ESCUELA SUPERIOR POLITECNICA DEL LITORAL

Facultad de Ingeniería en Electricidad y Computación

IMPLEMENTACIÓN DEL SISTEMA DE CIFRADO DE DATOS NORMALIZADO UTILIZANDO LA TECNOLOGÍA DE ARREGLOS DE PUERTAS PROGRAMABLES POR CAMPOS

TESIS DE GRADO

Previo a la obtención del Título de:

INGENIERO EN ELECTRONICA Y TELECOMUNICACIONES

Presentada por:

ANDRES RAFAEL OCHOA CEPEDA

EDUARDO DANIEL ICAZA VILLALVA

GUAYAQUIL – ECUADOR

AÑO

2008

AGRADECIMIENTO

Agradecemos a Dios, por Su gracia que nos sustento en todo momento; a nuestros padres, por su esfuerzo constante y abnegado; a nuestros amigos por su afecto, y ánimo; y a nuestra Directora por su guía y paciencia.

DEDICATORIA

A Dios, a nuestras familias y amigos.

TRIBUNAL DE GRADUACIÓN

Ing. Holger Cevallos SUBDECANO DE LA FIEC	Ing. Sara Ríos DIRECTORA DE TESIS
Ing. Hugo Villavicencio VOCAL PRINCIPAL	Ing. Ludmila Gorenkova VOCAL PRINCIPAL

DECLARACIÓN EXPRESA

"La responsabilidad del contenido de esta Tesis de Grado, nos corresponde exclusivamente; y el patrimonio intelectual de la misma a la Escuela Superior Politécnica del Litoral".

(Reglamento de Graduación de la ESPOL).

Andrés R. Ochoa Cepeda

Eduardo D. Icaza Villalva

RESUMEN

La meta principal de este trabajo es la implementación de un cifrador de datos, aplicando el Cifrado Normalizado de Datos (DES, *Data Encryption Standard*), en un circuito integrado de Arreglos de Puertas Programables por Campos (FPGA, *Field Programmable Gate Array*), empleando el Lenguaje de Descripción de Hardware de Circuitos Integrados de Alta Velocidad (VHDL, *Very High Speed Integrated Circuit Hardware Description Language*); valiéndonos de las técnicas y modelos de diseño, tanto de bloques de lógica combinatorial como de máquinas algorítmicas sincrónicas (ASM), aprendidos en los cursos de Sistemas Digitales. La motivación de este tipo de implementación es la rápidamente creciente importancia que cobra la Seguridad Informática y el potencial de los circuitos integrados modernos. Este trabajo queda como precedente en la institución para nuevos proyectos que podrían implementar protocolos de seguridad más avanzados o procedimientos más veloces.

ÍNDICE GENERAL

ABREVIATURAS

CAD	Diseño Asistido por Computador
CBC	Cifrado Realimentado
CFB	Cifrado con Realimentación
CLB	Bloque Lógico Configurable
CTR	Cifrado en Modo Contador
DCM	Administrador de Reloj Digital
DES	Cifrado de Datos Normalizado
DoS	Denegación de Servicio (ataque)
ECB	Modo de Cifrado Directo de algoritmos de bloque
FPGA	Arreglo de Puertas Programables por Campos
FIPS	Estándares Federales de Procesamiento de la Información
IBM	Internacional Business Machines
IOB	Bloque de Entradas y Salidas
IP	Protocolo de Internet
ISO	Organización Internacional para la Estandarización
JTAG	Norma IEEE 1149
MDC	Detector de Códigos Modificados
MD5	Algoritmo de Resumen de Mensaje 5
NBS	Oficina Nacional de Estandarización de los E.E.U.U.
NITS	Instituto Nacional de Estándares y Tecnologías de los E.E.U.U.
NSA	Agencia de Seguridad Nacional del los E.E.U.U.
OFB	Cifrado con Realimentación de la Salida
OSI	Modelo de Interconexión de Sistemas Abiertos

PAR	Ubicación y Enrutamiento
PLD	Dispositivos Lógicos Programables
SHA1	Algoritmo de Resumen de Mensaje Seguro – 1
SSL	Capa de Conexión Segura
TCP	Protocolo de Capa de Transporte
UART	Recepción – Transmisión Asíncrona Universal
UDP	Protocolo de Datagrama de Usuario
VHDL	Lenguaje de Descripción de Hardware de Circuitos Integrados de Muy Alta Velocidad
XST	Herramientas de Sintetización de Xilinx

SIMBOLOGÍA

A_c	Alfabeto de texto cifrado
A_m	Alfabeto de texto plano
c	Texto cifrado, o bloque de texto cifrado
C	Espacio de textos cifrados
$D_k(\bullet)$	Función de descifrado, usando la llave k
D	Familia de todas las funciones de descifrado D_k
$E_k(\bullet)$	Función de cifrado, usando la clave k
E	Familia de todas las funciones de cifrado E_k
k	Clave secreta de un criptosistema simétrico
K	Espacio de claves
K_p	Clave privada de un criptosistema asimétrico
K_P	Clave pública de un criptosistema asimétrico
m	Mensaje en texto plano
M	Espacio de textos planos
$\mathbb{N}_m$	Conjunto $\{1,2,3,...,m\}$
p	Bloque de texto plano
$r(\bullet)$	Función de resumen, o dispersión (hash)
$\mathbb{Z}_n$	Conjunto $\{0,1,2,...,n-1\}$

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

INTRODUCCIÓN

La Seguridad Informática, en especial los mecanismo que usan Criptografía para la transferencia segura de datos, y la Electrónica, en particular los Sistemas Digitales, son dos grandes campos que en los últimos años han tenido un desarrollo significativo y vertiginoso, ya que han captado la atención de un gran número de investigadores y entidades por su relevancia; el primero por su dedicación a la seguridad que en la actualidad las Telecomunicaciones demandan, y el segundo por su énfasis en proveer soluciones cada vez más capaces y compactas para casi cualquier ámbito o tipo de aplicación.

Este trabajo pertenece a un tipo de implementación que conjuga ambos campos: *la implementación de algoritmos criptográficos en hardware reconfigurable*. La tendencia a esta clase de aplicaciones es relativamente nueva y responde básicamente a dos necesidades, de mecanismos de seguridad de alta velocidad, y de mecanismos de seguridad embebidos. El primero para aplicarlos en canales de información troncales y el segundo para poder integrarlos como propiedad adicional de equipos de telecomunicaciones.

Más concretamente, nuestro trabajo consiste en la implementación del algoritmo criptográfico de Cifrado Estándar de Datos (DES, *Data Encryption Standard*) empleando Arreglos de Puertas Programables por Campos (FPGA, *Field*

Programmable Gate Array), haciendo uso del Lenguaje de Descripción de Hardware de Circuitos Integrados de Alta Velocidad (VHDL, *Very High Speed Integrated Circuit Hardware Description Language*). Este cometido lo logramos acudiendo a las especificaciones publicadas del DES y aplicando las técnicas de diseño aprendidas en los cursos de Sistemas Digitales, para finalmente sintetizarlo en una plataforma de desarrollo basada en un FPGA Virtex-II Pro de Xilinx Inc.

Para comprobar el buen funcionamiento del cifrador DES, se implementó un módulo de transmisión serial asincrónico estándar (8-N-1) para enviar los datos cifrados al computador; y al cifrador DES se lo dispuso en modo directo (ECB, *Electronic Codebook*). En el computador se instaló un software de monitoreo del puerto serial que permite observar los octetos recibidos en un formato conveniente (para nuestro caso, hexadecimal).

El documento está organizado en dos partes, cada una conformada por dos capítulos. Los dos primeros capítulos introducen al lector en el campo de la Seguridad Informática, la Criptografía y finalmente presentan al algoritmo DES, sus características y operación. Los dos siguientes capítulos explican los procedimientos usados en la implementación, las herramientas y plataformas tecnológicas utilizadas, el diseño del cifrador y los resultados obtenidos.

I CAPÍTULO

1 CONCEPTOS GENERALES DE SEGURIDAD INFORMÁTICA

En este capítulo introduciremos al lector en el contexto de la Seguridad Informática y la Criptografía.

Iniciaremos con la exposición de los conceptos y definiciones fundamentales de Seguridad de la Información; veremos de manera general los riesgos de seguridad y cómo afrontarlos. Continuaremos con la presentación de los Protocolos de Seguridad; veremos sus propiedades y ventajas; y abordaremos su aplicación en las redes actuales. Finalmente nos concentraremos en estudiar la Criptografía y los Sistemas Secretos; estudiaremos su clasificación en las ciencias, el modelo de un criptosistema y las técnicas criptográficas de seguridad más usadas.

1.1 Introducción a la Seguridad Informática

En sencillas palabras, *Seguridad Informática* o *Seguridad de la Información* es

la protección, resguardo o cautela de información. Según el Diccionario de la Lengua Española, información se define como un *«elemento de conocimiento susceptible de ser representado mediante signos convencionales o símbolos, y conservado, tratado o comunicado.»*[1] De esta manera, podemos definir, más formalmente, a la Seguridad de la Información como el conjunto de reglas, planes y acciones que permiten asegurar la información manteniendo las propiedades de confidencialidad, integridad y disponibilidad de la misma. Siendo,

- *confidencialidad,* que la información sea accesible sólo para aquéllos que están autorizados;
- *integridad*, que la información sólo puede ser creada y modificada por quien esté autorizado a hacerlo; y
- *disponibilidad*, que la información debe ser accesible para su consulta o modificación cuando se requiera.

Esto nos plantea dos marcos para la Seguridad de la Información:

- proteger la información almacenada, y
- resguardar la información mientras está siendo transmitida (y recibida) durante una conversación o sesión de comunicación.

1.1.1 Idea Básica de la Seguridad Informática

Para entender el flujo normal (sin anomalías) de la información vamos a

basarnos en el *Modelo de Shannon*[3]. Un flujo de información se lo puede representar esquemáticamente como lo muestra figura 1.1.

Figura 1.1 Diagrama esquemático general de la transferencia ideal de información.

Tal como se muestra en la figura, un sistema donde se transfiere o intercambia información, en general, consta esencialmente de cinco elementos:

1. Una *fuente de información,* la cual produce un mensaje o secuencia de mensajes (m) a ser comunicada al terminal de recepción.
2. Un *transmisor,* el cual opera de alguna manera sobre el mensaje para producir una señal (s) adecuada para transmitirse por el canal.
3. El *canal,* que es meramente el medio usado para transportar o conducir la señal desde el transmisor hasta el receptor.
4. El *receptor,* que comúnmente ejecuta la operación inversa a aquella hecha por el trasmisor, reconstruyendo el mensaje a partir de la señal.
5. El *destino,* que es la persona (o equipo) para quién el mensaje debe llegar.

En este proceso, la fuente de información emite el mensaje; éste es tomado por el transmisor, el cual lo transforma en algún tipo de señal y la envía a

través del canal; la señal viaja a través del canal; y ésta es recogida por el receptor, el cual la decodifica para reconstruir el mensaje; finalmente el destino recibe el mensaje. Una falla de seguridad en el sistema se puede encontrar en cualquier punto entre la fuente y el destino.

Como ejemplos de un flujo de información podemos citar:

- El intercambio de datos entre la unidad central de procesamiento de un sistema digital y un periférico:
    - El procesador es la fuente de información;
    - los buses de datos y de dirección son el canal; y
    - el periférico es el destino, dependiendo del caso.
- Un enlace de radio:
    - el equipo de la estación local es la fuente de información;
    - el MODEM de la estación local es el trasmisor;
    - el aire es el canal;
    - el MODEM de la estación remota es el receptor; y
    - el equipo de la estación remota es el destino.

1.1.2 Anomalías en la Comunicación

Se entiende por Ataques a la seguridad las acciones que pueden comprometer la seguridad de la información que pertenece a un sistema.

En el argot de la seguridad informática, se suelen utilizar nombres propios para los agentes que participan en un sistema, de acuerdo al papel que desempeñan. Así, tenemos:

Alicia Primer participante en el sistema. Generalmente la fuente de la información.

Bob Segundo participante en el sistema. Generalmente el destino de la información.

Eva Fisgón. (En Inglés, *Eve*, de *eavesdropper.*) Intruso que escucha sin autorización en el canal.

Manuel Atacante Malicioso. (En Inglés, *Mallory*, de *malicious attacker.*) Atacante que modifica los datos y actúa en el medio.

Oscar Oponente. Normalmente es un equivalente de Manuel.

Arturo Árbitro. (En Inglés, *Trent*, de *trusted arbitrator.*) Intermediario confiable.

Guillermo Guardián. (En Inglés, *Walter*, de *warden.*) Protege a Alicia y a Bob.

Víctor Verificador.

LA INTERRUPCIÓN

Este es uno de los problemas más grandes que hay. La interrupción de la transmisión de la información puede ser ocasionada por fallo del canal o de algún elemento del sistema, ya sea de forma natural o intencional. Este ataque afecta la *disponibilidad* de la información.

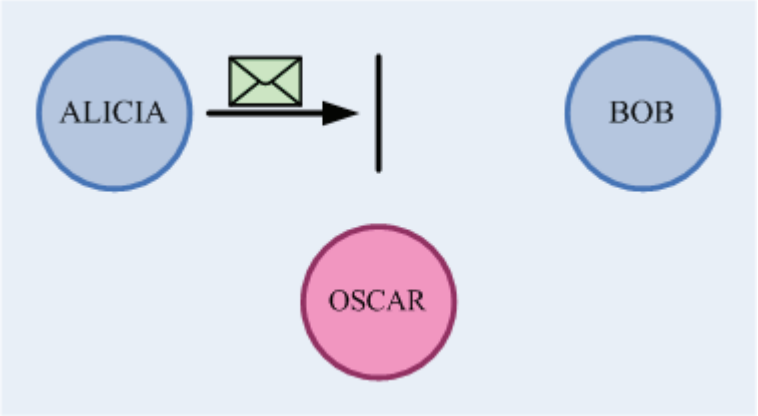

Figura 1.2 Ilustración de la interrupción.

Ejemplos:

- Cuando en un ordenador existe un virus que inhibe el funcionamiento de una aplicación.
- La comunicación vía radio, que es sensible al ruido; si un atacante genera demasiada interferencia, la comunicación se pierde.

LA INTERCEPTACIÓN

Sucede cuando un intruso escucha todo lo que pasa por el canal sin alterar nada. Esto es algo muy común, ya que muchas de las transmisiones son enviadas mediante protocolos que son conocidos por todos y generalmente los mensajes viajan tal cual se generan. Este ataque afecta la *confidencialidad* de la información.

Ejemplos:

- Los monitores o analizadores de protocolos (*sniffers*), que leen el espacio de memoria que usa el sistema operativo del ordenador para enviar o recibir

datos por algún puerto (serial, de red LAN, etc.).

- La transmisión radial ya que las ondas de radio se difunden en el medio y éstas pueden ser receptadas por cualquiera que tenga el equipo receptor adecuado.

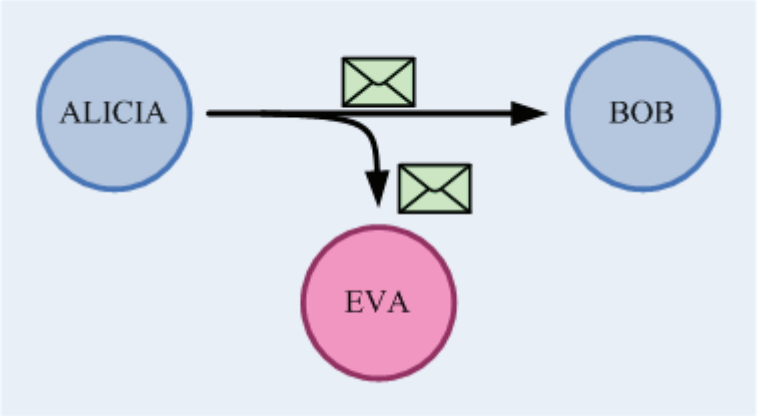

Figura 1.3 Ilustración de la interceptación.

