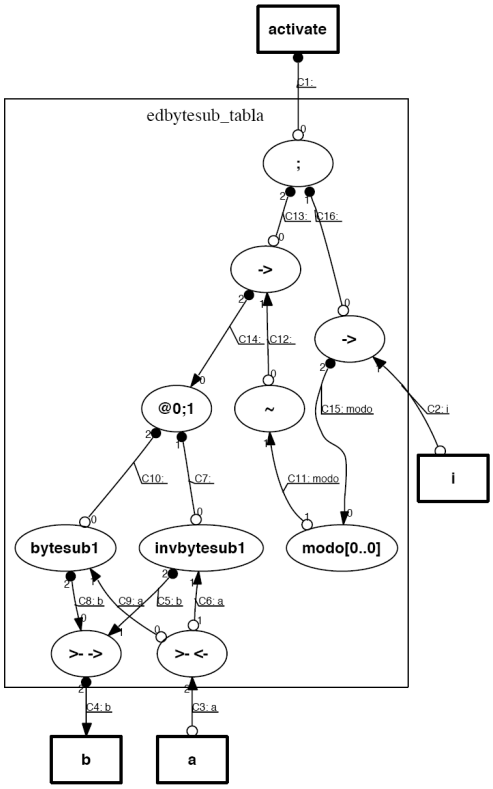


Figura 4.18: *Handshake* para *EDByteSubTabla*

4.2.1.2.5. Resultados de simulación funcional en *Balsa* e implementación en *FPGA*

La utilidad *breeze-cost* [25] de Balsa, se ha usado para estimar el área ocupada por cada circuito de *handshake* en los diseños realizados. Las unidades retornadas por *breeze-cost* son micrómetros (µm) lineales de celdas estándar sobre un viejo proceso 2LM CMOS de 1µm; la precisión de los valores es relativa al tamaño de las celdas usadas por *breeze-cost*, no a la precisión de estimación de *breeze-cost*. Las celdas primarias de esta librería básica presentan una grilla (*pitch*) de 45µm, adicionando un 50% de espacio de ruteo extra (el cual es típicamente requerido en los procesos 2LM) [28]. Balsa soporta las tecnologías AMS de 350ηm y SGS-ST de 180ηm, las cuales generan formatos del tipo *'netlist'* para producir implementaciones en Silicio. Balsa también soporta la tecnología apropiada para producir formatos utilizados en FPGAs de Xilinx.

La tabla 4.3 relaciona el costo de área relativa y el tiempo de retardo de la simulación funcional para cada procedimiento involucrado en el diseño de la Sbox asíncrona. Balsa genera el costo de área como una guía para que el diseñador sepa en qué proporción un cambio en la descripción de un circuito afecta su tamaño. Para la implementación en FPGA, Balsa genera un archivo compatible con *Verilog* para cada especificación asíncrona de circuito. El archivo en *Verilog* es sintetizado mediante el software *ISE* (*Integrated Software Environment*) versión 9.2i de *Xilinx*.

La tabla 4.4 muestra los resultados de la implementación asíncrona real en una FPGA Virtex II-Pro XCV2P30-6ff896 de Xilinx, con tecnología CMOS de 0.13µm y utilizando codificación balanceada ya sea *dual-rail* ó *1-de-4*.

Tabla 4.3: Referencia de costo de área y retardo de simulación funcional para Sbox asíncronas

Transformación	***Referencia de Costo de Área***	***Tiempo de Simulación Funcional (ns)***
ByteSub	***12742.0***	***61800***
InvByteSub	***12377.5***	***60200***
EDByteSub	***12924.0***	***Encriptación: 69200***
		Desencriptación: 67600
ByteSubTabla	***7855.0***	***8500***
InvByteSubTabla	***7855.0***	***8500***
EDByteSubTabla	***16231.5***	***Encriptación: 14400***
		Desencriptación: 14400

Tabla 4.4: Resultados de implementación en Virtex XCV2P30-6ff896 de Xilinx

	Retardo	
Transformación	***Dual-rail***	***1-de-4***
ByteSub	***217.268ns*** *Lógica 90.240ns (41.5%)* *Rutas 127.028ns (58.5%)*	***222.574ns*** *Lógica 92.038ns (41.4%)* *Rutas 130.538ns (58.6%)*
InvByteSub	***212.626ns*** *Lógica 88.361ns (41.6%)* *Rutas 124.265ns (58.4%)*	***224.424ns*** *Lógica 93.601ns (41.7%)* *Rutas 130.822ns (58.3%)*
EDByteSub (con inverso multiplicativo en tabla)	***334.00ns*** *Lógica 137.501ns (41.2%)* *Rutas 196.499ns (58.8%)*	*Balsa genera un error de síntesis cuando las descripciones de circuitos contienen definiciones de tablas con valores con estilo de codificación 1-de-4. Por tal razón no se genera archivo Verilog para que sea sintetizado por ISE 9.2i.*
ByteSubTabla	***252.360ns*** *Lógica 104.638ns (41.5%)* *Rutas 147.722ns (58.5%)*	
InvByteSubTabla	***252.360ns*** *Lógica 104.638ns (41.5%)* *Rutas 147.722ns, (58.5%)*	
EDByteSubTabla	***261.888ns*** *Lógica 108.081ns (41.3%)* *Rutas 153.807ns (58.7%)*	

El comportamiento esperado de las implementaciones en FPGA es que las transformaciones de sustitución de byte basadas en tablas sean más rápidas que las basadas en la metodología combinatoria PPRM, tal como lo indican las simulaciones hechas en Balsa. Sin embargo, los datos de las tabla 4.4, referentes a los tiempos de retardo de implementación en FPGA medidos con ISE de Xilinx muestran lo contrario.

La contradicción se puede explicar al observar que los circuitos implementados en FPGA registran mayor retardo en las rutas de interconexión (superiores al 58%) que en los elementos lógicos que conforman cada función (retardos entre el 41.2% y el 41.7%). Esta observación puede evidenciar que la FPGA utilizada en las implementaciones (Virtex XCV2P30-6ff896) no cuenta con suficientes recursos de interconexión que se adecuen a las exigencias de ruteo que requiere el diseño asíncrono. También puede evidenciar que los algoritmos de ruteo automático con los que cuenta el sistema *Xilinx ISE* no están pensados para optimizar interconexiones entre componentes asíncronos. Ambos casos apuntan a la conclusión que el estado actual de las FPGAs de Xilinx y su ambiente de diseño no han sido orientados a la implementación eficiente de circuitos asíncronos, sin embargo, es ésta la única herramienta de síntesis automática disponible que muestra un nivel importante de integración con el sistema Balsa.

NOTA: Los diagramas esquemáticos de los circuitos, los símbolos a nivel de transferencia de registros (RTL) y los reportes de área y rendimiento obtenidos mediante la síntesis de los archivos *Verilog*, pueden ser visualizados en los directorios de los proyectos asociados con cada diseño sintetizado en el programa *Xilin ISE 9.2i*. Tal información está contenida en un disco compacto anexado en este trabajo.

4.2.2. Funciones de desplazamiento de fila: *ShiftRow e InvShiftRow*

De acuerdo con el estándar FIPS-197 [2], en la transformación *ShiftRow* los bytes de las últimas tres filas de la matriz de estado se desplazan cíclicamente sobre diferente número de posiciones u *offsets*. La primera fila, $r = 0$, no se desplaza. Según el estándar, la transformación *ShiftRow* procede, de manera general para cualquier byte de la matriz de estado, de acuerdo con la siguiente expresión:

$$S'_{r,c} = S_{r,(c+\ Shift(r,Nb)) \bmod Nb} \quad \text{para } 0 < r < 4 \text{ y } 0 \le c < Nb, \qquad (3)$$

donde el valor desplazado *shift*(*r*,*Nb*) depende del número de fila *r*, por ejemplo, para *Nb*= 4 se tiene:

$$shift(1,4) = 1;\ shift(2,4) = 2\ ;\ shift(3,4) = 3 \qquad (4)$$

Esto tiene el efecto de mover bytes de posiciones altas a posiciones más bajas en la fila (valores más bajos de la columna *c* en una fila dada), mientras los bytes más bajos circulan hacia el tope de la fila. La figura 4.19 ilustra la transformación *ShiftRow* aplicada a la matriz de estado S, las filas 2, 3 y 4 se desplazan 1,2 y 3 posiciones a la izquierda respectivamente, formando la nueva matriz S'.

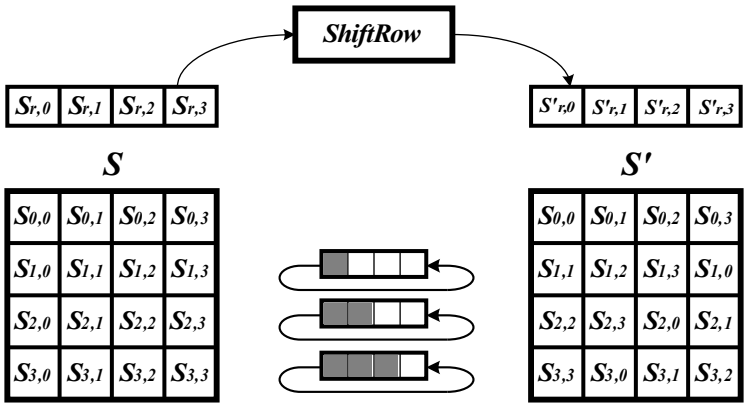


Figura 4.19: Función *ShiftRow* de desplazamiento de fila [2]

InvShiftRow es la inversa de la transformación *ShiftRow* [2]. La primera fila, *r*=0, no se desplaza. Los bytes en las últimas tres filas de la matriz de estado se desplazan cíclicamente *Nb-shift(r,Nb)* posiciones; el valor desplazado *shift(r,Nb),* dado por la ecuación (4), depende del número de fila. Según [2], la transformación *InvShiftRow* procede de manera general, para cualquier byte de la matriz de estado, de acuerdo con la siguiente expresión:

$$S_{r,c} = S'_{r,(c+\ Shift(r,Nb)) \bmod Nb} \qquad \text{para } 0 < r < 4 \text{ y } 0 \le c < Nb, \qquad (5)$$

La figura 4.20 ilustra la transformación *InvShiftRow* aplicada a la matriz de estado S', las filas 2, 3 y 4 se desplazan 1,2 y 3 posiciones a la derecha respectivamente, formando la nueva matriz *S*.

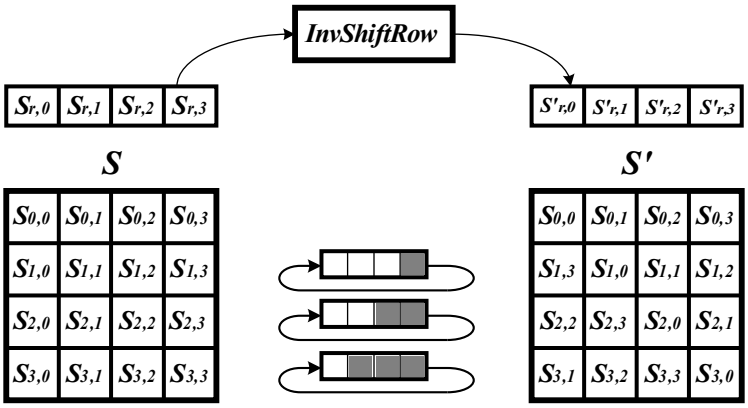


Figura 4.20: Función Inv*ShiftRow* de desplazamiento de fila de la matriz de estado [2]

Las transformaciones *ShiftRow* e *InvShiftRow* pueden ser implementadas en hardware de manera muy sencilla mediante líneas de interconexión, sin necesidad de componentes adicionales de hardware. Con el fin de indicar la actividad de *handshake* asociada con la implementación asíncrona, se muestra a continuación la especificación en Balsa de la transformación *ShiftRow*. La figura 4.21 muestra el código en Balsa de *ShiftRow*. Se puede observar la asignación de salida para cada byte de acuerdo con el desplazamiento asignado a cada fila de la matriz de estado.

```
import [balsa.types.basic]

procedure  shiftrow1(input  ent15, ent14, ent13, ent12, ent11, ent10, ent9, ent8, ent7, ent6, ent5, ent4, ent3,
ent2, ent1, ent0 : byte; output out15, out14, out13, out12, out11, out10,out9, out8,out7, out6, out5, out4,
out3,out2,out1,out0 : byte) is
variable x15,x14,x13,x12,x11,x10,x9,x8,x7,x6,x5,x4,x3,x2,x1,x0 : byte

begin
ent15  -> x15  || ent14  -> x14  || ent13  -> x13  || ent12  -> x12  || ent11  -> x11  || ent10  -> x10  ||ent9   ->
x9  || ent8   -> x8  || ent7   -> x7  || ent6   -> x6  || ent5   -> x5  || ent4   -> x4  || ent3   -> x3  || ent2  -> x2  ||
ent1   -> x1  || ent0 -> x0   ;

out15 <- x15  || out14 <- x10  || out13 <- x5   || out12 <- x0   || out11 <- x11  || out10 <- x6  || out9  <- x1   ||
out8  <- x12  || out7  <- x7   || out6  <- x2   || out5  <- x13  || out4  <- x8   || out3  <- x3   || out2  <- x14  ||  out1
<- x9   || out0  <- x4
end
```

Figura 4.21: Especificación de función *ShiftRow*

El diagrama de *handshake* simplificado correspondiente a la función *ShiftRow* se aprecia en la figura 4.22. El diagrama se dibuja simplificado debido a que este se hace ilegible cuando se dibujan todas las conexiones de control y de rutas de datos asociadas a las variables. Se observan dos procesos paralelos de *handshake* que permiten transferir el contenido de las 16 bytes de entrada a las salidas determinadas por la transformación *ShiftRow*.

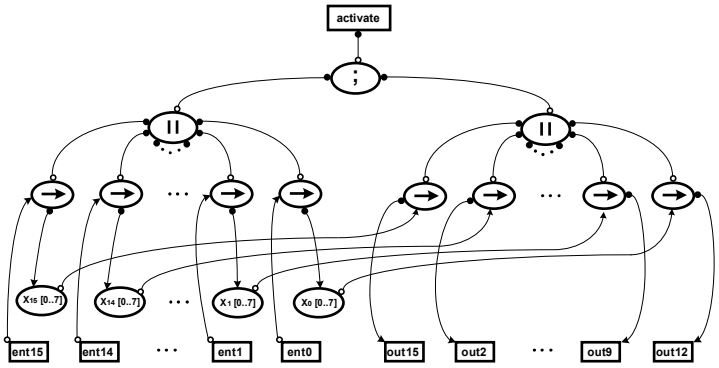


Figura 4.22: *Handshake* para transformación *ShiftRow*

Una observación cuidadosa de las figuras 4.19 y 4.20 permite concluir que las matrices de estado resultantes de la transformación *ShiftRow* e *InvShiftRow* difieren solamente en el contenido de las filas 2 y 4. Por lo anterior, ya que en ambas matrices las filas 1 y 3 son idénticas, se puede realizar la implementación de ambas transformaciones compartiendo recursos hardware. La figura 4.23 muestra la especificación en balsa para la transformación de desplazamiento de fila para encriptación/desencriptación, para este caso denominada *EDShiftRow*. La figura 4.24 muestra el diagrama esquemático para ambas transformaciones. El diagrama de *handshake* resultante para esta transformación no se incluye debido a su densidad en nivel de detalles.

```
-- Función de Desplazamiento de fila ShiftRow/InvShiftRow
import [balsa.types.basic]
procedure  edshiftrow (input i: bit; input ent15,ent14,ent13,ent12,ent11,ent10,ent9,ent8, ent7,ent6,ent5,ent4,ent3,
ent2,ent1,ent0: byte; output out15,out14,out13,out12,out11,out10,out9,out8,out7,out6,out5,out4,out3,out2,out1,out0 : byte)
is
variable x15,x14,x13,x12,x11,x10,x9,x8,x7,x6,x5,x4,x3,x2,x1,x0 : byte
variable modo : bit

begin
        i -> modo ;
        ent15  -> x15  || ent14  -> x14  || ent13  -> x13  || ent12  -> x12  || ent11  -> x11  || ent10  -> x10  ||
        ent9   -> x9  || ent8   -> x8  || ent7   -> x7  || ent6   -> x6  || ent5   -> x5  || ent4   -> x4  || ent3   -> x3  ||
        ent2   -> x2  || ent1   -> x1  || ent0   -> x0   ;

out15 <- x15 || out13 <- x5 || out11 <- x11 || out9  <- x1 || out7 <- x7  || out5  <- x13  || out3 <- x3 || out1 <- x9 ;

        if modo = 0b_0 then             -- Encriptar
out14 <- x10 || out12 <- x0  || out10 <- x6  || out8 <- x12 || out6 <- x2 || out4 <- x8 || out2 <- x14 || out0 <- x4
        | modo = 0b_1 then              -- Desencriptar
out14 <- x2 || out12 <- x8 || out10 <- x14 || out8 <- x4 || out6 <- x10 || out4 <- x0 || out2 <- x6 || out0 <- x12
        end
end
```

Figura 4.23: Código Balsa para *EDShiftRow*

El circuito de la figura 4.24 muestra que sólo los componentes de las filas 2 y 4, en este caso los bytes ubicados en posiciones pares de la matriz de estado (o bloque de entrada), difieren en la posición de destino, por lo que pueden ser manejados por un demultiplexor tal como se observa en la figura.

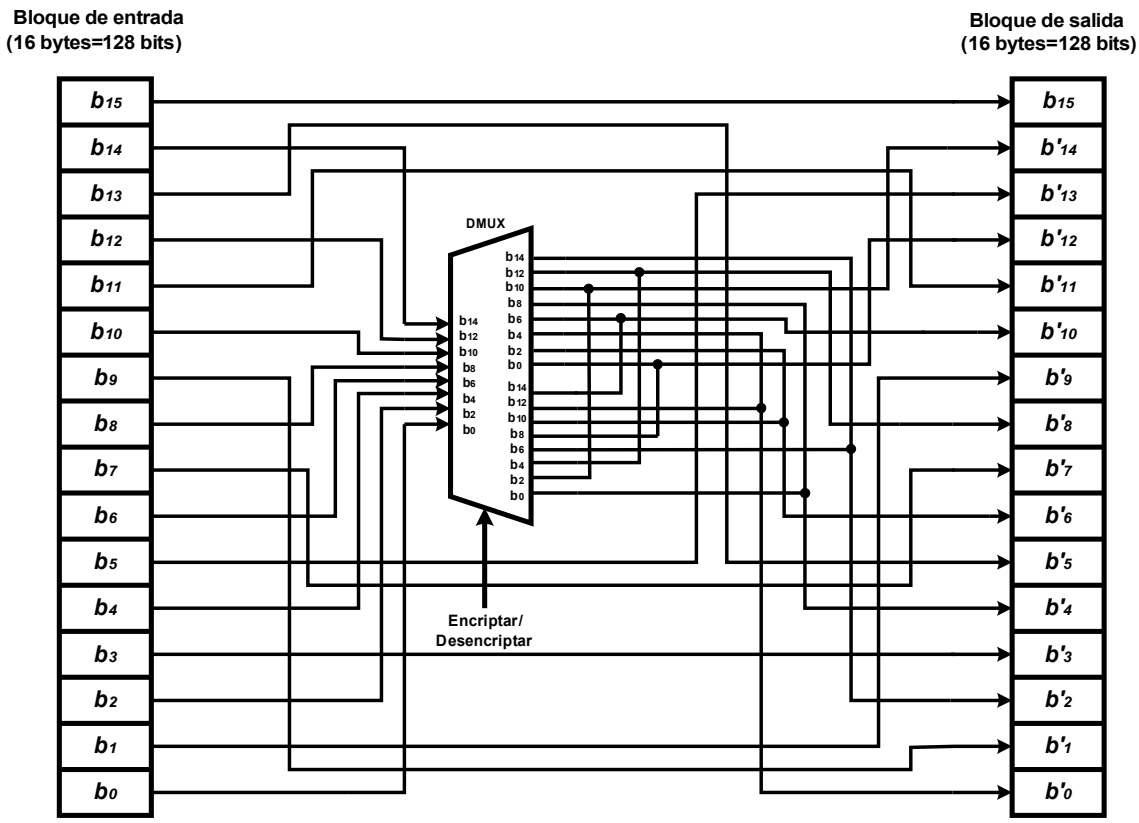


Figura 4.24: Diagrama de circuito para función de desplazamiento de fila para encriptar/desencriptar

Tabla 4.5: Referencia de costo de área y retardo de simulación funcional para transformaciones *ShiftRow/InvShiftRow* asíncronas

Transformación	***Referencia de Costo de Área***	***Retardo de Simulación Funcional (ns)***
ShiftRow	***5469.0***	***3900***
InvShiftRow	***5469.0***	***3900***
EDShiftRow	***8836.5***	***Encriptación: 10200***
		Desencriptación: 10200

Los resultados de esta tabla evidencian la equivalencia de recursos hardware utilizados para los procesos de encriptación y desencriptación por parte de las funciones de desplazamiento de fila.

Tabla 4.6: Resultados de implementación en Virtex XCV2P30-6ff896 de Xilinx

	Retardo	
Transformación	***Dual-rail***	***1-de-4***
ShiftRow	***31.788ns***	***38.348ns***
	Lógica 14.806ns (46.6%)	*Lógica 17.855ns (46.6%)*
	Rutas 16.982ns (53.4%)	*Rutas 20.493ns (53.4%)*
InvShiftRow	***31.788ns***	***38.348ns***
	Lógica 14.806ns (46.6%)	*Lógica 17.855ns (46.6%)*
	Rutas 16.982ns (53.4%)	*Rutas 20.493ns (53.4%)*
EDShiftRow	***47.404ns***	***53.967 ns***
	Lógica 21.066ns (44.4%)	*Lógica 24.115ns (44.7%)*
	Rutas 26.338ns (55.6%)	*Rutas 29.852ns (55.3%)*

Al analizar los resultados de implementación en FPGA presentados en la tabla 4.6, puede sorprender el hecho que, para ambos estilos de codificación, los retardos asociados con las rutas de interconexión de los componentes asíncronos sean mayores que los retardos inducidos por la lógica circuital. Además, las funciones de desplazamiento de fila, en ambos casos están implementadas principalmente por conmutación de líneas de las cuales se espera un retardo mínimo e inferior al introducido por los componentes lógicos asociados. Estos resultados refuerzan la idea en el sentido que las FPGAs utilizadas, a pesar de permitir el mapeo de circuitos asíncronos, no están adecuadas aún para implementar eficientemente tales circuitos.

4.2.3. Funciones de Transformación de Columna: *MixColumn e InvMixColumn*

4.2.3.1. Transformación *MixColumn*

Esta función opera columna por columna sobre la matriz de estado, tratando cada columna como un polinomio de cuatro términos. Las columnas son consideradas como polinomios sobre $GF(2^8)$ y multiplicados módulo $x^4 + 1$ con un polinomio fijo $c(x)$, dado por:

$$c(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\} \tag{6}$$

Lo anterior se puede escribir como una multiplicación de matrices de la siguiente forma:

$$S = c(x) \otimes S(x) \tag{7}$$

$$\begin{bmatrix} S'_{0,c} \\ S'_{1,c} \\ S'_{2,c} \\ S'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} S_{0,c} \\ S_{1,c} \\ S_{2,c} \\ S_{3,c} \end{bmatrix} \qquad \text{para } 0 \le c < Nb \tag{8}$$

Los bytes de cada columna pueden se reemplazados por el resultado obtenido de la multiplicación de acuerdo con las siguientes expresiones:

$$
\begin{aligned}
S'_{0,c} &= (\{02\} \bullet S_{0,c}) \oplus (\{03\} \bullet S_{1,c}) \oplus S_{2,c} \oplus S_{3,c} \\
S'_{1,c} &= S_{0,c} \oplus (\{02\} \bullet S_{1,c}) \oplus (\{03\} \bullet S_{2,c}) \oplus S_{3,c} \\
S'_{2,c} &= S_{0,c} \oplus S_{1,c} \oplus (\{02\} \bullet S_{2,c}) \oplus (\{03\} \bullet S_{3,c}) \\
S'_{3,c} &= (\{03\} \bullet S_{0,c}) \oplus S_{1,c} \oplus S_{2,c} \oplus (\{02\} \bullet S_{3,c})
\end{aligned}
\qquad (9)
$$

Las ecuaciones (9) muestran multiplicaciones de polinomios por potencias de *x* en $GF(2^8)$. Estas operaciones corresponden a multiplicaciones módulo un polinomio binario irreducible de grado 8 [3]. Para Rijndael, este polinomio es llamado *m(x)* y está dado por

$$m(x) = x^8 + x^4 + x^3 + x + 1 \qquad (10)$$

Para facilitar la implementación de la transformación *MixColumn* en hardware, los autores del algoritmo definieron la función llamada *xtime*, la cual simplifica la operación de un polinomio por potencias de *x*. Esta función consiste en aplicar un desplazamiento a la izquierda al valor que representa el polinomio además de una operación XOR con $1B_{16}$ (00011011_2) cuando el resultado de la multiplicación deba ser reducido módulo *m(x)*. Una manera sencilla de entender la función *xtime* es la siguiente:
Sea el polinomio

$$b(x) = b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0 \qquad (11)$$

Entonces

$$b(x) \bullet x = b_7x^8 + b_6x^7 + b_5x^6 + b_4x^5 + b_3x^4 + b_2x^3 + b_1x^2 + b_0x \qquad (12)$$

La multiplicación polinomial $b(x) \bullet x$ se obtiene reduciendo el anterior resultado módulo *m(x)*. Si $b_7 = 0$, esta reducción es la operación identidad; si $b_7 = 1$, se debe restar el resultado de *m(x)* (mediante una operación XOR). Según lo anterior, una multiplicación por *x* (o por 02_{16}) se puede implementar a nivel de byte como un desplazamiento a la izquierda seguido de una operación XOR condicional con $1B_{16}$. Esta operación está denotada como *b=xtime(a)*, donde *a* es el polinomio a multiplicar por *x*. Se pueden realizar multiplicaciones por potencias mayores de *x* mediante la aplicación repetida de *xtime* [3].

Al realizar la operación modular de (12) con *m(x)* se obtiene una expresión que permite implementar el circuito para *xtime* a nivel de bits:

$$xtime(a) = a(x) \bullet x \bmod m(x) = a_7x^8 + a_6x^7 + a_5x^6 + a_4x^5 + a_3x^4 + a_2x^3 + a_1x^2 + a_0x \bmod x^8 + x^4 + x^3 + x + 1$$

Entonces

$$b(x) = xtime(a) = a_6x^7 + a_5x^6 + a_4x^5 + (a_7 + a_3)x^4 + (a_7 + a_2)x^3 + a_1x^2 + (a + a_0)x + a_7 \qquad (13)$$

De esta expresión se deduce que:

$$b_7 = a_6, \quad b_6 = a_5, \quad b_5 = a_4, \quad b_4 = a_7 + a_3,$$
$$b_3 = a_7 + a_2, \quad b_2 = a_1, \quad b_1 = a_7 + a_0, \quad b_0 = a_7 \tag{14}$$

La Figura 4.25 representa el circuito combinatorio de la función *xtime* a nivel de bits implementado de acuerdo con los términos deducidos. Se puede ver que solo se requieren tres puertas XOR para la implementación de la función.

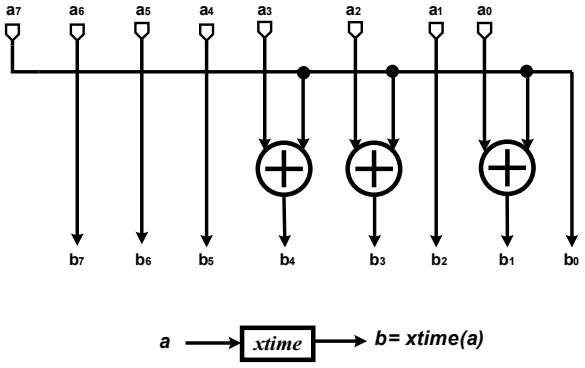


Figura 4.25: Representación circuital de la función *xtime* [29], [30]

La figura 4.26 muestra el código para la especificación asíncrona del circuito que implementa la función *xtime* en Balsa. El diagrama de *handshake* resultante de la compilación del código para *xtime* se aprecia en la figura 4.27.

```
import [balsa.types.basic]
procedure xtime1 (input i: byte; output o : byte)
is
variable u : byte
variable x,y : array 0 .. 7 of bit
begin
i -> u;
x := #u;
y[7] := x[6] ||
y[6] := x[5] ||
y[5] := x[4] ||
y[4] := x[3] xor x[7] ||
y[3] := x[2] xor x[7] ||
y[2] := x[1] ||
y[1] := x[0] xor x[7]  ||
y[0] := x[7]  ;
   o <-  (y as byte)
end
```

Figura 4.26: Especificación en Balsa de función *xtime*

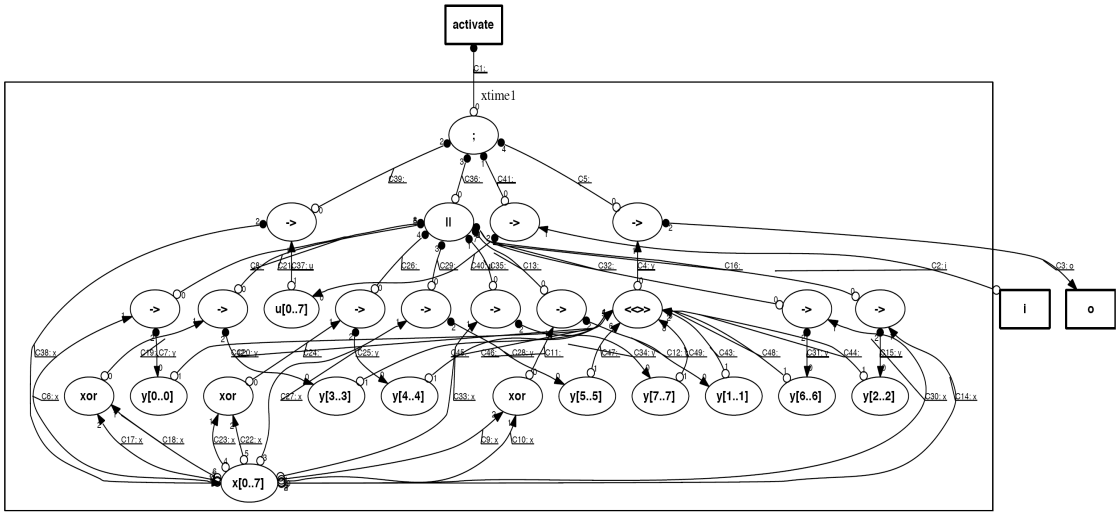


Figura 4.27: *Handshake* de función *xtime*

Las ecuaciones (9) muestran multiplicaciones de polinomios por potencias de *x* (representadas por los coeficientes 02_{16} y 03_{16}). Ya que multiplicar un polinomio A por *x* (equivalente a multiplicar por 02_{16}) consiste en aplicar la función *xtime* al polinomio, entonces multiplicar A por 03_{16} resulta sencillo si se tiene en cuenta que $03_{16}=00000011_2=0000\ 0010 \oplus 0000\ 0001$, es decir:

$$A \bullet \{03\} = A \bullet (\{02\} \oplus \{01\}) = A \bullet \{02\} \oplus A = xtime(A) \oplus A \qquad (15)$$

Con esta distribución [30],[31] se implementa fácilmente el circuito que realiza la transformación *MixColumn* a partir de (9) tal como se aprecia en la figura 4.28.