LA MODIFICACIÓN

Es cuando un ente no autorizado no sólo accede a una parte de la información, sino que además es capaz de modificar su contenido. Este ataque afecta la *integridad* de la información.

Ejemplos:

- La alteración de ficheros de datos y alteración de programas, por algún intruso, virus o *programa espía.*
- La modificación de mensajes, tales como el correo electrónico, mientras son transmitidos por Internet.

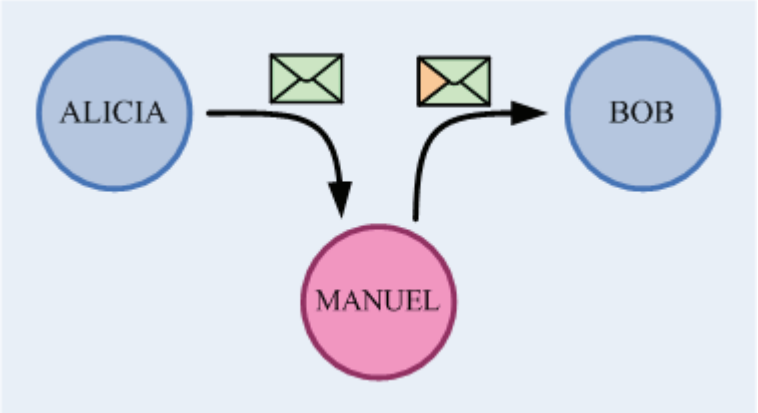

Figura 1.4 Ilustración de la alteración.

LA FABRICACIÓN

El atacante envía información haciéndose pasar por un usuario legítimo. Este ataque afecta la *autenticidad.*

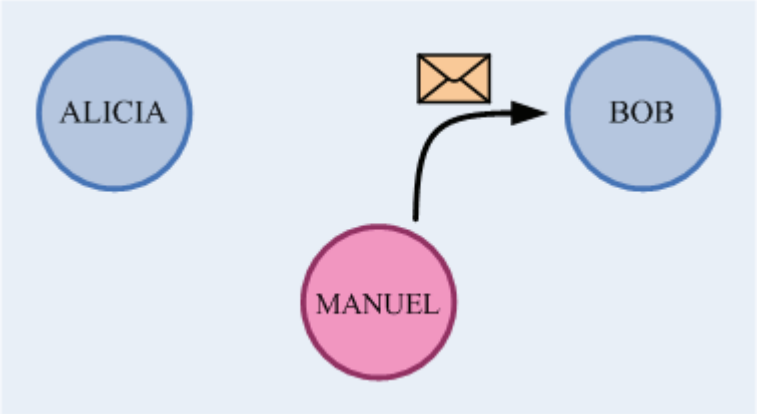

Figura 1.5 Ilustración de la fabricación.

Ejemplos:

- Los virus troyanos, que son programas que engañan al usuario haciéndolo creer que son archivos ejecutables que proceden de un origen legítimo.
- La estafa electrónica (en Inglés, *phishing*), donde el estafador tiene la

capacidad de duplicar una página web para hacer creer al visitante que se encuentra en la página original en lugar de la copiada, con el fin de adquirir información confidencial de forma fraudulenta (contraseñas, información detallada de tarjetas de crédito, etc.)

1.1.3 Definiciones y términos de Seguridad

SISTEMA

Entenderemos por sistema:

- un producto o componente, como es el caso de un protocolo de seguridad, una SmartCard o el hardware de un PC;
- una colección de componentes como los antes mencionados más un sistema operativo, de comunicaciones, y algunos otros componentes que constituyen la infraestructura de una organización;
- lo antes descrito más una o más aplicaciones (contabilidad, cuentas por pagar, etc.) Lo anterior más el personal de IT;
- lo anterior más usuarios internos y gerentes. Lo anterior más clientes y otros usuarios externos;
- lo anterior más el entorno incluyendo el medio, competidores, reguladores y políticos.

SUJETO

Persona física en cualquier rol, incluyendo el de operador, víctima, etc.

PRINCIPAL

Entidad que participa en un sistema de seguridad. Esa entidad puede ser un sujeto, una persona, un rol o una pieza de equipo (PC, SmartCard, etc.) Un grupo es un conjunto de principales. Un rol es una función asumida por diferentes personas en distintos momentos.

VULNERABILIDAD

Propiedad de un sistema o su ambiente, el cual en conjunción con una amenaza interna o externa puede conducir a una falla de seguridad, el cual es un estado de cosas contrario a la política de seguridad del sistema.

SECRETO

Término técnico que se refiere al efecto del mecanismo usado para limitar el número de principales que tienen acceso a la información.

PRIVACIDAD

Habilidad o el derecho de una persona u organización de proteger sus propios secretos.

1.1.4 Tomando medidas de seguridad

Para estimar las necesidades de seguridad de un sistema y evaluar y elegir los productos y políticas de seguridad, se necesita evaluar los siguientes

aspectos en la seguridad de la información:

1. ***Ataques a la seguridad:*** Qué acciones pueden comprometer la seguridad de la información que pertenece a un sistema.
2. ***Mecanismos de seguridad:*** Qué mecanismos hay que implementar para detectar, prevenir o recuperarse de un ataque a la seguridad de la información.
3. ***Servicios de seguridad:*** Qué servicios ofrecer al usuario respecto a la transferencia de información. Los servicios de seguridad tratan de contrarrestar los ataques y para ello hacen uso de los mecanismos de seguridad para proporcionar ese servicio.

Habiendo evaluado estos tres puntos, podemos diseñar el esquema de seguridad más conveniente (eficaz y de costo moderado) para nuestro sistema. Esto lo veremos mejor en la sección **1.2.3.**

1.2 Los protocolos de Seguridad Informática

1.2.1 Definiciones esenciales

Un *protocolo de seguridad* es una serie de pasos, que involucra a dos o más principales, diseñado para realizar una tarea particular.

1. Todos los principales deben conocer los pasos del protocolo de antemano.
2. Todos deben estar de acuerdo en seguir el protocolo.
3. El protocolo no admite ambigüedades.
4. El protocolo debe ser completo –define qué hacer en cualquier circunstancia posible.
5. No debe ser posible hacer más (o aprender más) que lo que el protocolo define.

Un *protocolo criptográfico* es un protocolo que usa funciones criptográficas en algunos o todos los pasos.

Una *arquitectura de seguridad* es un conjunto de protocolos de seguridad.

Una *política de seguridad* es una declaración sucinta de la estrategia de protección de un sistema.

Un *objetivo de seguridad* es una especificación más detallada que define los medios mediante los cuales se implementa una política de seguridad en un producto particular.

1.2.2 Alcances de un protocolo de seguridad

Un solo protocolo de seguridad no lo hace todo, dependiendo del caso, se

necesita un conjunto de protocolos (una arquitectura o sistema de seguridad), donde cada protocolo se encargue de un ámbito o aspecto en particular. Una arquitectura de seguridad debería estar en capacidad de proveer los siguientes servicios al sistema protegido:

- *Confidencialidad,* asegurando que la información almacenada y la transmitida sean asequibles únicamente para lectura por parte de participantes autorizados.
- *Autentificación,* asegurando que el origen de la información es correctamente identificado, con la certeza de que no es falso.
- *Integridad,* asegurando que solo los participantes autorizados estén en la capacidad ya sea para modificar los activos del sistema (configuraciones, información almacenada, etc.) o para transmitir información.
- *"No repudio",* asegurando que ni el remitente ni el destinatario de un mensaje puedan negar que la transmisión de dicho mensaje se dio.
- *Control de acceso,* asegurando que el acceso a los recursos de información sea controlado.
- *Disponibilidad,* asegurando que los recursos necesarios del sistema estén disponibles cuando sea necesario.

Tal como se mencionó al principio de esta sección, no existe un mecanismo único que pueda proveer todos estos servicios; sin embargo, hay uno en particular que cubre la mayoría de los mecanismos de seguridad usados: las *Técnicas Criptográficas*. Este tema será analizado con más detenimiento más

adelante.

1.2.3 El Modelado de Riesgos

El modelado de riesgos es el primer paso a dar cuando uno piensa en el diseño de un sistema seguro. ¿Cuáles son los riesgos reales que se afrontan con el sistema? ¿Con qué frecuencia pueden ocurrir? Y sobre todo ¿Cuál es su costo? y ¿Cuánto estamos dispuestos a invertir para garantizar que no ocurran? Esta actividad requiere de una visión sistémica de lo que pueden significar los riesgos de seguridad. No es suficiente con hacer un listado completo de los posibles riegos de seguridad.

La idea básica es que debemos tomar en cuenta todas las vulnerabilidades o posibles ataques y estimar las pérdidas asociadas a estos ataques desarrollando una expectativa anual de pérdidas. Por ejemplo el riesgo de seguridad puede representar una pérdida de USD 5 000 por cada ataque y se puede estimar que se realicen 10 ataques por año. Entonces la pérdida estimada para ese modelo de seguridad es de USD 50 000 por año. Eso da cuenta de cuanto está dispuesta la empresa atacada a invertir para evitar estos ataques.

Algunos riesgos o tipos de ataque tienen una probabilidad muy baja de incidencia. Las industrias de seguro hacen este tipo de cálculo todo el tiempo y allí podemos conseguir múltiples ejemplos de como iniciar este trabajo. Para

riesgos asociados al ataque de sistemas computacionales hay múltiples herramientas disponibles para su análisis, estas herramientas proveen mecanismos para la realización de este tipo de análisis.

El foco debe mantenerse en los grandes riesgos y en todo caso despreciar aquellos hechos que tienen baja probabilidad o bajo costos. Por esto es muy importante poder determinar un “costo esperado” o una “esperanza de ataques exitosos” asociados a estos riesgos.

La ingeniería de seguridad procede secuencialmente desde los requerimientos de seguridad hasta la solución, no desde la tecnología más novedosa. Esto significa que lo primero que hay que hacer es modelar el tipo de ataques al que se está sujeto, a partir de esto crear una política de seguridad y luego a partir de esto escoger la tecnología a aplicar para evitar los riesgos antes modelados. Los riesgos determinan la política y esta define la tecnología a utilizar, los pasos a seguir serían los siguientes:

1. Comprender los riesgos reales del sistema y evaluar las probabilidades de esos riesgos.
2. Describir la política de seguridad requerida para defenderse de esos ataques o riesgos.
3. Diseñar las medidas de seguridad destinadas a contrarrestar esos riesgos.

Una política de seguridad para un sistema define los objetivos. La política

debe establecer:

1. Quién es responsable y por qué de la implementación, del reforzamiento, de la auditoría y revisión;
2. Cuáles son las políticas de seguridad básicas de la red; y
3. Por qué ellas son implementadas en la manera en que lo son.

En pocas palabras la política de seguridad debe establecer el porque y no el como. Los comos son parte de la táctica. Las medidas de seguridad deben proteger no solo contra las amenazas sino también contra los problemas no previstos. La seguridad absoluta en los sistemas es algo imposible de lograr sin embargo algunos principios básicos pueden ayudar a fortalecer la seguridad de los sistemas, a continuación se listan algunas de estas medidas:

1. Compartimentación de los recursos, o en otras palabras, nunca colocar *"todos los huevos en una sola canasta".*
2. Proteger los componentes que puedan ser considerados más débiles del sistema.
3. Usar *puntos de entrada,* o de *acceso,* bien definidos.
4. Usar técnicas de defensa combinadas para aumentar el nivel de resistencia de un sistema.
5. Manejo de fallas en un entorno seguro.
6. Sacar provecho de la imprevisibilidad.
7. La simplicidad siempre es una buena premisa, mientras mas simple es un

sistema más sencillo es garantizar su seguridad.

8. Preparar a los usuarios para las potenciales fallas de seguridad.
9. Cuestionar permanentemente la seguridad del sistema y hacer esfuerzos permanentes desde nuestro lado para romper las medidas que nosotros mismos hemos diseñado.
10. Tener buenos mecanismos de detección, al menos debemos saber que hemos sido atacados o que estamos siendo atacados.
11. Tener buenos mecanismos para detectar los ataques; esto incluye: Detección, localización, identificación y evaluación.
12. Tener respuestas preparadas a los potenciales ataques.
13. Estar vigilantes, en forma permanente.
14. Vigilar a los vigilantes, aunque suene paranoico.
15. Tener mecanismos de recuperación.

1.2.4 Seguridad Informática en Redes de Comunicación

El estándar ISO 7498 establece el modelo de referencia para la Interconexión de Sistemas Abiertos (OSI, *Open System Interconnection*) para las redes de comunicación. Este modelo divide en 7 capas el proceso de transmisión de la información entre equipos informáticos, donde cada capa se encarga de ejecutar una determinada parte del proceso global.

Tal como se muestra en la figura 1.6, cada capa establece el ámbito de acción

de cada servicio, función o mecanismo dentro de la comunicación. Cada capa puede ser referida por su nombre o por el nivel (N) que ocupa en la pila.

Figura 1.6 Pila de capas del modelos OSI.

El modelo OSI permite la interconexión de sistemas informáticos de modo que se pueda alcanzar una comunicación útil entre procesos de aplicación, pero para ello se deben establecer mecanismos de seguridad para proteger la información intercambiada entre procesos. Tales mecanismos deberían hacer que el coste de obtener o modificar los datos sea mayor que el valor potencial de obtener o modificar los datos.

Es necesario por tanto un estándar de arquitectura de seguridad en el modelo OSI que permita comenzar la tarea de implementar los servicios de seguridad en productos comerciales de modo que un sistema OSI no sólo pueda

comunicarse con otro, sino que además pueda hacerlo con la adecuada seguridad.

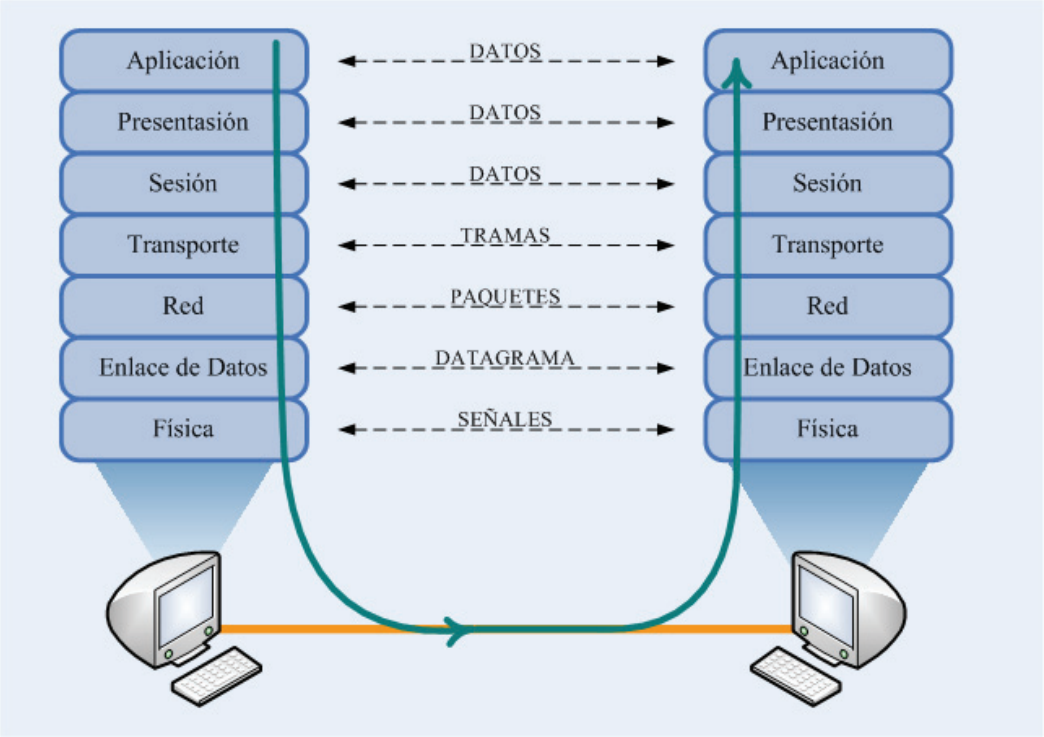

Figura 1.7 Secuencia de transmisión de información entre dos nodos, según el modelo OSI, y la transición de la información a través de la pila de protocolos.

La arquitectura de seguridad propuesta para el modelo OSI definida en el estándar ISO 7498 (parte 2) proporciona una descripción de los servicios de seguridad y mecanismos asociados, los cuales pueden ser proporcionados por los niveles del modelo de referencia OSI y define los niveles dentro del modelo de referencia donde se pueden proporcionar los servicios y mecanismos.

La arquitectura de seguridad propuesta para el modelo OSI contempla cinco

elementos: definición de servicios de seguridad, definición de mecanismos de seguridad, definición de una serie de principios de estructuración de servicios de seguridad en los niveles de la arquitectura OSI, implantación de servicios de seguridad en los niveles OSI y finalmente definición de los mecanismos asociados a cada servicio de seguridad.

La normalización ISO 7498-2 hace uso de las siguientes definiciones:

- *Servicio de Seguridad (N):* Es la capacidad que el nivel N y de los niveles inferiores ofrecen a las entidades de nivel $N+1$ en el campo de la seguridad en el interfaz entre el nivel N y el nivel $N+1$ por medio de las primitivas de servicio.
- *Función de seguridad (N):* Es una función relativa a la seguridad de acuerdo al servicio proporcionado al nivel N, controlado por el control lógico de la entidad (N).
- *Mecanismo de seguridad (N):* Mecanismo de nivel N que realiza una parte de una función de seguridad de nivel N.

Los servicios de seguridad definidos son autenticación, control de acceso, confidencialidad de datos, integridad de datos y no repudio. Para proporcionar estos servicios de seguridad es necesario incorporar en niveles apropiados del modelo de referencia OSI mecanismos de seguridad (cifrado, firma digital, mecanismos de control de acceso, integridad de datos, intercambio de autenticación, etc.)

En este modelo una entidad de nivel N se compone de tres partes:

- ***Control Lógico:*** Realiza la lógica del protocolo usando funciones y variables.
- ***Mecanismos:*** Controlados por el elemento de control opera con la variables realizando funciones.
- ***Variables:*** La mayoría de las cuales son locales a cada entidad.

La figura 1.8 muestra una descripción gráfica de la estructura de una entidad par de un nivel (N) así como su interrelación con el entorno.

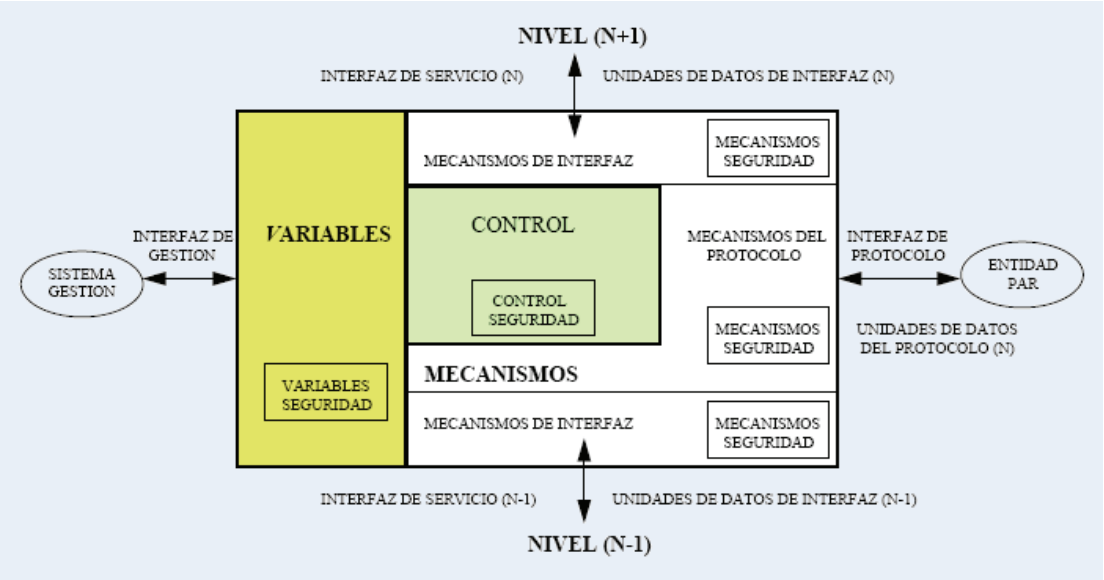

Figura 1.8 Estructura de seguridad de una entidad par en un nivel N. Tomado de [8].

De esta manera la arquitectura de seguridad del modelo OSI establece las normas de diseño y estudio para generar protocolos de seguridad en cada capa para así incrementar la seguridad en una red de comunicaciones.

1.2.5 El problema de la comunicación segura en las redes TCP/IP

La seguridad de las conexiones en red merece en la actualidad una atención especial por el impacto que representan los fallos. Los ataques más comunes a los sistemas en red involucran (en orden):

1. **Desbordamiento de buffers.-** Los ataques de desbordamiento de buffer escriben más datos en el buffer de los que éste puede soportar. El desbordamiento de buffer permite cambiar el flujo de ejecución de un programa.
2. **Secuestro de Sesiones.-** El atacante roba, comparte, termina, monitorea o registra alguna sesión de terminal que está en progreso. A veces usando estratagemas informáticos y otras usando Ingeniería Social.
3. **Debilidades en los protocolos de red.-** La mayoría de los protocolos de red (como TCP/IP) son vulnerables porque no fueron diseñados pensando en la seguridad, sino sólo en la conectividad.

A continuación vamos a mencionar de manera sucinta algunas de las medidas, herramientas y soluciones que ayudan a mejorar los niveles de seguridad en las redes TCP/IP.

LOS CORTAFUEGOS

Los cortafuegos (*firewalls*) representan un bloqueo entre la red privada o protegida y otras redes, típicamente públicas o Internet (que es supuesta

como no segura y no confiable).

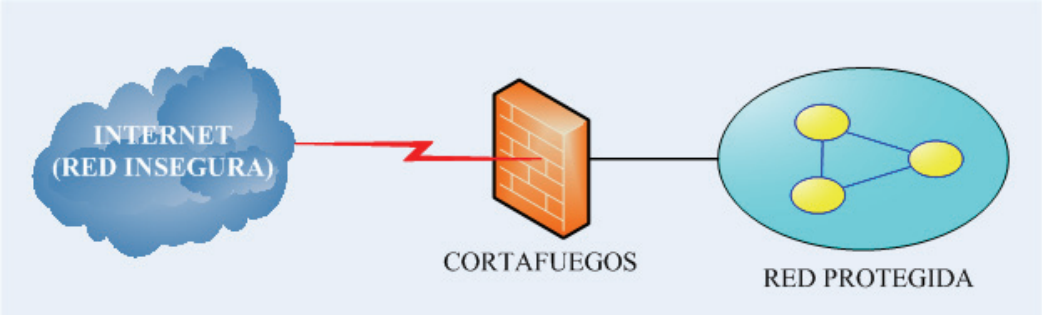

Figura 1.9 Cortafuegos en una red TCP/IP.

El propósito de un cortafuego es prevenir comunicaciones indeseadas y las no autorizadas hacia el interior de la red protegida, como hacia el exterior de la misma. En este caso la seguridad está situada en un solo punto y, dependiendo de la red, se pueden requerir precauciones de seguridad adicionales.

SEGURIDAD EN LA CAPA DE RED

La seguridad a nivel IP, denominada IPSec, comprende las siguientes 3 áreas funcionales:

- **Autenticación (e Integridad):** Se asegura de que un paquete IP fue, de verdad, transmitido por aquel que se identifica como fuente de dicho paquete en la cabecera del mismo. También provee la seguridad de que el paquete no ha sido alterado.
- **Confidencialidad:** Dota a los nodos de comunicación con la habilidad de cifrar los mensajes para prevenir espionaje o fisgoneo de terceras partes.

- **Manejo de claves:** Permite el intercambio seguro de claves.

Cuando IPSec es implementado en un cortafuego o en un enrutador, eso provee un fuerte control de seguridad que puede ser aplicado a todo el tráfico que cruza el perímetro. Además, como funciona en el nivel 3, o en capa de red, y por ende, debajo de la capa de transporte, es transparente para las aplicaciones que corren en cada nodo.

SEGURIDAD EN LA CAPA DE TRANSPORTE

El protocolo de Capa de Conexión Segura (SSL, *Secure Sockets Layer*) fue diseñado para hacer uso de TCP proveyendo un servicio de seguridad de extremo a extremo. Es implementado, en general, entre la capa de trasporte y la de aplicación.

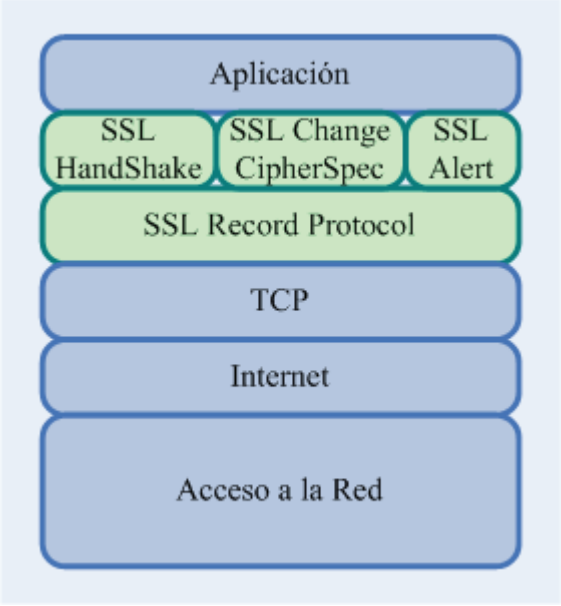

Figura 1.10 Ubicación de las subcapas de SSL en la pila TCP/IP.

SSL está compuesto de 2 subcapas tal como se muestra en la figura 1.10; algunas arquitecturas de seguridad lo implementan en la capa de presentación y otras lo incluyen en la misma capa de aplicación pero antes de que el programa principal reciba los datos. Entre las ventajas de SSL está el hecho de que es transparente para la aplicación que está corriendo en el nodo de comunicación, además de ser más minucioso que IPSec ya que interactúa con la capa de transporte. En cambio, entre las desventajas tenemos que es únicamente aplicable en redes basadas en TCP y no en UDP; y, además, este protocolo no trabaja con servidores Proxy (como por ejemplo, los cortafuegos), ya que un Proxy es un *intermediario*, y SSL fue diseñado para proveer seguridad en una conexión directa entre un cliente y un servidor.

En conclusión, en la actualidad se encuentran disponibles muchos métodos y herramientas de seguridad diseñadas para fortalecer la confiabilidad de redes basadas en TCP/IP, y sólo un correcto análisis de las amenazas del sistema ayudará a seleccionar los más adecuados.

1.3 La Criptografía y los Criptosistema

1.3.1 Definiciones

Según el Diccionario de la Real Academia, la palabra Criptografía proviene del griego $\kappa\rho\upsilon\pi\tau\acute{o}\varsigma$, que significa oculto, y $\gamma\rho\alpha\varphi\epsilon\iota\nu$, que significa escritura, y su

definición es: *"Arte de escribir con clave secreta o de un modo enigmático"*, o en un sentido más amplio sería aplicar alguna técnica para hacer ininteligible un mensaje.

En su clasificación dentro de las ciencias, la Criptografía proviene de la rama de las Matemáticas llamada *La Teoría de la Información*; esta rama de las ciencias se divide en: *Teoría de Códigos* y en *Criptología.* Y a su vez la Criptología se divide en Criptoanálisis y Criptografía.

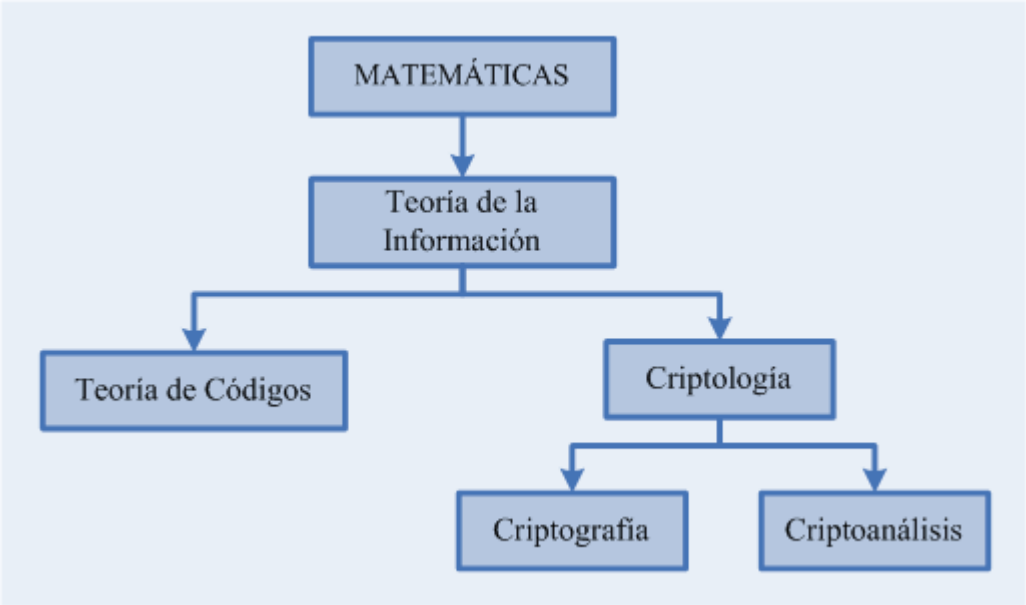

Figura 1.11 Clasificación de la Criptografía dentro de las ciencias.

De esta manera, en un sentido más completo, la Criptografía es la ciencia encargada de diseñar funciones o dispositivos, capaces de transformar mensajes legibles o en claro a mensajes cifrados de tal manera que esta transformación (*cifrar*) y su transformación inversa (*descifrar*) sólo pueden ser

factibles con el conocimiento de una o más claves. En contraparte, el Criptoanálisis es la ciencia que estudia los métodos que se utilizan para, a partir de uno o varios mensajes cifrados, recuperar los mensajes en claro en ausencia de la(s) clave(s), o encontrar la(s) clave(s) con las que fueron cifrados dichos mensajes.

La Criptografía se puede clasificar históricamente en dos: La *Criptografía Clásica* y la *Criptografía Moderna*.

La Criptografía Clásica es aquella que se utilizó desde antes de la época actual hasta la mitad del siglo XX. También puede entenderse como la Criptografía no computarizada o mejor dicho no digitalizada. Los métodos utilizados eran variados, algunos muy simples y otros muy complicados de criptoanalizar para su época.