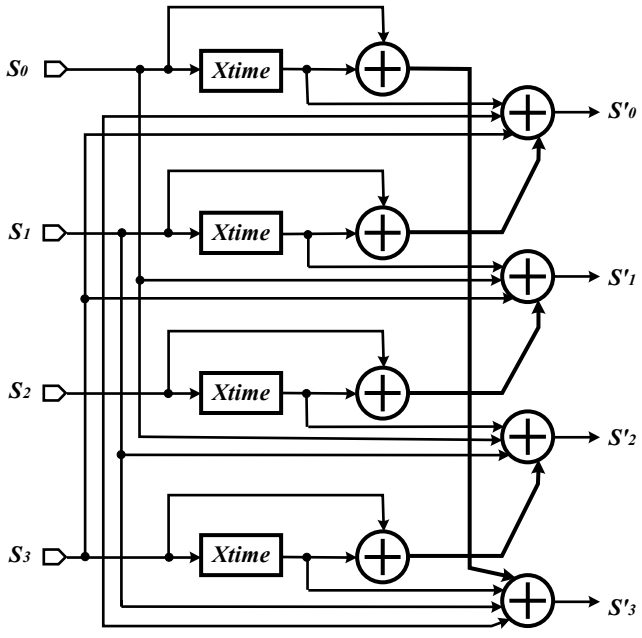


Figura 4.28: Diagrama de circuito de la transformación *MixColumn* [11],[20]

```
import [balsa.types.basic]
import [Xtime]
procedure previoxtime1(input  a3,a2,a1,a0: byte ; output  b3,b2,b1,b0: byte) is
variable  x3,x2,x1,x0 : byte
begin
a3 -> x3 ||  a2 -> x2 ||  a1 -> x1 ||  a0 -> x0 ;
b3 <- x3 xor x2   ||  b2 <- x2 xor x1   ||  b1 <- x1 xor x0   ||  b0 <- x3 xor x0
end
procedure mixcolumn1 (input  i3,i2,i1,i0: byte ; output  o3,o2,o1,o0: byte) is
variable  x3,x2,x1,x0 : byte
variable  y3,y2,y1,y0 : byte
variable  z : byte
channel   ch3,ch2,ch1,ch0    : byte
begin
i3 -> x3 || i2 -> x2 || i1 -> x1 || i0 -> x0 ;   z := x3 xor x2 xor x1 xor x0  ||
previoxtime1 (i3,i2,i1,i0,ch3,ch2,ch1,ch0) ||
xtime1(ch3,-> y3) || xtime1(ch2,-> y2) || xtime1(ch1,-> y1) || xtime1(ch0,-> y0) ;
o3 <- z xor y3 xor x3 ||
o2 <- z xor y2 xor x2 ||
o1 <- z xor y1 xor x1 ||
o0 <- z xor y0 xor x0
end
```

Figura 4.29: Código Balsa para transformación *MixColumn*

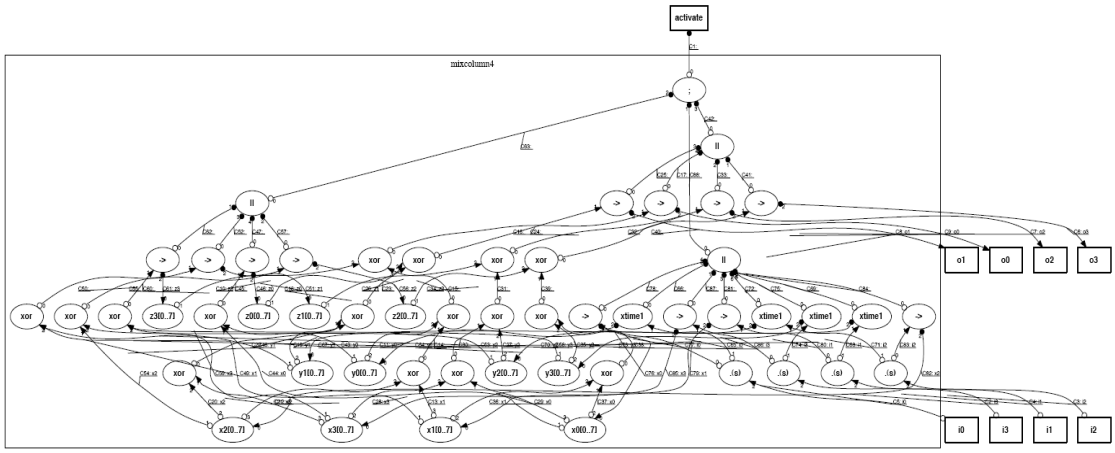


Figura 4.30: *Handshake* de transformación *MixColumn*

La transformación del bloque de datos completo se logra ejecutando en paralelo cuatro veces la transformación *MixColumn*. Cada Circuito recibe como entrada una columna de la matriz de estado y genera las salidas correspondientes de la nueva matriz, tal como lo muestra el diagrama de *handshake* simplificado de la figura 4.31.

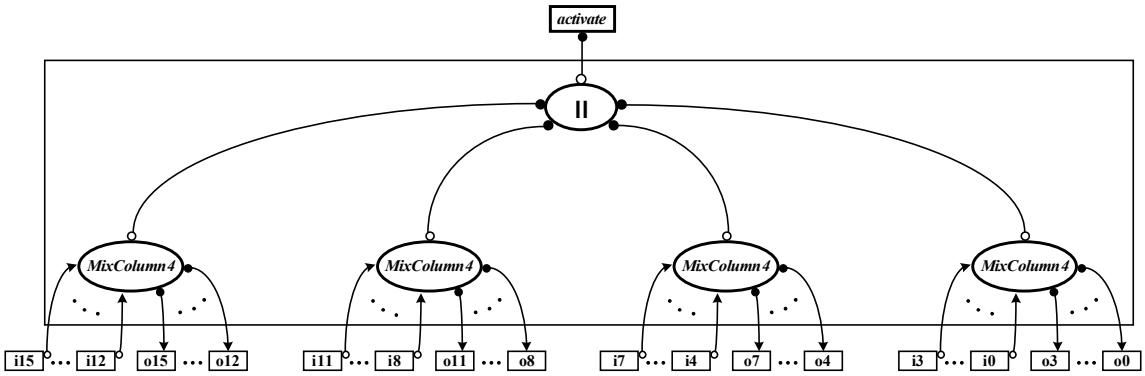


Figura 4.31: *Handshake* de transformación *MixColumn* para 16 bytes

4.2.3.2. Transformación *InvMixColumn*

InvMixColumn es la inversa de la transformación *MixColumn*, y opera sobre la matriz de estado columna por columna, tratando cada columna como un polinomio de cuatro términos. Las columnas son consideradas como polinomios sobre $GF(2^8)$ multiplicados módulo $x^4 + 1$ con un polinomio fijo $c^{-1}(x)$, en adelante llamado $d(x)$, dado por:

$$d(x) = c^{-1}(x) = \{0b\}x^3 + \{0d\}x^2 + \{09\}x + \{0e\} \tag{16}$$

La ecuación (16) se puede escribir como una multiplicación de matrices de la siguiente forma:

$$S' = c^{-1}(x) \otimes S(x) \tag{17}$$

$$\begin{bmatrix} S'_{0,c} \\ S'_{1,c} \\ S'_{2,c} \\ S'_{3,c} \end{bmatrix} = \begin{bmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{bmatrix} \begin{bmatrix} S_{0,c} \\ S_{1,c} \\ S_{2,c} \\ S_{3,c} \end{bmatrix} \qquad \text{para} \quad 0 \le c < Nb \tag{18}$$

Al resolver la multiplicación anterior se obtienen las siguientes expresiones para reemplazar cada byte de la columna c [3]:

$$\begin{aligned} S'_{0,c} &= (\{0e\} \bullet S_{0,c}) \oplus (\{0b\} \bullet S_{1,c}) \oplus (\{0d\} \bullet S_{2,c}) \oplus (\{09\} \bullet S_{3,c}) \\ S'_{1,c} &= (\{09\} \bullet S_{0,c}) \oplus (\{0e\} \bullet S_{1,c}) \oplus (\{0b\} \bullet S_{2,c}) \oplus (\{0d\} \bullet S_{3,c}) \\ S'_{2,c} &= (\{0d\} \bullet S_{0,c}) \oplus (\{09\} \bullet S_{1,c}) \oplus (\{0e\} \bullet S_{2,c}) \oplus (\{0b\} \bullet S_{3,c}) \\ S'_{3,c} &= (\{0b\} \bullet S_{0,c}) \oplus (\{0d\} \bullet S_{1,c}) \oplus (\{09\} \bullet S_{2,c}) \oplus (\{0e\} \bullet S_{3,c}) \end{aligned} \tag{19}$$

Ya que en $GF(2^8)$ la multiplicación es distributiva respecto de la adición, se puede reescribir la multiplicación de dos elementos como una combinación lineal de productos,

por ejemplo, se puede expresar $x \bullet \{0b\}$ como $x \bullet \{08 \oplus 02 \oplus 01\} = \{x \bullet 08\} \oplus \{x \bullet 02\} \oplus \{x \bullet 01\}$ para cualquier $x \in GF(2^8)$. Teniendo en cuenta esta propiedad, los coeficientes en (14) se pueden expresar como sumas de elementos en potencias de dos, lo cual facilita la implementación del circuito que genera los coeficientes $09_{16}, 0b_{16}, 0d_{16}, 0e_{16}$ a partir de la apropiada interconexión de circuitos *xtime*. El circuito mostrado en la figura 4.32 es un multiplicador que genera los coeficientes requeridos y constituye la base de la implementación del circuito para *InvMixColumn*. La especificación en Balsa del multiplicador y el diagrama de *handshake* resultante se observan en figura 4.33 y figura 4.34, respectivamente.

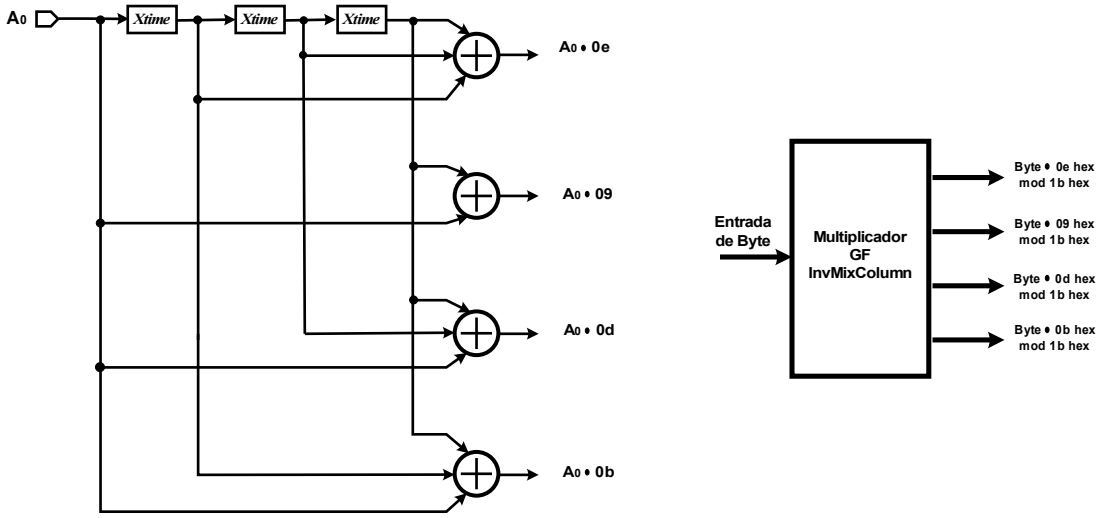


Figura 4.32: Multiplicador para función *InvMixColumn* [31]

```
import [balsa.types.basic]
import [Xtime]
procedure mult_GF(input i: byte ; output  o3,o2,o1,o0: byte) is
variable z3,z2,z1,z0 : byte
channel  ch2,ch1,ch0 : byte
begin
i -> z3 ||
xtime1 (i,->z2) ||
xtime1(i,ch2) || xtime1 (ch2,->z1) ||
xtime1(i,ch1) || xtime1 (ch1,ch0) || xtime1 (ch0,->z0);
o3 <- z2 xor z1 xor z0 ||
o2 <- z3 xor z0          ||
o1 <- z3 xor z1 xor z0 ||
o0 <- z3 xor z2 xor z0
end
```

Figura 4.33: Código Balsa del multiplicador para *InvMixColumn*

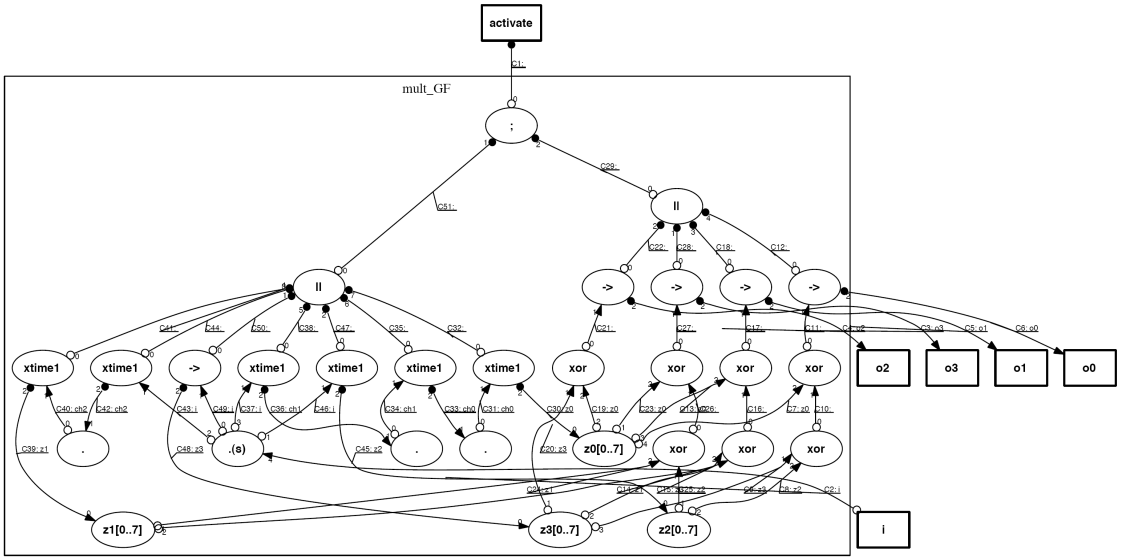


Figura 4.34: *Handshake* del multiplicador para *InvMixColumn*

Como la transformación *InvMixColumn* opera sobre una columna a la vez, se requieren cuatro circuitos multiplicadores para realizar la transformación en paralelo de los cuatro bytes de la columna. El código en Balsa para esta implementación, como se aprecia en la figura 4.35, hace uso de cuatro multiplicadores (*mult_GF*), cuyas salidas se conectan como entradas a las diferentes puertas XOR de salida, tal como lo expresa (19). La figura 4.36 muestra el diagrama de *handshake* correspondiente. En este diagrama se observa al operador paralelo '||' conectando los cuatro procesos asociados con los multiplicadores y sus conexiones de *handshake* con las variables internas. Las variables a su vez indican las conexiones hacia las puertas XOR y las salidas respectivas.

```
import [balsa.types.basic]
import [Multiplicador_GF]
procedure invmixcolumn4 (input i3,i2,i1,i0: byte ; output o3,o2,o1,o0: byte) is
variable u15,u14,u13,u12,u11,u10,u9,u8,u7,u6,u5,u4,u3,u2,u1,u0 : byte
begin
mult_GF(i3,->u15,->u14,->u13,->u12)  ||
mult_GF(i2,->u11,->u10,->u9,->u8)  ||
mult_GF(i1,->u7,->u6,->u5,->u4)  ||
mult_GF(i0,->u3,->u2,->u1,->u0)  ;
o3 <- u15 xor u8 xor u5 xor u2  ||
o2 <- u14 xor u11 xor u4 xor u1 ||
o1 <- u13 xor u10 xor u7 xor u0 ||
o0 <- u12 xor u9 xor u6 xor u3
end
```

Figura 4.35: Código Balsa de transformación *InvMixColumn*

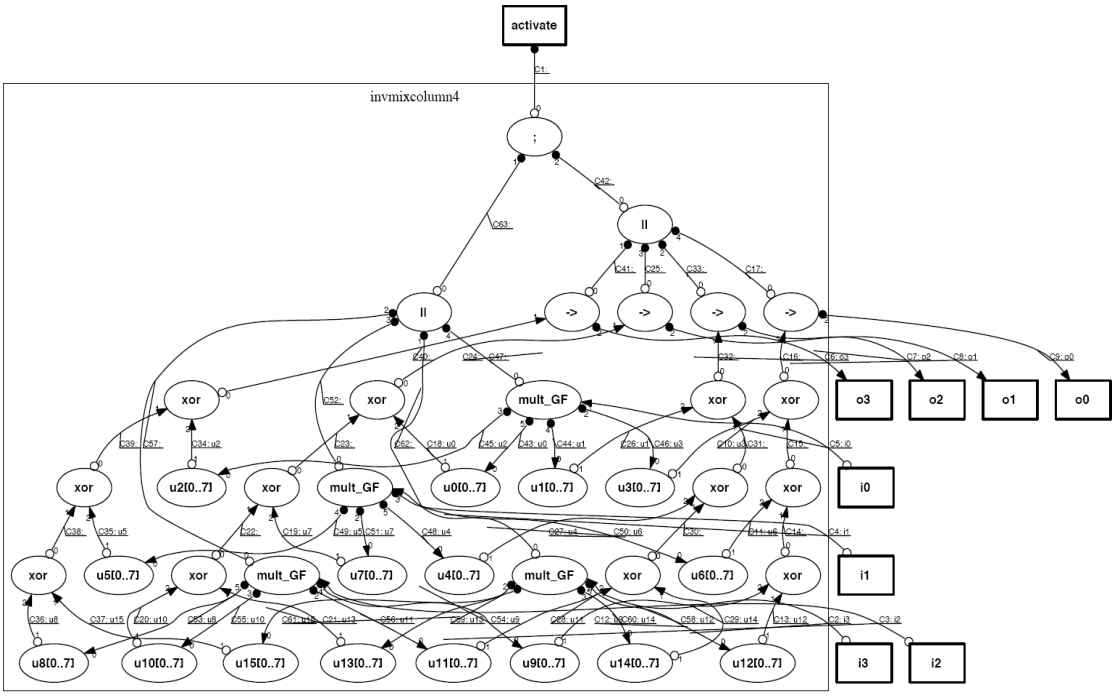


Figura 4.36: *Handshake* de transformación *InvMixColumn*

El diagrama de circuito para realizar la transformación *InvMixColumn* (mostrado en la figura 4.37) se deduce de (19) teniendo como base las salidas del multiplicador. La figura 4.38 presenta el código de la transformación *InvMixColumn* en Balsa para 16 bytes. La figura 4.40 ilustra el diagrama de *handshake* simplificado de la implementación de la función para la matriz de estado.

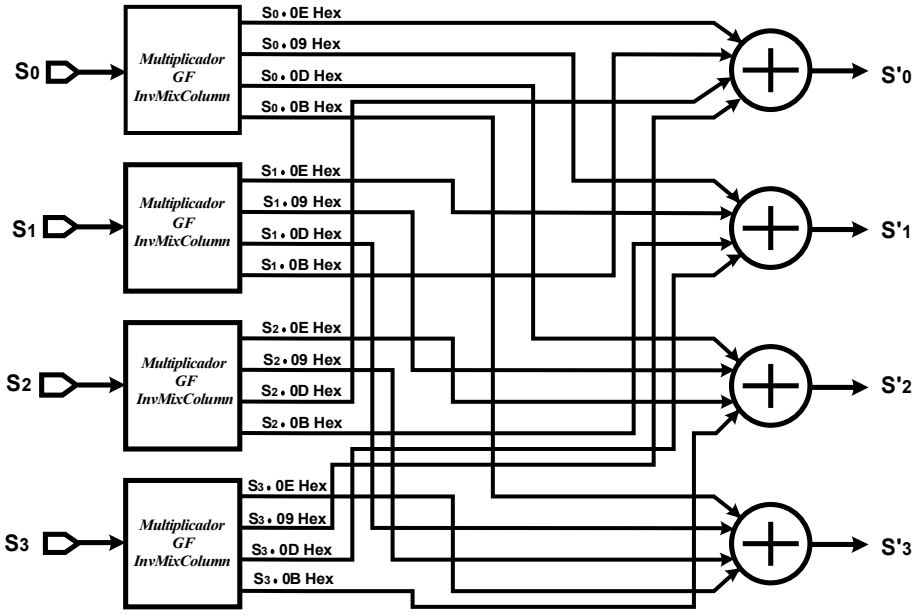


Figura 4.37: Diagrama de circuito para transformación *InvMixColumn* [31]

```
import [balsa.types.basic]
import [InvMixColumn_4]
procedure invmixcolumn16 (input
i15,i14,i13,i12,i11,i10,i9,i8,i7,i6,i5, i4, i3,i2,i1,i0: byte ; output
o15,o14,o13,o12,o11,o10,o9,o8,o7,o6,o5,o4,o3,o2,o1,o0: byte) is
begin
invmixcolumn4 (i15,i14,i13,i12,o15,o14,o13,o12) ||
invmixcolumn4 (i11,i10,i9,i8,o11,o10,o9,o8) ||
invmixcolumn4 (i7,i6,i5,i4,o7,o6,o5,o4) ||
invmixcolumn4 (i3,i2,i1,i0,o3,o2,o1,o0)
end
```

Figura 4.38: Código Balsa para transformación *InvMixColumn* de 16 bytes

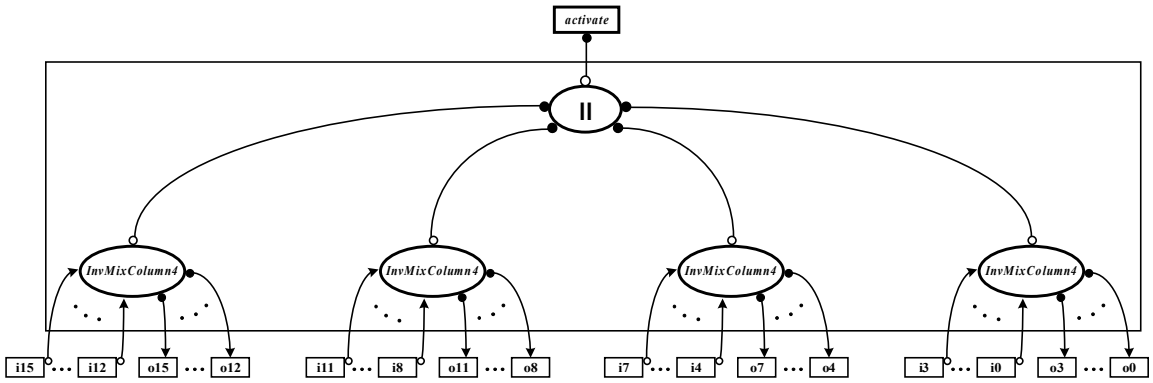


Figura 4.39: *Handshake* de transformación *InvMixColumn* para 16 bytes

4.2.3.3. Transformación de Columna para encriptación/desencriptación (*EDMixColumn*)

Una forma muy simple de implementar un circuito para encriptar/desencriptar consiste en seleccionar el circuito de acuerdo al proceso que se esté llevando a cabo por medio de un multiplexor, como se ilustra en la figura 4.40, sin embargo, el costo en hardware es alto si se utilizan ambas transformaciones por separado. De las ecuaciones (9) y (19) se observa que los coeficientes de *d(x)* son más complejos que los de *c(x),* por lo que la implementación hardware de la transformación *InvMixColumn* es más compleja, más lenta y ocupa mayor área que la transformación *MixColumn*. Afortunadamente, es posible aprovechar la complejidad de *d(x)* para obtener un circuito que realice ambos procesos de forma eficiente, tal como se describe en las siguientes secciones.

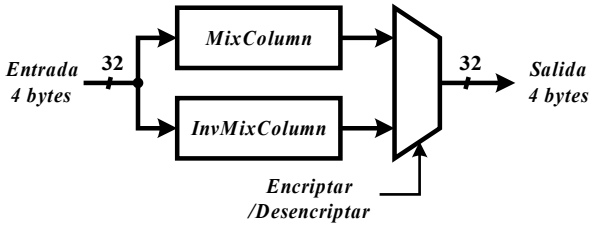


Figura 4.40: Transformación de columna para encriptar/desencriptar sin optimización.

4.2.3.3.1. Descomposición de *InvMixColumn*

Para reducir el costo del hardware, la transformación *InvMixColumn* se puede descomponer en forma serial para que comparta recursos lógicos con *MixColumn* a nivel de palabras de 32 bits. Esta idea se ha planteado en varias publicaciones [29], [32]-[34]. En [32] se analizó, además, una descomposición paralela de *InvMixColumn*. En este trabajo, las transformaciones de columna para encriptación/desencriptación se denominaron *EDMixColumn*.

La descomposición paralela de *InvMixColumn*, propuesta por [35], se basa en el hecho que *d(x)* se puede expresar usando *c(x)* de la siguiente manera:

$$d(x) = c(x) + e(x) \tag{20}$$

Donde *e(x)* es una extensión polinomial definida como

$$e(x) = \{08\}x^3 + \{0c\}x^2 + \{08\}x + \{0c\} \tag{21}$$

La descomposición serial de *InvMixColumn*, se sustenta en el hecho que el inverso *d(x)* está dado por la fórmula

$$d(x) = c^{-1}(x) = c^3(x) \tag{22}$$

Esta ecuación sugiere que la operación *InvMixColumn* se puede realizar repitiendo tres veces *MixColumn*, ya que la ecuación (22) se puede expresar como

$$d(x) = c(x) + f(x) \tag{23}$$

Donde

$$f(x) = c^2(x) = \{04\}x^2 + \{05\} \tag{24}$$

de esta ecuación se deduce que la transformación *InvMixColumn* se puede implementar usando la función *MixColumn* y el polinomio *f(x)*. Al comparar *f(x)* con *c(x),* dada en (6), se puede ver que *f(x)* es más simple por cuanto dos de sus coeficientes son cero.

La ecuación (24) también se obtuvo en [36] de la siguiente forma:

Ya que $$c(x) \bullet d(x) = \{01\} \tag{25}$$

Si se multiplica a ambos lados de (20) por *d(x)* se obtiene:

$$c(x) \bullet d^2(x) = d(x) \tag{26}$$

Donde

$$d(x) = \{04\}x^2 + \{05\} \tag{27}$$

La ecuación (27) se puede implementar en hardware como se muestra en la figura 4.41.

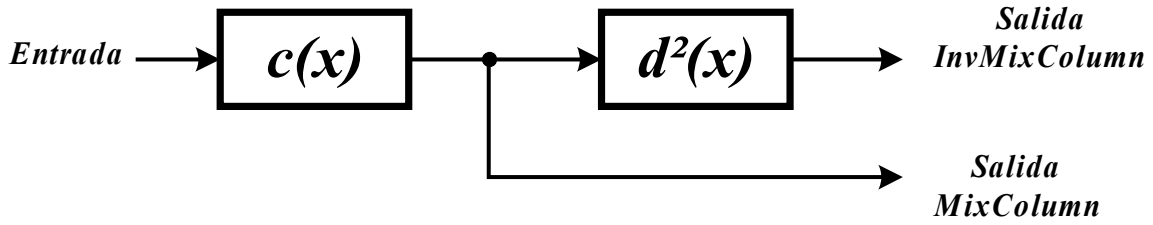


Figura 4.41: Implementación de *MixColumn* e *InvMixColumn* [35]

Tanto la descomposición serial como la paralela de *InvMixColumn* permiten compartir recursos a nivel de byte. En la descomposición serial, *c(x)* y *d(x)* tienen que ser optimizadas en forma separada, mientras que la descomposición paralela permite la optimización de *c(x)* y *d(x)* a la vez.

4.2.3.3.1.1. Descomposición serial de *InvMixColumn*: *EDMixColumn_serie*

Para comprender la descomposición serial de *InvMixColumn* [32], basta con hacer un análisis del byte más significativo de la operación *MixColumn,* el cual, de acuerdo con (9), se puede expresar como:

$$\begin{aligned} b_3 &= \{02\}a_3 + \{03\}a_2 + a_1 + a_0 \\ &= (a_3 + a_2 + a_1 + a_0) + \{02\}(a_3 + a_2) + a_3 \end{aligned} \tag{28}$$

Términos similares se obtienen para los bytes b_2, b_1 y b_0. La figura 4.42 muestra la implementación *EDMixColumn_serie*, en ella se puede observar que el término $(a_0+a_1+a_2+a_3)$ es común a todos los bytes de entrada de la implementación *MixColumn* (encerrada en líneas punteadas).
La implementación de la transformación *InvMixColumn* por medio de la descomposición serial también se puede derivar del uso de la ecuación (19), por ejemplo, el byte b_3 se puede expresar como:

$$b_3 = \{05\}b_3 + \{04\}b_1 = \{04\}(b_3 + b_1) + b_3 \tag{29}$$

En esta expresión, b'_3 y b'_1 son el primer y tercer byte de salida de la función *MixColumn,* por lo que el primer y tercer byte de la función *d(x)* comparten el término $\{04\}(b'_3 + b'_1)$ de la ecuación anterior. Se puede usar la misma aproximación para el segundo y cuarto byte.

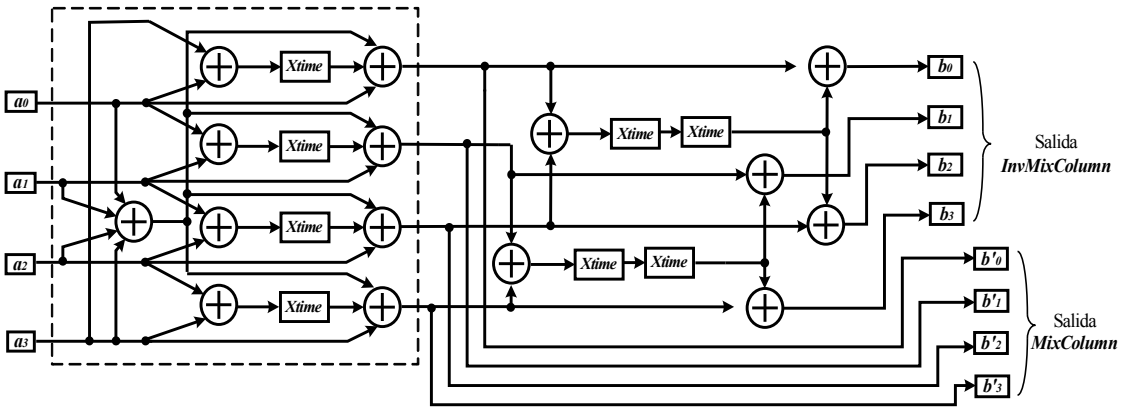


Figura 4.42: Descomposición serial de *InvMixColumn, EDMixColumn_serie* [32].

4.2.3.3.1.2. Descomposición paralela de *InvMixColumn*: *EDMixColumn_paralelo*

En la descomposición paralela de *InvMixColumn,* el polinomio *c(x)* y la extensión polinomial *e(x)* dada en (21), pueden ser optimizadas por separado o en conjunto. El byte b_3 de la función *InvMixColumn* se puede descomponer, según [32], de la siguiente manera:

$$\begin{aligned} b_3 &= \{0e\}a_3 + \{0b\}a_2 + \{0d\}a_1 + \{09\}a_0 \\ &= \{02\}(a_3 + a_2) + a_2 + a_1 + a_0 + \{04\}(\{02\}(a_3 + a_2) + \{02\}(a_1 + a_0) + (a_3 + a_1)) \end{aligned} \quad (30)$$

El término $\{02\}(a_3+a_2)+a_2+a_1+a_0$ representa el byte más significativo de la función *MixColumn*, y el término $\{04\}(\{02\}(a_3+a_2)+\{02\}(a_1+a_0)) + \{04\}(a_3+a_1)$ representa el primer byte de la función de extensión.