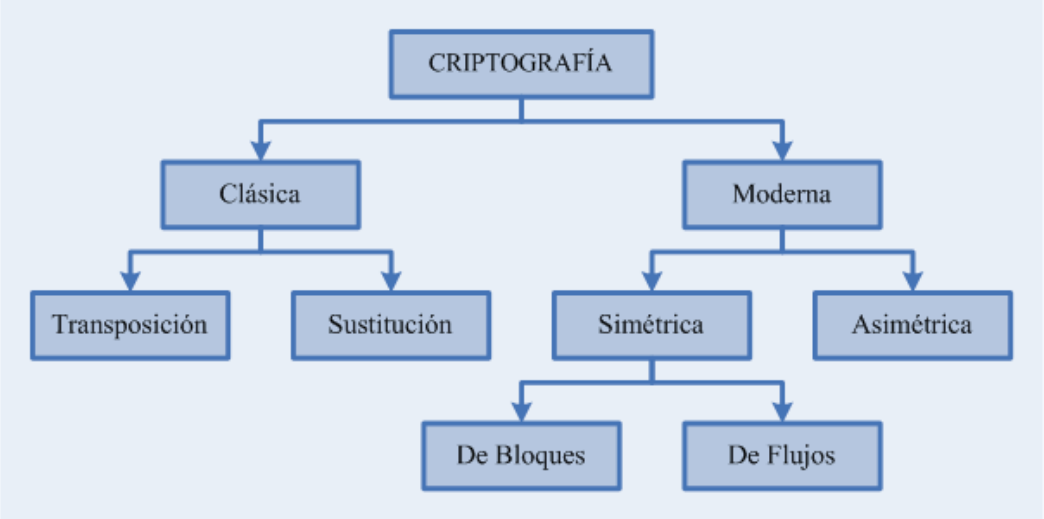

Figura 1.12 Clasificación general de la Criptografía.

Se puede decir que la Criptografía Moderna se inició después de tres hechos: el primero fue la publicación de los trabajos desarrollados por Claude Shannon, *"A Mathematical Theory of Communication"* (1948) y *"Communication Theory of Secrecy Systems"* (1949); el segundo, la publicación del Estándar de Cifrado de Datos (DES, *Data Encryption Standard*) en 1974; y el tercero, la aparición del estudio hecho por Whitfield Diffie y Martin Hellman, *"New directions in Cryptography"*, en 1976.

1.3.2 Técnicas de la Criptografía Clásica

Como ya se mencionó anteriormente, la Criptografía Clásica es muy antigua. Las técnicas criptográficas eran muy ingeniosas y se usaban para enviar mensajes secretos entre las personas que tenían el poder, o, en época de guerra, para enviar instrucciones. A diferencia de la Criptografía Moderna, el algoritmo del sistema criptográfico se mantenía en secreto.

Estas técnicas tienen en común que pueden ser empleadas usando simplemente lápiz y papel, y que pueden ser criptoanalizadas casi de la misma forma. De hecho, con la ayuda de las computadoras, los mensajes cifrados empleando estos códigos son fácilmente descifrables, por lo que cayeron rápidamente en desuso.

La Criptografía Clásica también incluye la construcción de máquinas, que

mediante mecanismos, comúnmente engranes o rotores, transformaban un mensaje en claro a un mensaje cifrado. Todos los algoritmos criptográficos clásicos son simétricos, lo cual es evidente si atendemos al hecho de que hasta mediados de los años setenta no había nacido la Criptografía Asimétrica.

Tanto en los mecanismos de la Criptografía Clásica como en los algoritmos de la Criptografía Moderna los mensajes intercambiados (originales y cifrados) entre los principales son sucesiones finitas de símbolos (o caracteres, según sea el caso) que pertenecen a algún alfabeto.

Es decir, que si M es el conjunto de todos los posibles mensajes originales m, A_m es el alfabeto con el que se generan dichos mensajes, y además, C es el conjunto de todos los mensajes cifrados resultantes posibles c, y A_c es el alfabeto utilizado para generar los mismos, entonces $M = A_m^*$ y $C = A_c^*$.

Para los algoritmos de la Criptografía Moderna $A_m = A_c = \{0,1\}$, es decir, que los mensajes son sucesiones finitas de dígitos binarios; para los mecanismos de la Criptografía Clásica A_m generalmente correspondía al alfabeto del lenguaje natural de los principales (e.g. el alfabeto Griego, el alfabeto Inglés, el alfabeto Alemán, etc.), mientras que A_c si bien podía ser igual a A_m, también podía tratarse de un alfabeto vernacular.

1.3.2.1 Permutación

La transposición consiste en la reordenación (o permutación) de los símbolos del mensaje original mediante un mecanismo específico y de acuerdo a una clave establecida entre los principales.

El objetivo de la transposición el de *difuminar* el mensaje. Con este método al reordenar el criptograma aparecerán exactamente los mismos caracteres que en el texto en claro.

Los algoritmos criptográficos de transposición se dividen en cuatro tipos generales: por Grupos, por Series, por Columnas y por Filas, tal como lo muestra la figura 1.13.

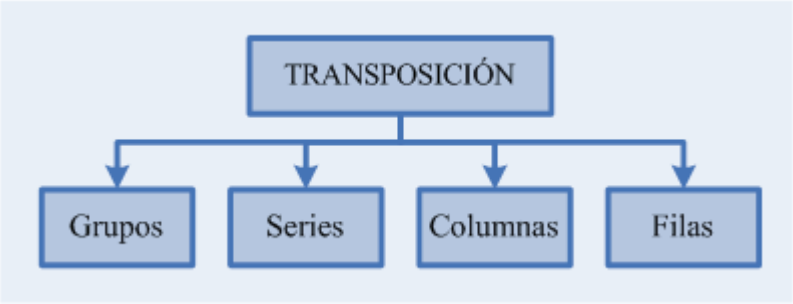

Figura 1.13 Clasificación de los métodos criptográficos por transposición.

TRANSPOSICIÓN POR GRUPOS

Este método consiste en tomar el texto original y dividirlo en grupos de n símbolos, obteniendo algo como lo siguiente:

$$\underbrace{m_1 m_2 m_3 \ldots m_n}_{n \text{ símbolos}} \quad \underbrace{m_{n+1} m_{n+2} m_{n+3} \ldots m_{n+n}}_{n \text{ símbolos}} \quad \underbrace{m_{2n+1} m_{2n+2} m_{2n+3} \ldots m_{2n+n}}_{n \text{ símbolos}} \ \ldots$$

Si llegaran a faltar símbolos en el último grupo, para completar los n, se selecciona un símbolo para rellenar (generalmente, en Castellano y en Inglés, es la letra X). La secuencia $[1,2,3,...,n]$ representa la ubicación relativa de cada símbolo en su grupo. Se toman los símbolos de cada grupo y se los reordena dentro del mismo grupo de acuerdo a una clave, la cual es la secuencia original permutada.

Para descifrar el mensaje se utiliza el mismo algoritmo pero con una clave que haga una permutación inversa.

Ejemplo:

Tamaño de grupo: 5

Clave: 43521

Mensaje: Transposición por grupos

Bloques: TRANS POSIC IONPO RGRUP OSXXX

Criptograma: NASRTISCOPPNOOIURPGRXXXSO

Clave de descifrado: 54213

Una máquina antigua que implementaba este método de cifrado era la *Escítala* que la usaban los lacedemonios de Esparta para intercambiar mensajes en tiempos de guerra.

TRANSPOSICIÓN POR SERIES

El mensaje se ordena como una cadena de submensajes, de forma que el mensaje original se transmite como la unión de varias series de símbolos que lo componen. Cada submensaje sigue una función específica; cada función se aplica al mensaje, una tras otra, sin tomar en cuenta los símbolos tomados por las anteriores.

Al no tener período, este algoritmo posee mayor fortaleza, residiendo ésta en el secreto y complejidad de las series utilizadas. Para descifrar sólo debemos saber el orden en el que están las series y cuáles son, así podremos tener la ubicación correspondiente de cada símbolo y recuperar el mensaje sin problemas.

Ejemplo:

Secuencia: $s_1(m) = \{m_i / i \ es \ primo\}$

$s_2(m) = \{m_i / i \ es \ par\}$

$s_3(m) = \{m_i / i \ es \ impar\}$

Mensaje: Transposición por series

$s_1(m)$ $m_1 m_2 m_3 m_5 m_7 m_{11} m_{13} m_{17} m_{19} =$ TRASOINSR

$s_2(m - s_1)$ $m_4 m_6 m_8 m_{10} m_{12} m_{14} m_{16} m_{18} m_{20} m_{22} =$ NPSCOPREIS

$s_3(m - s_1 - s_2)$ $m_9 m_{15} m_{21} =$ IOE

Criptograma: TRASOINSRNPSCOPREISIOE

TRANSPOSICIÓN POR COLUMNAS

El algoritmo de transposición por columnas es un método de cifrado sencillo que consiste en rellenar por filas una tabla de transposición, de n columnas, con los caracteres del mensaje original y leerlos por columnas con el fin de formar el mensaje cifrado. Existen numerosas variantes de esta técnica, cada cuál más ingeniosa.

Ejemplo:

Clave: $n = 6$

Mensaje: Transposición por columnas

Tabla:

T	R	A	N	S	P
O	S	I	C	I	O
N		P	O	R	
C	O	L	U	M	N
A	S	X	X	X	X

Criptograma: TONCARS OSAIPLXNCOUXSIRMXPO NX

TRANSPOSICIÓN POR FILAS

De forma similar al cifrado por columnas, en esta operación se escribe el mensaje en forma vertical, por ejemplo de arriba hacia abajo, con un cierto número de filas n y luego se lee el criptograma en forma horizontal.

Ejemplo:

Clave: $n = 3$

Mensaje: Cifrado por transposición por filas

Tabla:

C	R	O	O	T	N	O	C	N	O	F	A
I	A		R	R	S	S	I		R	I	S
F	D	P		A	P	I	O	P		L	X

Criptograma: CROOTNOCNOFAIA RRSSI RISFDP APIOP LX

1.3.2.2 Sustitución

Los métodos de sustitución consisten en establecer una correspondencia entre las letras del alfabeto en el que está escrito el mensaje original y los elementos de otro conjunto, que puede ser el mismo alfabeto u otro distinto.

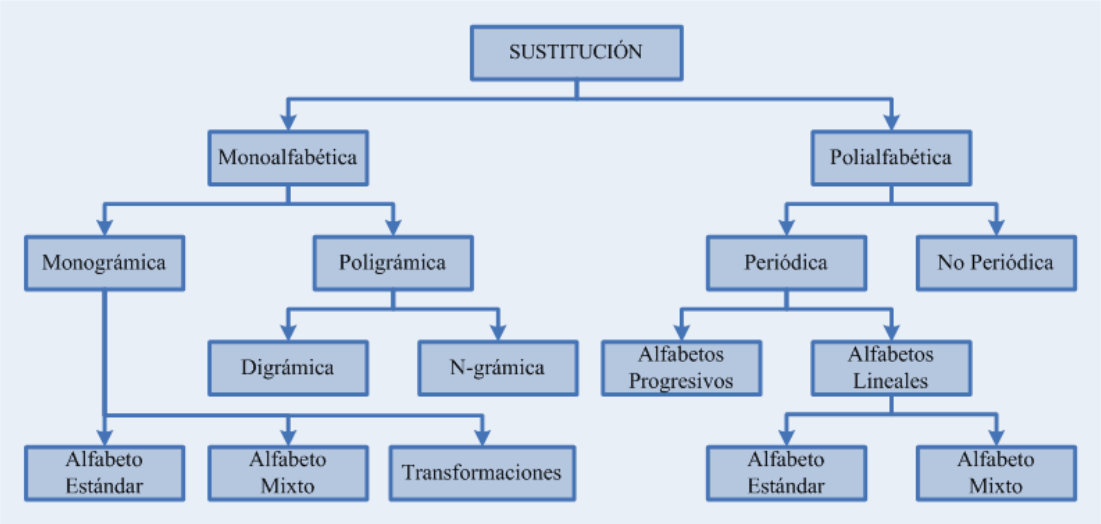

Figura 1.14 Clasificación de los métodos criptográficos por sustitución.

La figura 1.14 muestra la extensa subdivisión del cifrado por sustitución, la explicación detallada de cada caso no pertenece al ámbito de este trabajo, a continuación vamos a describir las dos clases más generales.

SUSTITUCIÓN MONOALFABÉTICA

En las sustituciones monoalfabéticas, cada letra del mensaje original se sustituye por una única letra cifrada, y que es siempre la misma.

Como ejemplo podemos citar el Cifrado de César, llamado así porque es el método que empleaba Julio César para enviar mensajes secretos. Este método (*monográmico*) consiste en cambiar cada símbolo (letra) del mensaje, por otro situado n lugares más adelante en el mismo alfabeto; y a las n últimas letras se las hace corresponder con las n primeras del alfabeto. El descifrado es la operación inversa; la clave es el número n. O sea,

$$\begin{cases} c_i = (m_i + n) \bmod N\left(A_m\right) \\ m_i = (c_i - n) \bmod N\left(A_m\right) \end{cases}$$

Ejemplo:

Alfabeto: $A_m = \{Alfabeto\ Castellano\}\ ;\ N\left(A_m\right) = 27$

Clave: $n = 3$

Mensaje: Cifrado de César

Criptograma: FLIUDGRGHFHVDU

SUSTITUCIÓN POLIALFABÉTICA

Los cifrados de sustitución polialfabética basan su funcionamiento en sustituciones múltiples. La sustitución aplicada a cada símbolo varía en

función de la posición que ocupe éste dentro del texto claro. Dicho de otra manera, la sustitución polialfabética corresponde a la aplicación cíclica de n cifrados monoalfabéticos.

Un ejemplo típico de cifrado polialfabético es el Cifrado de Vigènere que debe su nombre a Blaise de Vigènere, su creador, y que data del siglo XVI. Se establece una secuencia clave $k = k_1 k_2 ... k_d$, se divide el mensaje claro en subcadenas de longitud d, y a cada símbolo se lo cifra usando el Cifrado de César. De manera formal,

$$\begin{cases} c_i = \left(m_i + k_{(i \bmod d)} \right) \bmod N\left(A_m \right) \\ m_i = \left(c_i - k_{(i \bmod d)} \right) \bmod N\left(A_m \right) \end{cases}$$

Ejemplo:

Alfabeto: $A_m = \{ Alfabeto\ Castellano \};\ N\left(A_m \right) = 27$

Clave: ejemplo $\equiv$ $k = (5,10,5,13,16,12,15);\ d = 7$

Mensaje: Cifrado de Vigenere

Criptograma: HRKEOÑDHÑAUVPBJBJ

1.3.3 Técnicas de la Criptografía Moderna

El estudio de la Criptografía Moderna se basa en la descripción matemática

de un *sistema secreto*, o *criptosistema*, hecha por C. E. Shannon en su trabajo *Communication Theory of Secrecy Systems*. Los distintos tipos de algoritmos, técnicas y estrategias de cifrado van a ser derivaciones o modificaciones a este modelo.

MODELO DE UN CRIPTOSISTEMA

Un *sistema secreto* o *criptosistema* se lo define como una quíntupla (M, C, E, D), donde:

- Los alfabetos del texto plano y del texto cifrado son $A_m = A_c = \{0,1\}$.
- M es el espacio de todos los mensajes sin cifrar m (texto en claro, *plaintext*) que pueden ser generados por la fuente.
- C es el espacio de todos los posibles criptogramas c.
- K es el espacio de todas las claves k que se pueden emplear en el criptosistema.
- E es la familia de *transformaciones de cifrado* que se aplican a cada mensaje m, para obtener un criptograma c.

$$E = \{E_k\}_{k \in K} \tag{1.1}$$

$$\begin{aligned} E_k : M \times K &\to C \\ (m,k) &\mapsto E_k(m) \end{aligned} \tag{1.2}$$

- D es la familia de transformaciones de descifrado, análoga a E.

$$D = \{D_k\}_{k \in K} \tag{1.3}$$

$$D_k : C \times K \to M$$
$$(c,k) \mapsto D_k(c) \qquad \textbf{(1.4)}$$

El siguiente diagrama muestra el esquema general de un criptosistema, sus componentes y su papel.

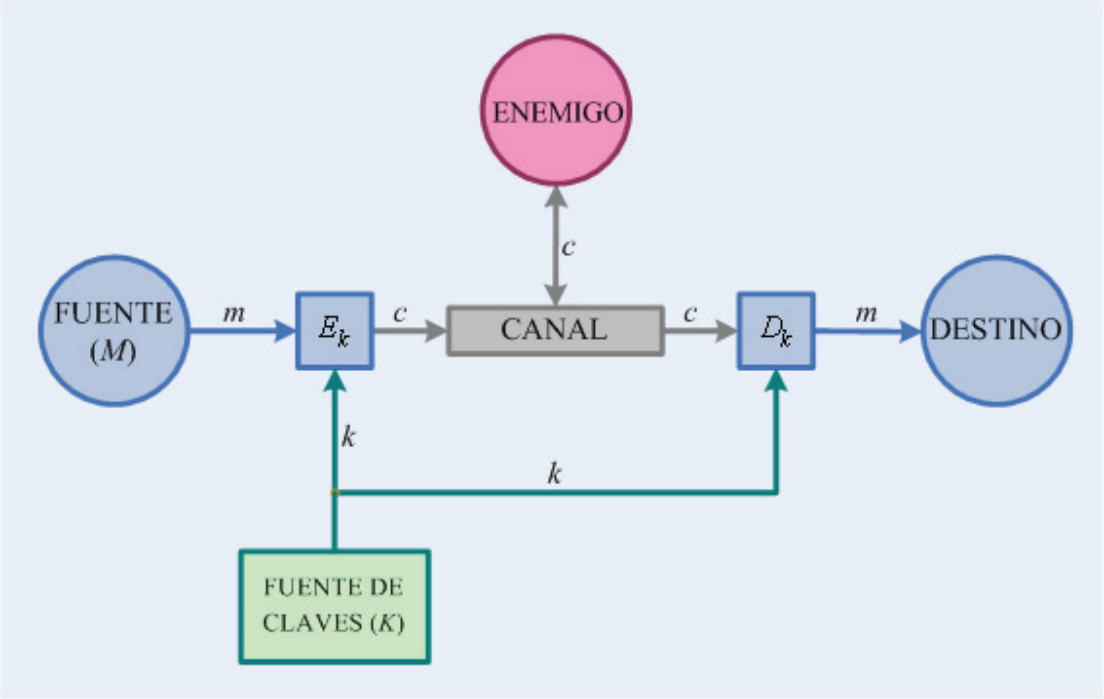

Figura 1.15 Esquema general de un criptosistema.

Como podemos observar, la fuente posee un algoritmo cifrador, el destino uno descifrador y ambos deben poseer la misma clave para poder cifrar y descifrar el mensaje. Además, se observa que en el medio se encuentra un enemigo, que en general es un criptoanalista, que tiene acceso al mensaje cifrado e inclusive podría tener la facultad de *fabricar* un mensaje falso.

Todo criptosistema debe cumplir las propiedades:

$$\forall m \in M \ \forall k \in K \ \left[E_k(m) \neq E_{k'}(m) \right] \quad \textbf{(1.5)}$$

$$\forall m \in M \ \forall k \in K \ \left[D_k\left(E_k(m)\right) = m \right] \quad \textbf{(1.6)}$$

La propiedad **(1.5)** quiere decir que si tenemos un mensaje y lo ciframos con distintas claves, también los criptogramas resultantes deben ser diferentes; y la **(1.6)** significa que si tenemos un mensaje, lo ciframos empleando una clave y luego lo desciframos empleando la misma clave, debemos obtener de nuevo el mensaje original.

Además, para que un criptosistema sea viable de implementar debe satisfacer:

- Que las transformaciones de cifrado y descifrado, conociendo la clave, deben ser computacionalmente eficientes; y
- Que la seguridad del sistema debe depender exclusivamente del secreto de las claves, siendo Las funciones E_k y D_k tales que sin el conocimiento de las claves no pueda descifrarse un mensaje.

La seguridad de un sistema criptográfico depende de un parámetro denominado *fuerza*, que define el grado de dificultad para *romper* el sistema. Romper un criptosistema significa que un criptoanalista puede (en un porcentaje mayor del puramente aleatorio):

- Reconstruir el texto en claro a partir del texto cifrado, o sea romper el secreto.
- Hacer que el receptor acepte mensajes no generados por un emisor autorizado, esto es, romper la autenticidad.

Los algoritmos E_k y D_k deben buscarse para que el sistema sea lo más fuerte posible. Con los medios adecuados, prácticamente todo sistema puede romperse si se dispone del tiempo suficiente. No debe procurarse que el sistema sea totalmente inatacable sino que sea *computacionalmente irrompible*, esto es, que conociendo las funciones E_k y D_k, y teniendo múltiples parejas mensaje-criptograma, sea computacionalmente muy costoso determinar la clave k.

En la Criptografía Moderna existen dos tipos fundamentales de criptosistemas: los Criptosistemas Simétricos o de Clave Privada, y los Criptosistemas Asimétricos o de Clave Pública.

1.3.3.1 Criptografía Simétrica

Los sistemas criptográficos simétricos son aquellos que emplean la misma clave tanto para cifrar como para descifrar. La simetría, entonces, es evidente al observar que cada principal requiere tener implementadas las mismas funciones de cifrado y descifrado al igual que la misma clave.

Generalmente los algoritmos de cifrado son computacionalmente sencillos, y en la práctica, muchas veces se da que $E_k = D_k$.

Entre las ventajas de estos criptosistemas tenemos que, por la sencillez de las funciones de cifrado y descifrado, pueden ejecutarse a alta velocidad; que cuando $E_k = D_k$ se puede usar el mismo circuito integrado para cifrar y descifrar; y que son buenos para formar sistemas compuestos. Mientras tanto, por otro lado, entre las desventajas de estos sistemas tenemos que en un contexto de comunicaciones la distribución de las claves debe hacerse a través de algún medio seguro; que es necesario cambiar las claves con cierta frecuencia; y que en una red, existe una gran cantidad de claves a ser manejadas.

Existen dos tipos de sistemas de clave simétrica: De Bloque y De Flujo.

1.3.3.1.1 Criptografía Simétrica de Bloques

Las técnicas de cifrado simétrico por bloques consisten básicamente en *fragmentar* el mensaje en n bloques de tamaño fijo $p_i \in \{0,1\}^L$ (generalmente L es múltiplo de 8), y aplicar la función de cifrado a cada uno de ellos.

El mecanismo típico consiste en tomar el mensaje de la fuente y guardarlo en un buffer, se lee el buffer por porciones de L bits, se ejecuta la función de cifrado sobre cada bloque y se acumula el resultado en otro buffer.

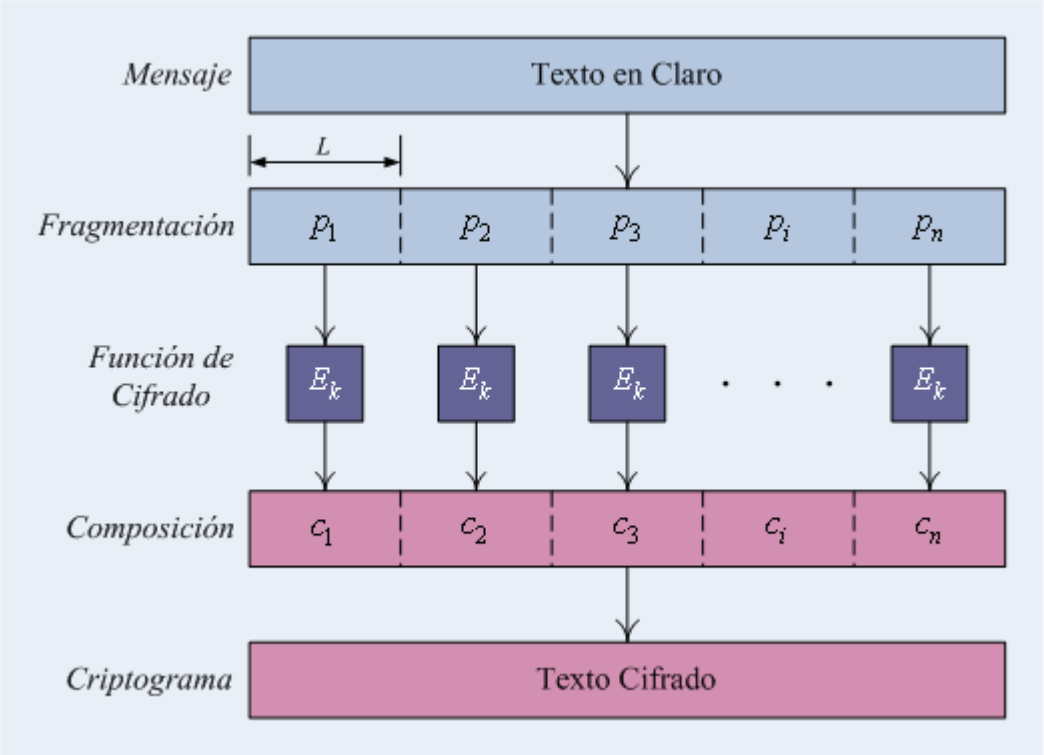

Figura 1.16 Esquema de cifrado por bloques.

La gran mayoría de estos algoritmos de cifrado se apoyan en los conceptos de *confusión* y *difusión* inicialmente propuestos por Shannon.[12] La *confusión* consiste en tratar de ocultar la relación que existe entre el texto claro, el texto cifrado y la clave (un buen mecanismo de confusión hará demasiado complicado extraer relaciones estadísticas entre las tres cosas), por su parte la *difusión* trata de repartir la influencia de cada bit del mensaje original lo más posible entre el mensaje cifrado.

La confusión por sí sola sería suficiente, ya que si establecemos una tabla de sustitución completamente diferente para cada clave con todos los textos claros posibles tendremos un sistema extremadamente seguro. Sin embargo, dichas tablas ocuparían cantidades astronómicas de memoria, por lo que en la práctica serían inviables. Lo que en realidad se hace para conseguir algoritmos fuertes, sin necesidad de almacenar tablas enormes, es intercalar la confusión (usando sustituciones simples, con tablas pequeñas) y la difusión (empleando permutaciones).

La combinación de confusión y difusión se conoce como *cifrado de producto*. La mayoría de los algoritmos simétricos de bloque se basan en diferentes capas de sustituciones y permutaciones, estructura que denominaremos *Red de Sustitución-Permutación*. En muchos casos esto consiste en una operación combinada de sustituciones y permutaciones, repetida n veces. Esto se puede comprender mejor al observar la *Red de Feistel* en la sección **2.2.2**.

1.3.3.1.2 Criptografía Simétrica de Flujo

Las técnicas de cifrado en flujo de clave secreta se basan en una transformación variante en el tiempo de los símbolos del texto en claro. La diferencia fundamental entre el cifrado en flujo y el cifrado en bloque consiste en la presencia o no de memoria interna en el sistema, así como

el tamaño de la unidad mínima a cifrar, y además que permiten cifrar mensajes de longitud arbitraria, sin necesidad de dividirlos en bloques para codificarlos por separado.

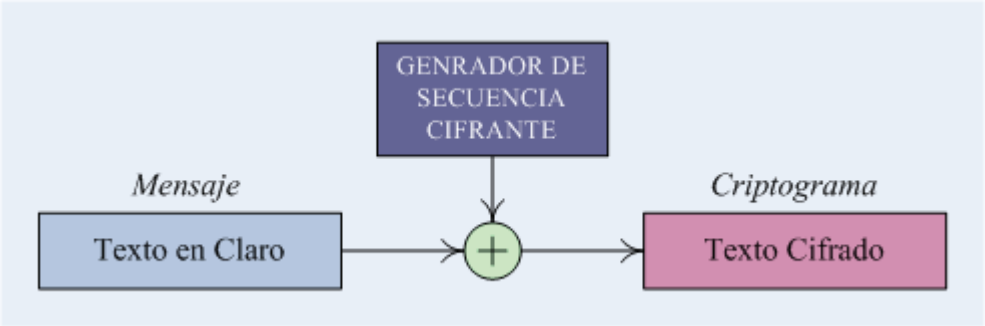

Figura 1.17 Esquema de un cifrador de flujo.

En general, un cifrador en flujo consiste en un generador de clave que provee una secuencia de bits *criptográficamente aleatoria* (llamada secuencia *cifrante* o *cifradora*) la cual se suma en módulo 2, segmento a segmento, al mensaje (operación equivalente a la disyunción exclusiva, XOR, bit a bit) para formar la secuencia de salida. Cada bit de salida de la secuencia cifradora pseudoaleatoria depende del estado interno del generador en ese momento.

De esta manera, los algoritmos de cifrado simétrico de flujo no son más que la especificación del generador pseudoaleatorio. Los generadores que se emplean como cifrado de flujo pueden dividirse en dos grandes grupos, dependiendo de los parámetros que se empleen para calcular el valor de cada porción de la secuencia: *Generadores Sincrónicos* y *Generadores*

Asincrónicos.

GENERADORES SINCRÓNICOS

Un generador *sincrónico* es aquel en el que la secuencia es calculada de forma independiente tanto del texto en claro como del texto cifrado. El caso general, ilustrado en la figura 1.18, viene dado por las siguientes ecuaciones:

$$\left\| \begin{array}{l} s_{i+1} := g(s_i, k) \\ o_i := h(s_i, k) \\ c_i := w(m_i, o_i) \end{array} \right. \qquad \textbf{(1.7)}$$

Donde k es la clave, s_i es el estado interno del generador, s_0 es el estado inicial, o_i es la salida en el instante i, m_i y c_i son la i-ésima porción del texto claro y cifrado respectivamente, y w es una función reversible, usualmente una suma en módulo 2 (disyunción exclusiva, XOR). En muchos casos, la función h depende únicamente de s_i, siendo $k = s_0$.

Cuando empleamos un generador de estas características, necesitamos que tanto el emisor como el receptor estén sincronizados para que el texto pueda descifrarse. Si durante la transmisión se pierde o inserta algún bit, ya no se estará aplicando en el receptor un XOR con la misma secuencia,

por lo que el resto del mensaje será imposible de descifrar. Esto nos obliga a emplear tanto técnicas de verificación como de restablecimiento de la sincronía.

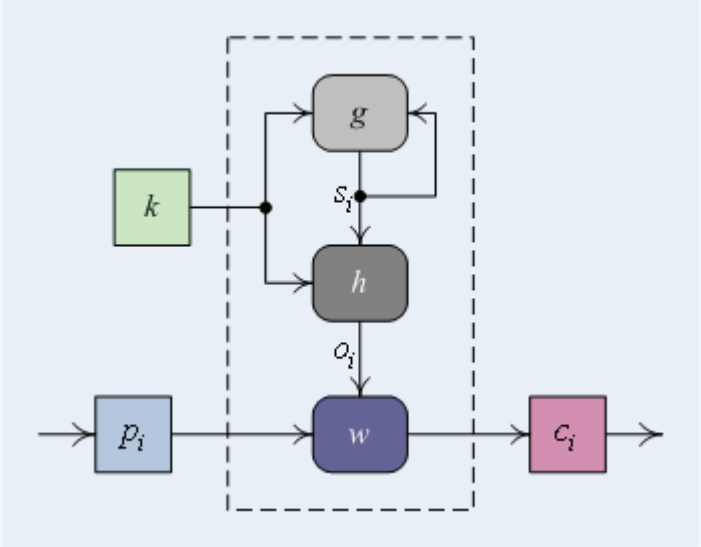

Figura 1.18 Esquema de generadores de secuencia sincrónicos.