Para ilustrar la forma en que la descomposición paralela de la función *InvMixColumn* permite disminuir área, el byte b_3 también se puede expresar como sigue:

$$\begin{aligned} b_3 &= \{0e\}a_3 + \{0b\}a_2 + \{0d\}a_1 + \{09\}a_0 \\ &= (a_3 + a_2 + a_1 + a_0) + \{02\}(a_3 + a_2) + a_3 + \{02\}(\{04\}(a_3 + a_1) + \{04\}(a_2 + a_0)) + \{04\}(a_3 + a_1) \end{aligned} \quad (31)$$

En este caso, el término $(a_3+a_2+a_1+a_0)+\{02\}(a_3+a_2)+a_3$ representa el byte más significativo de la función *MixColumn*, y el término $\{02\}(\{04\}(a_3+a_1)+\{04\}(a_2+a_0))$ $+\{04\}(a_3+a_1)$ representa el byte más significativo de la función de extensión *e(x)*. De esta manera se implementa el circuito para las funciones *MixColumn* e *InvMixColumn* de forma concurrente, el cual se aprecia en la figura 4.43. En este se puede ver la arquitectura paralela y la forma en que los cuatro bytes de salida de la función de extensión comparten el término $\{02\}(\{04\}(a_3+a_1)+ \{04\}(a_2+a_0))$.

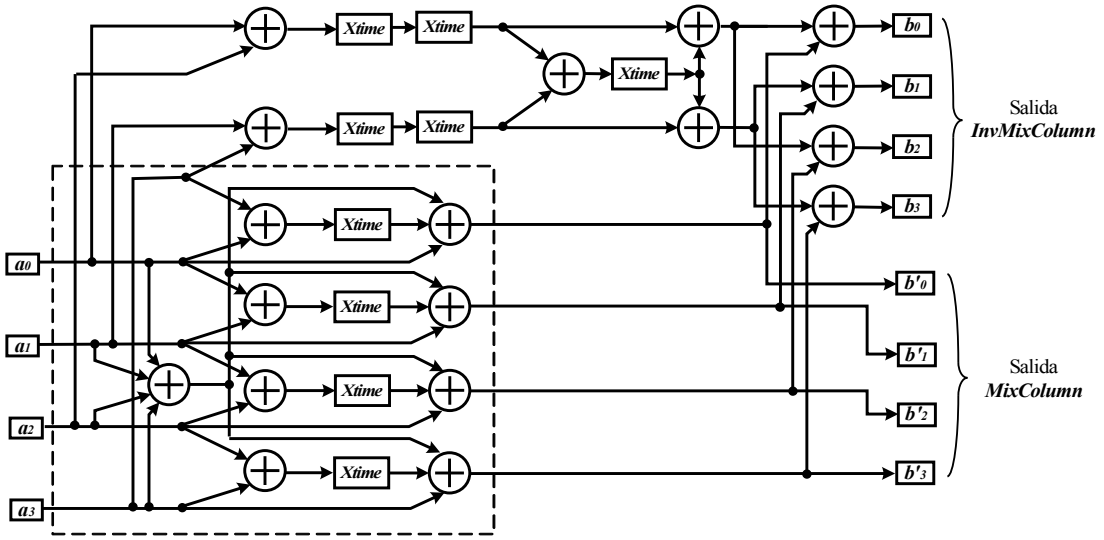


Figura 4.43: Descomposición paralela de *InvMixColumn, EDMixColumn_paralelo* [32].

4.2.3.4. Transformación *EDMixColumn* asíncrona

Con base en lo visto en el subapartado 4.2.3.3, y por considerar que los circuitos descritos anteriormente son eficientes en la utilización de área, al ser implementados con el estilo síncrono, se decidió implementarlos utilizando el estilo asíncrono. La figura 4.45 ilustra la arquitectura general de la transformación *EDMixColumn* asíncrona utilizando protocolos de codificación *dual-rail* ó *1-de-4*.

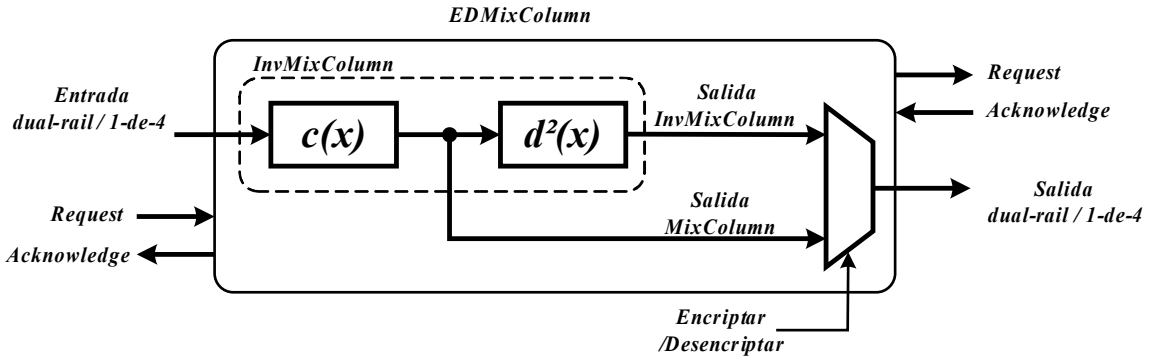


Figura 4.44: Arquitectura para *EDMixColumn* serial/paralela

Las especificaciones asíncronas en Balsa y los diagramas de *handshake* para *EDMixColumn_serie* y *EDMixColumn_paralelo* se ilustran en las figura 4.45 a 4.48.

```
import [balsa.types.basic]
import [Xtime]

procedure previoxtime1_serie(input  a3,a2,a1,a0: byte ; output  b3,b2,b1,b0: byte) is
variable  x3,x2,x1,x0 : byte
begin
        a3 -> x3 ||  a2 -> x2 ||  a1 -> x1 ||  a0 -> x0 ;
        b3 <- x3 xor x2   ||  b2 <- x2 xor x1   ||  b1 <- x1 xor x0   ||  b0 <- x3 xor x0
end

-- Primera parte de Circuito InvMixColumn / Circuito MixColumn
procedure parte1_serie (input  i3,i2,i1,i0: byte ; output  o3,o2,o1,o0: byte) is
variable  x3,x2,x1,x0 : byte
variable  y3,y2,y1,y0 : byte
variable  z : byte
channel   ch3,ch2,ch1,ch0    : byte
begin
        i3 -> x3 ||  i2 -> x2 ||  i1 -> x1 ||  i0 -> x0 ;
        z := x3 xor x2 xor x1 xor x0  ||
        previoxtime1_serie (i3,i2,i1,i0,ch3,ch2,ch1,ch0) ||
        xtime1(ch3,-> y3) || xtime1(ch2,-> y2) ||  xtime1(ch1,-> y1) ||  xtime1(ch0,-> y0) ;
        o3 <- z xor y3 xor x3 ||
        o2 <- z xor y2 xor x2 ||
        o1 <- z xor y1 xor x1 ||
        o0 <- z xor y0 xor x0
end

procedure previoxtime2_serie (input b3,b2,b1,b0: byte ; output c1,c0: byte) is
variable v3,v2,v1,v0: byte
begin
        parte1_serie (b3,b2,b1,b0,->v3,->v2,->v1,->v0);
        c1 <- v3 xor v1 ||  c0 <- v2 xor v0
end

procedure edmixcolumn4 (input i: bit ; input  i3,i2,i1,i0: byte ; output  o3,o2,o1,o0: byte) is
variable  x3,x2,x1,x0 : byte
variable  y3,y2,y1,y0 : byte
variable  z3,z2,z1,z0 : byte
variable  w1,w0 : byte
variable  modo : bit
channel    cha1,cha0    : byte
channel    chc1,chc0    : byte
begin
        parte1_serie (i3,i2,i1,i0,->z3,->z2,->z1,->z0)||
        previoxtime2_serie (i3,i2,i1,i0,chc1,chc0)    ||
        xtime1 (chc1,cha1) || xtime1 (cha1,->w1) ||
        xtime1 (chc0,cha0) || xtime1 (cha0,->w0) ||
        i -> modo ;
        if modo = 0b_0 then
        o3 <- z3 ||  o2 <- z2 ||  o1 <- z1 ||  o0 <- z0
        |  modo = 0b_1 then
        o3 <- z3 xor w1 ||  o2 <- z2 xor w0 ||  o1 <- z1 xor w1 ||  o0 <- z0 xor w0
        end   -- if
end
```

Figura 4.45: Código Balsa para *EDMixColum_serie*

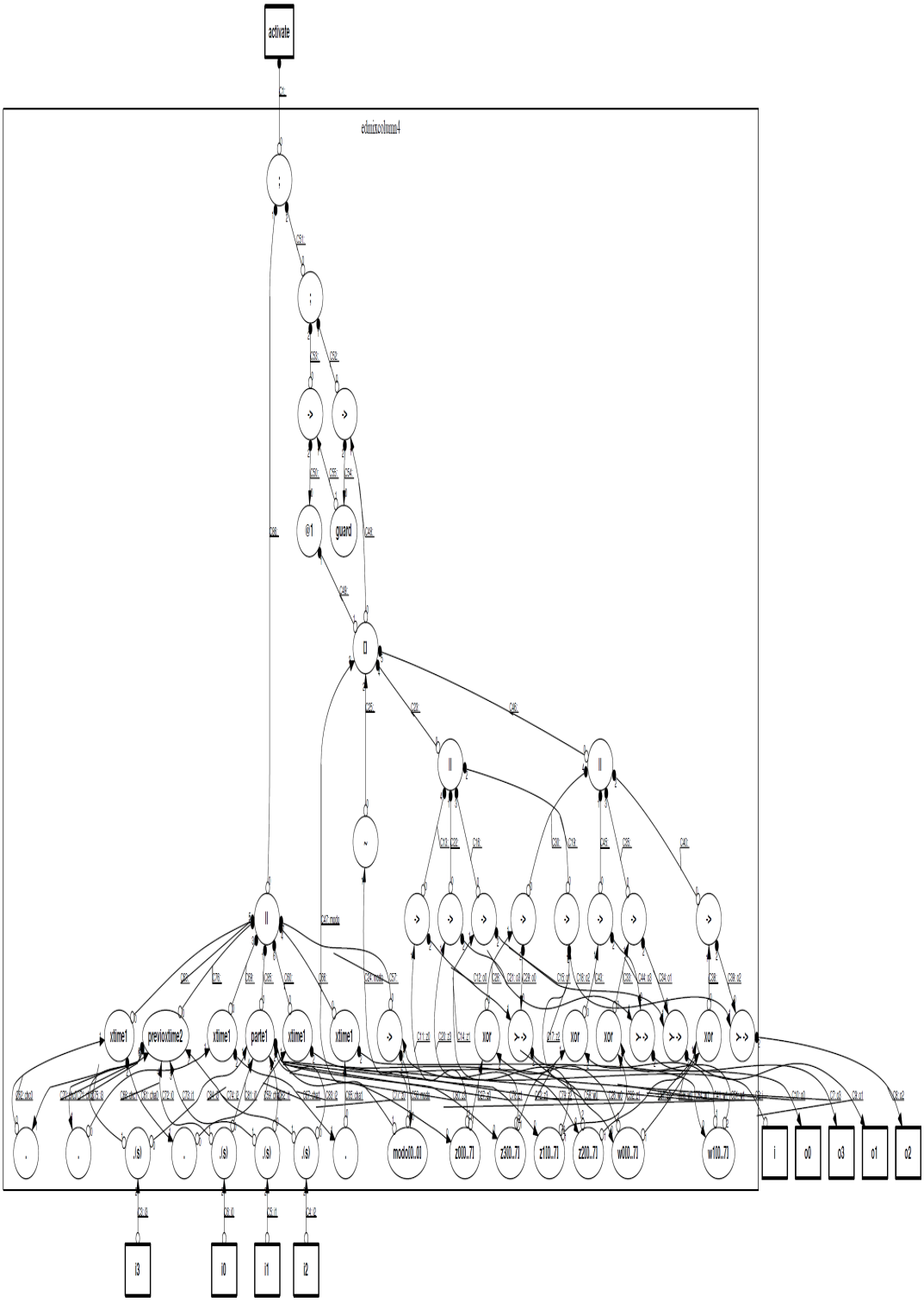


Figura 4.46: *Handshake* de transformación *EDMixColum_serie*

```
import [balsa.types.basic]
import [Xtime]

procedure previoxtime1_pll (input  a3,a2,a1,a0: byte ; output  b5,b4,b3,b2,b1,b0: byte) is
variable  x3,x2,x1,x0 : byte
begin

a3 -> x3 ||  a2 -> x2 ||  a1 -> x1 ||  a0 -> x0 ;
b5 <- x3 xor x1 || b4 <- x2 xor x0 || b3 <- x3 xor x2 || b2 <- x2 xor x1 ||  b1 <- x1 xor x0  ||  b0 <- x3
xor x0
end

procedure parte1_pll (input  i3,i2,i1,i0: byte ; output o6,o5,o4,o3,o2,o1,o0: byte) is
variable  x3,x2,x1,x0 : byte
variable  y5,y4,y3,y2,y1,y0 : byte
variable  z : byte
channel  cho5,cho4,ch5,ch4,ch3,ch2,ch1,ch0: byte
begin
            i3 -> x3 ||  i2 -> x2 ||  i1 -> x1 ||  i0 -> x0 ;
           z := x3 xor x2 xor x1 xor x0  ||
           previoxtime1_pll (i3,i2,i1,i0,ch5,ch4,ch3,ch2,ch1,ch0) ||
           xtime1(ch5,cho5) || xtime1(cho5,-> y5) ||  xtime1(ch4,cho4) || xtime1(cho4,-> y4) ||
           xtime1(ch3,-> y3) || xtime1(ch2,-> y2) ||  xtime1(ch1,-> y1) ||  xtime1(ch0,-> y0) ;

           o6 <- y5 xor y4 ||  o5 <- y5 ||  o4 <- y4 ||
           o3 <- z xor y3 xor x3 ||
           o2 <- z xor y2 xor x2 ||
           o1 <- z xor y1 xor x1 ||
           o0 <- z xor y0 xor x0
end

procedure edmixcolumn4 (input i: bit ; input  i3,i2,i1,i0: byte ; output  o3,o2,o1,o0: byte) is
variable  x3,x2,x1,x0 : byte
variable  y3,y2,y1,y0 : byte
variable  z6,z5,z4,z3,z2,z1,z0 : byte
variable  w1,w0 : byte
variable  modo : bit
channel   ch : byte

begin
           parte1_pll (i3,i2,i1,i0,ch,->z5,->z4,->z3,->z2,->z1,->z0)||
           xtime1 (ch,->z6) ||
           i -> modo ;
           if modo = 0b_0 then
             o3 <- z3   ||  o2 <- z2   ||  o1 <- z1   || o0 <- z0
           |  modo = 0b_1 then
           o3 <- z6 xor z5 xor z3 ||  o2 <- z6 xor z4 xor z2 || o1 <- z6 xor z5 xor z1 ||  o0 <- z6 xor z4
xor z0
           end   -- if
end
```

Figura 4.47: Código Balsa para *EDMixColum_paralelo*

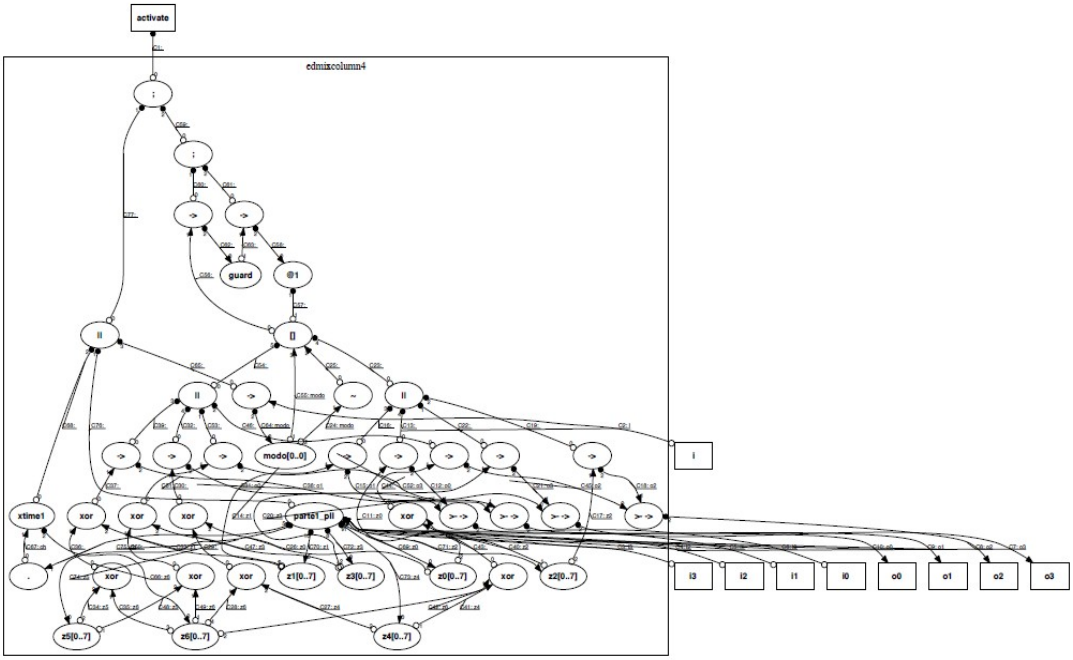


Figura 4.48. *Handshake* de transformación *EDMixColum_paralelo*

Resultados

La tabla 4.7 muestra los resultados obtenidos de las implementaciones asíncronas tanto para las arquitecturas tradicionales *MixColumn* e *InvMixColumn* de 128 bits, como para las implementaciones basadas en las arquitecturas serial y paralela desarrolladas en [32]. Los resultados de la implementación en FPGA usando los protocolos asíncronos de codificación *dual-rail* de dos fases y *1-de-4*, se observan en la tabla 4.8.

Tabla 4.7: Referencia de costo de área y retardo de simulación funcional para circuitos de transformación de columna asíncronos.

Transformación	***Número de XOR***	***Costo Área***	***Retardo (ns)***	***área*retardo***
MixColumn	*28*	*11746.25*	*17000*	16,968,625
InvMixColumn	*76*	*17195.00*	*35000*	601,825,000
EDMixColumn	*104*	*30715.5*	*Encript: 25900* *Desenc: 43900*	795,531,450 1,348,410,450
EDMixColumn_serie	*45*	*23722.75*	*Encript: 48600* *Desenc: 49400*	1,152,925,650 1,171,903,850
EDMixColumn_paralelo	*51*	*25721.5*	*Encript: 43400* *Desenc: 45000*	1,116,313,100 1,157,467,500

La tabla 4.8 muestra que las implementaciones *EDMixColumn_serie* y *EDMixColumn_paralelo*, conllevan un ahorro considerable de puertas XOR (de 2 entradas) respecto de la implementación *EDMixColumn* típica en la que no se comparten recursos hardware. Esto se refleja claramente en el costo de área reportado. Aunque las implementaciones individuales para *MixColumn* e *InvMixColumn* presentan menor

retardo, se puede ver que el costo de área para estas es muy grande, lo que se refleja como un valor muy alto en la columna de figura de mérito *'área*retardo'*.

Tabla 4.8: Resultados de implementación en Virtex XCV2P30-6ff896 de Xilinx

Transformación	***Retardo***	
	Dual-rail	***1-de-4***
MixColumn	***86.701ns*** *Lógica 37.029ns (42.7%)* *Rutas 49.672ns (57.3%)*	***107.992ns*** *Lógica 46.025ns (42.6%)* *Rutas 61.967ns (57.4%)*
InvMixColumn	***176.884ns*** *Lógica 73.650ns (41.6%)* *Rutas 103.233ns (58.4%)*	***217.393ns*** *Lógica 90.472ns (41.6%)* *Rutas 126.921ns (58.4%)*
EDMixColumn	***276.258ns*** *Lógica 114.260ns (41.4%)* *Rutas 161.999ns (58.6%)*	***284.885ns*** *Lógica 118.01ns (41.4%)* *Rutas 166.869ns (58.6%)*
EDMixColumn_serie	***355.077ns*** *Lógica 145.952ns (41.1%)* *Rutas 209.125ns (58.9%)*	***415.565ns*** *Lógica 171.233ns (41.2%)* *Rutas 244.342ns (58.8%)*
EDMixColumn_paralelo	***337.292ns*** *Lógica 138.754ns (41.1%)* *Rutas 139.781ns (58.9%)*	***382.711ns*** *Lógica 157.765ns (41.2%)* *Rutas 224.946ns (58.8%)*

La tabla 4.8 se puede analizar de varias formas. Inicialmente se confirma el hecho que el circuito que implementa la transformación *EDMixColumn_paralelo* presenta un mejor desempeño (menor retardo) que el circuito de *EDMixColumn_serie*. De otro lado, el circuito *EDMixColumn* se evidencia como el de mejor desempeño comparado con las implementaciones de descomposición serial y paralela, esto se debe a que su implementación, a pesar de contener más compuertas lógicas, incluye un número menor de niveles lógicos. El análisis realizado es válido para ambos tipos de codificación.

Los datos sobre las implementaciones registradas en la tabla 4.9, de nuevo evidencian que las rutas de interconexión conllevan el mayor porcentaje de retardo total de cada circuito (alrededor del 58%), lo que indica (como se menciona en el apartado 4.2.3.5) que el sistema ISE y las FPGAs de Xilinx están pensadas principalmente para sistemas síncronos.

4.2.4. Función de adición de clave: Transformación *AddRoundKey*

En la transformación *AddRoundkey*, la clave de ronda se adhiere a la matriz de estado mediante una operación XOR (figura 4.50). Cada clave de ronda consiste de *Nb* palabras provenientes de la unidad de distribución de claves (la cual se describe en la sección 4.3). Según el estándar FIPS-197 [2], cada una de las *Nb* palabras se adicionan en las columnas de la matriz de estado de acuerdo con la siguiente expresión:

$$[S'_{0,c}, S'_{1,c}, S'_{2,c}, S'_{3,c}] = [S_{0,c}, S_{1,c}, S_{2,c}, S_{3,c}] \oplus [w_{rond*Nb+c}] \quad \text{para} \quad 0 \le c < Nb, \qquad (32)$$

Donde [w_i] corresponde a la *i-ésima* palabra de distribución, y *rond* es un valor en el rango $0 \leq rond \leq Nr$.
En el proceso de encriptación, la adición de clave de ronda inicial ocurre cuando *rond*=0, antes de la primera aplicación de la función de ronda. La aplicación de la transformación *AddRoundKey* a las *Nr* rondas de encriptación se presenta cuando $1 \leq rond \leq Nr$.

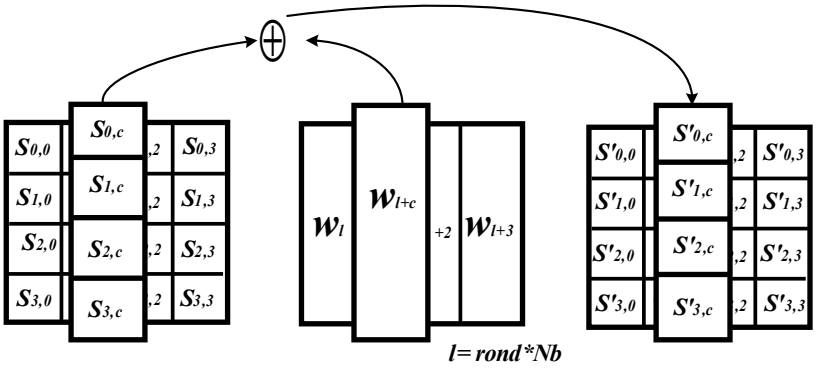


Figura 4.49: transformación *AddRoundKey* [2]

Inversa de transformación *AddRoundKey*

La función *AddRoundKey*, descrita anteriormente, es su propia inversa, ya que solamente involucra la aplicación de la operación XOR.

La tabla 4.9 consigna los resultados de la simulación funcional en Balsa y su implementación en *FPGA* Virtex II-Pro XCV2P30-6ff896 mediante la herramienta *Xilinx ISE 9.2i.*

Tabla 4.9: Resultados de simulación en Balsa e implementación en FPGA de *AddRoundKey*

Transformación AddRoundKey	***Balsa***		***Retardo simulado con Xilinx ISE***	
	Costo Área	***Retardo (ns)***	***Dual-rail***	***1 de 4***
	12377.0	*4700*	***33.290ns*** *Lógica 15.432ns (46.4%)* *Rutas 17.858ns (53.6%)*	***39.850ns*** *Lógica 18.481ns (46.4%)* *Rutas 21.369ns (53.6%)*

4.3. Conclusiones del capítulo

4.3.1. Los resultados de implementación obtenidos en las simulaciones en Balsa muestran que las transformaciones de sustitución de byte, *ByeSub* e *InvByteSub* basadas en tablas presentan mejor desempeño que las transformaciones basadas en la metodología PPRM. Sin embargo, los resultados reportados sobre los retardos de implementación en FPGA medidos con ISE 9.2i de Xilinx muestran que las implementaciones de sustitución de byte basadas en Sbox combinatorias, PPRM, muestran mejor desempeño que las Sbox implementadas con tablas.

4.3.2. Las transformaciones implementadas evidencian una buena integración entre la herramienta de especificación asíncrona Balsa y la plataforma de desarrollo *Xilinx ISE 9.2i*, sin embargo, hace evidente que las FPGAs, utilizadas para implementar el circuito final, presentan restricciones para mapear circuitos asíncronos.

4.3.3. Los reportes de síntesis obtenidos con el sistema *Xilinx ISE 9.2i*, en los que se observa que los circuitos implementados en FPGA registran mayor retardo en las rutas de interconexión (superiores al 58%) que en los elementos lógicos que conforman cada función, indican que el estado actual de las FPGAs de Xilinx y su ambiente de diseño no han sido orientados a la implementación eficiente de circuitos asíncronos.

4.3.4. Una limitante en la implementación asíncrona de las funciones del algoritmo AES, y en general del diseño asíncrono, es que aún no existen sistemas de desarrollo o circuitos reprogramables comerciales orientados al estilo de diseño asíncrono. La mayoría de las herramientas comerciales de hoy (incluido el sistema *ISE 9.2i de Xilinx*) están diseñadas para lograr implementaciones concebidas únicamente para sistemas síncronos. Por esta razón, el desempeño de los circuitos asíncronos, al ser implementados dentro de plataformas síncronas como las FPGAs, está condicionado por los elementos que las conforman, en especial por los recursos de interconexión y las rutas de los relojes globales, además, por los *Bloques Lógicos Configurables* (*CLBs*) y los *Bloques de Entrada/Salida* (*IOBs*) cuyos elementos de almacenamiento básicos son *flip-flops*.

Referencias

[1] **NIST**, ***National Institute of Standards and Technology***. (2000). Disponible en: http://csrc.nist.gov/CryptoToolkit/aes/rijndael/

[2] **FIPS**, **Federal Information Processing Standards.** "S*pecification for the ADVANCED ENCRYPTION STANDARD (AES)"*. Publication 197, November 26. 2001a. Disponible en: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

[3] **Daemen, J., Rijmen, V.** *"AES Proposal: Rijndael"*. Proton Worl Int. I, Zweefvliegtuigstraat, Brussel, Belgium; Katholieke Universiteit Leuven, ESAT-COSIC, Heverlee, Belgium. Document V. 2. 1999.

[4] **Segredo, A., Zabala, E., Bellora, G.** *"Diseño de un Procesador Criptográfico Rijndael en FPGA"*. Facultad de Ingeniería, Bernard Wand-Polak. Ingeniería en Telecomunicaciones. Universidad ORT. Uruguay, 2003.

[5] **Muñoz, A.** *"Seguridad Europea para EEUU, Algoritmo de Rijndael"*. Madrid, España. 2004. Disponible en:

[6] **Angel, J.J.** *"AES–Advanced Encryption Standard"*. P-p 78-81, 2005. Disponible en: http://anubis.server01.org/storage/docs/cyberpunk/AES_v2005_jjaa.pdf
§
[7] **Lu, Ch., Tseng, Sh.** *"Integrated Design of AES (Advanced Encryption Standard) Encrypter and Decrypter"*. Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures, and Processors (ASAP'02), 2002.

[8] **Morioka, S., Satoh, A.** *"An Optimized S-Box Circuit Architecture for Low Power AES Design"*. Cryptographic Hardware and Embedded Systems (CHES 2002), LNCS 2523, pp.172-186, Springer-Verlag Berlin Heidelberg, 2003.

[9] **Bryant, R.E**. *"Graph-Based Algorithms for Boolean Function Manipulation*". *IEEE Trans. on Computers*, Vol. C-35, No. 8, pp. 677–691, 1986.

[10] **Morioka, S., Satoh A.** *"A 10-Gbps Full-AES Crypto Design With a Twisted BDD S-Box Architecture"*. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol 12, No 7, July 2004.

[11] **Guajardo, M.** "*Arithmetic Architectures for Finite Fields GF(p^m) with Cryptographic Applications"*.Dissertation, Doktor-Ingenieurs der Fakult¨at f¨ur Elektrotechnik und Informations technikan der Ruhr-Universit¨at Bochum von aus Caracas, Venezuela. Bochum, 2004.

[12] **Morioka, S., Katayama Y.**. *"O(log2m) Iterative Algorithm for Multiplicative Inverse in GF(2m)"*. IEEE Intl. Symp. On Info. Theory (ISIT2000), pp. 449 ff., 2000.

[13] Itoh, T., Tsujii, S. *"A Fast Algorithm for Computing Multiplicative Inverses in GF(2m) using Normal Bases". Information and Computation*, Vol.78, No. 3, pp. 171–177, 1988.

[14] Rodriguez F., Morales G., Saquib N., Cruz N. *"Parallel Itoh-Tsujii Multiplicative Inversion Algorithm for Special Class of Trinomials".* Disponible en: http://www.delta.cs.cinvestav.mx/~francisco/arith/irv03.pdf

[15] Drescher, W., Bachmann, K., Fettweis, G. *"VLSI Architecture for non-Sequential Inversion over GF(2^m) using the Euclidean Algorithm".* Dresden University of Technology.

[16] Guajardo J. & C. Paar. *"Efficient Algorithms for Elliptic Curve Cryptosystems". CRYPTO'97*, LNCS Vol. 1294, pp. 342–356, 1997.

[17] Rudra A., Dubey P.K., Jutla C.S., Kumar V., Rao J.R., Rohatgi P. *"Efficient Rijndael encryption implementation with composite field arithmetic". Proc.* Cryptographic Hardware and Embedded Systems (CHES 2001), LNCS Vol. 2162, pp. 175–188, Paris, France 2001.

[18] Satoh A., Morioka S., Takano K., & Munetoh S. *"A Compact Rijndael Hardware Architecture with S-Box Optimization".* Advances in Cryptology – *ASIACRYPT 2001*, LNCS Vol. 2248, pp. 239–254, 2001.