Otro problema muy común con este tipo de técnicas es que si algún bit del criptograma es alterado, la sincronización no se pierde, pero el texto claro se verá modificado en la misma posición. Esta característica podría permitir a un atacante introducir cambios en nuestros mensajes, simplemente conociendo qué bits debe alterar. Para evitar esto, deben emplearse mecanismos de verificación que garanticen la integridad del mensaje recibido, como las funciones resumen (ver sección **1.3.3.2.1**).

GENERADORES ASINCRÓNICOS

Un generador de secuencia *asincrónico* o *auto-sincronizado* es aquel en el

que la secuencia generada es función de una semilla, más una cantidad fija de los bits anteriores de la propia secuencia, como puede verse en la figura 1.19. Formalmente:

$$\begin{aligned} o_i &:= h\left(k, c_{i-t}, c_{i-t+1}, \ldots, c_{i-1}\right) \\ c_i &:= w\left(m_i, o_i\right) \end{aligned} \qquad \textbf{(1.8)}$$

Donde k es la clave, m_i y c_i son la i-ésima porción del texto claro y cifrado respectivamente y w es una función reversible. Los valores $c_{i-t}, c_{i-t+1}, \ldots, c_{i-1}$ constituyen el estado s_i del generador.

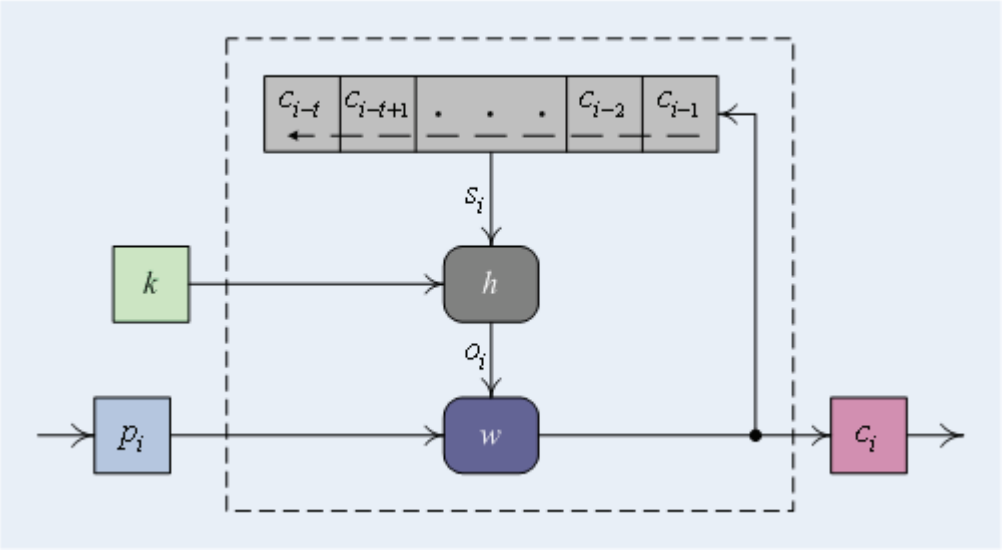

Figura 1.19 Esquema de generadores de secuencia asincrónicos.

Esta familia de generadores es resistente a la pérdida o inserción de información, ya que acaba por volver a sincronizarse de forma automática,

en cuanto llegan t bloques correctos de forma consecutiva. También es sensible a la alteración de un mensaje, ya que si se modifica la unidad de información c_i, el receptor tendrá valores erróneos de entrada en su función h hasta que se alcance el bloque c_{i+t}, momento a partir del cual la transmisión habrá recuperado la sincronización. En cualquier caso, al igual que con los generadores sincrónicos, habrá que introducir mecanismos de verificación.

Una propiedad interesante de estos generadores es la dispersión de las propiedades estadísticas del texto claro a lo largo de todo el mensaje cifrado, ya que cada dígito del mensaje influye en todo el criptograma. Esto hace que los generadores asincrónicos se consideren en general más resistentes frente a ataques basados en la redundancia del texto en claro.

1.3.3.1.3 Modos de Operación

En esta sección comentaremos algunos métodos para aplicar cifrados por bloques a mensajes de gran longitud. El estándar FIPS 81 expedido por el Instituto Nacional de Estándares y Tecnologías (NITS) define cuatro modos de operación para cifradores simétricos de bloque: Modo Directo, ECB; Cifrado de Bloques Encadenados, CBC; Cifrado Realimentado, CFB; y Cifrado con Realimentación de la Salida, OFB; además nosotros

incluiremos un modo adicional, el Modo Contador, CTR.

MODO DIRECTO

El modo Directo ó ECB es el método más rápido y obvio de aplicar un algoritmo de cifrado por bloques. Simplemente se subdivide el mensaje que se quiere codificar en n bloques $p_i \in \{0,1\}^L$, y se cifra cada uno empleando la misma función de cifrado y la misma clave k (ver figura 1.16).

$$c_i := E_k(p_i) \; : \; 1 \le i \le n \qquad \textbf{(1.9)}$$

$$p_i := D_k(c_i) \; : \; 1 \le i \le n \qquad \textbf{(1.10)}$$

Si la longitud del texto que queremos cifrar no es un múltiplo exacto del tamaño de bloque, entonces tenemos que añadir información al final para que sí lo sea. El mecanismo más sencillo consiste en *rellenar* con ceros (o algún otro patrón) el último bloque que se codifica.

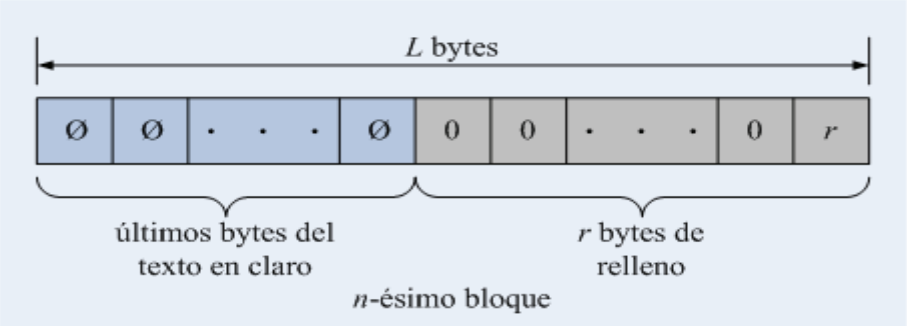

Figura 1.20 Técnica de relleno.

Cuando se descifra, para saber por dónde hay que *cortar* el bloque rellenado, lo que se suele hacer es añadir como último byte del último bloque el número de bytes que se han añadido. Por ejemplo, si el tamaño de bloque fuera 64 bits (8 bytes), y nos sobraran cinco bytes al final, añadiríamos dos ceros y un tres, para completar los ocho bytes necesarios en el último bloque (ver figura 1.20).

A favor del ECB tenemos que permite codificar los bloques independientemente de su orden, lo cual es adecuado para codificar bases de datos o ficheros en los que se requiera un acceso aleatorio. También es resistente a errores, pues si uno de los bloques sufriera una alteración, el resto quedaría intacto. Por otro lado, si el mensaje presenta patrones repetitivos, el texto cifrado también los presentará, y eso es peligroso, sobre todo cuando se codifica información muy redundante (como ficheros de texto), o con patrones comunes al inicio y final (como el correo electrónico). Un contrincante puede en estos casos efectuar un ataque estadístico y extraer bastante información.

Otro riesgo bastante importante que presenta el modo ECB es el de la sustitución de bloques. El atacante puede cambiar un bloque sin mayores problemas, y alterar los mensajes incluso desconociendo la clave y el algoritmo empleados. Simplemente se escucha una comunicación de la que se conozca el contenido; luego se escuchan otras comunicaciones y se sustituyen los bloques correspondientes sin siquiera habernos

molestado en descifrar.

El modo ECB fue seleccionado como el modo de operación del DES para el diseño del cifrador de datos que es implementado en este trabajo.

MODO CIFRADO DE BLOQUES ENCADENADOS

El modo de Cifrado de Bloques Encadenados ó CBC incorpora un mecanismo de retroalimentación en el cifrado por bloques. El cifrado consiste en dividir el mensaje en n bloques $p_i \in \{0,1\}^L$, sumar en módulo 2 (XOR) el texto plano con el texto cifrado anterior y aplicarle la función de cifrado de bloque E_k. Las siguientes ecuaciones definen el cifrado y descifrado CBC respectivamente:

$$c_i := E_k\left(p_i \oplus c_{i-1}\right) \ : \ 1 \le i \le n \qquad \textbf{(1.11)}$$

$$p_i := D_k\left(c_i\right) \oplus c_{i-1} \ : \ 1 \le i \le n \qquad \textbf{(1.12)}$$

El modo CBC utiliza c_{i-1} para hacer más aleatorio el texto en claro; esto esconde los patrones y repeticiones. Para habilitar el cifrado del primer bloque del texto en claro se define $c_0 := VI$, donde VI debería ser enviado de forma segura al destinatario y escogido aleatoriamente; al hacer esto último uno se asegura que un mismo texto plano al ser cifrado resulta en un criptograma distinto así se use la misma clave. Si la longitud del texto

no es múltiplo de L hay que *rellenar* como en el ECB.

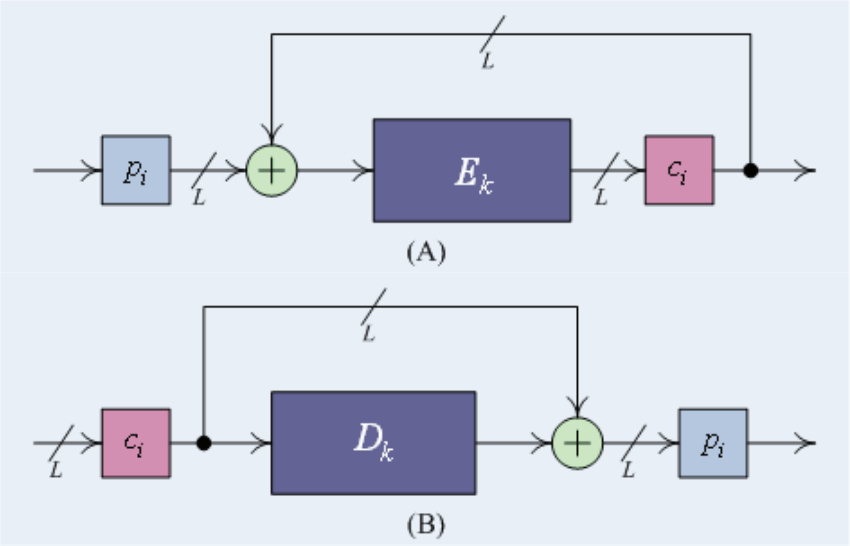

Figura 1.21 Esquema de operación en modo CBC: (A) Cifrado; (B) Descifrado.

El descifrado CBC tiene una propagación limitada de errores: los errores en el bloque cifrado i-ésimo provocan que el bloque i-ésimo del texto plano recuperado se estropee completamente y que se reproduzca en el siguiente.

MODO CIFRADO REALIMENTADO

El modo de Cifrado Realimentado ó CBC no empieza a codificar (o decodificar) hasta que no se tiene que transmitir (o se ha recibido) un bloque completo (L bits) de información. Esta circunstancia puede convertirse en un serio inconveniente, por ejemplo en el caso de terminales, que deberían poder transmitir cada carácter que pulsa el usuario de manera individual. Una posible solución sería emplear un

bloque completo para transmitir cada byte y rellenar el resto con ceros, pero esto hará que tengamos únicamente 256 mensajes diferentes en nuestra transmisión y que un atacante pueda efectuar un sencillo análisis estadístico para comprometerla. Otra opción sería rellenar el bloque con información aleatoria, aunque seguiríamos desperdiciando gran parte del ancho de banda de la transmisión.

El modo de operación CFB permite codificar la información en unidades inferiores al tamaño del bloque (L) del cifrador simétrico seleccionado, lo cual permite aprovechar totalmente la capacidad de transmisión del canal de comunicaciones, manteniendo además un nivel de seguridad adecuado.

El CFB transforma a un cifrador de bloque en un *cifrador de flujo asincrónico* (ver sección **1.3.3.1.2**). Para cifrar, el texto plano es dividido en n bloques $p_i \in \{0,1\}^r, r \leq L$; el estado $s_i \in \{0,1\}^L$ es pasado por el cifrador de bloques, los r bits más significativos de la salida son sumados en módulo 2 (XOR) al bloque del texto plano, y el flujo cifrado es realimentado al estado. Formalmente:

$$\left\| \begin{array}{l} s_{i+1} \coloneqq 2^r . s_i + c_i \\ o_i \coloneqq E_k(s_i) \big/ 2^{L-r} \\ c_i \coloneqq p_i \oplus o_i \end{array} \right. \quad \textbf{(1.13)}$$

$$\begin{aligned} s_{i+1} &:= 2^r . s_i + c_i \\ o_i &:= E_k\left(s_i\right) \big/ 2^{L-r} \\ p_i &:= c_i \oplus o_i \end{aligned} \qquad \textbf{(1.14)}$$

Los esquemas **(1.13)** y **(1.14)** son los procedimientos de cifrado CFB y descifrado CFB respectivamente. El proceso es inicializado con un vector inicial $s_0 = VI$. Si el último bloque está incompleto, a r se lo define con el valor de equivalente al número de bits que sobran. En la práctica son usados los valores $r = 1$ (flujo de bits) y $r = 8$ (flujo de bytes), esto resulta en un aprovechamiento alto del ancho de banda.

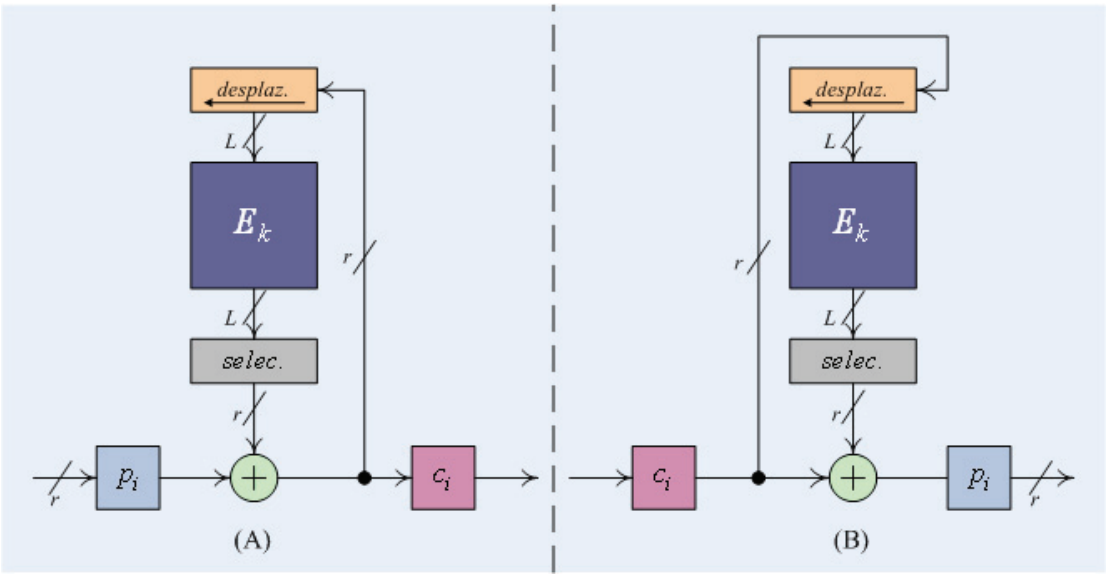

Figura 1.22 Esquema de operación en modo CFB: (A) Cifrado; (B) Descifrado.

El modo CFB es L/r veces más lento que el CBC, ya que sólo r bits son usados por cada operación de cifrado. (Nótese que si $r = L$ el modo CFB

opera igual que el CBC.)

La propagación de los errores en el CFB también es limitada, un error en el i-ésimo bloque cifrado va a ser copiado en el i-ésimo bloque plano, de ahí los siguientes L bits del texto plano (un bloque en claro más) van a ser descartados debido a que el error va a permanecer L/r pasos en el registro de estado.

MODO CIFRADO CON REALIMENTACIÓN DE SALIDA

El modo de Cifrado con Realimentación de Salida u OFB es similar al CFB, pero con la distinción de que el OFB transforma un algoritmo criptográfico simétrico de bloque en un *algoritmo de flujo asincrónico* (ver sección **1.3.3.1.2**).

Para cifrar, el texto plano es dividido en n bloques $p_i \in \{0,1\}^r, r \leq L$; el estado $s_i \in \{0,1\}^L$ es pasado por la función de cifrado de bloques, los r bits más significativos de la salida son sumados en módulo 2 (XOR) al bloque del texto plano, y a su vez realimentados al estado.

$$\left\| \begin{array}{l} s_{i+1} := 2^r . s_i + o_i \\ o_i := E_k(s_i) / 2^{L-r} \\ c_i := p_i \oplus o_i \end{array} \right. \quad \textbf{(1.15)}$$

$$\begin{aligned} s_{i+1} &:= 2^r . s_i + o_i \\ o_i &:= E_k\left(s_i\right) \big/ 2^{L-r} \\ p_i &:= c_i \oplus o_i \end{aligned} \qquad \textbf{(1.16)}$$

Los esquemas **(1.15)** y **(1.16)** son los procedimientos de cifrado OFB y descifrado OFB respectivamente. El proceso es inicializado con un vector inicial $s_0 = VI$. Si el último bloque está incompleto, a r se lo define con el valor de equivalente al número de bits que sobran. En la práctica son usados los valores $r = 1$ (flujo de bits) y $r = 8$ (flujo de bytes), para lograr un aprovechamiento eficiente del ancho de banda. El modo OFB es L/r veces más lento que el CBC, ya que sólo r bits son usados por cada operación de cifrado.

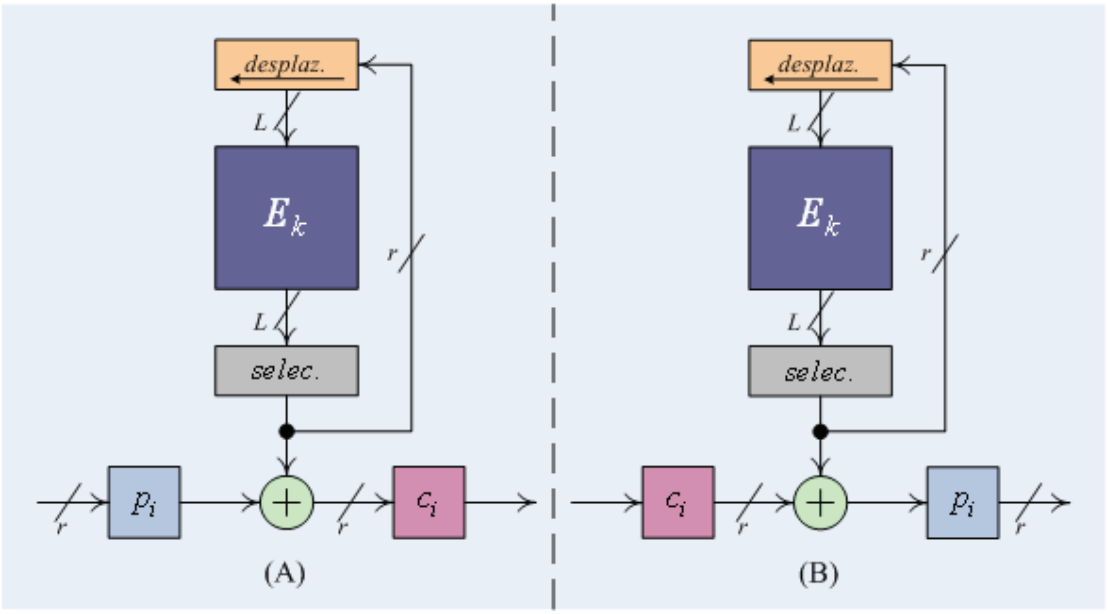

Figura 1.23 Esquema de operación en modo OFB: (A) Cifrado; (B) Descifrado.

Entre las ventajas más importantes del OFB tenemos de que no propaga el error: un bit errado en el texto cifrado genera únicamente un bit errado en el texto plano en la ubicación correspondiente. Además, dado que la clave cifrante es independiente del texto plano, puede ser generada previamente. Este modo es adecuado si la latencia entre la recepción y el descifrado es importante.

MODO CONTADOR

El modo Contador ó CTR es otra forma de convertir un cifrador de bloque en un *cifrador sincrónico de flujo*, pero a diferencia de los otros modos de operación, la clave cifrante es computada aplicando el cifrador de bloques a valores generados por lo que denominaremos un *contador*; el cual es esencialmente cualquier función que no repita el valor de su salida por un tiempo muy largo.

Para cifrar, el texto plano es dividido en n bloques $p_i \in \{0,1\}^r, r \le L$; el estado $s_i \in \{0,1\}^L$ es pasado por la función de cifrado de bloques, los r bits más significativos de la salida son sumados en módulo 2 (XOR) al bloque del texto plano; y el estado se actualiza usando la función cnt.

$$\left\| \begin{array}{l} s_{i+1} := cnt(s_i) \\ c_i := p_i \oplus \left[E_k(s_i) / 2^{L-r} \right] \end{array} \right. \qquad \textbf{(1.17)}$$

$$\left\| \begin{array}{l} s_{i+1} := cnt\left(s_i\right) \\ p_i := c_i \oplus \left[E_k\left(s_i\right) / 2^{L-r} \right] \end{array} \right. \qquad \textbf{(1.18)}$$

Los esquemas **(1.17)** y **(1.18)** son los procedimientos de cifrado CTR y descifrado CTR respectivamente. El contador es inicializado con un vector inicial $s_0 = VI$.

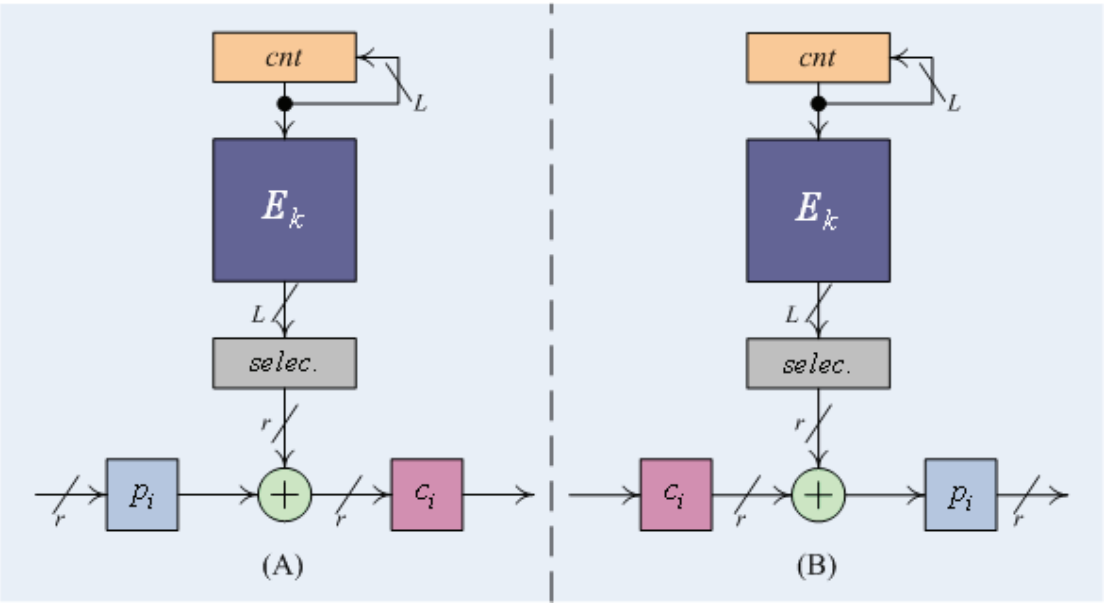

Figura 1.24 Esquema de operación en modo CTR: (A) Cifrado; (B) Descifrado.

Generalmente *cnt* se describe como:

$$cnt\left(s_i\right) = \left(s_0 + i\right) \bmod 2^L \qquad \textbf{(1.19)}$$

Existen dos variaciones del CTR, el *CTR Determinístico* (detCTR) y el

CTR Aleatorio (randCTR). El detCTR inicia con $VI = 0$ cada vez que se envía un nuevo mensaje; mientras que el randCTR adopta un valor aleatorio de VI para cada mensaje, lo cual lo hace más seguro.

1.3.3.2 Criptografía Asimétrica

Los algoritmos de clave pública, o algoritmos asimétricos, han demostrado su interés para ser empleados en redes de comunicación inseguras (Internet). Introducidos por Whitfield Difie y Martin Hellman a mediados de los años 70, su novedad fundamental con respecto a la Criptografía Simétrica es que las claves no son únicas, sino que forman pares. El algoritmo más popular por su sencillez es el RSA, que ha sobrevivido a multitud de ataques, si bien necesita una longitud de clave considerable. Otros algoritmos son los de ElGamal y Rabin.

Los algoritmos asimétricos emplean generalmente longitudes de clave mucho mayores que los simétricos. Por ejemplo, mientras que para algoritmos simétricos se considera segura una clave de 128 bits, para algoritmos asimétricos (si exceptuamos aquellos basados en curvas elípticas) se recomiendan claves de al menos 1024 bits. Además, la complejidad de cálculo que poseen estos últimos los hace considerablemente más lentos que los algoritmos de cifrado simétricos. En la práctica los métodos asimétricos se emplean únicamente para codificar la

clave de sesión (simétrica) de cada mensaje o transacción particular.

Los algoritmos asimétricos poseen dos claves diferentes en lugar de una, K_p y K_P, denominadas *clave privada* y *clave pública* respectivamente Una de ellas se emplea para codificar, mientras que la otra para decodificar.

$$D_{K_p}\left(E_{K_P}(m)\right) = m \ \vee \ D_{K_P}\left(E_{K_p}(m)\right) = m \qquad \textbf{(1.20)}$$

Dependiendo de la aplicación que le demos al algoritmo, la clave pública será la de cifrado o viceversa. Para que estos criptosistemas sean seguros también ha de cumplirse que a partir de una de las claves resulte extremadamente difícil calcular la otra.

PROTECCIÓN DE LA INFORMACIÓN

Una de las aplicaciones inmediatas de los algoritmos asimétricos es el cifrado de la información sin tener que transmitir la clave de decodificación, lo cual permite su uso en canales inseguros.

Supongamos que A quiere enviar un mensaje a B (figura 1.23). Para ello solicita a B su clave pública K_P. A genera entonces el mensaje cifrado $E_{K_P}(m)$. Una vez hecho esto únicamente quien posea la clave K_p, en nuestro ejemplo B, podrá recuperar el mensaje original m.

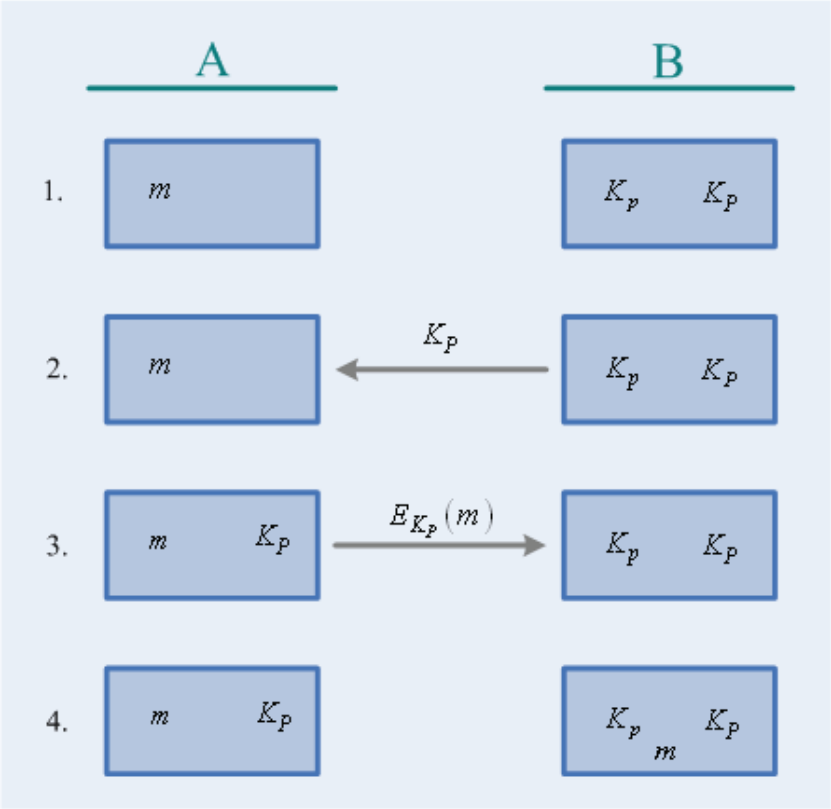

Figura 1.25 Transmisión de información empleando algoritmos asimétricos.

Nótese que para este tipo de aplicación, la clave que se hace pública es aquella que permite codificar los mensajes, mientras que la clave privada es aquella que permite descifrarlos.

AUTENTIFICACIÓN

La segunda aplicación de los algoritmos asimétricos es la autentificación de mensajes, con ayuda de *funciones de resumen* (ver sección **1.3.3.2.1**), que nos permiten obtener una *firma digital* a partir de un mensaje. Dicha firma es mucho más pequeña que el mensaje original, y es muy difícil encontrar otro mensaje que de lugar a la misma. Supongamos que A recibe un mensaje m

de B y quiere comprobar su autenticidad. Para ello B genera un resumen del mensaje $r(m)$ (ver figura 1.24) y lo codifica empleando la clave de cifrado, que en este caso será privada.

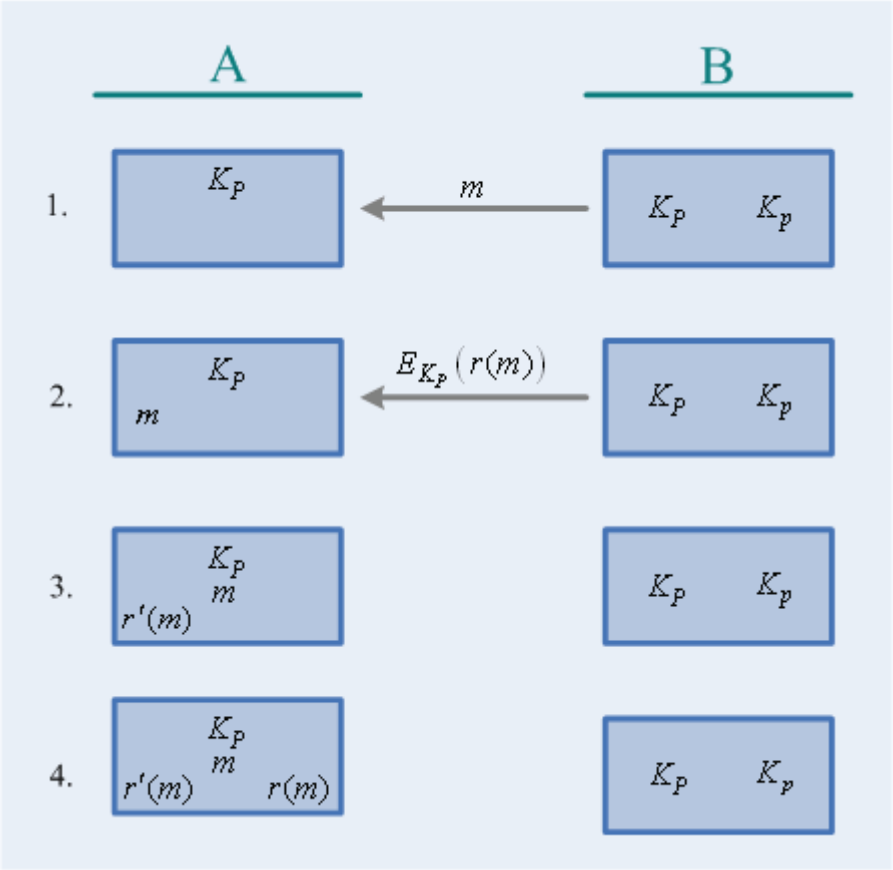

Figura 1.26 Autentificación de información empleando algoritmos asimétricos.