[19] Jarvinen K.U., Tommiska M.T., Skytt¨a J.O. "*A fully pipelined memoryless 17.8 Gbps AES-128 encryptor*". International Symposium on Field-Programmable Gate Arrays (FPGA 2003), Monterey, CA, 2003.

[20] Verbauwhede I., Schaumont P., Kuo H. *"Design and performance testing of a 2.29-*
GB/s rijndael processor". IEEE Journal of Solid-State Circuits, Volume: 38 Issue:3, March 2003

[21] Lin T.F., Su C.P., Huang C.T., Wu C.W. *"A high-throughput low-cost AES cipher chip".* IEEE Asia-Pacific Conference on ASIC, 2002

[22] Wolkerstorfer J., Oswald E., Lamberger M. *"An ASIC Implementation of the AES Sboxes".* The Cryptographer's Track at the RSA Conference, San Jose, CA, 2002

[23] Satoh A., Morioka S., Takano K., Munetoh S. *"A Compact Rijndael Hardware Architecture with S-Box Optimization".* Theory and Application of Cryptology and Information Security (ASIACRYPT 2001), Gold Coast, Australia, 2001

[24] Rijmen, V. *¨Efficient Implementation of the Rijndael S-box¨.* Katholieke Universiteit Leuven, Dept. ESAT, Heverlee, Belgium, 2000. Disponible en: http://www.iaik.tu-graz.ac.at/research/krypto/AES/old/~rijmen/rijndael/Sbox.pdf

[25] **D. Edwards, L. Bardsley, L. Janin, W. Toms.** "Balsa: a Tutorial Guide". Version 3.5.The Advanced Processor Technologies Group, APT Group. Manchester University. Manchester, England. 2006. Disponible en: http://intranet.cs.man.ac.uk/apt

[26] http://www.iesalfaro.com/gh/Dicciona/Diccionario de Arte y Diseño -P.htm

[27] www.camaraalcoy.net/Servicios_web/glosario/Glosario/P.htm

[28] **Bardsley, A.** *"Implementing Balsa handshake Circuits"*, Ph.D. dissertation, directed by D. Edwards, faculty of Science & Engineering, University of Manchester, 2000. Disponible en:
ftp://ftp.cs.man.ac.uk/pub/amulet/theses/bardsley_phd.pdf

[29] **Huang, Y.H., Lin, Y.S., Hung, K.Y., Lin, K.C.** "Efficient Implementation of AES IP". IEEE, Asia Pacific, Conference on Circuits and Systems APCCAS-2006, pp 1418-1421. Dec 2006.

[30] **Li, H., Friggstad, Z.** *"And Efficient Architecture for the AES MixColumns Operation"*. IEEE International Symposium on Circuits and Systems, 2005, ISCAS. Vol. 5, pp. 4637-4640. May 2005.

[31] **Cordero, V., Silva, C.** *"Design of an ASIC CORE for data encryption and decryption Using the NIST Advanced Encryption standard"*. Microelectronics Group – Pontificia Universidad Católica del Perú. IX Workshop de Iberchip, La Habana, Cuba. 2003.

[32] **Fischer, V., Drutarovský, M., Chodowiec, P., Gramain, F.** *"InvMixColumn Descomposition and Multilevel Resource Sharing in AES Implementations". IEEE Transactions* on Very Large Scale Integration (VLSI) Systems, Vol. 13, No 8, August 2005.

[33] **Hsiao, S.F., Chen, M.C., Tu, C.H.** "Memory-Free Low-Cost Designs of Advanced Encryption Standard Using Common Subexpression Elimination for Subfunctions in Transformations". IEEE Trans. on circuits and systems-I: Regular papers, vol. 53, No. 3, pp. 615-626, March 2006.

[34] **Lu, C.C., Tseng, S.Y.** "Integrated Designs of AES (Advanced Encryption Standard) Encrypter and Decrypter". IEEE Proceedings on Application-Specific Systems, Architectures, and Processors (ASAP'02), Jul 2002.

[35] **Wolkerstorfer, J.** "An ASIC implementation of the AES MixColumn operation". Proceedings Austrochip 2001, pp. 129-132. Vienna, Austria, Oct. 12, 2001,

[36] **Chodowiec, P., Gaj, K.** *"Very Compact FPGA Implementation of the AES Algorithm".* CHES 2003, LCNS 2779, pp. 319-333, *Springer-Verlag* Berlin Heidelberg 2003.

Anexo 4.A:
Función Sbox especificada en Tabla

```
-- Descripción en Balsa 3.5.1
-- Función de Sustitución de byte
-- Tabla Sbox,  ByteSubTabla

import [balsa.types.basic]

procedure bytesub1(input  ent : byte; output out : byte) is

variable u : byte

begin
        ent -> u;
                if  u = 0x00 then out <- 0x63   |  u = 0x01 then out <- 0x7c   |  u = 0x02 then out <- 0x77
                |  u = 0x03 then out <- 0x7b   |  u = 0x04 then out <- 0xf2   |  u = 0x05 then out <- 0x6b
                |  u = 0x06 then out <- 0x6f   |  u = 0x07 then out <- 0xc5   |  u = 0x08 then out <- 0x30
                |  u = 0x09 then out <- 0x01   |  u = 0x0a then out <- 0x67   |  u = 0x0b then out <- 0x2b
                |  u = 0x0c then out <- 0xfe   |  u = 0x0d then out <- 0xd7   |  u = 0x0e then out <- 0xab
                |  u = 0x0f then out <- 0x76   |  u = 0x10 then out <- 0xca   |  u = 0x11 then out <- 0x82
                |  u = 0x12 then out <- 0xc9   |  u = 0x13 then out <- 0x7d   |  u = 0x14 then out <- 0xfa
                |  u = 0x15 then out <- 0x59   |  u = 0x16 then out <- 0x47   |  u = 0x17 then out <- 0xf0
                |  u = 0x18 then out <- 0xad   |  u = 0x19 then out <- 0xd4   |  u = 0x1a then out <- 0xa2
                |  u = 0x1b then out <- 0xaf   |  u = 0x1c then out <- 0x9c   |  u = 0x1d then out <- 0xa4
                |  u = 0x1e then out <- 0x72   |  u = 0x1f then out <- 0xc0   |  u = 0x20 then out <- 0xb7
                |  u = 0x21 then out <- 0xfd   |  u = 0x22 then out <- 0x93   |  u = 0x23 then out <- 0x26
                |  u = 0x24 then out <- 0x36   |  u = 0x25 then out <- 0x3f   |  u = 0x26 then out <- 0xf7
                |  u = 0x27 then out <- 0xcc   |  u = 0x28 then out <- 0x34   |  u = 0x29 then out <- 0xa5
                |  u = 0x2a then out <- 0xe5   |  u = 0x2b then out <- 0xf1   |  u = 0x2c then out <- 0x71
                |  u = 0x2d then out <- 0xd8   |  u = 0x2e then out <- 0x31   |  u = 0x2f then out <- 0x15
                |  u = 0x30 then out <- 0x04   |  u = 0x31 then out <- 0xc7   |  u = 0x32 then out <- 0x23
                |  u = 0x33 then out <- 0xc3   |  u = 0x34 then out <- 0x18   |  u = 0x35 then out <- 0x96
                |  u = 0x36 then out <- 0x05   |  u = 0x37 then out <- 0x9a   |  u = 0x38 then out <- 0x07
                |  u = 0x39 then out <- 0x12   |  u = 0x3a then out <- 0x80   |  u = 0x3b then out <- 0xe2
                |  u = 0x3c then out <- 0xeb   |  u = 0x3d then out <- 0x27   |  u = 0x3e then out <- 0xb2
                |  u = 0x3f then out <- 0x75   |  u = 0x40 then out <- 0x09   |  u = 0x41 then out <- 0x83
                |  u = 0x42 then out <- 0x2c   |  u = 0x43 then out <- 0x1a   |  u = 0x44 then out <- 0x1b
                |  u = 0x45 then out <- 0x6e   |  u = 0x46 then out <- 0x5a   |  u = 0x47 then out <- 0xa0
                |  u = 0x48 then out <- 0x52   |  u = 0x49 then out <- 0x3b   |  u = 0x4a then out <- 0xd6
                |  u = 0x4b then out <- 0xb3   |  u = 0x4c then out <- 0x29   |  u = 0x4d then out <- 0xe3
                |  u = 0x4e then out <- 0x2f   |  u = 0x4f then out <- 0x84   |  u = 0x50 then out <- 0x53
                |  u = 0x51 then out <- 0xd1   |  u = 0x52 then out <- 0x00   |  u = 0x53 then out <- 0xed
                |  u = 0x54 then out <- 0x20   |  u = 0x55 then out <- 0xfc   |  u = 0x56 then out <- 0xb1
                |  u = 0x57 then out <- 0x5b   |  u = 0x58 then out <- 0x6a   |  u = 0x59 then out <- 0xcb
                |  u = 0x5a then out <- 0xbe   |  u = 0x5b then out <- 0x39   |  u = 0x5c then out <- 0x4a
                |  u = 0x5d then out <- 0x4c   |  u = 0x5e then out <- 0x58   |  u = 0x5f then out <- 0xcf
                |  u = 0x60 then out <- 0xd0   |  u = 0x61 then out <- 0xef   |  u = 0x62 then out <- 0xaa
                |  u = 0x63 then out <- 0xfb   |  u = 0x64 then out <- 0x43   |  u = 0x65 then out <- 0x4d
                |  u = 0x66 then out <- 0x33   |  u = 0x67 then out <- 0x85   |  u = 0x68 then out <- 0x45
                |  u = 0x69 then out <- 0xf9   |  u = 0x6a then out <- 0x02   |  u = 0x6b then out <- 0x7f
                |  u = 0x6c then out <- 0x50   |  u = 0x6d then out <- 0x3c   |  u = 0x6e then out <- 0x9f
                |  u = 0x6f then out <- 0xa8   |  u = 0x70 then out <- 0x51   |  u = 0x71 then out <- 0xa3
                |  u = 0x72 then out <- 0x40   |  u = 0x73 then out <- 0x8f   |  u = 0x74 then out <- 0x92
                |  u = 0x75 then out <- 0x9d   |  u = 0x76 then out <- 0x38   |  u = 0x77 then out <- 0xf5
                |  u = 0x78 then out <- 0xbc   |  u = 0x79 then out <- 0xb6   |  u = 0x7a then out <- 0xda
                |  u = 0x7b then out <- 0x21   |  u = 0x7c then out <- 0x10   |  u = 0x7d then out <- 0xff
                |  u = 0x7e then out <- 0xf3   |  u = 0x7f then out <- 0xd2   |  u = 0x80 then out <- 0xcd
                |  u = 0x81 then out <- 0x0c   |  u = 0x82 then out <- 0x13   |  u = 0x83 then out <- 0xec
                |  u = 0x84 then out <- 0x5f   |  u = 0x85 then out <- 0x97   |  u = 0x86 then out <- 0x44
                |  u = 0x87 then out <- 0x17   |  u = 0x88 then out <- 0xc4   |  u = 0x89 then out <- 0xa7
                |  u = 0x8a then out <- 0x7e   |  u = 0x8b then out <- 0x3d   |  u = 0x8c then out <- 0x64
                |  u = 0x8d then out <- 0x5d   |  u = 0x8e then out <- 0x19   |  u = 0x8f then out <- 0x73
                |  u = 0x90 then out <- 0x60   |  u = 0x91 then out <- 0x81   |  u = 0x92 then out <- 0x4f
                |  u = 0x93 then out <- 0xdc   |  u = 0x94 then out <- 0x22   |  u = 0x95 then out <- 0x2a
                |  u = 0x96 then out <- 0x90   |  u = 0x97 then out <- 0x88   |  u = 0x98 then out <- 0x46
                |  u = 0x99 then out <- 0xee   |  u = 0x9a then out <- 0xb8   |  u = 0x9b then out <- 0x14
```

```
                | u = 0x9c then out <- 0xde     | u = 0x9d then out <- 0x5e     | u = 0x9e then out <- 0x0b
                | u = 0x9f then out <- 0xdb     | u = 0xa0 then out <- 0xe0     | u = 0xa1 then out <- 0x32
                | u = 0xa2 then out <- 0x3a     | u = 0xa3 then out <- 0x0a     | u = 0xa4 then out <- 0x49
                | u = 0xa5 then out <- 0x06     | u = 0xa6 then out <- 0x24     | u = 0xa7 then out <- 0x5c
                | u = 0xa8 then out <- 0xc2     | u = 0xa9 then out <- 0xd3     | u = 0xaa then out <- 0xac
                | u = 0xab then out <- 0x62     | u = 0xac then out <- 0x91     | u = 0xad then out <- 0x95
                | u = 0xae then out <- 0xe4     | u = 0xaf then out <- 0x79     | u = 0xb0 then out <- 0xe7
                | u = 0xb1 then out <- 0xc8     | u = 0xb2 then out <- 0x37     | u = 0xb3 then out <- 0x6d
                | u = 0xb4 then out <- 0x8d     | u = 0xb5 then out <- 0xd5     | u = 0xb6 then out <- 0x4e
                | u = 0xb7 then out <- 0xa9     | u = 0xb8 then out <- 0x6c     | u = 0xb9 then out <- 0x56
                | u = 0xba then out <- 0xf4     | u = 0xbb then out <- 0xea     | u = 0xbc then out <- 0x65
                | u = 0xbd then out <- 0x7a     | u = 0xbe then out <- 0xae     | u = 0xbf then out <- 0x08
                | u = 0xc0 then out <- 0xba     | u = 0xc1 then out <- 0x78     | u = 0xc2 then out <- 0x25
                | u = 0xc3 then out <- 0x2e     | u = 0xc4 then out <- 0x1c     | u = 0xc5 then out <- 0xa6
                | u = 0xc6 then out <- 0xb4     | u = 0xc7 then out <- 0xc6     | u = 0xc8 then out <- 0xe8
                | u = 0xc9 then out <- 0xdd     | u = 0xca then out <- 0x74     | u = 0xcb then out <- 0x1f
                | u = 0xcc then out <- 0x4b     | u = 0xcd then out <- 0xbd     | u = 0xce then out <- 0x8b
                | u = 0xcf then out <- 0x8a     | u = 0xd0 then out <- 0x70     | u = 0xd1 then out <- 0x3e
                | u = 0xd2 then out <- 0xb5     | u = 0xd3 then out <- 0x66     | u = 0xd4 then out <- 0x48
                | u = 0xd5 then out <- 0x03     | u = 0xd6 then out <- 0xf6     | u = 0xd7 then out <- 0x0e
                | u = 0xd8 then out <- 0x61     | u = 0xd9 then out <- 0x35     | u = 0xda then out <- 0x57
                | u = 0xdb then out <- 0xb9     | u = 0xdc then out <- 0x86     | u = 0xdd then out <- 0xc1
                | u = 0xde then out <- 0x1d     | u = 0xdf then out <- 0x9e     | u = 0xe0 then out <- 0xe1
                | u = 0xe1 then out <- 0xf8     | u = 0xe2 then out <- 0x98     | u = 0xe3 then out <- 0x11
                | u = 0xe4 then out <- 0x69     | u = 0xe5 then out <- 0xd9     | u = 0xe6 then out <- 0x8e
                | u = 0xe7 then out <- 0x94     | u = 0xe8 then out <- 0x9b     | u = 0xe9 then out <- 0x1e
                | u = 0xea then out <- 0x87     | u = 0xeb then out <- 0xe9     | u = 0xec then out <- 0xce
                | u = 0xed then out <- 0x55     | u = 0xee then out <- 0x28     | u = 0xef then out <- 0xdf
                | u = 0xf0 then out <- 0x8c     | u = 0xf1 then out <- 0xa1     | u = 0xf2 then out <- 0x89
                | u = 0xf3 then out <- 0x0d     | u = 0xf4 then out <- 0xbf     | u = 0xf5 then out <- 0xe6
                | u = 0xf6 then out <- 0x42     | u = 0xf7 then out <- 0x68     | u = 0xf8 then out <- 0x41
                | u = 0xf9 then out <- 0x99     | u = 0xfa then out <- 0x2d     | u = 0xfb then out <- 0x0f
                | u = 0xfc then out <- 0xb0     | u = 0xfd then out <- 0x54     | u = 0xfe then out <- 0xbb
                    | u = 0xff then out <- 0x16
                    end
end
```

Anexo 4.B:
Función Sbox Inversa especificada en Tabla

```
-- Descripción en Balsa 3.5.1
-- Función de Sustitución de byte Inversa
-- Tabla Sbox Inversa InvByteSubTabla

import [balsa.types.basic]

procedure invbytesub1(input  ent : byte; output out : byte) is

variable u : byte

begin
        ent -> u;
        if  u = 0x00 then out <- 0x52 |  u = 0x01 then out <- 0x09 |  u = 0x02 then out <- 0x6a
        |  u = 0x03 then out <- 0xd5 |  u = 0x04 then out <- 0x30 |  u = 0x05 then out <- 0x36
        |  u = 0x06 then out <- 0xa5 |  u = 0x07 then out <- 0x38 |  u = 0x08 then out <- 0xbf
        |  u = 0x09 then out <- 0x40 |  u = 0x0a then out <- 0xa3 |  u = 0x0b then out <- 0x9e
        |  u = 0x0c then out <- 0x81 |  u = 0x0d then out <- 0xf3 |  u = 0x0e then out <- 0xd7
        |  u = 0x0f then out <- 0xfb |  u = 0x10 then out <- 0x7c |  u = 0x11 then out <- 0xe3
        |  u = 0x12 then out <- 0x39 |  u = 0x13 then out <- 0x82 |  u = 0x14 then out <- 0x9b
        |  u = 0x15 then out <- 0x2f |  u = 0x16 then out <- 0xff |  u = 0x17 then out <- 0x87
        |  u = 0x18 then out <- 0x34 |  u = 0x19 then out <- 0x8e |  u = 0x1a then out <- 0x43
        |  u = 0x1b then out <- 0x44 |  u = 0x1c then out <- 0xc4 |  u = 0x1d then out <- 0xde
        |  u = 0x1e then out <- 0xe9 |  u = 0x1f then out <- 0xcb |  u = 0x20 then out <- 0x54
        |  u = 0x21 then out <- 0x7b |  u = 0x22 then out <- 0x94 |  u = 0x23 then out <- 0x32
        |  u = 0x24 then out <- 0xa6 |  u = 0x25 then out <- 0x62 |  u = 0x26 then out <- 0x23
        |  u = 0x27 then out <- 0x3d |  u = 0x28 then out <- 0xee |  u = 0x29 then out <- 0x4c
        |  u = 0x2a then out <- 0x95 |  u = 0x2b then out <- 0x0b |  u = 0x2c then out <- 0x42
        |  u = 0x2d then out <- 0xfa |  u = 0x2e then out <- 0xc3 |  u = 0x2f then out <- 0x4e
        |  u = 0x30 then out <- 0x08 |  u = 0x31 then out <- 0x2e |  u = 0x32 then out <- 0xa1
        |  u = 0x33 then out <- 0x66 |  u = 0x34 then out <- 0x28 |  u = 0x35 then out <- 0xd9
        |  u = 0x36 then out <- 0x24 |  u = 0x37 then out <- 0xb2 |  u = 0x38 then out <- 0x76
        |  u = 0x39 then out <- 0x5b |  u = 0x3a then out <- 0xa2 |  u = 0x3b then out <- 0x49
        |  u = 0x3c then out <- 0x6d |  u = 0x3d then out <- 0x8b |  u = 0x3e then out <- 0xd1
        |  u = 0x3f then out <- 0x25 |  u = 0x40 then out <- 0x72 |  u = 0x41 then out <- 0xf8
        |  u = 0x42 then out <- 0xf6 |  u = 0x43 then out <- 0x64 |  u = 0x44 then out <- 0x86
        |  u = 0x45 then out <- 0x68 |  u = 0x46 then out <- 0x98 |  u = 0x47 then out <- 0x16
        |  u = 0x48 then out <- 0xd4 |  u = 0x49 then out <- 0xa4 |  u = 0x4a then out <- 0x5c
        |  u = 0x4b then out <- 0xcc |  u = 0x4c then out <- 0x5d |  u = 0x4d then out <- 0x65
        |  u = 0x4e then out <- 0xb6 |  u = 0x4f then out <- 0x92 |  u = 0x50 then out <- 0x6c
        |  u = 0x51 then out <- 0x70 |  u = 0x52 then out <- 0x48 |  u = 0x53 then out <- 0x50
        |  u = 0x54 then out <- 0xfd |  u = 0x55 then out <- 0xed |  u = 0x56 then out <- 0xb9
        |  u = 0x57 then out <- 0xda |  u = 0x58 then out <- 0x5e |  u = 0x59 then out <- 0x15
        |  u = 0x5a then out <- 0x46 |  u = 0x5b then out <- 0x57 |  u = 0x5c then out <- 0xa7
        |  u = 0x5d then out <- 0x8d |  u = 0x5e then out <- 0x9d |  u = 0x5f then out <- 0x84
        |  u = 0x60 then out <- 0x90 |  u = 0x61 then out <- 0xd8 |  u = 0x62 then out <- 0xab
        |  u = 0x63 then out <- 0x00 |  u = 0x64 then out <- 0x8c |  u = 0x65 then out <- 0xbc
        |  u = 0x66 then out <- 0xd3 |  u = 0x67 then out <- 0x0a |  u = 0x68 then out <- 0xf7
        |  u = 0x69 then out <- 0xe4 |  u = 0x6a then out <- 0x58 |  u = 0x6b then out <- 0x05
        |  u = 0x6c then out <- 0xb8 |  u = 0x6d then out <- 0xb3 |  u = 0x6e then out <- 0x45
        |  u = 0x6f then out <- 0x06 |  u = 0x70 then out <- 0xd0 |  u = 0x71 then out <- 0x2c
        |  u = 0x72 then out <- 0x1e |  u = 0x73 then out <- 0x8f |  u = 0x74 then out <- 0xca
        |  u = 0x75 then out <- 0x3f |  u = 0x76 then out <- 0x0f |  u = 0x77 then out <- 0x02
        |  u = 0x78 then out <- 0xc1 |  u = 0x79 then out <- 0xaf |  u = 0x7a then out <- 0xbd
        |  u = 0x7b then out <- 0x03 |  u = 0x7c then out <- 0x01 |  u = 0x7d then out <- 0x13
        |  u = 0x7e then out <- 0x8a |  u = 0x7f then out <- 0x6b |  u = 0x80 then out <- 0x3a
        |  u = 0x81 then out <- 0x91 |  u = 0x82 then out <- 0x11 |  u = 0x83 then out <- 0x41
```

```
        | u = 0x84 then out <- 0x4f  | u = 0x85 then out <- 0x67  | u = 0x86 then out <- 0xdc
        | u = 0x87 then out <- 0xea  | u = 0x88 then out <- 0x97  | u = 0x89 then out <- 0xf2
        | u = 0x8a then out <- 0xcf  | u = 0x8b then out <- 0xce  | u = 0x8c then out <- 0xf0
        | u = 0x8d then out <- 0xb4  | u = 0x8e then out <- 0xe6  | u = 0x8f then out <- 0x73
        | u = 0x90 then out <- 0x96  | u = 0x91 then out <- 0xac  | u = 0x92 then out <- 0x74
        | u = 0x93 then out <- 0x22  | u = 0x94 then out <- 0xe7  | u = 0x95 then out <- 0xad
        | u = 0x96 then out <- 0x35  | u = 0x97 then out <- 0x85  | u = 0x98 then out <- 0xe2
        | u = 0x99 then out <- 0xf9  | u = 0x9a then out <- 0x37  | u = 0x9b then out <- 0xe8
        | u = 0x9c then out <- 0x1c  | u = 0x9d then out <- 0x75  | u = 0x9e then out <- 0xdf
        | u = 0x9f then out <- 0x6e  | u = 0xa0 then out <- 0x47  | u = 0xa1 then out <- 0xf1
        | u = 0xa2 then out <- 0x1a  | u = 0xa3 then out <- 0x71  | u = 0xa4 then out <- 0x1d
        | u = 0xa5 then out <- 0x29  | u = 0xa6 then out <- 0xc5  | u = 0xa7 then out <- 0x89
        | u = 0xa8 then out <- 0x6f  | u = 0xa9 then out <- 0xb7  | u = 0xaa then out <- 0x62
        | u = 0xab then out <- 0x0e  | u = 0xac then out <- 0xaa  | u = 0xad then out <- 0x18
        | u = 0xae then out <- 0xbe  | u = 0xaf then out <- 0x1b  | u = 0xb0 then out <- 0xfc
        | u = 0xb1 then out <- 0x56  | u = 0xb2 then out <- 0x3e  | u = 0xb3 then out <- 0x4b
        | u = 0xb4 then out <- 0xc6  | u = 0xb5 then out <- 0xd2  | u = 0xb6 then out <- 0x79
        | u = 0xb7 then out <- 0x20  | u = 0xb8 then out <- 0x9a  | u = 0xb9 then out <- 0xdb
        | u = 0xba then out <- 0xc0  | u = 0xbb then out <- 0xfe  | u = 0xbc then out <- 0x78
        | u = 0xbd then out <- 0xcd  | u = 0xbe then out <- 0x5a  | u = 0xbf then out <- 0xf4
        | u = 0xc0 then out <- 0x1f  | u = 0xc1 then out <- 0xdd  | u = 0xc2 then out <- 0xa8
        | u = 0xc3 then out <- 0x33  | u = 0xc4 then out <- 0x88  | u = 0xc5 then out <- 0x07
        | u = 0xc6 then out <- 0xc7  | u = 0xc7 then out <- 0x31  | u = 0xc8 then out <- 0xb1
        | u = 0xc9 then out <- 0x12  | u = 0xca then out <- 0x10  | u = 0xcb then out <- 0x59
        | u = 0xcc then out <- 0x27  | u = 0xcd then out <- 0x80  | u = 0xce then out <- 0xec
        | u = 0xcf then out <- 0x5f  | u = 0xd0 then out <- 0x60  | u = 0xd1 then out <- 0x51
        | u = 0xd2 then out <- 0x7f  | u = 0xd3 then out <- 0xa9  | u = 0xd4 then out <- 0x19
        | u = 0xd5 then out <- 0xb5  | u = 0xd6 then out <- 0x4a  | u = 0xd7 then out <- 0x0d
        | u = 0xd8 then out <- 0x2d  | u = 0xd9 then out <- 0xe5  | u = 0xda then out <- 0x7a
        | u = 0xdb then out <- 0x9f  | u = 0xdc then out <- 0x93  | u = 0xdd then out <- 0xc9
        | u = 0xde then out <- 0x9c  | u = 0xdf then out <- 0xef  | u = 0xe0 then out <- 0xa0
        | u = 0xe1 then out <- 0xe0  | u = 0xe2 then out <- 0x3b  | u = 0xe3 then out <- 0x4d
        | u = 0xe4 then out <- 0xae  | u = 0xe5 then out <- 0x2a  | u = 0xe6 then out <- 0xf5
        | u = 0xe7 then out <- 0xb0  | u = 0xe8 then out <- 0xc8  | u = 0xe9 then out <- 0xeb
        | u = 0xea then out <- 0xbb  | u = 0xeb then out <- 0x3e  | u = 0xec then out <- 0x83
        | u = 0xed then out <- 0x53  | u = 0xee then out <- 0x99  | u = 0xef then out <- 0x61
        | u = 0xf0 then out <- 0x17  | u = 0xf1 then out <- 0x2b  | u = 0xf2 then out <- 0x04
        | u = 0xf3 then out <- 0x7e  | u = 0xf4 then out <- 0xba  | u = 0xf5 then out <- 0x77
        | u = 0xf6 then out <- 0xd6  | u = 0xf7 then out <- 0x26  | u = 0xf8 then out <- 0xe1
        | u = 0xf9 then out <- 0x69  | u = 0xfa then out <- 0x14  | u = 0xfb then out <- 0x63
        | u = 0xfc then out <- 0x55  | u = 0xfd then out <- 0x21  | u = 0xfe then out <- 0x0c
        | u = 0xff then out <- 0x7d
      end
end
```

Capítulo 5: Implementación asíncrona del algoritmo de Rijndael

Introducción

En este capítulo se describe la arquitectura utilizada en la implementación asíncrona del estándar de encriptación avanzado (AES-128: *Advanced Encryption Standard* – clave de 128 bits) o algoritmo de Rijndael. Se muestran los resultados de simulación obtenidos mediante la herramienta Balsa [1], [2] para los procesos de encriptación y desencriptación con base en las especificaciones del estándar de la *Federal Information Processing Standards- Publication 197* (*FIPS-197*) [3] del *NIST* (*National Institute of Standards and Technology*) de Estados Unidos [4].

En la literatura solo se han reportado dos implementaciones asíncronas del estándar AES para encriptación basados en *ASICs*: en [5] se reporta una arquitectura QDI balanceada con tecnología de 0.13µm, 1.2V, codificación *1-de-N* y un retardo de 850*ns*. En [6], [7] se reporta una arquitectura *pipeline* con codificación *dual-rail* con tecnología CMOS de 0.35µm y un retardo de 800*ns*. Estos reportes no mencionan resultados de rendimientos de bloques individuales del algoritmo. La literatura tampoco reporta resultados de implementaciones AES asíncronas sobre *FPGAs*.

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 claves*. Ambas unidades interactúan mediante la unidad de adición de claves (circuito *AddRoundKey*) en cada ronda de ejecución del algoritmo. En las siguientes secciones se describen estas dos partes. Se presentan las descripciones asíncronas en el sistema Balsa y los resultados de simulación funcional obtenidos para la unidad de claves y la unidad de encriptación/desencriptación.

5.1. Unidad de Claves

Los circuitos de la unidad de claves incluidos en esta sección están basados en la descripción realizada por los autores del algoritmo de Rijndael [8]. El algoritmo AES toma la clave de encriptación, *K*, y realiza una rutina de expansión para generar una clave expandida. La rutina de expansión genera un total de $Nb*(Nr + 1)$ palabras: el algoritmo requiere un conjunto inicial de *Nb* palabras, y cada una de las *Nr* rondas requiere *Nb* palabras de datos de clave. La clave expandida resultante consta de un arreglo lineal de cuatro bytes, denotadas por [*wi*], con *i* en el rango de $0 \leq i < Nb*(Nr+1)$. 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:

$$SubclaveRonda_i = f(SubclaveRonda_{i-1}) \quad \text{para todo } 0<i<11 \qquad (1)$$

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:

$$K_0 = (w_3, w_2, w_1, w_0) \quad (2)$$

Las primeras *Nk* 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 *Nk* posiciones mas cercana, *w*[*i-Nk*]. Para las palabras que están en posiciones múltiplo de *Nk*, 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 Sbox (*SubWord*) a cada byte.

La rutina de expansión para claves de encriptación de 256 bits (*Nk*=8) es ligeramente diferente a las rutinas para claves de 128 y 192 bits. Si *Nk*=8 e *i-4* es un múltiplo de *Nk*, entonces la función *SubWord* se aplica a *w*[i-1] antes de la XOR.

La figura 5.1 ilustra detalles de la unidad de expansión de clave del algoritmo AES-128 para encriptación (*SubclaveRonda_enc*).

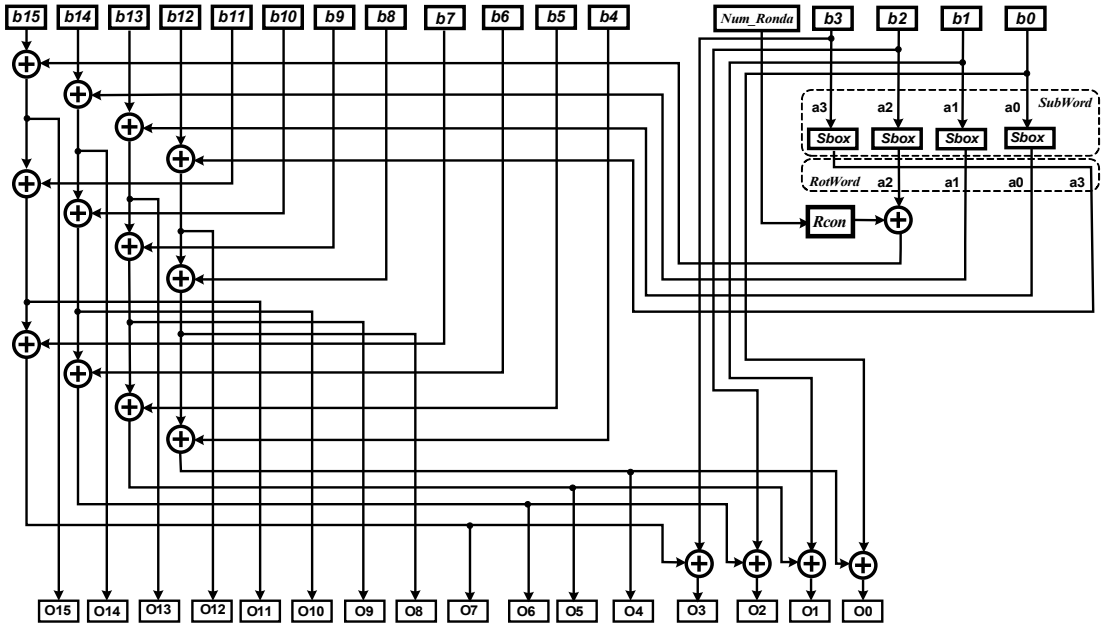


Figura 5.1: Unidad de generación de subclaves de encriptación (*SubclaveRonda_enc*).

La figura 5.1 muestra que la columna menos significativa de la matriz de subclave (columna 4) se transforma de manera distinta a las otras columnas de la clave. A la columna se le aplica, mediante Sbox, la transformación *SubWord* (equivalente a una sustitución de byte para cada elemento de la columna). La columna resultante se modifica mediante una rotación circular de una posición, de tal manera que el byte más significativo de la columna pasa a ser el menos significativo. La siguiente función es *RCon* (adición de constante de ronda) y consiste en realizar una operación XOR entre el

vector columna {*Kr*,0,0,0} y la columna resultante. El valor del byte *Kr* (*constante de ronda*) depende del número de ronda que se este llevando a cabo. Las transformaciones de esta columna son similares para los procesos de encriptación y desencriptación.

La figura 5.2 ilustra el código en Balsa para la implementación asíncrona del circuito correspondiente a la unidad de expansión de clave para encriptación. El diagrama de *handshake* que representa el funcionamiento del circuito se excluye debido a su tamaño.

```
-- Generación de Subclave para ronda de encriptación de Algoritmo de Rijndael (AES-128)
import [balsa.types.basic]
import [SubWord]

procedure subclaveronda (input nr : nibble; input i15,i14,i13,i12,i11,i10,i9,i8,i7,i6,i5,i4,i3,i2,i1,i0 : byte; output nextnr : nibble; output
o15,o14,o13,o12,o11,o10,o9,o8,o7,o6,o5,o4,o3,o2,o1,o0 : byte) is
variable num_ronda : nibble
variable a3,a2,a1,a0,u3,x3,x2,x1,x0,y3,y2,y1,y0,z3,z2,z1,z0 :  byte
variable kr,w15,w14,w13,w12,w11,w10,w9,w8,w7,w6,w5,w4,w3,w2,w1,w0 : byte

begin       nr -> num_ronda ||
            i15 -> w15 || i14 -> w14 || i13 -> w13 || i12 -> w12 || i11 -> w11 || i10 -> w10 || i9  -> w9  || i8  -> w8  ||
            i7  -> w7  || i6  -> w6  || i5  -> w5  || i4  -> w4  || i3  -> w3  || i2  -> w2  || i1  -> w1  || i0  -> w0  ||

            subword1 (i3,i2,i1,i0,->a3,->a2,->a1,->a0) ;  -- Sustitución de byte para columna 4 de matriz de clave

-- Generación de Constante de Ronda (Rcon) para la Columna 4
    if num_ronda = 0 then kr := 0x01
            | num_ronda = 1 then kr := 0x02
            | num_ronda = 2 then kr := 0x04
            | num_ronda = 3 then kr := 0x08
            | num_ronda = 4 then kr := 0x10
            | num_ronda = 5 then kr := 0x20
            | num_ronda = 6 then kr := 0x40
            | num_ronda = 7 then kr := 0x80
            | num_ronda = 8 then kr := 0x1b
            | num_ronda = 9 then kr := 0x36
   end ;
            nextnr <-  (num_ronda + 1 as nibble) ;
-- Rotación de byte (RotWord) y adicion de constante de Ronda
            u3  := a2 xor kr  ;

            x3 :=  u3 xor w15 ||
            x2 :=  w14 ||
            x1 :=  w13 ||
            x0 :=  w12  ;
            y3 := x3 xor w11 ||
            y2 := x2 xor w10 ||
            y1 := x1 xor w9  ||
            y0 := x0 xor w8   ;
            z3 := y3 xor w7  ||
            z2 := y2 xor w6  ||
            z1 := y1 xor w5  ||
            z0 := y0 xor w4   ;

-- Matriz de Estado de salida

            o15 <-  x3 || o14 <-  x2 || o13 <-  x1 || o12 <-  x0 ||
            o11 <-  y3 || o10 <-  y2 || o9  <-  y1 || o8  <-  y0 ||
            o7  <-  z3 || o6  <-  z2 || o5  <-  z1 || o4  <-  z0 ||
            o3  <-  z3 xor w3 || o2  <-  z2 xor w2 || o1  <-  z1 xor w1 || o0  <-  z0 xor w0
end
```

Figura 5.2: Código Balsa para *SubclaveRonda_enc*

Las claves de ronda usadas en el proceso de desencriptación son iguales a las generadas

en el proceso de encriptación, pero se usan en orden inverso; es así como la última clave generada en la encriptación es la clave inicial utilizada en la desencriptación. El circuito de distribución de clave para el proceso de desencriptación (*SubclaveRonda_dec*) se observa en la figura 5.3, mientras que la especificación asíncrona en Balsa se observa en la figura 5.4.

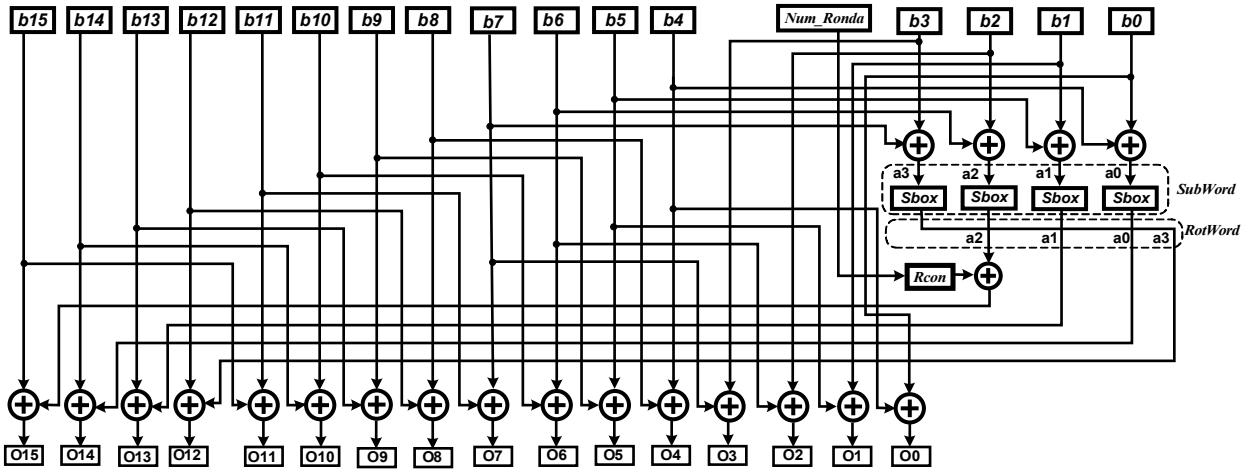


Figura 5.3: Unidad de generación de subclaves de desencriptación (*SubclaveRonda_dec*).

En el circuito de la figura 5.3 se resaltan dos aspectos: el primero tiene que ver con el hecho que la secuencia de las funciones *SubWord* y *RotWord* puede ser conmutada, ya que la función *Rotword* no implica elementos de circuito y se puede implementar de manera sencilla tan solo mediante la conmutación de las conexiones etiquetadas como *a3, a2, a1, y a0*. El segundo aspecto está relacionado con la implementación de la función de adición de constante de ronda, la cual consiste en realizar una operación XOR entre un valor (definido previamente en el bloque *Rcon*) el cual depende del número de ronda (*Num_Ronda*) que se esté llevando a cabo y el valor resultante de la operación *RotWord*. Ya que el vector de constante de ronda es {*Kr*,0,0,0} la adición de constante de ronda tan solo requiere realizar la operación XOR entre *Kr* y el byte más significativo (*a2*) de la función *RotWord.*

Los circuitos de las figuras 5.1 y 5.3 son similares en lo referente al circuito que transforma la columna menos significativa de la clave. También se puede ver que en ambos circuitos las salidas *O15 a O11* se forman por una operación XOR entre las entradas *b15 a b11* y las salidas *a1, a0, a3* (de *RotWord)* y la salida resultante de la adición entre *Rcon* y *a2*. Tales similitudes se pueden aprovechar para construir un único circuito para encriptación/desencriptación (*EDSubclaveRonda*) y obtener así un ahorro significativo de hardware, tal como se muestra en la figura 5.5.

```
-- Generación de Subclave para ronda de desencriptación de Algoritmo de Rijndael (AES-128)

import [balsa.types.basic]
import [SubWord]
import [XorSubClaveInv]

procedure subclaverondainv (input nr : nibble; input  i15,i14,i13,i12,i11,i10,i9,i8,i7,i6,i5,i4,i3,i2,i1,i0 : byte; output nextnr : nibble; output
o15,o14,o13,o12,o11,o10,o9,o8,o7,o6,o5,o4,o3,o2,o1,o0 : byte) is

channel cha,chb,chc,chd : byte
variable num_ronda : nibble
variable a3,a2,a1,a0,u3 : byte
variable kr,w15,w14,w13,w12,w11,w10,w9,w8,w7,w6,w5,w4,w3,w2,w1,w0 : byte

begin
        nr -> num_ronda ||
        i15 -> w15 || i14 -> w14 || i13 -> w13 || i12 -> w12 || i11 -> w11 || i10 -> w10 || i9  -> w9  || i8  -> w8  ||
        i7  -> w7  || i6  -> w6  || i5  -> w5  || i4  -> w4  || i3  -> w3  || i2  -> w2  || i1  -> w1  || i0  -> w0  ||

-- Sustitución de byte para columna 4 de matriz de clave

xorsubclaveinv1 (i7,i6,i5,i4,i3,i2,i1,i0,cha,chb,chc,chd) ||
subword1 (cha,chb,chc,chd,->a3,->a2,->a1,->a0) ;

-- Constante de Ronda (Rcon) para la Columna 4

    if num_ronda = 0 then kr := 0x36
        | num_ronda = 1 then kr := 0x1b
        | num_ronda = 2 then kr := 0x80
        | num_ronda = 3 then kr := 0x40
        | num_ronda = 4 then kr := 0x20
        | num_ronda = 5 then kr := 0x10
        | num_ronda = 6 then kr := 0x08
        | num_ronda = 7 then kr := 0x04
        | num_ronda = 8 then kr := 0x02
        | num_ronda = 9 then kr := 0x01
        end ;

        nextnr <-  (num_ronda + 1 as nibble) ;

-- Rotación de byte (RotWord) y adición de constante de Ronda
        u3  := a2 xor kr   ;

        o15 <- u3 xor w15 ||
        o14 <-  w14 ||
        o13 <-  w13 ||
        o12 <-  w12 ||

        o11 <- w15 xor w11 ||
        o10 <- w14 xor w10 ||
        o9  <- w13 xor w9  ||
        o8  <- w12 xor w8  ||

        o7  <- w11 xor w7  ||
        o6  <- w10 xor w6  ||
        o5  <- w9  xor w5  ||
        o4  <- w8  xor w4  ||

        o3  <- w7  xor w3  ||
        o2  <- w6  xor w2  ||
        o1  <- w5  xor w1  ||
        o0  <- w4  xor w0
end
```

Figura 5.4: Código Balsa para *SubclaveRonda_dec*.

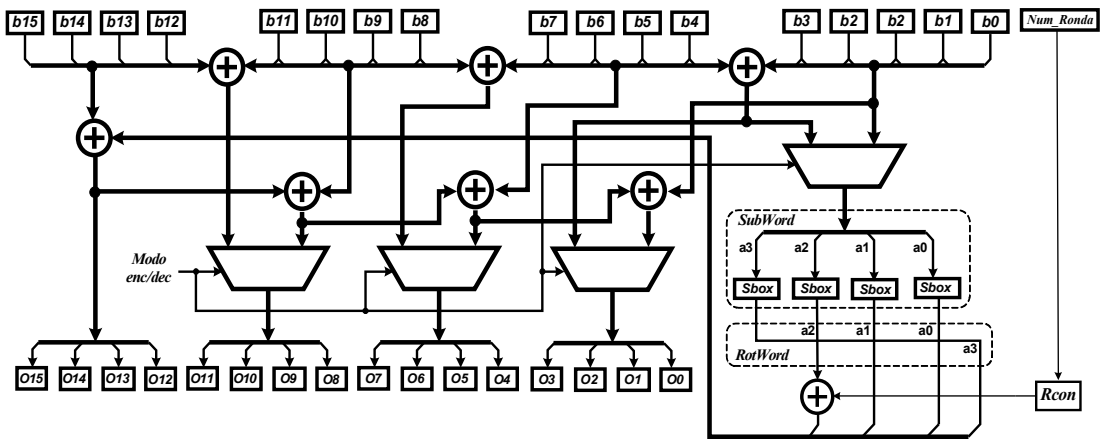


Figura 5.5: Circuito de generación de subclaves de encriptación/desencriptación para AES-128.

El circuito representado en la figura 5.5 se puede utilizar para implementar la unidad de distribución de claves para encriptación o desencriptación de 16 bytes; la entrada de *Modo enc/dec* de los multiplexores permite seleccionar el proceso deseado. Se puede observar que el circuito para las funciones *SubWord*, *RotWord* y adición de constante de ronda, es el mismo para encriptación y desencriptación al igual que las salidas *015* a *012*. El bloque *Rcon* recibe como entrada el número de ronda almacenado en la variable *Num_Ronda* y entrega como salida el valor de la constante de ronda, *Kr,* según el proceso sea de encriptación o desencriptación.

La tabla 5.1 contiene los resultados de las simulaciones de los circuitos usando Balsa. Cada circuito se implementó de dos maneras distintas: la primera consistió en utilizar la versión combinatoria PPRM de las Sbox para conformar la transformación *SubWord*; la segunda forma utilizó tablas Sbox directas.

Tabla 5.1: Costo de área y retardo para circuitos asíncronos de distribución de clave.

Transformación	***Costo Área***	***Retardo (ns)***
SubclaveRonda_enc ***(con Sbox PPRM)***	*14396.5*	*85300*
SubclaveRonda_dec ***(con Sbox PPRM)***	*14554.5*	*80100*
EDSubclaveRonda ***(con Sbox PPRM)***	*32141.25*	*Encriptación: 97300* *Desencriptación: 92100*
SubclaveRonda_enc ***(con Tabla Sbox)***	*15316.0*	*32400*
SubclaveRonda_dec ***(con Tabla Sbox)***	*15434.0*	*28200*
EDSubclaveRonda ***(con Tabla Sbox)***	*25428.5*	*Encriptación: 47400* *Desencriptación: 44900*

Los reportes de esta tabla muestran claramente que los circuitos de Subclave de ronda que utilizan tablas Sbox para la implementar la función SubWord presentan desempeños superiores al 50%, que los circuitos que utilizan la metodología combinatoria PPRM.

La tabla 5.2 contiene los reportes de tiempos de retardo de los circuitos sintetizados para FPGA. Sólo se obtuvieron datos para la codificación *dual-rail* ya que el sistema Balsa no genera archivos *Verilog* sintetizables con codificación *1-de-4* que contengan definiciones de tablas (como es el caso de *SubWord* implementado con tablas Sbox).

Tabla 5.2: Resultados de implementación en Virtex XCV2P30-6ff896 de Xilinx

Transformación	***Retardo***
	Dual-rail
SubclaveRonda_enc ***(con Sbox PPRM)***	***371.783ns*** *Lógica 153.151ns (41.2%)* *Rutas 218.632ns (58.8%)*
SubclaveRonda_dec ***(con Sbox PPRM)***	***352.564ns*** *Lógica 145.327ns (41.2%)* *Rutas 207.237ns (58.8%)*
SubclaveRonda_enc ***(con Tabla Sbox)***	***405.597ns*** *Lógica 166.610ns (41.1%)* *Rutas 238.988ns (58.9%)*

5.2. Algoritmo de Rijndael: proceso de encriptación

El proceso de encriptación del algoritmo se describe en la figura 5.6. Las funciones de transformación (descritas en el capítulo 4) se han agrupado para facilitar su integración y descripción en Balsa. Inicialmente la unidad *AddRoundkey* recibe el bloque de entrada y la clave inicial (ambos de 16 bytes), al mismo tiempo la clave inicial se envía a la unidad *Subclave_Ronda* para comenzar a generar las diferentes subclaves utilizadas en *Ronda_Normal* y en *Ronda_Final*. Las transformaciones agrupadas en *Ronda_Normal* y *Subclave_Ronda* se ejecutan de manera iterativa e interactúan mediante la unidad *AddRoundKey*. La transformación *MixColumn* no se ejecuta en la *Ronda_Final* del proceso de encriptación.

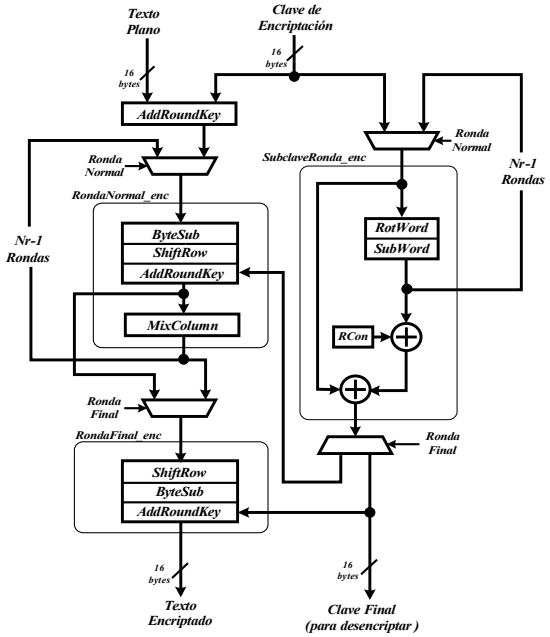


Figura 5.6: Diagrama de bloques del algoritmo de Rijndael para encriptación

5.2.1. Bloque *RondaNormal_enc*

Este bloque, el cual agrupa las cuatro funciones de transformación del algoritmo, se ejecuta *Nr-1* veces de forma iterativa. Para su descripción en Balsa no es necesario incluir el bloque *ShiftRow* ya que éste se puede implementar mediante la conmutación de canales entre los bloques *ByteSub* y *MixColumn*. Esto es posible ya que se puede cambiar el orden de la secuencia de ejecución de las funciones de transformación debido a dos propiedades del algoritmo AES [3]:

1. Las transformaciones *ByteSub* y *ShiftRow* conmutan; es decir, una transformación *ByteSub* seguida por una transformación *ShiftRow* es equivalente a una transformación *ShiftRow* seguida por una transformación *ByteSub*. Esta propiedad también se aplica a las funciones para desencriptación *InvByteSub* e *InvShiftRow*.
2. Las funciones de transformación de columna, *MixColumn* e *InvMixColumn*, son lineales con respecto a la entrada de columna, lo cual significa que

   MixColumn(matriz de estado XOR Clave de Ronda)
   = MixColumn(matriz de estado) XOR MixColumn(Clave de Ronda).

Estas dos propiedades permiten invertir el orden de secuencia de las transformaciones *ByteSub* y *ShiftRow*. También se puede invertir el orden de las transformaciones *AddRoundKey* y *MixColumn* siempre que las columnas de la clave de encriptación sean modificadas usando la transformación *MixColumn*.

En la figura 5.7 se observa el código en lenguaje Balsa para las funciones agrupadas en *Ronda_Normal* mediante canales de comunicación. La función *AddRoundKey*, aunque hace parte de esta ronda, no se muestra debido a que se implementa fuera de la función por conveniencia en la tarea de programación con Balsa.

```
import [balsa.types.basic]
import [ByteSub16]
-- import [ShiftRow]
import [MixColumn]

procedure rondanormal (input w15,w14,w13,w12,w11,w10,w9,w8,w7,w6,w5,w4,w3,w2,w1,w0 :
byte; output s15,s14,s13,s12,s11,s10,s9,s8,s7,s6,s5,s4,s3,s2,s1,s0 : byte) is

channel ca15,ca14,ca13,ca12,ca11,ca10,ca9,ca8,ca7,ca6,ca5,ca4,ca3,ca2,ca1,ca0 : byte

begin
        bytesub16(w15,w14,w13,w12,w11,w10,w9,w8,w7,w6,w5,w4,w3,w2,w1,w0,ca15,ca14,ca13,
        ca12,ca11,ca10,ca9,ca8,ca7,ca6,ca5,ca4,ca3,ca2,ca1,ca0) ||

--ShiftRow se implementa aqui reorganizando el orden de los canales de entrada a MixColumn

        mixcolumn1(ca15,ca10,ca5,ca0,ca11,ca6,ca1,ca12,ca7,ca2,ca13,ca8,ca3,ca14,ca9,ca4,s15,s14
        ,s13,s12, s11,s10,s9,s8,s7,s6,s5,s4,s3,s2,s1,s0)
end
```

Figura 5.7: Código en Balsa para *RondaNormal_enc*

5.2.2. Bloque *RondaFinal_enc*

Este bloque, cuya codificación en Balsa se aprecia en la figura 5.8, agrupa las funciones *ShiftRow*, *ByteSub* y *AddRoundKey*. Gracias a las propiedades de AES, descritas en la sección anterior, se puede invertir la secuencia de ejecución de las transformaciones. La transformación *AddRoundKey*, también en este caso, se especifica fuera de la de esta ronda por reglas propias de Balsa.

```
import [balsa.types.basic]
import [ShiftRow]
import [ByteSub16]

procedure  rondafinal (input  w15,w14,w13,w12,w11,w10,w9,w8,w7,w6,w5,w4,w3,w2,w1,w0 : byte; output
s15,s14,s13,s12,s11,s10,s9,s8,s7,s6,s5,s4,s3,s2,s1,s0 : byte) is

channel ca15,ca14,ca13,ca12,ca11,ca10,ca9,ca8,ca7,ca6,ca5,ca4,ca3,ca2,ca1,ca0 : byte

begin
        bytesub16(w15,w14,w13,w12,w11,w10,w9,w8,w7,w6,w5,w4,w3,w2,w1,w0,ca15,ca14,ca13,ca12,
        ca11,ca10,ca9,ca8,ca7, ca6,ca5,ca4,ca3,ca2,ca1,ca0) ||

        shiftrow1(ca15,ca14,ca13,ca12,ca11,ca10,ca9,ca8,ca7,ca6,ca5,ca4,ca3,ca2,ca1,ca0,s15,s14,s13,s12,
        s11,s10,s9,s8,s7,s6,s5,s4,s3,s2,s1,s0)
end
```

Figura 5.8: Código en Balsa para *RondaFinal_enc*

5.2.3. Resultados de implementación asíncrona de AES-128 para encriptación

El programa escrito en lenguaje Balsa que describe el circuito que implementa el algoritmo de Rijndael para el proceso de encriptación, *Rijndael_enc,* se puede observar en el anexo 5.A, al final de este capítulo. Mediante los procedimientos definidos en el código se interconectan las funciones de transformación (descritas en el capítulo 4) por medio de canales de *handshake*. El procedimiento *RondaNormal_enc* se ejecuta de manera recursiva nueve veces, mientras que el procedimiento *RondaFinal_enc*, como es obvio, solo se ejecuta una vez, sumando el total de diez rondas que completan el proceso de encriptación. La implementación del algoritmo se ha realizado con base en las especificaciones del estándar *FIPS-197* [3]. Se utilizó el mismo ejemplo del estándar para realizar la simulación de la implementación asíncrona del algoritmo en Balsa.

Los siguientes son los valores utilizados como entrada de datos (*matriz de estado*) y clave de encriptación, cada uno con longitud de 16 bytes ($Nb = 4$ y $Nk = 4$), ambos en formato hexadecimal:

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

El sistema Balsa tiene dos maneras de obtener los resultados de simulación funcional: de forma gráfica, mediante el visor de ondas *GTKWave* y en forma de texto, utilizando una

ventana de la interfaz de usuario. Esta última es bastante útil en la medida que los resultados se visualizan de forma rápida. La simulación mediante el uso del visor gráfico se utiliza cuando se necesita observar los tiempos de *handshake* de las señales asociadas con los distintos canales.

La figura 5.9 muestra los resultados desplegados en modo texto luego de procesar los datos de entrada y la clave de encriptación. En esta versión de Balsa los datos se despliegan en formato decimal, por lo que se requiere hacer la conversión a formato hexadecimal y confirmar su equivalencia con los resultados mostrados en el ejemplo del estándar FIPS. El anexo 5.B muestra los resultados que, según el estándar, se deben obtener al final de cada ronda de encriptación.

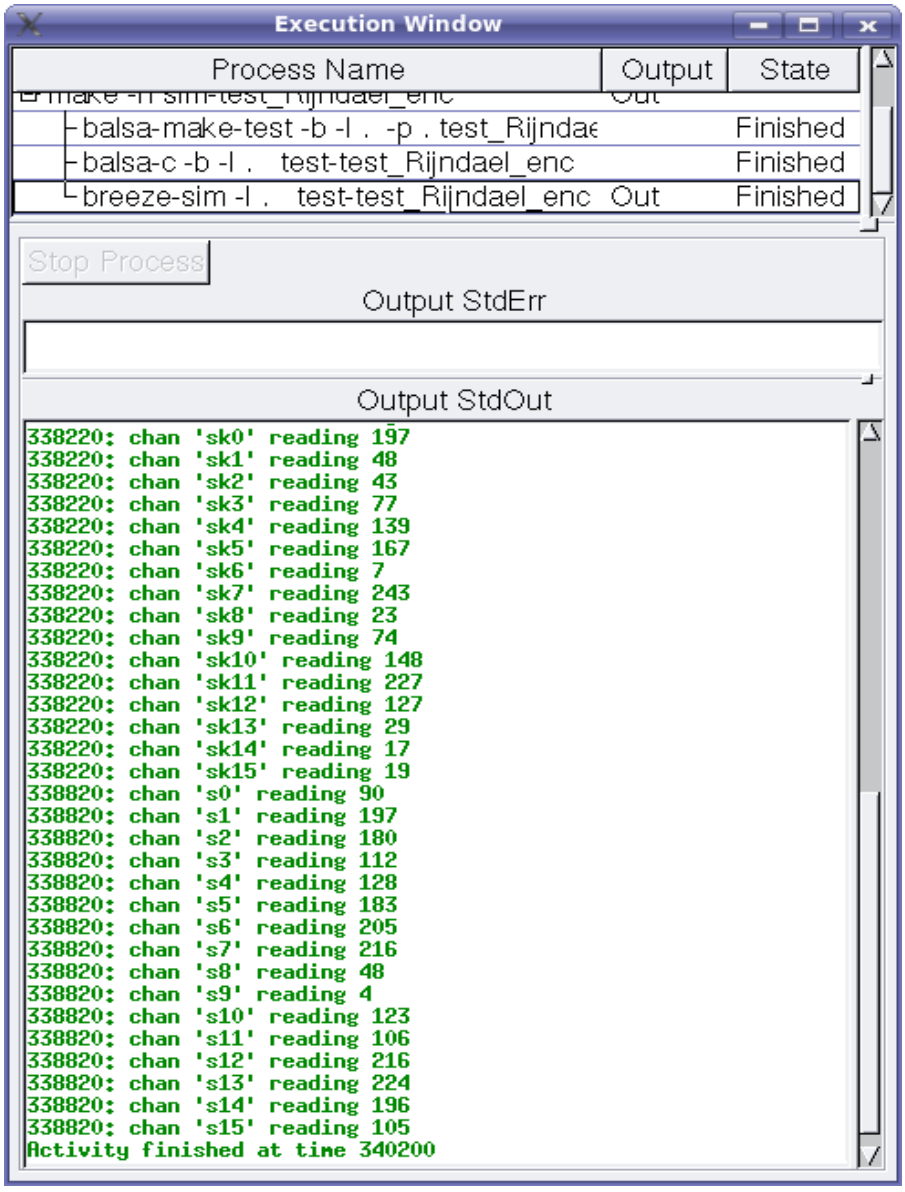


Figura 5.9: Resultados del proceso de encriptación *(Rijndael_enc)* desplegados en modo texto

En la simulación funcional de los circuitos especificados en Balsa se ha comprobado que las transformaciones que utilizan Sbox introducen el mayor retardo en la implementación del algoritmo. Por tal razón, se han llevado a cabo simulaciones en las que las transformaciones de sustitución de byte utilizan la versión combinatoria PPRM de Sbox y luego funciones Sbox definidas mediante tablas. Los resultados de la simulación asíncrona del algoritmo completo para encriptación con los dos tipos de Sbox se han consignado en la tabla 5.3.

Tabla 5.3: Resultados de simulación asíncrona del algoritmo de Rijndael: proceso de encriptación usando claves de 128 bits (AES-128)

Tipo de Sbox utilizadas	***Función***	***Costo Área***	***Retardo (ns)***
PPRM	*RondaNormal_enc*	*25950.25*	*77700*
	RondaFinal_enc	*6211.0*	*65100*
	Rijndael_enc	***291488.25***	***872300***
Tablas	*RondaNormal_enc*	*25950.25*	*23500*
	RondaFinal_enc	*6211.0*	*10600*
	Rijndael_enc	***291488***	***340200***

Los resultados observados en esta tabla indican que los tiempos de retardo para encriptación disminuyen bastante al utilizar transformaciones de sustitución de byte que utilizan funciones Sbox definidas en tablas, en lugar de utilizar la funciones Sbox definidas mediante circuitos combinatorios PPRM. Según los datos, el porcentaje de la mejora obtenida en el desempeño, al usar tablas Sbox para la implementación completa del algoritmo de encriptación, es de 60.1%.

En las simulaciones, Balsa utiliza el visor de ondas *GTKWave*. Este visor de formas de onda despliega un conjunto de canales sobre el lado derecho de la pantalla. Las señales de *request* se indican con color rojo y las señales de *acknowledge* con color verde. Al lado izquierdo se listan los nombres de las señales; el signo "=" indica el estado de las señales de *handshake* (*r* : *request* y *a*: *acknowledge*) según la posición del cursor vertical de tiempo, "+" indica nivel alto y "-" nivel bajo, luego aparece el dato asociado con el canal.

Las figuras 5.10 y 5.11 ilustran los diagramas de tiempo con los resultados obtenidos en la simulación asíncrona de Rijndael. Esta simulación corresponde a la descripción del algoritmo que incluye las funciones *ByteSub* y *SubWord* definidas mediante tablas Sbox. Los 16 bytes de texto encriptado en formato hexadecimal se pueden comprobar con el resultado final del ejemplo de encriptación del anexo 5.B.

El vector de bytes obtenido como texto encriptado y la última clave calculada en esta simulación se utilizan como vectores de entrada en el proceso de desencriptación, que se describe en la siguiente sección. Como es de esperarse, los resultados del proceso de desencriptación deben ser los mismos vectores de entrada utilizados para encriptación. La ventana de la figura 5.10 despliega las señales para los ocho bytes menos significativos del texto encriptado. La columna del lado izquierdo muestra el nombre de cada salida, el estado de las señales *request* y *acknowledge* según la posición del cursor vertical, y el dato de salida en formato hexadecimal.

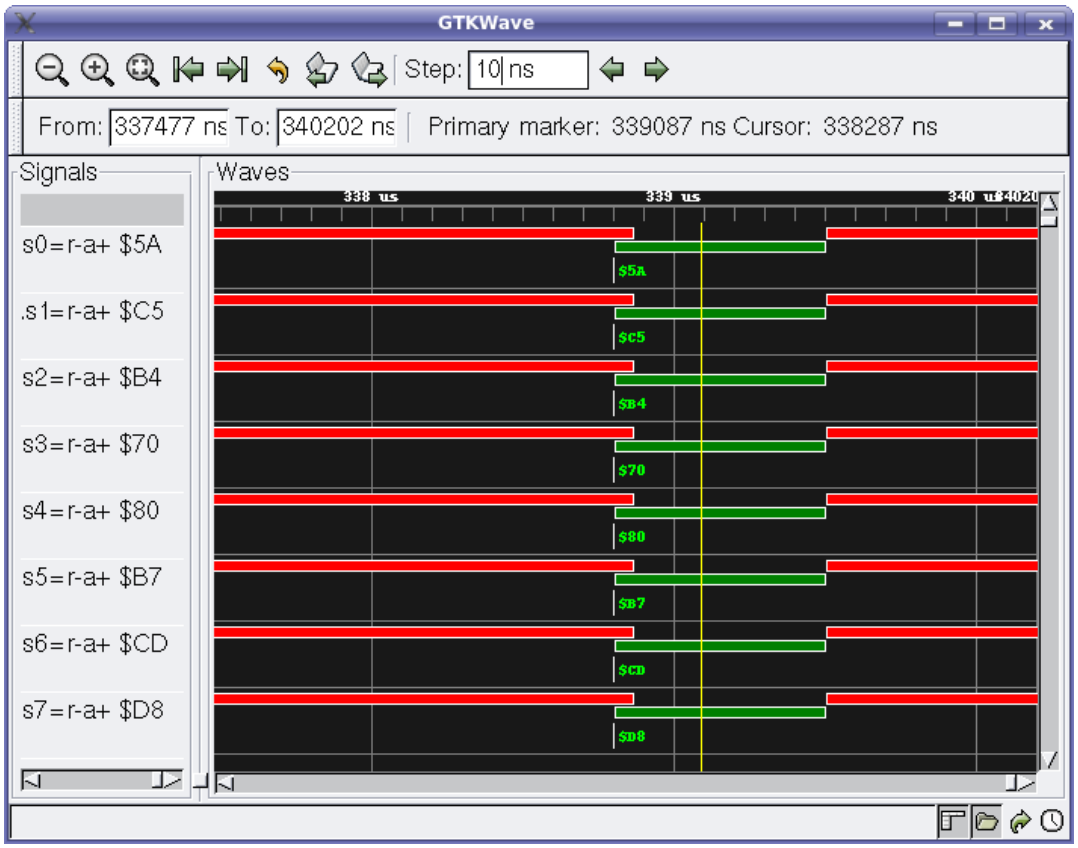


Figura 5.10: Resultado de simulación de algoritmo de Rijndael, dato encriptado: bytes S0 a S7

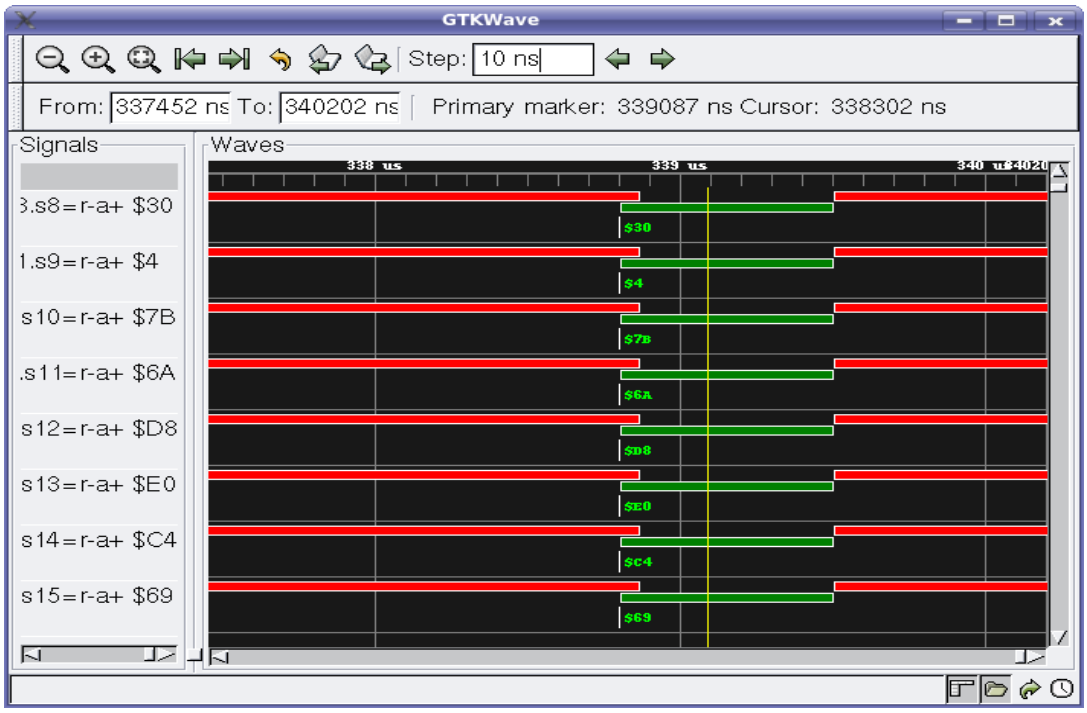


Figura 5.11: Resultado de simulación de algoritmo de Rijndael, dato encriptado: bytes S8 a S15

En la parte superior, arriba de las formas de onda, se observa una barra en la que se indica los tiempos inicial y final en que las señales están enmarcadas. También indica el instante de tiempo en que está posicionado el cursor vertical (línea amarilla).

Todas las salidas de byte responden a las señales de *handshake* correspondientes con la misma temporización debido a que ellas se procesan en paralelo. La línea de color rojo corresponde a la señal de *request* la cual se encuentra desactivada en la posición del cursor (r-); la línea verde corresponde a la señal de *acknowledge*, la cual se encuentra activada en la posición del cursor (a+). El dato de salida se indica por su valor en hexadecimal y por una pequeña línea vertical que señala el instante en el que éste queda disponible en la variable de salida.

Las señales indican que una vez se hace presente el dato en el canal y se ha activado la señal *request* (solicitud de transmisión de dato) se inicia un ciclo de *acknowledge* (aceptación de dato) que permite que el dato pueda ser almacenado en una variable o elemento de salida. Una vez el dato ha sido almacenado, la señal de *acknowledge* se desactiva y la de *request* se activa de nuevo a la espera de un nuevo dato en el canal, sin embargo, debido a que en este caso no hay más datos para procesar, la señal *request* también se desactivará al finalizar el proceso de encriptación.

Al momento de establecerse el dato sobre el canal en 338.8µs se establece también una simultaneidad de activación entre *request* y *acknowledge* que dura 58ηs. La señal *acknowledge* permanece activa durante 500ηs; luego de esto la señal *request* se activa de nuevo hasta el final del proceso en 340.2µs. Las figuras 5.12 y 5.13 ilustran los diagramas de tiempo con los resultados de la clave de salida en formato hexadecimal. Los 16 bytes de la clave y el tiempo de retardo corresponden a la implementación del algoritmo utilizando Sbox especificadas mediante tablas.

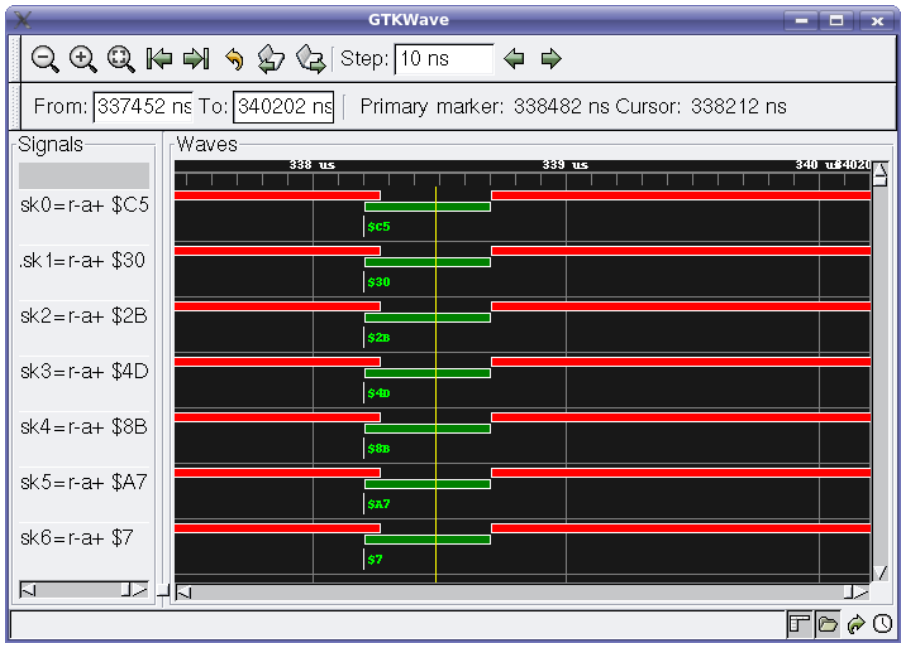


Figura 5.12: Resultado de simulación de algoritmo de Rijndael, clave de salida: bytes Sk0 a Sk7

La última clave calculada se establece en la salida en 338.2µs, 0.6µs antes que se establezcan las salidas encriptadas. Aquí también se presenta una simultaneidad de activación entre *request* y *acknowledge* que dura 58ns y la señal *acknowledge* permanece activa durante 500ns.

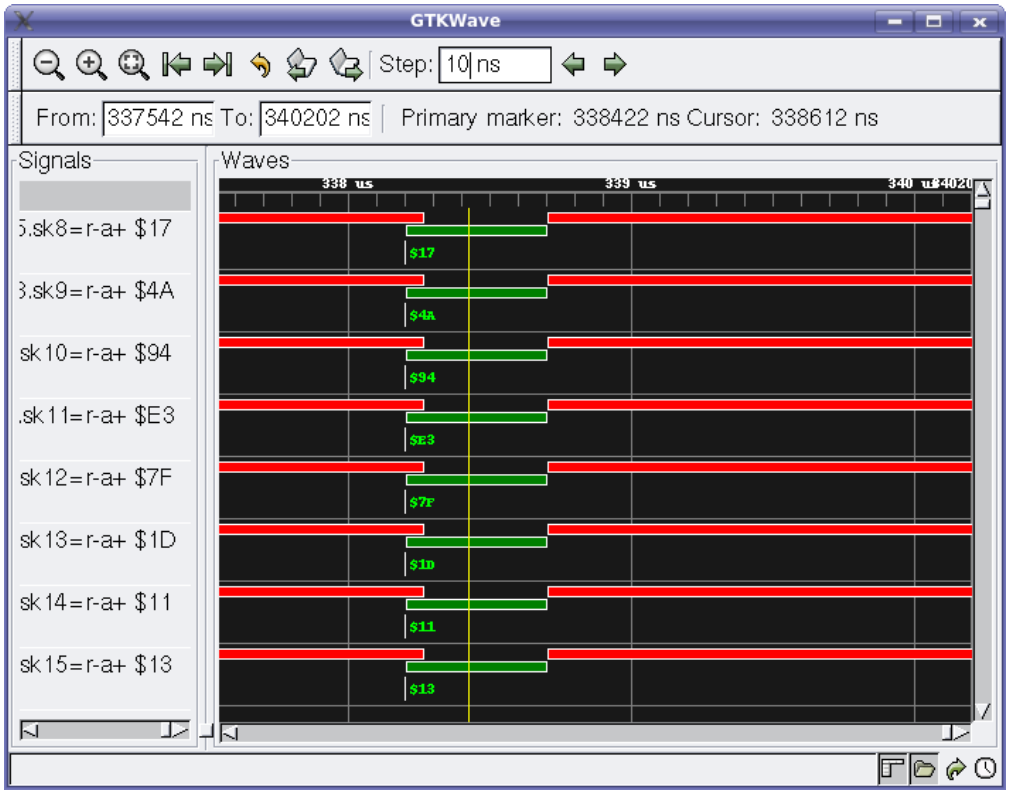


Figura 5.13: Resultado de simulación de algoritmo de Rijndael, clave de salida: bytes Sk8 a Sk15

El texto encriptado y la última clave generada, mostrados en las figuras anteriores, se muestran como vectores en la gráfica 5.14. Estos vectores se utilizan como entradas para el proceso de desencriptación, el cual se describe en la siguiente sección.

	S15	S14	S13	S12	S11	S10	S9	S8	S7	S6	S5	S4	S3	S2	S1	S0
Texto encriptado	69	c4	e0	d8	6a	7b	04	30	d8	cd	b7	80	70	b4	c5	5a
	MSB															*LSB*

	Sk15	Sk14	Sk13	Sk12	Sk11	Sk10	Sk9	Sk8	Sk7	Sk6	Sk5	Sk4	Sk3	Sk2	Sk1	Sk0
Última clave generada	13	11	1d	7f	e3	94	4a	17	f3	07	a7	8b	4d	2b	30	c5
	MSB															*LSB*

Figura 5.14: Vectores de salida del algoritmo de Rijndael obtenidos en el proceso de encriptación

5.3 Algoritmo de Rijndael: proceso de desencriptación

La secuencia de las transformaciones para el proceso de desencriptación es diferente a la secuencia de transformaciones para encriptación (como se muestra en la figura 5.6). Sin embargo, gracias a las propiedades del estándar AES, descritas en la sección 5.2.1, es posible implementar el proceso de desencriptación con la misma secuencia usada en el proceso de encriptación, tal como se aprecia en el diagrama de bloques de la figura 5.15.

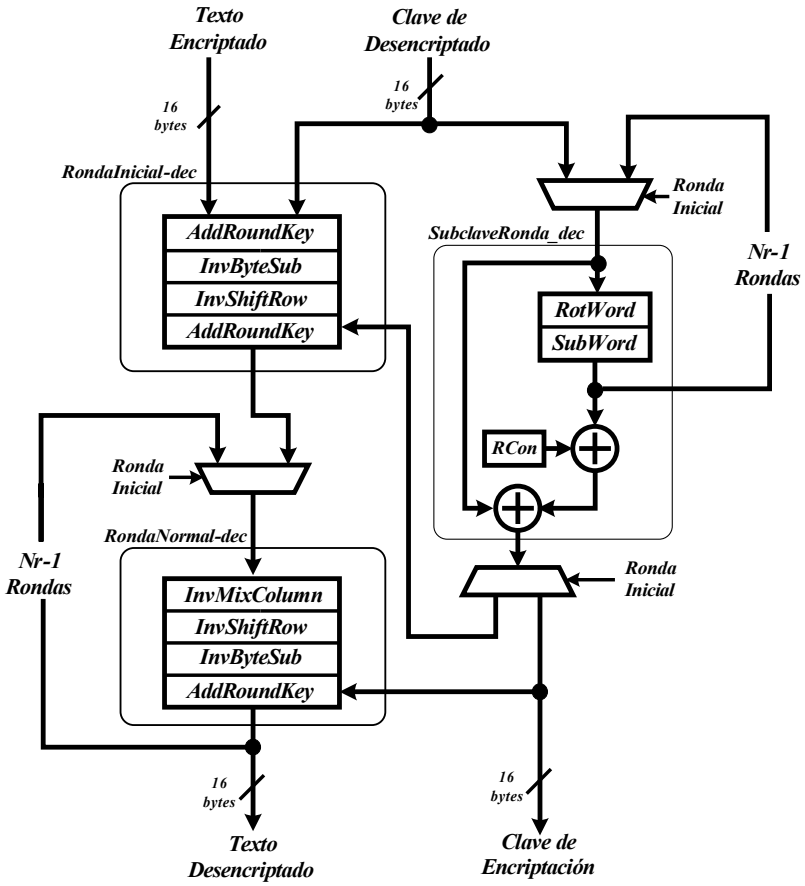


Figura 5.15: Diagrama de bloques del algoritmo de Rijndael para desencriptación

5.3.1. Bloque *RondaInicial_dec*

Esta ronda, cuya codificación en Balsa se observa en la figura 5.16, agrupa las funciones *AddroundKey* (al comienzo y al final del bloque), *ShiftRow e InvByteSub*, sin embargo, gracias a las propiedades de conmutación de AES, es posible invertir la secuencia para *InvShiftRow* e *InvByteSub*. Esto posibilita organizar la conexión de canales entre las transformaciones *AddRoundKey* (la primera del bloque) e *InvByteSub*, de tal manera que quede implementada la transformación *InvShiftRow,* evitando la necesidad de incluirla explícitamente dentro de la ronda. La transformación *AddRoundKey* al final de este bloque, se escribe fuera de la función por conveniencia en el desarrollo de la codificación en Balsa.

```
import [balsa.types.basic]
import [AddRoundKey]
import [InvByteSub16]

procedure  invrondainicial (input
d15,d14,d13,d12,d11,d10,d9,d8,d7,d6,d5,d4,d3,d2,d1,d0,k15,k14,k13,k12,
k11,k10,k9,k8,k7,k6,k5,k4,k3,k2,k1,k0 : byte ; output
s15,s14,s13,s12,s11,s10,s9,s8,s7,s6,s5,s4,s3,s2,s1,s0 : byte) is
channel ca15,ca14,ca13,ca12,ca11,ca10,ca9,ca8,ca7,ca6,ca5,ca4,ca3,ca2,ca1,ca0 : byte

begin
        addroundk1(d15,d14,d13,d12,d11,d10,d9,d8,d7,d6,d5,d4,d3,d2,d1,d0,k15,k14,k13,k12,k11,
        k10,k9,k8,k7,k6,k5,k4,k3,k2,k1,k0,ca15,ca14,ca13,ca12,ca11,ca10,ca9,ca8,ca7,ca6,ca5,ca4,ca
        3,ca2,ca1,ca0) ||

--InvShiftRow se implementa aqui reorganizando el orden de los canales de entrada a InvByteSub

        invbytesub16(ca15,ca2,ca5,ca8,ca11,ca14,ca1,ca4,ca7,ca10,ca13,ca0,ca3,ca6,ca9,ca12,s15,
        s14, s13,s12,s11, s10,s9,s8,s7,s6,s5,s4,s3,s2,s1,s0)
end
```

Figura 5.16: Código en Balsa para *RondaInicial_dec*

5.3.2. Bloque *RondaNormal_dec*

La ronda normal para desencriptación también permite el cambio en la secuencia de las transformaciones, de esa manera se hace posible implementar de manera implícita, ahorrando hardware, la función *InvShiftRow*, tal como se aprecia en la codificación en lenguaje Balsa de la figura 5.17.

```
import [balsa.types.basic]
import [InvMixColumn]
import [InvByteSub16]

procedure  invrondanormal (input d15,d14,d13,d12,d11,d10,d9,d8,d7,d6,d5,d4,d3,d2,d1,d0 : byte; out-
put s15,s14,s13,s12,s11,s10,s9,s8,s7,s6,s5,s4,s3,s2,s1,s0 : byte) is

channel ca15,ca14,ca13,ca12,ca11,ca10,ca9,ca8,ca7,ca6,ca5,ca4,ca3,ca2,ca1,ca0 : byte
channel cb15,cb14,cb13,cb12,cb11,cb10,cb9,cb8,cb7,cb6,cb5,cb4,cb3,cb2,cb1,cb0 : byte

begin
        invmixcolumn1(d15,d14,d13,d12,d11,d10,d9,d8,d7,d6,d5,d4,d3,d2,d1,d0,ca15,ca14,ca13,
        ca12,ca11,ca10,ca9,ca8,ca7,ca6,ca5,ca4,ca3,ca2,ca1,ca0) ||

-- Al reorganizar el orden de los canales de entrada a InvByteSub se implementa InvShiftRow

        invbytesub16(ca15,ca2,ca5,ca8,ca11,ca14,ca1,ca4,ca7,ca10,ca13,ca0,ca3,ca6,ca9,ca12,s15,s14
        ,s13,s12,s11,s10,s9, s8,s7,s6,s5,s4,s3,s2,s1,s0)
end
```

Figura 5.17: Código en Balsa para *RondaNormal_dec*

5.3.3. Resultados de implementación asíncrona de AES-128 para desencriptación

El código del programa escrito en lenguaje Balsa, *Rijndael_dec*, que describe el circuito que implementa el algoritmo de Rijndael para el proceso de desencriptación con claves de 128 bits (AES-128) se puede observar en el anexo 5.C de este capítulo. La función *RondaNormal_dec* se ejecuta de manera recursiva nueve veces, y junto con la ejecución de la función *RondaInicial_dec*, suma las diez rondas que completan el proceso de desencriptación.

Los siguientes son los vectores utilizados como entrada de datos encriptados y clave de desencriptación, cada uno con longitud de 16 bytes (Nb = 4 y Nk = 4) y en formato hexadecimal, se debe notar que son los mismos vectores obtenidos del proceso de encriptación.

Texto encriptado = 69 c4 e0 d8 6a 7b 04 30 d8 cd b7 80 70 b4 c5 5a
Clave de desencriptación = 13 11 1d 7f e3 94 4a 17 f3 07 a7 8b 4d 2b 30 c5

Los resultados obtenidos al utilizar estos valores después de cada ronda de desencriptación se muestran en el Anexo 5.D. De igual forma que para el algoritmo de encriptación, la simulación asíncrona del algoritmo de Rijndael para desencriptación se realizó teniendo en cuenta los dos tipos de Sbox utilizadas por las funciones de sustitución de byte. Los resultados se observan en la tabla 5.4. El tiempo de retardo obtenido en la simulación para la función *Rijndael_dec* indica que este es mucho menor al utilizar transformaciones de sustitución de byte que utilizan funciones Sbox definidas en tablas. Los datos de la tabla 5.4 permiten calcular un incremento de 54.8% en el desempeño respecto de la implementación que utiliza Sbox PPRM.

Tabla 5.4: Resultados de simulación asíncrona del algoritmo de Rijndael: proceso de desencriptación usando claves de 128 bits (AES-128)

Tipo de Sbox utilizadas	***Función***	***Costo Área***	***Retardo (ns)***
PPRM	*RondaNormal_dec*	*1006.75*	*93900*
	RondaInicial_dec	*13119.0*	*64500*
	Rijndael_dec	***248883.5***	***944500***
Tablas	*RondaNormal_dec*	*1006.75*	*42300*
	RondaFinal_dec	*13119.0*	*12900*
	Rijndael_dec	***248883.5***	***426900***

La figura 5.18 muestra los resultados desplegados en modo texto luego de procesar los datos de entrada y la clave de desencriptación. En esta versión de Balsa los datos se despliegan en formato decimal, para confirmar su equivalencia con los resultados mostrados en el ejemplo del estándar FIPS se requiere hacer la conversión a formato hexadecimal.

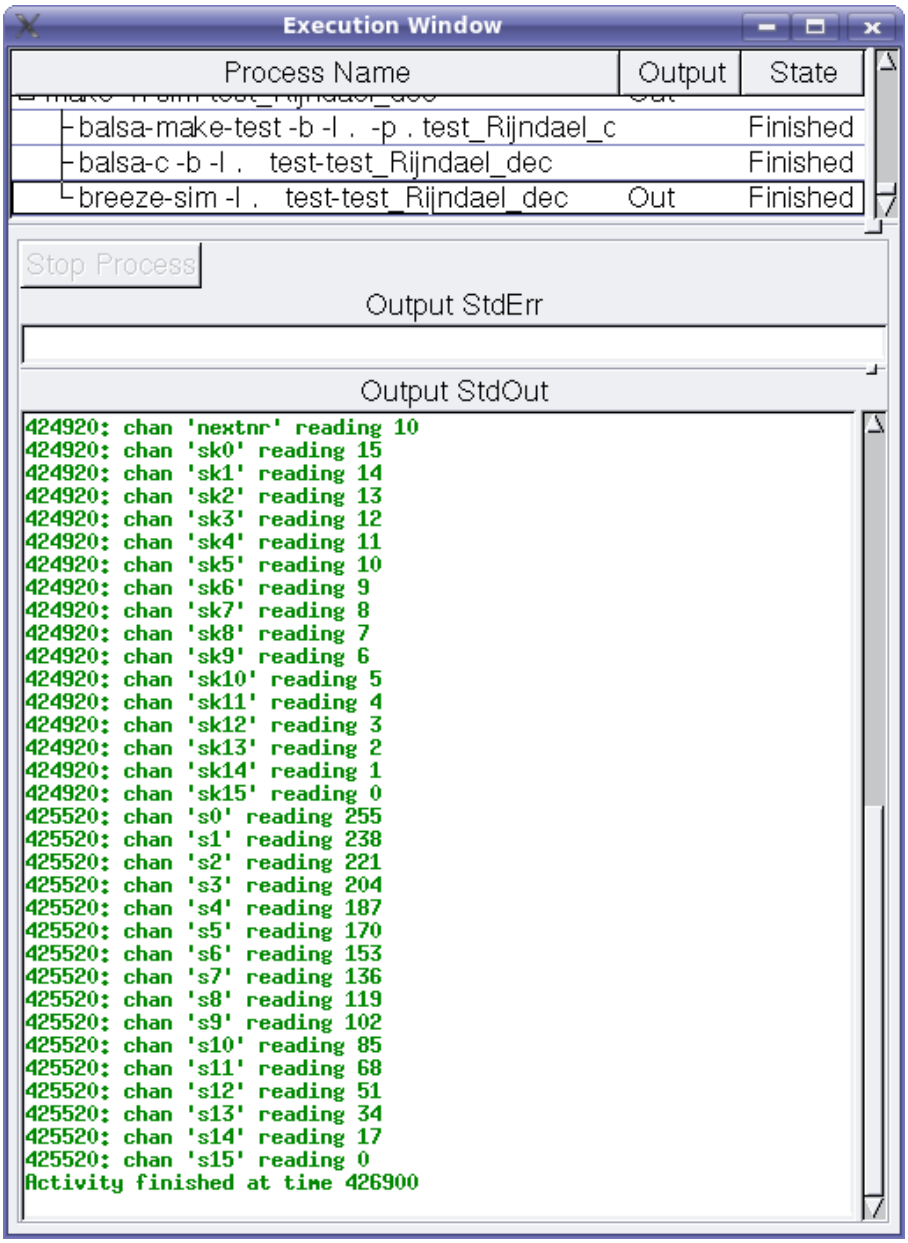


Figura 5.18: Resultados de la desencriptación *(Rijndael_dec)* desplegados en modo texto

Las figuras 5.19 y 5.20 ilustran los diagramas de tiempo con los resultados obtenidos en la simulación asíncrona de Rijndael. Esta simulación corresponde a la descripción del algoritmo que implementa las funciones *InvByteSub* y *SubWord* definidas mediante tablas Sbox.

Los 16 bytes de salida desencriptados (texto plano inicial) son los mismos utilizados como entrada en el proceso de encriptación de la sección 5.2.3. De esta manera y de acuerdo con los vectores de salida se confirma el correcto funcionamiento del proceso de desencriptación.

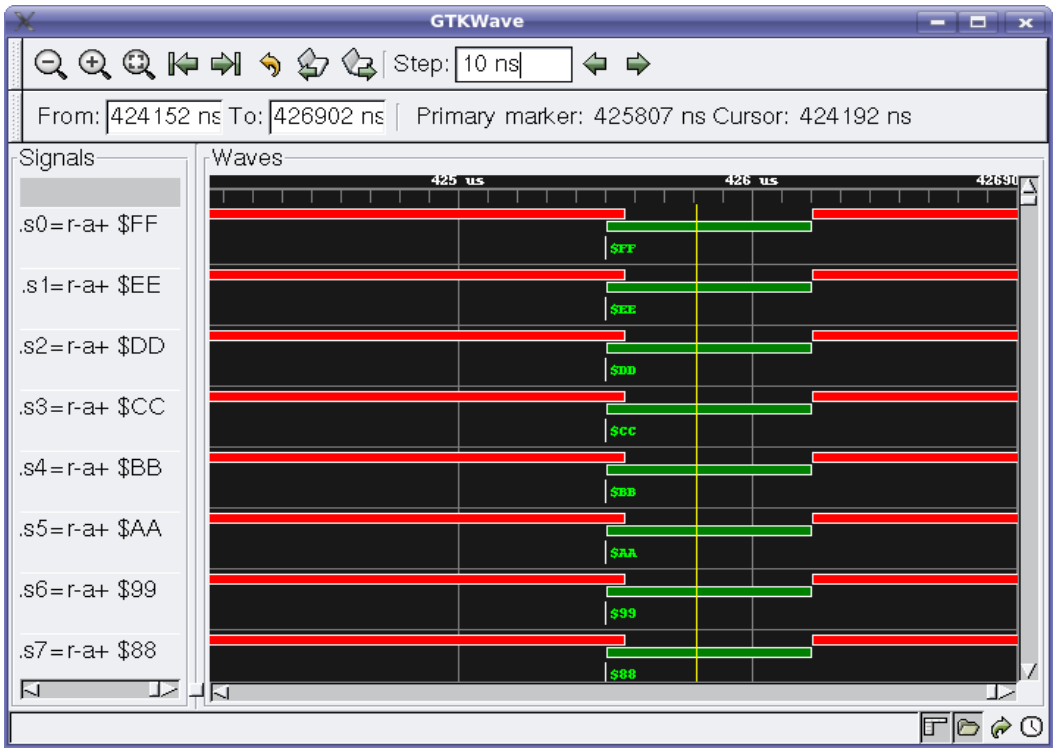


Figura 5.19: Resultado de simulación de algoritmo de Rijndael, bloque de datos desencriptado (texto plano): bytes S0 a S7

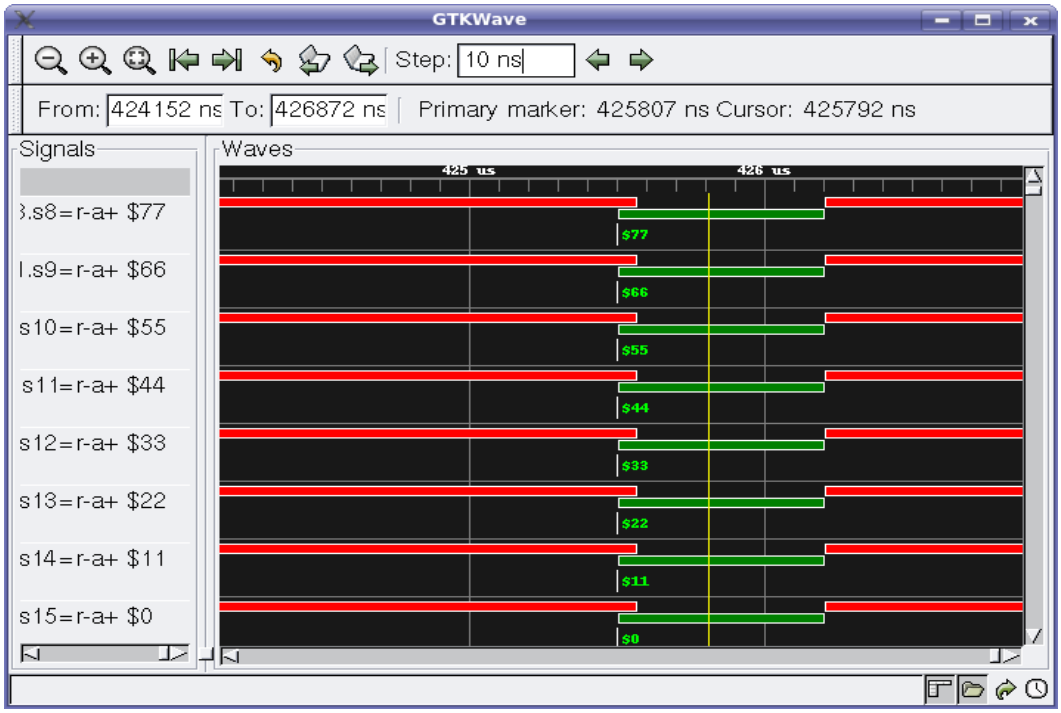


Figura 5.20: Resultado de simulación de algoritmo de Rijndael, bloque de datos desencriptado: bytes S8 a S15

Las figuras 5.21 y 5.22 muestran los 16 bytes de la salida de la unidad de claves de desencriptación.

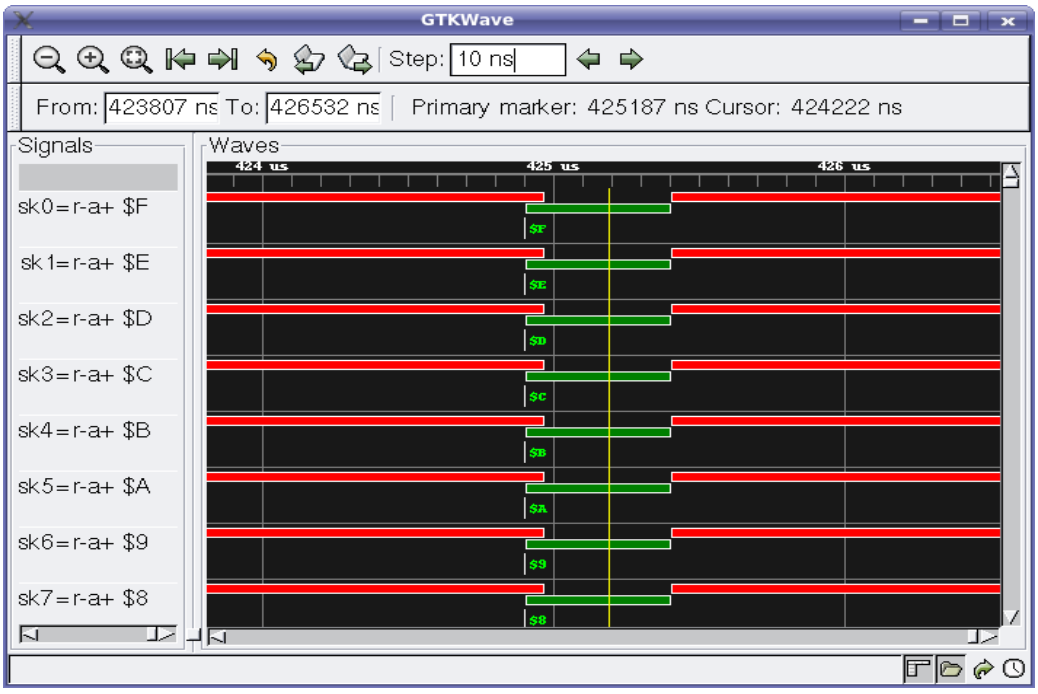


Figura 5.21: Resultado de simulación de algoritmo para desencriptación, clave de salida: bytes Sk0 a Sk7

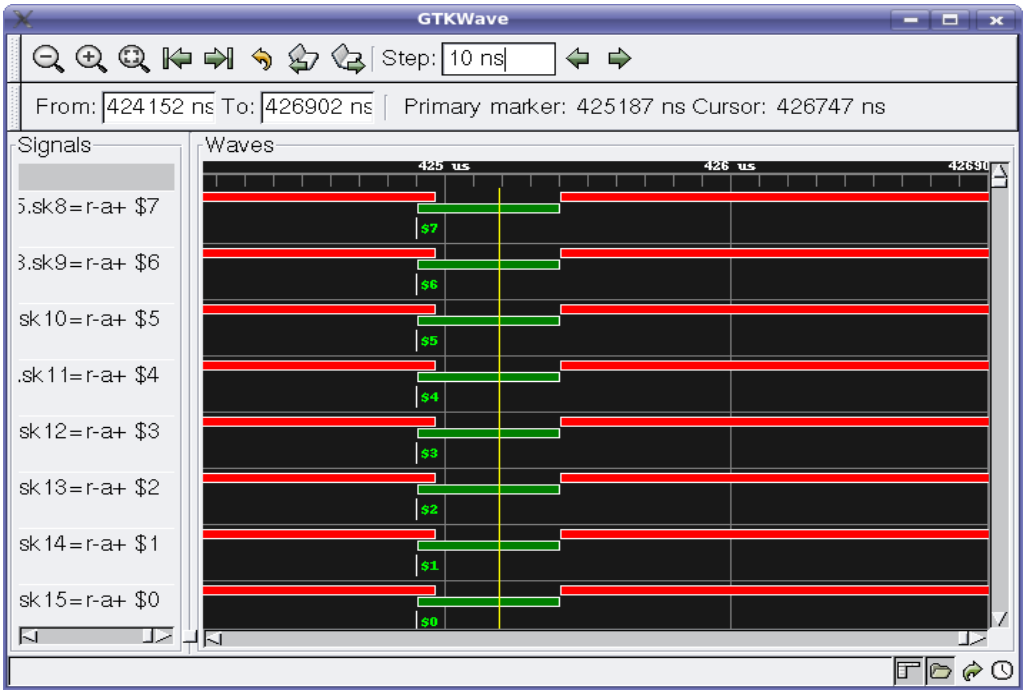


Figura 5.22: Resultado de simulación de algoritmo para desencriptación, clave de salida: bytes Sk8 a Sk15

En la figura 5.23 se presentan los vectores de salida obtenidos en el proceso de desencriptación. Se debe notar que estos vectores de salida son los mismos vectores de entrada utilizados como ejemplo en el proceso de encriptación. Es así como el texto desencriptado equivale al texto plano inicial y la clave desencriptada es la misma clave inicial de encriptación.

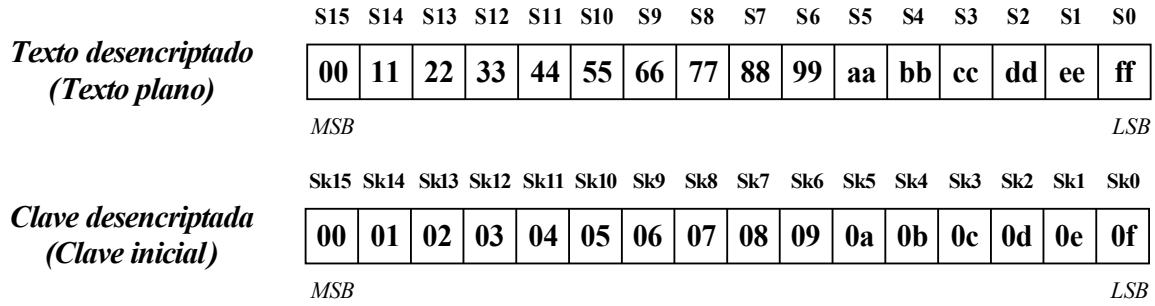


Figura 5.23: Vectores de salida del algoritmo de Rijndael para el proceso de desencriptación

5.4. Resultados de simulación en Balsa para encriptación/desencriptación

La descripción del algoritmo para encriptación/desencriptación, *EDRijndael*, se realizó de dos maneras: con Sbox combinatorias PPRM, y con Sbox implementadas en tablas. Los datos de entrada y las claves son las mismas utilizadas en los ejemplos anteriores de encriptación y desencriptación.

El programa en lenguaje Balsa que implementa ambos procesos se incluye en el anexo 5.E. Los resultados de la simulación asíncrona se muestran en la tabla 5.5.

Tabla 5.5: Resultados de simulación asíncrona del algoritmo de encriptación/desencriptación de Rijndael usando claves de 128 bits (AES-128)

Función	***Retardo (ns)*** *(Simulación en Balsa)*
EDRijndael *(Sbox PPRM)*	*Encriptación: 881200* *Desencriptación: 953400*
EDRijndael *(Sbox en Tablas)*	*Encriptación: 349100* *Desencriptación: 435800*

Los datos muestran que al utilizar tablas para implementar las Sbox de las funciones utilizadas en ambos procesos se obtiene un incremento en el desempeño aproximado de 68.38% para encriptación y de 54.29% para desencriptación.

5.5. Comentarios sobre la implementación asíncrona del algoritmo de Rijndael en FPGA

La implementación del algoritmo de Rijndael sobre FPGA sólo fue posible a nivel de funciones individuales. Casi todas las transformaciones básicas del algoritmo, especificadas de forma asíncrona mediante el sistema Balsa, pudieron ser implementadas

en FPGA, utilizando para ello los estilos asíncronos de codificación *dual-rail* y *1-de-4*. De las implementaciones en FPGA se exceptúan aquellas funciones o transformaciones que definen tablas para implementar los circuitos con el estilo *1-de-4*; una razón de ello es que el sistema Balsa no genera el correspondiente archivo en formato *Verilog* para este tipo de implementaciones. Balsa genera el siguiente reporte de error para los casos mencionados:

> *"ERROR: gen-> simple-gate -> default-gate: can´t find description for gate type 'one-of-four- dual-rail- dims inequal' ".*

Hasta la fecha este error no aparece reportado en la página de errores de Balsa ubicada en: http://solem.cs.man.ac.uk/index.cgi [9].

El tamaño de la FPGA ha sido otro factor limitante en la implementación completa del algoritmo. Aunque en la mayoría de las implementaciones los recursos lógicos de la FPGA (*CLBs*) han sido suficientes, los recursos de interconexión han sido utilizados al máximo y en algunos casos superados, forzando a la herramienta a utilizar bloques de entrada/salida (*IOBs*) como rutas alternas para conexión de componentes asíncronos, lo cual aumenta aún mas el tiempo de retardo asociado a rutas de interconexión.

A pesar que el sistema Balsa permitió obtener resultados de simulación funcional y también archivos de descripción asíncrona del algoritmo de Rijndael en formato *Verilog* para los procesos de encriptación y desencriptación, la implementación asíncrona completa del algoritmo en FPGA no ha sido posible.

Al tratar de realizar la síntesis del archivo de especificación del algoritmo en *Verilog* mediante el Sistema *ISE 9.2i de Xilinx*, este genera el reporte de error *"Portability:3",* el cual está relacionado, entre otros aspectos, con la excesiva aparición, en el código fuente, de instrucciones tipo "*if anidadas*". Este error está reportado en la página de *Xilinx* [10] donde se proponen varias soluciones de acuerdo con la versión de la herramienta utilizada, sin embargo, tales soluciones suponen un origen síncrono del problema, por lo que hasta el momento no se ha encontrado una solución que permita eliminar el error encontrado y lograr la síntesis de la especificación asíncrona completa del algoritmo de Rijndael mediante la herramienta *ISE 9.2i* para su implementación en FPGA.

La tabla 5.6 recopila las funciones de transformación que obtuvieron los menores tiempos de retardo utilizando la codificación balanceada *dual-rail*. Estos tiempos son el resultado de implementaciones sobre la FPGA Virtex XCV2P30-6ff896 de Xilinx. Con base en estos tiempos individuales se realiza el ejercicio de calcular el tiempo de encriptación total del algoritmo, en el supuesto que no existiesen los inconvenientes de implementación mencionados con anterioridad. El cálculo se realiza con base en la agrupación de las funciones mediante rondas definidas en las secciones 5.2.1 (*RondaNormal_enc*) y 5.2.2 (*RondaFinal_enc*).

Tabla 5.6: Tiempos de retardo de implementación de funciones de encriptación
Utilizando codificación *dual-rail* balanceada

Transformación	***Retardo***
	Dual-rail
SubclaveRonda_enc	***405.597ns***
ByteSubTabla	***252.360ns***
ShiftRow	***31.788ns***
MixColumn	***86.701ns***
AddRoundKey	***33.290ns***

De acuerdo con la descripción asíncrona del algoritmo para encriptación (Anexo 5.A), el primer nivel de ejecución del algoritmo involucra las funciones en la siguiente forma:

AddRoundKey (*33.290ns*)
RondaNormal: *ByteSubTabla* (*252.360ns*) + *MixColumn* (*86.701ns*), la función *ShiftRow* seimplementa mediante la conmutación de líneas entre estas dos funciones, por lo que no introduce retardo. Lo anterior suma *372.751ns*.

En el primer nivel de ejecución se requiere ejecutar la función de adición de clave como parte de la ronda normal, sin embargo, es necesario esperar a que la función *SubclaveRonda_enc* se ejecute, ya que esta lo hace en paralelo con las funciones de encriptación.
SubclaveRonda_enc (*405.597ns*)
AddRoundKey (*33.290ns*)
Es claro que, debido a que el cálculo de la subclave de ronda demora más tiempo, *SubclaveRonda_enc* + *AddRoundKey* determina el tiempo de retardo del primer nivel, lo cual suma *438.887ns.*

Los siguientes niveles de ejecución de encriptación (9 rondas) involucran las funciones *RondaNormal, SubclaveRonda_enc y AddRoundKey,* pero es claro que debido a su mayor retardo y a que se ejecuta en paralelo con *Ronda_Normal,* la función *SubclaveRonda_enc* sumada con *AddRoundKey* determina, también en este caso, el tiempo de retardo total de estos niveles de ejecución de la encriptación.

Total para 9 rondas *438.887ns x 9 = 3949.983ns*

El último nivel de encriptación:
RondaFinal_enc: *ByteSubTabla* (*252.360ns*) + *ShiftRow* (*31.788ns*) = *284.148ns*

En este nivel también *SubclaveRonda_enc* se ejecuta en paralelo y se requiere una última adición de clave, así que el tiempo de retardo en este nivel es de *438.887ns.*

Tiempo total de encriptación = 438.887ns +3949.983ns +438.887ns = ***4827.757ns***

Las funciones de transformación para desencriptación que obtuvieron los menores tiempos de retardo utilizando la codificación balanceada *dual-rail* se pueden ver en la tabla 5.7. Como en el proceso de encriptación, estos tiempos permiten realizar el ejercicio de calcular el tiempo de desencriptación total del algoritmo, en el supuesto que

no existiesen los inconvenientes de implementación. El cálculo se realiza con base en la agrupación de las funciones mediante rondas definidas en las secciones 5.3.1 (*RondaInicial_dec*) y 5.3.2 (*RondaNormal_dec*).

Tabla 5.7: Tiempos de retardo de implementación de funciones de desencriptación
Utilizando codificación *dual-rail* balanceada

Transformación	***Retardo***
	Dual-rail
SubclaveRonda_dec	***352.562ns***
InvByteSubTabla	***212.6266ns***
InvShiftRow	***31.788ns***
InvMixColumn	***176.884ns***
AddRoundKey	***33.290ns***

De acuerdo con la descripción asíncrona del algoritmo para desencriptación (Anexo 5.C), el primer nivel de ejecución del algoritmo involucra las funciones en el siguiente orden:

RondaInicial: AddRoundKey (*33.290ns*) + *InvByteSubTabla* (*212.626ns*)
+ *InvShiftRow* (*31.788ns*) = *277.704ns*

RondaInicial genera el estado de la primera ronda, el cual también se necesita para obtener la clave de segunda ronda.

SubclaveRonda_dec (*352.562ns*) se ejecuta en paralelo con *RondaInicial*, pero de acuerdo con el código del anexo 5.C se requiere de nuevo la ejecución de la función *AddRoundkey* para generar la subclave de la segunda ronda. Por lo tanto este tiempo inicial es:*352.562ns + 33.290ns =385.852ns*
Luego de ejecutarse *RondaInicial* se ejecutan las funciones de *RondaNormal_dec* de forma iterativa (se ejecuta recursivamente para 9 rondas). Para una sola ronda se tiene:

RondaNormal_dec = InvMixColumn (*176.884ns*) + *InvShiftRow* (*31.778ns*)
+ *InvByteSubTabla* (*212.626ns*) + *AddRoundKey* (*33.290ns*)

Como la función *InvShiftRow* se implementa fácilmente conmutando las salidas de *InvMixColumn* hacia *InvByteSub,* su tiempo de retardo no se tiene en cuenta, así el retardo de *RondaNormal_dec* es:

InvMixColumn (*176.884ns*) + *InvByteSubTabla* (*212.626ns*) + *AddRoundKey* (*33.290ns*)
= *422.800ns*

Para 9 rondas se tiene:

RondaNormal_dec (422.800ns) x *9 rondas = 3805.200ns*

De los cálculos anteriores se concluye que el tiempo de retardo está determinado por la generación del siguiente estado y este, a su vez, depende de la subclave generada para la siguiente ronda. De acuerdo con esto, el tiempo total de desencriptación se calcula sumando el tiempo de *Ronda_Inicial* y el tiempo de ejecución de las 9 iteraciones de *Ronda_Normal_dec*.

Tiempo total de desencriptación = 3805.200ns +385.852ns= ***4191.052ns***

5.6. Conclusiones del capítulo

5.6.1. Los resultados obtenidos en la simulación funcional del algoritmo de Rijndael para claves de 128 bits, indican que el desempeño de la implementación está determinado, en gran medida, por los tiempos de retardo introducidos por los circuitos que implementan las funciones de sustitución de byte: *ByteSub, InvByteSub y SubWord* implementadas mediante Sbox.

5.6.2. Los resultados también indican que el mejor desempeño en la simulación de Rijndael, para encriptación y desencriptación, se obtuvo al utilizar funciones Sbox basadas en tablas para implementar las transformaciones de sustitución de byte.

5.6.3. A pesar de existir una integración entre la herramienta de especificación asíncrona Balsa y la plataforma de desarrollo *Xilinx ISE 9.2i* se hace evidente que las FPGAs, utilizadas para implementar el circuito final, presentan restricciones para mapear circuitos asíncronos, como lo corroboran los resultados en los que más del 55% del retardo de una implementación asíncrona se debe a las rutas de interconexión de los circuitos lógicos.

5.6.4. La implementación completa del algoritmo de Rijndael en FPGA no pudo ser lograda debido a las limitaciones que presentan estos dispositivos a nivel de recursos de interconexión para sistemas asíncronos.

5.6.5. En la literatura revisada no se han encontrado aun reportes de resultados de implementaciones asíncronas del estándar AES sobre *FPGAs*. Este trabajo presenta los primeros reportes sobre implementación asíncrona en FPGA, a nivel de funciones individuales que conforman el algoritmo de Rijndael.

5.6.6. Hasta el momento sólo se han reportado dos implementaciones asíncronas para encriptación basados en *ASICs*. La tabla 5.1 muestra los reportes de las implementaciones de Rijndael en *ASIC* y también los resultados de simulación funcional del algoritmo obtenidos con el sistema Balsa en este trabajo de tesis.

Tabla 5.8: Tabla comparativa de implementaciones asíncronas del algoritmo de Rijndael

Referencia	***Desempeño***	***Tamaño (mm^2)***	***Comentarios especiales***
[5] (*ASIC*)	215 Mbits/seg	0.088 (clave 128 bits)	• V_{dd} = 1.2v • Tecnología CMOS 0.13µm • Codificación 1 de N • Modelo de circuito *QDI* • Arquitectura Balanceada • Solo se implementa el proceso de encriptación
	721 Mbits/seg	0.192 (clave 192 bits)	
	892 Mbits/seg	0.427 (clave 256 bits)	
[6], [7] (*ASIC*)	800 ns	1.897 (clave 128 bits)	• Tecnología CMOS 0.35µm • Modelo de circuito *SI* • Codificación *dual-rail* • Arquitectura Balanceada • Sólo se presentan resultados de simulación para encriptación
En este trabajo de Tesis	340200 ns (Encriptación) 426900 ns (desencriptación)		• Resultados de Simulación con Balsa para los procesos de encriptación/desencriptación • Claves de 128 bits • Implementación en *FPGA Virtex XCV2P30-6ff896* de Xilinx para las funciones individuales • Funciones individuales implementadas con codificación *dual-rail*, y 1-de-4 • *Dual-rail* de Arquitectura Balanceada

Referencias

[1] **Edwards D., Bardsley L., Janin L., Toms W.** *¨Balsa: a Tutorial Guide¨*. version 3.5. The Advanced Processor Technologies Group, APT Group. Manchester University. Manchester, England. 2006. Disponible en:
http://intranet.cs.man.ac.uk/apt

[2] **Bardsley A.** *"Implementing Balsa handshake Circuits"*, Ph.D. dissertation, directed by D. Edwards, faculty of Science & Engineering, University of Manchester, 2000. Disponible en:
ftp://ftp.cs.man.ac.uk/pub/amulet/theses/bardsley_phd.pdf

[3] ***FIPS***, ***Federal Information Processing Standards.*** "S*pecification for the ADVANCED ENCRYPTION STANDARD (AES)"*. Publication 197, November 26. 2001. Disponible en: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

[4] ***NIST***, ***National Institute of Standards and Technology***. (2000). Disponible en: http://csrc.nist.gov/CryptoToolkit/aes/rijndael/

[5] **Bouesse F., Renaudin M., Germain F.** *"Asynchronous AES Crypto- Processor Including Secured and Optimized Blocks"*. TIMA Laboratory, Grenoble, France; SGDN/ DCISS, Paris, France. Journal of Integrated Circuits and Systems, Vol 1, No 1, pp 5-13. March 2004.

[6] **Shang D., Burns F., Bystrov A., Koelmans A., Sokolov D., Yakovlev A.** *"A Low and Balanced Power Implementation of the AES Security Mechanism Using Self-Timed Circuits"*. E. Macii et al. (Eds.): PATMOS 2004, LNCS 3254, pp. 471–480, 2004.c. Springer-Verlag Berlin Heidelberg 2004.

[7] **Shang D., Burns F., Bystrov A., Koelmans A., Sokolov D., Yakovlev A.** ***"****High-security asynchronous circuit implementation of AES"*. IEE Proceedings online no. 20050088, doi:10.1049/ip-cdt:20050088. IEE Proc.-Comput. Digit. Tech., Vol. 153, No. 2, March 2006.

[8] **Daemen J., Rijmen V.** *"AES Proposal: Rijndael"*. Proton World Int. I, Zweefvliegtuigstraat, Brussel, Belgium; Katholieke Universiteit Leuven, ESAT-COSIC, Heverlee, Belgium. Document V. 2. 1999. Disponible en:
http://www.daimi.au.dk/~ivan/rijndael.pdf

[9] ***APT Group.*** *"Bugzilla*, *Balsa Bug Report Main Page"*. The Advanced Processor Technologies Group, Manchester University. Manchester, England. 2008. Disponible en: http://solem.cs.man.ac.uk/index.cgi

[10] Xilinx. http://www.xilinx.com/support/mysupport.htm

Anexo 5.A:

Descripción Asíncrona en Balsa del Algoritmo de Rijndael para encriptación

```
-- Programa en Lenguaje Balsa para descripción asíncrona del circuito
-- que implementa el Algoritmo de Encriptación de Rijndael

import [balsa.types.basic]
import [AddRoundKey]
import [RondaNormal_enc]
import [SubclaveRonda_enc]
import [RondaFinal_enc]

procedure  ronda_recursiva (parameter nivel: cardinal; input nr:nibble; input
d15,d14,d13,d12,d11,d10,d9,d8,d7,d6,d5,d4, d3,d2,d1,d0,k15,k14,k13,k12,k11,k10,k9,k8,k7,k6,k5,k4,k3,k2,k1,k0 :
byte; output nextnr: nibble; output s15,s14,s13, s12,
s11,s10,s9,s8,s7,s6,s5,s4,s3,s2,s1,s0,sk15,sk14,sk13,sk12,sk11,sk10,sk9,sk8,sk7,sk6,sk5,sk4,sk3,sk2,sk1,sk0 : byte) is

channel ca15,ca14,ca13,ca12,ca11,ca10,ca9,ca8,ca7,ca6,ca5,ca4,ca3,ca2,ca1,ca0 : byte
channel chk15,chk14,chk13,chk12,chk11,chk10,chk9,chk8,chk7,chk6,chk5,chk4,chk3,chk2,chk1,chk0: byte
variable cb15,cb14,cb13,cb12,cb11,cb10,cb9,cb8,cb7,cb6,cb5,cb4,cb3,cb2,cb1,cb0 : byte
variable cc15,cc14,cc13,cc12,cc11,cc10,cc9,cc8,cc7,cc6,cc5,cc4,cc3,cc2,cc1,cc0 : byte
channel chnr  : nibble
variable SigNr : nibble

begin

if nivel= 0 then  print error, "El parametro nivel no debe ser 0"

|  nivel= 1 then

begin

--  Operación de adición de Clave
addroundk1 (d15,d14,d13,d12,d11,d10,d9,d8,d7,d6,d5,d4,d3,d2,d1,d0,k15,k14,k13,k12,k11,k10,k9,k8,k7,k6,k5,k4,
k3,k2,k1,k0, ca15,ca14,ca13,ca12,ca11,ca10,ca9,ca8,ca7,ca6,ca5,ca4,ca3,ca2,ca1,ca0) ||

-- Ronda Normal: operaciones ShiftRow, ByteSub y MixColumn
Rondanormal (ca15,ca14,ca13,ca12,ca11,ca10,ca9,ca8,ca7,ca6,ca5,ca4,ca3,ca2,ca1,ca0,->cb15,->cb14,->cb13,-
>cb12,->cb11, ->cb10,->cb9,->cb8,->cb7,->cb6,->cb5,->cb4,->cb3,->cb2,->cb1,->cb0) ||

-- Generación de Subclave de ronda siguiente
subclaveronda (nr,k15,k14,k13,k12,k11,k10,k9,k8,k7,k6,k5,k4,k3,k2,k1,k0,->SigNr,->cc15,->cc14,->cc13,->cc12,-
>cc11,->cc10, ->cc9,->cc8,->cc7,->cc6,->cc5,->cc4,->cc3,->cc2,->cc1,->cc0) ;

sk15<- cc15 || sk14<- cc14 || sk13<- cc13 || sk12<- cc12 || sk11<- cc11 || sk10<- cc10 || sk9<- cc9 || sk8<- cc8 ||
sk7<- cc7 || sk6<- cc6 || sk5<- cc5 || sk4<- cc4 || sk3<- cc3  || sk2<- cc2  || sk1<- cc1  || sk0<- cc0  ||

nextnr <-  SigNr  ||

-- Adición de Subclave de ronda para generar bloque de ronda siguiente
s15<- cb15 xor cc15  ||
s14<- cb14 xor cc14  ||
s13<- cb13 xor cc13  ||
s12<- cb12 xor cc12  ||
s11<- cb11 xor cc11  ||
s10<- cb10 xor cc10  ||
s9<- cb9 xor cc9  ||
s8<- cb8 xor cc8  ||
```

```
s7<- cb7 xor cc7  ||
s6<- cb6 xor cc6  ||
s5<- cb5 xor cc5  ||
s4<- cb4 xor cc4  ||
s3<- cb3 xor cc3  ||
s2<- cb2 xor cc2  ||
s1<- cb1 xor cc1  ||

s0<- cb0 xor cc0
end
| nivel>= 2 then

begin

ronda_recursiva
(nivel-1,nr,d15,d14,d13,d12,d11,d10,d9,d8,d7,d6,d5,d4,d3,d2,d1,d0,k15,k14,k13,k12,k11,k10,k9,k8,k7,k6,k5,k4, k3,
k2,
k1,k0,chnr,ca15,ca14,ca13,ca12,ca11,ca10,ca9,ca8,ca7,ca6,ca5,ca4,ca3,ca2,ca1,ca0,chk15,chk14,chk13,chk12,chk11,c
hk10,chk9,chk8,chk7,chk6,chk5,chk4,chk3,chk2,chk1,chk0) ||

rondanormal (ca15,ca14,ca13,ca12,ca11,ca10,ca9,ca8,ca7,ca6,ca5,ca4,ca3,ca2,ca1,ca0,->cb15,->cb14,->cb13,->cb12,
->cb11, ->cb10, ->cb9, ->cb8,->cb7,->cb6,->cb5,->cb4,->cb3,->cb2,->cb1,->cb0) ||

subclaveronda (chnr,chk15,chk14,chk13,chk12,chk11,chk10,chk9,chk8,chk7,chk6,chk5,chk4,chk3,chk2,chk1,chk0,-
>SigNr,->cc15, ->cc14,->cc13,->cc12,->cc11,->cc10,->cc9,->cc8,->cc7,->cc6,->cc5,->cc4,->cc3,->cc2,->cc1,->cc0) ;

sk15<- cc15  || sk14<- cc14  || sk13<- cc13  || sk12<- cc12  || sk11<- cc11  || sk10<- cc10  || sk9<- cc9  || sk8<- cc8  ||
sk7<- cc7  || sk6<- cc6  || sk5<- cc5  || sk4<- cc4  || sk3<- cc3  || sk2<- cc2  || sk1<- cc1  || sk0<- cc0  ||

nextnr <- SigNr ||
s15<- cb15 xor cc15  ||
s14<- cb14 xor cc14  ||
s13<- cb13 xor cc13  ||
s12<- cb12 xor cc12  ||
s11<- cb11 xor cc11  ||
s10<- cb10 xor cc10  ||
s9<- cb9 xor cc9  ||
s8<- cb8 xor cc8  ||
s7<- cb7 xor cc7  ||
s6<- cb6 xor cc6  ||
s5<- cb5 xor cc5  ||
s4<- cb4 xor cc4  ||
s3<- cb3 xor cc3  ||
s2<- cb2 xor cc2  ||
s1<- cb1 xor cc1  ||
s0<- cb0 xor cc0

end  --begin
end  --if
end

procedure main_encrip (input nr:nibble; input
d15,d14,d13,d12,d11,d10,d9,d8,d7,d6,d5,d4,d3,d2,d1,d0,k15,k14,k13,k12,k11,k10, k9,k8, k7,k6,k5, k4,k3,k2,k1,k0 :
byte; output nextnr: nibble; output  s15,s14,s13,s12,s11,s10,s9,s8,s7,s6,s5,s4,s3,s2,s1,s0,sk15, sk14,sk13,sk12,sk11,
sk10,sk9,sk8,sk7,sk6,sk5,sk4,sk3,sk2,sk1,sk0 : byte) is

channel ch15,ch14,ch13,ch12,ch11,ch10,ch9,ch8,ch7,ch6,ch5,ch4,ch3,ch2,ch1,ch0 : byte
variable cb15,cb14,cb13,cb12,cb11,cb10,cb9,cb8,cb7,cb6,cb5,cb4,cb3,cb2,cb1,cb0 : byte
channel chk15,chk14,chk13,chk12,chk11,chk10,chk9,chk8,chk7,chk6,chk5,chk4,chk3,chk2,chk1,chk0: byte
variable cc15,cc14,cc13,cc12,cc11,cc10,cc9,cc8,cc7,cc6,cc5,cc4,cc3,cc2,cc1,cc0 : byte
variable SigNr : nibble
channel chnr  : nibble
```

```
begin

ronda_recursiva
(9,nr,d15,d14,d13,d12,d11,d10,d9,d8,d7,d6,d5,d4,d3,d2,d1,d0,k15,k14,k13,k12,k11,k10,k9,k8,k7,k6,k5,k4,k3,
k2,k1,k0, chnr,ch15,ch14,ch13,ch12,ch11,ch10,ch9,ch8,ch7,ch6,ch5,ch4,ch3,ch2,ch1,ch0,chk15,chk14,chk13,chk12,
chk11, chk10, chk9,chk8,chk7,chk6,chk5,chk4,chk3,chk2,chk1,chk0)  ||

rondafinal (ch15,ch14,ch13,ch12,ch11,ch10,ch9,ch8,ch7,ch6,ch5,ch4,ch3,ch2,ch1,ch0,->cb15,->cb14,->cb13,->cb12,-
>cb11, ->cb10, ->cb9, ->cb8,->cb7,->cb6,->cb5,->cb4,->cb3,->cb2,->cb1,->cb0) ||

subclaveronda (chnr,chk15,chk14,chk13,chk12,chk11,chk10,chk9,chk8,chk7,chk6,chk5,chk4,chk3,chk2,chk1,chk0,-
>SigNr, ->cc15, ->cc14,->cc13,->cc12,->cc11,->cc10,->cc9,->cc8,->cc7,->cc6,->cc5,->cc4,->cc3,->cc2,->cc1,-
>cc0) ;

sk15<- cc15 || sk14<- cc14 || sk13<- cc13 || sk12<- cc12 || sk11<- cc11 || sk10<- cc10 || sk9<- cc9 || sk8<- cc8 ||
sk7<- cc7 || sk6<- cc6 || sk5<- cc5 || sk4<- cc4 || sk3<- cc3  || sk2<- cc2  || sk1<- cc1  || sk0<- cc0  ||

nextnr <-  SigNr  ||

-- Adición de Subclave de ronda final
-- Como cb y cc son variables estas no se manipulan mediante 'procedure'

s15<- cb15 xor cc15  ||
s14<- cb14 xor cc14  ||
s13<- cb13 xor cc13  ||
s12<- cb12 xor cc12  ||
s11<- cb11 xor cc11  ||
s10<- cb10 xor cc10  ||
s9<- cb9 xor cc9  ||
s8<- cb8 xor cc8  ||
s7<- cb7 xor cc7  ||
s6<- cb6 xor cc6  ||
s5<- cb5 xor cc5  ||
s4<- cb4 xor cc4  ||
s3<- cb3 xor cc3  ||
s2<- cb2 xor cc2  ||
s1<- cb1 xor cc1  ||
s0<- cb0 xor cc0

end
```

Anexo 5.B: Vectores de ejemplo para encriptación

Texto plano: 00112233445566778899aabbccddeeff

Clave de encriptación: 000102030405060708090a0b0c0d0e0f

```
round[ 0].input 00112233445566778899aabbccddeeff
round[ 0].k_sch 000102030405060708090a0b0c0d0e0f
round[ 1].start 00102030405060708090a0b0c0d0e0f0
round[ 1].s_box 63cab7040953d051cd60e0e7ba70e18c
round[ 1].s_row 6353e08c0960e104cd70b751bacad0e7
round[ 1].m_col 5f72641557f5bc92f7be3b291db9f91a
round[ 1].k_sch d6aa74fdd2af72fadaa678f1d6ab76fe
round[ 2].start 89d810e8855ace682d1843d8cb128fe4
round[ 2].s_box a761ca9b97be8b45d8ad1a611fc97369
round[ 2].s_row a7be1a6997ad739bd8c9ca451f618b61
round[ 2].m_col ff87968431d86a51645151fa773ad009
round[ 2].k_sch b692cf0b643dbdf1be9bc5006830b3fe
round[ 3].start 4915598f55e5d7a0daca94fa1f0a63f7
round[ 3].s_box 3b59cb73fcd90ee05774222dc067fb68
round[ 3].s_row 3bd92268fc74fb735767cbe0c0590e2d
round[ 3].m_col 4c9c1e66f771f0762c3f868e534df256
round[ 3].k_sch b6ff744ed2c2c9bf6c590cbf0469bf41
round[ 4].start fa636a2825b339c940668a3157244d17
round[ 4].s_box 2dfb02343f6d12dd09337ec75b36e3f0
round[ 4].s_row 2d6d7ef03f33e334093602dd5bfb12c7
round[ 4].m_col 6385b79ffc538df997be478e7547d691
round[ 4].k_sch 47f7f7bc95353e03f96c32bcfd058dfd
round[ 5].start 247240236966b3fa6ed2753288425b6c
round[ 5].s_box 36400926f9336d2d9fb59d23c42c3950
round[ 5].s_row 36339d50f9b539269f2c092dc4406d23
round[ 5].m_col f4bcd45432e554d075f1d6c51dd03b3c
round[ 5].k_sch 3caaa3e8a99f9deb50f3af57adf622aa
round[ 6].start c81677bc9b7ac93b25027992b0261996
round[ 6].s_box e847f56514dadde23f77b64fe7f7d490
round[ 6].s_row e8dab6901477d4653ff7f5e2e747dd4f
round[ 6].m_col 9816ee7400f87f556b2c049c8e5ad036
round[ 6].k_sch 5e390f7df7a69296a7553dc10aa31f6b
round[ 7].start c62fe109f75eedc3cc79395d84f9cf5d
round[ 7].s_box b415f8016858552e4bb6124c5f998a4c
round[ 7].s_row b458124c68b68a014b99f82e5f15554c
round[ 7].m_col c57e1c159a9bd286f05f4be098c63439
round[ 7].k_sch 14f9701ae35fe28c440adf4d4ea9c026
round[ 8].start d1876c0f79c4300ab45594add66ff41f
round[ 8].s_box 3e175076b61c04678dfc2295f6a8bfc0
round[ 8].s_row 3e1c22c0b6fcbf768da85067f6170495
round[ 8].m_col baa03de7a1f9b56ed5512cba5f414d23
round[ 8].k_sch 47438735a41c65b9e016baf4aebf7ad2
round[ 9].start fde3bad205e5d0d73547964ef1fe37f1
round[ 9].s_box 5411f4b56bd9700e96a0902fa1bb9aa1
round[ 9].s_row 54d990a16ba09ab596bbf40ea111702f
round[ 9].m_col e9f74eec023020f61bf2ccf2353c21c7
round[ 9].k_sch 549932d1f08557681093ed9cbe2c974e
round[10].start bd6e7c3df2b5779e0b61216e8b10b689
round[10].s_box 7a9f102789d5f50b2beffd9f3dca4ea7
round[10].s_row 7ad5fda789ef4e272bca100b3d9ff59f
round[10].k_sch 13111d7fe3944a17f307a78b4d2b30c5
round[10].output 69c4e0d86a7b0430d8cdb78070b4c55a
```

Anexo 5.C:
Descripción Asíncrona en Balsa del Algoritmo de Rijndael para desencriptación

```
import [balsa.types.basic]
import [RondaInicial_dec]
import [RondaNormal_dec]
import [SubclaveRonda_dec]

procedure  invronda_recursiva (parameter nivel: cardinal;input nr:nibble; input
d15,d14,d13,d12,d11,d10,d9,d8,d7,d6,d5,d4,d3,d2,d1, d0,k15, k14,k13,k12,k11,k10,k9,k8,k7,k6,k5,k4,k3,k2,k1,k0 :
byte; output nextnr: nibble; output s15,s14,s13,s12,s11,s10,s9,s8, s7,s6,
s5,s4,s3,s2,s1,s0,sk15,sk14,sk13,sk12,sk11,sk10,sk9,sk8, sk7,sk6,sk5, sk4,sk3,sk2,sk1,sk0 : byte) is

variable cs15,cs14,cs13,cs12,cs11,cs10,cs9,cs8,cs7,cs6,cs5,cs4,cs3,cs2,cs1,cs0 :byte
variable ck15,ck14,ck13,ck12,ck11,ck10,ck9,ck8,ck7,ck6,ck5,ck4,ck3,ck2,ck1,ck0 : byte
channel chs15,chs14,chs13,chs12,chs11,chs10,chs9,chs8,chs7,chs6,chs5,chs4,chs3,chs2,chs1,chs0 : byte
channel chk15,chk14,chk13,chk12,chk11,chk10,chk9,chk8,chk7,chk6,chk5,chk4,chk3,chk2,chk1,chk0 : byte
channel chnr : nibble
variable SigNr : nibble

begin

if nivel= 0 then  print error, "El parámetro nivel no debe ser 0"

|  nivel= 1 then

begin

-- Ronda Inicial: AddRoundKey,InvShiftRow, InvByteSub, subclave de Ronda Inversa
invrondainicial (d15,d14,d13,d12,d11,d10,d9,d8,d7,d6,d5,d4,d3,d2,d1,d0,k15,k14,k13,k12,k11, k10,k9,k8, k7, k6,
k5,k4, k3, k2, k1,k0, ->cs15, ->cs14,->cs13,->cs12,->cs11,->cs10,->cs9,->cs8,->cs7,->cs6,->cs5,->cs4,->cs3,->cs2,
->cs1,->cs0 ) ||

subclaverondainv (nr,k15,k14,k13,k12,k11,k10,k9,k8,k7,k6,k5,k4,k3,k2,k1,k0,->SigNr,->ck15,->ck14,->ck13,-
>ck12,->ck11,-> ck10,->ck9,->ck8,->ck7,->ck6,->ck5,->ck4,->ck3,->ck2,->ck1,->ck0) ;

-- Salida de subclave de ronda siguiente
sk15<-ck15 || sk14<-ck14 || sk13<-ck13 || sk12<-ck12 || sk11<-ck11 || sk10<-ck10 || sk9<-ck9 || sk8<-ck8 ||
sk7<-ck7 || sk6<-ck6 || sk5<-ck5 || sk4<-ck4 || sk3<-ck3  || sk2<-ck2 || sk1<-ck1 || sk0<-ck0 ||

nextnr <-  SigNr  ||

-- Adición de Subclave de ronda inicial para generar bloque  de datos de segunda ronda
s15<- cs15 xor ck15  ||
s14<- cs14 xor ck14  ||
s13<- cs13 xor ck13  ||
s12<- cs12 xor ck12  ||
s11<- cs11 xor ck11  ||
s10<- cs10 xor ck10  ||
s9<- cs9 xor ck9  ||
s8<- cs8 xor ck8  ||
s7<- cs7 xor ck7  ||
s6<- cs6 xor ck6  ||
s5<- cs5 xor ck5  ||
s4<- cs4 xor ck4  ||
s3<- cs3 xor ck3  ||
s2<- cs2 xor ck2  ||
s1<- cs1 xor ck1  ||
s0<- cs0 xor ck0
```

```
end

| nivel>= 2 then

begin

invronda_recursiva (nivel-1 ,nr,d15,d14,d13,d12,d11,d10,d9,d8,d7,d6,d5,d4,d3,d2,d1,d0,k15,k14,k13,k12,k11,k10,k9,
k8,k7,k6,k5,k4,k3,k2,k1,k0,chnr,chs15,chs14,chs13,chs12,chs11,chs10,chs9,chs8,chs7,chs6,chs5,chs4,chs3,chs2,chs1,
chs0,chk15,chk14,chk13,chk12,chk11,chk10,chk9,chk8,chk7,chk6,chk5,chk4,chk3,chk2,chk1,chk0) ||

invrondanormal (chs15,chs14,chs13,chs12,chs11,chs10,chs9,chs8,chs7,chs6,chs5,chs4,chs3,chs2,chs1,chs0,->cs15,-
>cs14,->cs13,->cs12, ->cs11,->cs10,->cs9,->cs8,->cs7,->cs6,->cs5,->cs4,->cs3,->cs2,->cs1,->cs0) ||

subclaverondainv
(chnr,chk15,chk14,chk13,chk12,chk11,chk10,chk9,chk8,chk7,chk6,chk5,chk4,chk3,chk2,chk1,chk0,->SigNr,->ck15,-
>ck14,->ck13,->ck12,->ck11,->ck10,->ck9,->ck8,->ck7,->ck6,->ck5,->ck4,->ck3,->ck2,->ck1,->ck0) ;

sk15<-ck15 || sk14<-ck14 || sk13<-ck13 || sk12<-ck12 || sk11<-ck11 || sk10<-ck10 || sk9<-ck9 || sk8<-ck8 ||
sk7<-ck7 || sk6<-ck6 || sk5<-ck5 || sk4<-ck4 || sk3<-ck3  || sk2<-ck2 || sk1<-ck1 || sk0<-ck0 ||

nextnr <-  SigNr  ||

s15<- cs15 xor ck15  ||
s14<- cs14 xor ck14  ||
s13<- cs13 xor ck13  ||
s12<- cs12 xor ck12  ||
s11<- cs11 xor ck11  ||
s10<- cs10 xor ck10  ||
s9<- cs9 xor ck9  ||
s8<- cs8 xor ck8  ||
s7<- cs7 xor ck7  ||
s6<- cs6 xor ck6  ||
s5<- cs5 xor ck5  ||
s4<- cs4 xor ck4  ||
s3<- cs3 xor ck3  ||
s2<- cs2 xor ck2  ||
s1<- cs1 xor ck1  ||
s0<- cs0 xor ck0

end  --begin
end  --if
end

procedure main_decrip (input nr:nibble; input
d15,d14,d13,d12,d11,d10,d9,d8,d7,d6,d5,d4,d3,d2,d1,d0,k15,k14,k13,k12,k11, k10, k9, k8,k7, k6, k5,k4,
k3,k2,k1,k0 : byte; output nextnr: nibble; output s15,s14,s13,s12,s11,s10,s9,s8,s7,s6,s5,s4,s3,s2,s1,s0, sk15,sk14,sk13,
sk12, sk11,sk10,sk9,sk8,sk7,sk6,sk5,sk4,sk3,sk2,sk1,sk0 : byte) is

begin

invronda_recursiva (10,nr,d15,d14,d13,d12,d11,d10,d9,d8,d7,d6,d5,d4,d3,d2,d1,d0,k15,k14,k13,k12,k11,
k10,k9,k8,k7,k6,k5, k4, k3,k2,k1,
k0,nextnr,s15,s14,s13,s12,s11,s10,s9,s8,s7,s6,s5,s4,s3,s2,s1,s0,sk15,sk14,sk13,sk12,sk11,sk10,sk9,sk8,sk7,sk6,sk5,sk
4, sk3,sk2, sk1,sk0)

end
```

Anexo 5.D: Vectores de ejemplo para desencriptación

Texto encriptado = 69 c4 e0 d8 6a 7b 04 30 d8 cd b7 80 70 b4 c5 5a

Clave de desencriptación = 13 11 1d 7f e3 94 4a 17 f3 07 a7 8b 4d 2b 30 c5

```
round[ 0].iinput 69c4e0d86a7b0430d8cdb78070b4c55a
round[ 0].ik_sch 13111d7fe3944a17f307a78b4d2b30c5
round[ 1].istart 7ad5fda789ef4e272bca100b3d9ff59f
round[ 1].is_row 7a9f102789d5f50b2beffd9f3dca4ea7
round[ 1].is_box bd6e7c3df2b5779e0b61216e8b10b689
round[ 1].ik_sch 549932d1f08557681093ed9cbe2c974e
round[ 1].ik_add e9f74eec023020f61bf2ccf2353c21c7
round[ 2].istart 54d990a16ba09ab596bbf40ea111702f
round[ 2].is_row 5411f4b56bd9700e96a0902fa1bb9aa1
round[ 2].is_box fde3bad205e5d0d73547964ef1fe37f1
round[ 2].ik_sch 47438735a41c65b9e016baf4aebf7ad2
round[ 2].ik_add baa03de7a1f9b56ed5512cba5f414d23
round[ 3].istart 3e1c22c0b6fcbf768da85067f6170495
round[ 3].is_row 3e175076b61c04678dfc2295f6a8bfc0
round[ 3].is_box d1876c0f79c4300ab45594add66ff41f
round[ 3].ik_sch 14f9701ae35fe28c440adf4d4ea9c026
round[ 3].ik_add c57e1c159a9bd286f05f4be098c63439
round[ 4].istart b458124c68b68a014b99f82e5f15554c
round[ 4].is_row b415f8016858552e4bb6124c5f998a4c
round[ 4].is_box c62fe109f75eedc3cc79395d84f9cf5d
round[ 4].ik_sch 5e390f7df7a69296a7553dc10aa31f6b
round[ 4].ik_add 9816ee7400f87f556b2c049c8e5ad036
round[ 5].istart e8dab6901477d4653ff7f5e2e747dd4f
round[ 5].is_row e847f56514dadde23f77b64fe7f7d490
round[ 5].is_box c81677bc9b7ac93b25027992b0261996
round[ 5].ik_sch 3caaa3e8a99f9deb50f3af57adf622aa
round[ 5].ik_add f4bcd45432e554d075f1d6c51dd03b3c
round[ 6].istart 36339d50f9b539269f2c092dc4406d23
round[ 6].is_row 36400926f9336d2d9fb59d23c42c3950
round[ 6].is_box 247240236966b3fa6ed2753288425b6c
round[ 6].ik_sch 47f7f7bc95353e03f96c32bcfd058dfd
round[ 6].ik_add 6385b79ffc538df997be478e7547d691
round[ 7].istart 2d6d7ef03f33e334093602dd5bfb12c7
round[ 7].is_row 2dfb02343f6d12dd09337ec75b36e3f0
round[ 7].is_box fa636a2825b339c940668a3157244d17
round[ 7].ik_sch b6ff744ed2c2c9bf6c590cbf0469bf41
round[ 7].ik_add 4c9c1e66f771f0762c3f868e534df256
round[ 8].istart 3bd92268fc74fb735767cbe0c0590e2d
round[ 8].is_row 3b59cb73fcd90ee05774222dc067fb68
round[ 8].is_box 4915598f55e5d7a0daca94fa1f0a63f7
round[ 8].ik_sch b692cf0b643dbdf1be9bc5006830b3fe
round[ 8].ik_add ff87968431d86a51645151fa773ad009
round[ 9].istart a7be1a6997ad739bd8c9ca451f618b61
round[ 9].is_row a761ca9b97be8b45d8ad1a611fc97369
round[ 9].is_box 89d810e8855ace682d1843d8cb128fe4
round[ 9].ik_sch d6aa74fdd2af72fadaa678f1d6ab76fe
round[ 9].ik_add 5f72641557f5bc92f7be3b291db9f91a
round[10].istart 6353e08c0960e104cd70b751bacad0e7
round[10].is_row 63cab7040953d051cd60e0e7ba70e18c
round[10].is_box 00102030405060708090a0b0c0d0e0f0
round[10].ik_sch 000102030405060708090a0b0c0d0e0f
round[10].ioutput 00112233445566778899aabbccddeeff
```

Anexo 5.E:
Descripción Asíncrona en Balsa del Algoritmo de Rijndael para encriptación/desencriptación

```
import [balsa.types.basic]
import [Rijndael_Encrip_Recursivo]
import [Rijndael_Decrip_Recursivo]

procedure Encrip_Decrip (input modo: bit; input nr:nibble; input
d15,d14,d13,d12,d11,d10,d9,d8,d7,d6,d5,d4,d3,d2, d1,d0, k15, k14, k13,
k12,k11,k10,k9,k8,k7,k6,k5,k4,k3,k2,k1,k0 : byte; output nextnr: nibble; output s15,s14, s13, s12,s11,
s10, s9,s8,s7,
s6,s5,s4,s3,s2,s1,s0,sk15,sk14,sk13,sk12,sk11,sk10,sk9,sk8,sk7,sk6,sk5,sk4,sk3,sk2,sk1,sk0 : byte) is

variable enc_dec : bit

begin

modo -> enc_dec ;

        if enc_dec = 0b_0 then

main_encrip
(nr,d15,d14,d13,d12,d11,d10,d9,d8,d7,d6,d5,d4,d3,d2,d1,d0,k15,k14,k13,k12,k11,k10,k9,k8,k7,k6,
k5,k4, k3, k2,
k1,k0,nextnr,s15,s14,s13,s12,s11,s10,s9,s8,s7,s6,s5,s4,s3,s2,s1,s0,sk15,sk14,sk13,sk12,sk11, sk10,
sk9,sk8,sk7,sk6,sk5, sk4,sk3,sk2,sk1,sk0)

        | enc_dec = 0b_1 then

main_decrip
(nr,d15,d14,d13,d12,d11,d10,d9,d8,d7,d6,d5,d4,d3,d2,d1,d0,k15,k14,k13,k12,k11,k10,k9,k8,k7, k6, k5,
k4, k3, k2,
k1,k0,nextnr,s15,s14,s13,s12,s11,s10,s9,s8,s7,s6,s5,s4,s3,s2,s1,s0,sk15,sk14,sk13,sk12,sk11,sk10,sk9,
sk8, sk7,sk6, sk5, sk4,sk3,sk2,sk1,sk0)

        end
end
```

Capítulo 6: Conclusiones y trabajo futuro

6.1. Conclusiones

6.1.1 El estado del arte indica que el diseño asíncrono es un área creciente de investigación que tiene un impacto importante en el desarrollo de herramientas de diseño como también de circuitos integrados comerciales. Un ejemplo de lo anterior lo constituyen los esfuerzos de la Comunidad Económica Europea orientados al apoyo de la investigación en esta área de conocimiento, lo que ha derivado en la aparición de nuevas herramientas de diseño, como también la vinculación de universidades, institutos de investigación y organizaciones industriales al desarrollo de objetivos concretos.

6.1.2. Se logró la síntesis asíncrona del algoritmo de Rijndael para claves de 128 bits (AES-128) de acuerdo con el estándar *FIPS-197* del NIST de Estados Unidos. Para ello se utilizó el sistema de especificación y síntesis asíncrona conocido como Balsa, desarrollado por el grupo *APT* (*Advanced Processor Technologies*) de la Universidad de Manchester en Inglaterra. Los bloques que conforman el algoritmo han sido especificados, en cada caso, con arquitecturas optimizadas a nivel de circuito lógico.

6.1.3. Los resultados obtenidos en la simulación funcional del algoritmo de Rijndael (AES-128), indican que el desempeño de la implementación está determinado, pricipalmente, por los tiempos de retardo asociados con los circuitos que implementan las transformaciones de sustitución de byte: *ByteSub, InvByteSub y SubWord* implementadas mediante funciones Sbox.

6.1.4. El mejor desempeño en la simulación del algoritmo de Rijndael, de acuerdo con los resultados obtenidos con el sistema Balsa, para los procesos de encriptación y desencriptación, se obtuvo al utilizar las transformaciones de sustitución de byte que usaron funciones Sbox basadas en tablas.

6.1.5. La implementación completa del algoritmo de Rijndael en FPGA no pudo ser lograda debido a las limitaciones que presentan estos dispositivos a nivel de recursos de interconexión para sistemas asíncronos.

6.1.6. En la literatura revisada no se han encontrado aun reportes de resultados de implementaciones asíncronas del estándar AES sobre *FPGAs*. Este trabajo presenta los primeros reportes sobre implementación asíncrona en FPGA, a nivel de funciones individuales que conforman el algoritmo de Rijndael, implementadas todas ellas con codificación *dual-rail* y arquitectura balanceada, tanto para encriptación como para desencriptación. Los resultados de la implementación en la FPGA *Virtex XCV2P30-6ff896* de Xilinx se consignan en la tabla 6.1. La columna de '*Reporte estimado total'* corresponde a tiempos 'supuestos' de encriptación y desencriptación bajo la suposición que el circuito total se ha ruteado por completo en la FPGA.

Tabla 6.1: Tiempos de retardo de implementación de las funciones que conforman el algoritmo de Rijndael para encriptación y desencriptación utilizando codificación *dual-rail* balanceada sobre la FPGA *Virtex XCV2P30-6ff896* de Xilinx

Proceso	***Transformación***	***Retardo***	***Retardo estimado total***
Encriptación	*SubclaveRonda_enc*	*371.783ns*	***4827.757ns***
	ByteSub	*217.268ns*	
	ShiftRow	*31.788ns*	
	MixColumn	*86.701ns*	
	AddRoundKey	*33.290ns*	
Desencriptación	*SubclaveRonda_dec*	*352.562ns*	***4191.052ns***
	InvByteSub	*212.626ns*	
	InvShiftRow	*31.788ns*	
	InvMixColumn	*176.884ns*	
	AddRoundKey	*33.290ns*	

6.1.7. Hasta el momento sólo se han reportado dos implementaciones asíncronas para encriptación basados en *ASICs,* tales reportes no mencionan resultados de desempeño de bloques individuales del algoritmo. La tabla 6.2 muestra los reportes de las implementaciones de Rijndael en *ASIC* y también los resultados de simulación funcional del algoritmo obtenidos en este trabajo de tesis utilizando el sistema Balsa.

Tabla 6.2: Implementaciones asíncronas del algoritmo de Rijndael

Referencia	***Desempeño***	***Tamaño (mm^2)***	***Comentarios especiales***
[1] (*ASIC*)	215 Mbits/seg	0.088 (clave 128 bits)	V_{dd} = 1.2v Tecnología CMOS 0.13µm Codificación 1 de N Modelo de circuito *QDI* Arquitectura Balanceada Solo se implementa el proceso de encriptación
	721 Mbits/seg	0.192 (clave 192 bits)	
	892 Mbits/seg	0.427 (clave 256 bits)	
[2],[3] (*ASIC*)	800 ns	1.897 (clave 128 bits)	Tecnología CMOS 0.35µm Modelo de circuito *SI* Codificación *dual-rail* Arquitectura Balanceada Sólo se presentan resultados de simulación para encriptación
En este trabajo de Tesis	340200 ns (Encriptación) 426900 ns (desencriptación)		Resultados de Simulación con Balsa para los procesos de encriptación/desencriptación Claves de 128 bits Implementación en *FPGA Virtex XCV2P30-6ff896* de Xilinx para las funciones individuales Funciones individuales implementadas con codificación *dual-rail*, y 1-de-4 *Dual-rail* de Arquitectura Balanceada

6.1.8. Aunque existe una buena integración entre la herramienta de especificación asíncrona Balsa y la plataforma de desarrollo *Xilinx ISE 9.2i,* es claro que las FPGAs utilizadas para implementar el circuito final evidencian dificultades para mapear circuitos asíncronos, como lo corroboran los resultados en los que más del 55% del retardo de una implementación asíncrona es causado por las rutas de interconexión de los circuitos lógicos.

6.1.9. Los resultados obtenidos en las simulaciones en Balsa muestran que las transformaciones de sustitución de byte, *ByeSub* e *InvByteSub* basadas en tablas presentan mejor desempeño que las transformaciones basadas en la metodología PPRM. Sin embargo, los resultados reportados sobre los retardos de implementación en FPGA medidos con ISE 9.2i de Xilinx muestran que las implementaciones de sustitución de byte basadas en Sbox combinatorias, PPRM, muestran mejor desempeño que las Sbox implementadas con tablas. Esta contradicción, respecto de las simulaciones previas con el sistema Balsa, refuerzan el hecho que el sistema de desarrollo de Xilinx y las FPGAs están orientados al diseño síncrono.

6.1.10. Los resultados de las implementaciones de las distintas transformaciones que conforman el algoritmo de Rijndael sintetizadas con el sistema Xilinx ISE 9.2i, indican que los circuitos implementados en FPGA registran mayor retardo en las rutas de interconexión (superiores al 58%) que en los elementos lógicos que conforman cada función, lo anterior fortalece la idea que indica que el estado actual de las FPGAs de Xilinx y su ambiente de diseño no han sido orientados a la implementación eficiente de circuitos asíncronos.

6.1.11. A pesar que el sistema Balsa es la herramienta no comercial mas avanzada para la especificación y síntesis de circuitos digitales asíncronos, aun no está completamente desarrollada y continúa en proceso de depuración.

6.1.12. El desarrollo de herramientas para *test* y verificación formal de los circuitos asíncronos es aún incipiente, lo cual constituye una de las principales desventajas respecto del estilo de diseño síncrono.

6.2. Trabajo Futuro

Dadas las condiciones de avance de las herramientas de diseño asíncrono, el trabajo debe orientarse en el corto plazo a estudiar estrategias que permitan adecuar eficientemente las plataformas de hardware comercial a las metodologías asíncronas existentes.

Desde el punto de vista del algoritmo de Rijndael, el trabajo futuro estará orientado a la implementación del mismo en un circuito integrado (*ASIC*), adicionando modos de configuración que permitan mejorar al máximo la seguridad del mismo ante cualquier tipo de ataque no invasivo.

Son muchas las sub-áreas que abarca el diseño asíncrono. Cada una de ellas constituye un reto importante de investigación y desarrollo. A continuación se presenta una lista con tópicos de interés:

- Arquitecturas Mixtas síncronas/asíncronas, interfaces y circuitos.
- Técnicas de diseño, síntesis y verificación para sistemas GALS.
- Interacción síncrona-asíncrona a diferentes niveles.
- Lógica asíncrona de alta-velocidad/baja-potencia para memorias e interconexiones.
- Diseño de alto-nivel y síntesis de circuitos auto-temporizados.
- Diseño físico de lógica sin reloj y *pipelines*.
- Métodos Formales para desempeño y análisis de diseños asíncronos.
- Test, confiabilidad, seguridad y tolerancia a la radiación.
- CAD para diseño y validación asíncrona.
- *Asynchronous System-on-Chip (SoC), System-in-Package (SiP).*
- Nuevas Arquitecturas asíncronas.
- Asincronismo y tolerancia de latencia en diseño a nivel de sistema.
- Motivaciones de estudios de caso, comparaciones y aplicaciones.
- Diseño de sistemas embebidos con arquitecturas/implementaciones asíncronas.
- Diseño asíncrono para manufactura.

Referencias

[1] Bouesse F., Renaudin M., Germain F. *"Asynchronous AES Crypto- Processor Including Secured and Optimized Blocks"*. TIMA Laboratory, Grenoble, France; SGDN/ DCISS, Paris, France. Journal of Integrated Circuits and Systems, Vol 1, No 1, pp 5-13. March 2004.

[2] Shang D., Burns F., Bystrov A., Koelmans A., Sokolov D., Yakovlev A. *"A Low and Balanced Power Implementation of the AES Security Mechanism Using Self-Timed Circuits"*. E. Macii et al. (Eds.): PATMOS 2004, LNCS 3254, pp. 471–480, 2004.c. Springer-Verlag Berlin Heidelberg 2004.

[3] Shang D., Burns F., Bystrov A., Koelmans A., Sokolov D., Yakovlev A. ***"****High-security asynchronous circuit implementation of AES"*. IEE Proceedings online no. 20050088, doi:10.1049/ip-cdt:20050088. IEE Proc.-Comput. Digit. Tech., Vol. 153, No. 2, March 2006.