La clave de descifrado se habrá hecho pública previamente, y debe estar en poder de A. B envía entonces a A el criptograma correspondiente a $r(m)$. A puede ahora generar su propia $r'(m)$ y compararla con el valor $r(m)$ obtenido del criptograma enviado por B. Si coinciden, el mensaje será auténtico, puesto que el único que posee la clave para codificar es

precisamente B. Nótese que en este caso la clave que se emplea para cifrar es la clave privada, justo al revés que para la simple codificación de mensajes.

En muchos de los algoritmos asimétricos ambas claves sirven tanto para cifrar como para descifrar, de manera que si empleamos una para codificar, la otra permitirá decodificar y viceversa. Esto ocurre con el algoritmo RSA, en el que un único par de claves es suficiente para codificar y autentificar.

1.3.3.2.1 Funciones de Resumen (Hash)

En la sección anterior vimos que la Criptografía Asimétrica permitía autentificar información, es decir, poder asegurar que un mensaje m proviene de un emisor A y no de cualquier otro. Asimismo vimos que la autentificación debía hacerse empleando una función resumen y no codificando el mensaje completo. Las *funciones resumen* (*hash*) son también conocidas como MDC (modification detection codes), y son las que permiten crear *firmas digitales*.

Sabemos que un mensaje m puede ser autentificado codificando con la clave privada K_p al resultado de aplicarle una función resumen, $E_{K_p}\left(r(m)\right)$. Esa información adicional (que denominaremos *firma* o

signatura del mensaje m) sólo puede ser generada por el poseedor de la clave privada K_p. Cualquiera que tenga la clave pública correspondiente estará en condiciones de decodificar y verificar la firma. Para que sea segura, la función resumen $r(x)$ debe cumplir además ciertas características:

- $r(m)$ debe ser de longitud fija, independientemente de la longitud de m.
- Dado m, tiene que ser fácil calcular $r(m)$.
- Dado $r(m)$, recuperar m debe ser computacionalmente intratable.
- Dado m, obtener un m' tal que $r(m) = r(m')$ tiene que ser computacionalmente intratable.

En general, las funciones resumen se basan en la idea de *funciones de compresión*, que dan como resultado bloques de longitud n a partir de bloques de longitud m. Estas funciones se encadenan de forma iterativa, haciendo que la entrada en el paso i sea función del i-ésimo bloque del mensaje y de la salida del paso $i-1$ (ver figura 1.25).

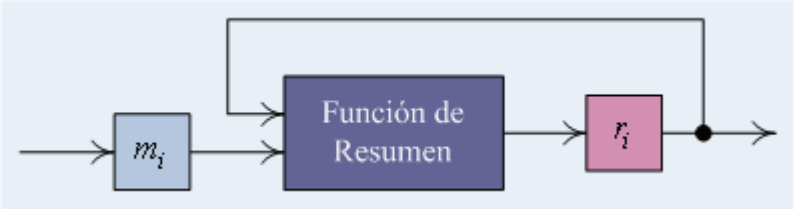

Figura 1.27 Estructura iterativa de una función de resumen.

En general, se suele incluir en alguno de los bloques del mensaje m, al principio o al final, información sobre la longitud total del mensaje. De esta forma se reducen las probabilidades de que dos mensajes con diferentes longitudes den el mismo valor en su resumen. Las funciones de resumen son *unidireccionales*.

Entre los algoritmos de generación de firmas más usados tenemos el Algoritmo de Resumen de Mensaje 5 (MD5) y el Algoritmo de Resumen Seguro 1 (SHA-1).

II CAPÍTULO

2 EL CIFRADO DE DATOS NORMALIZADO

En este capítulo describiremos en detalle el algoritmo seleccionado para implementar el cifrador de este proyecto, el Estándar de Cifrado de Datos (DES, Data Encryption Standard), su funcionamiento, características y propiedades.

2.1 Generalidades del Cifrado Normalizado de Datos

El DES es el algoritmo simétrico más extendido mundialmente. Se basa en el algoritmo Lucifer, que había sido desarrollado por IBM a principios de los setenta. Fue adoptado como estándar por el Gobierno de los EE.UU. para comunicaciones no clasificadas en 1976. Originalmente la Agencia de Seguridad Nacional del los EE.UU. (NSA) lo diseñó para ser implementado por hardware, creyendo que los detalles iban a ser mantenidos en secreto, pero la Oficina Nacional de Estandarización (NBS), ahora Instituto Nacional de Estándares y Tecnologías (NITS), publicó su especificación en el estándar FIPS 46, suficientemente detallado (con algunas modificaciones) como para que

cualquiera pudiera implementarlo por software. No fue casualidad que el siguiente algoritmo adoptado (el Skipjack) fuera mantenido en secreto.

Las modificaciones introducidas por el NITS consistieron básicamente en la reducción de la longitud de clave, de 128 bits a 64 bits (56 bits de clave como tal y 8 bits de paridad), y de los bloques de datos, de 128 bits a 64 bits. La descripción oficial del DES se la encuentra en el documento FIPS PUB 46-3[19].

A mediados de 1998, se demostró que un ataque por *fuerza bruta*† al DES era viable, debido a la escasa longitud que emplea en su clave. No obstante, el DES aún no ha demostrado ninguna debilidad grave desde el punto de vista teórico, por lo que su estudio sigue siendo plenamente interesante.

2.2 El Proceso de Cifrado

El algoritmo DES es un cifrador de producto, basado en un *Cifrador Feistel* con algunas inclusiones. Valiéndonos de las definiciones de la sección **1.3.3**, podemos describir formalmente al DES como sigue:

- El alfabeto utilizado tanto para los textos planos como para los textos cifrados es el alfabeto binario: $A_m = A_c = \{0,1\}$;

† Un ataque por fuerza bruta (Brute-force attack) consiste en probar todas las posibles soluciones sin ningún tipo de criterio que guíe la búsqueda.

- Los espacios de texto plano y texto cifrado son $M = C = \{0,1\}^{64}$, i.e. $L = 64$;
- El espacio de claves es $K = \{0,1\}^{56}$, aunque oficialmente las claves son de longitud 64. Esto se debe a que en el estándar el octavo bit de cada byte es tomado como *bit de paridad* (8 en total), pero se usan únicamente los primeros 7 bits de cada byte para computar el criptograma (56 en total).

El DES consta de cuatro etapas principales: Un Cifrador Feistel (F), un Esquema de Claves (K_s) y dos Permutaciones, una antes (P_I) y otra después (P_F) del Cifrado Feistel.

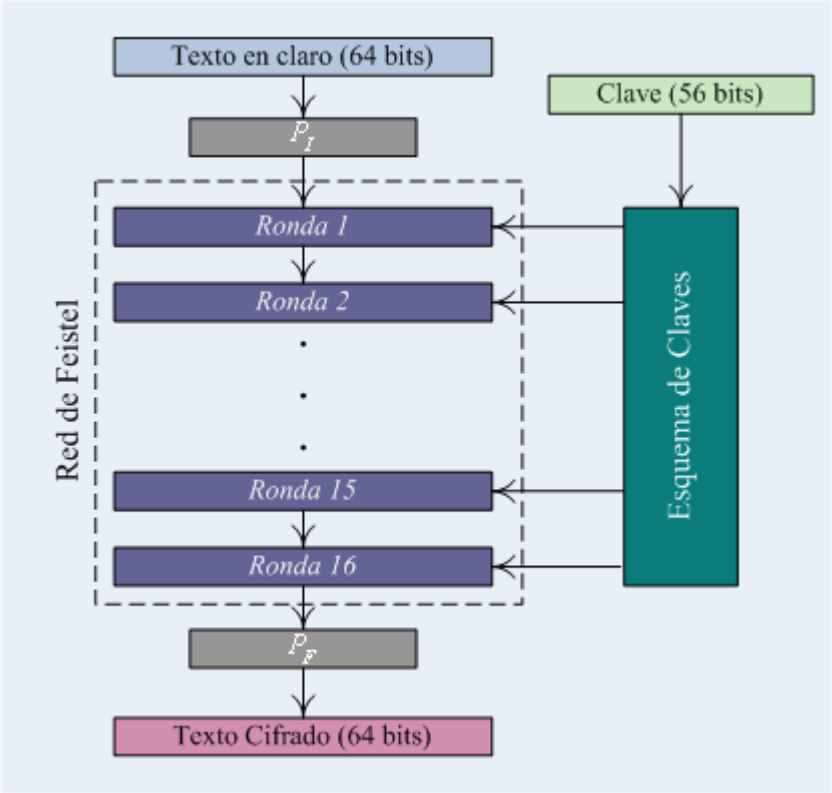

Figura 2.1 Esquema simplificado de operación del DES.

La función de cifrado DES se la define formalmente como:

$$E_k : \{0,1\}^{64} \times \{0,1\}^{56} \to \{0,1\}^{64}$$
$$(m,k) \mapsto E_k(m) = P_F\left(F\left[P_I(m), K_s(k)\right]\right) \qquad \textbf{(2.1)}$$

Cada uno de sus componentes se describe a continuación.

2.2.1 Permutación Inicial

Antes de entrar a la primera ronda de la Red de Feistel, se aplica una transposición, denominada *permutación inicial* P_I, al texto claro. P_I es una permutación sobre el conjunto $\mathbb{N}_{64}$. Si el arreglo P_0 representa el las posiciones originales de los bits del texto plano, donde

$$P_0 = \begin{Bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 \\ 17 & 18 & 19 & 20 & 21 & 22 & 23 & 24 \\ 25 & 26 & 27 & 28 & 29 & 30 & 31 & 32 \\ 33 & 34 & 35 & 36 & 37 & 38 & 39 & 40 \\ 41 & 42 & 43 & 44 & 45 & 46 & 47 & 48 \\ 49 & 50 & 51 & 52 & 53 & 54 & 55 & 56 \\ 57 & 58 & 59 & 60 & 61 & 62 & 63 & 64 \end{Bmatrix} \qquad \textbf{(2.2)}$$

entonces, el patrón de transposición de P_I es representado como:

$$P_I = \begin{Bmatrix} 58 & 50 & 42 & 34 & 26 & 18 & 10 & 2 \\ 60 & 52 & 44 & 36 & 28 & 20 & 12 & 4 \\ 62 & 54 & 46 & 38 & 30 & 22 & 14 & 6 \\ 64 & 56 & 48 & 40 & 32 & 24 & 16 & 8 \\ 57 & 49 & 41 & 33 & 25 & 17 & 9 & 1 \\ 59 & 51 & 43 & 35 & 27 & 19 & 11 & 3 \\ 61 & 53 & 45 & 37 & 29 & 21 & 13 & 5 \\ 63 & 55 & 47 & 39 & 31 & 23 & 15 & 7 \end{Bmatrix} \quad \textbf{(2.3)}$$

Esto quiere decir que el bit 58 del bloque original fue trasladado a la primera posición, el bit 50 a la segunda, mientras que el primer bit fue movido a la posición 40, el segundo bit a la posición 8, y así según el arreglo. (Usaremos este mismo tipo de representación para todas las demás permutaciones.)

El resultado de esta etapa pasará a la Red Feistel como $F_0 := P_I(m)$.

2.2.2 Red de Feistel

La Red de Feistel es una *red de sustitución-permutación* (ver sección **1.3.3.1.1**) hecha precisamente para diseñar cifradores simétricos. Fue descrita por Horst Feistel mientras trabajaba en el algoritmo *Lucifer*, en IBM.†

† La Red de Feistel es utilizada en algunos algoritmos simétricos de bloque como DES, Lucifer, FEAL, CAST, Blowfish, IDEA, RC5 y Skipjack.

Un Cifrador de Feistel, formalmente, se define como una transformación $F:\{0,1\}^{2t} \to \{0,1\}^{2t}; t \in \mathbb{N}$. El proceso consiste en la repetición sucesiva de una estructura de cifrado sencilla denominada *ronda*. Cada *ronda* va una a continuación de la otra, en cascada: cada una recibe un bloque de $2t$ bits, lo trata, y le pasa el resultado a la siguiente que tratará dicho valor y le pasará el resultado a la siguiente, y así r veces.

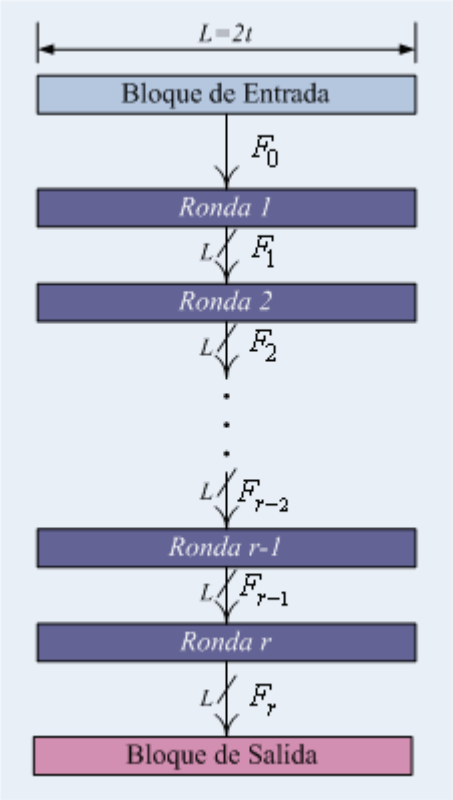

Figura 2.2 Esquema de operación de una Red de Feistel.

La Red de Feistel que incorpora el DES está definida para bloque de datos de $L = 2t = 64$ y con un número de rondas $r = 16$.

2.2.2.1 Ronda Sencilla de Feistel

Cada ronda individual de la Red de Feistel es una función que consiste en un *cifrador de producto*; cada una posee la misma estructura iterativa, que vamos a describir a continuación. Para $i = 1, 2, ..., r$:

1. Se divide cada bloque entrante F_{i-1} en dos mitades, $L_{i-1}, R_{i-1} \in \{0,1\}^t$:

$$\left\| \begin{array}{l} L_{i-1} := F_{i-1} / 2^t \\ R_{i-1} := F_{i-1} \bmod 2^t \end{array} \right. \quad \textbf{(2.4)}$$

L_{i-1} representa a los t bits más significativos y R_{i-1} a los t bits menos significativos: $F_{i-1} \equiv L_{i-1} \,||\, R_{i-1}$

2. Se aplica el cifrado de producto:

$$\left\| \begin{array}{l} L_i := R_{i-1} \\ R_i := L_{i-1} \oplus f_i\left(R_{i-1}\right) \end{array} \right. \quad \textbf{(2.5)}$$

donde f_i es la denominada *función de ronda;* propia de cada algoritmo.

3. Se compone el bloque de salida:†

$$F_i := L_i \,||\, R_i \quad \textbf{(2.6)}$$

---

† En algunos algoritmos, como el DES, en la última ronda (r) las mitades del bloque de salida se intercambian, esto es, $F_r := R_r \| L_r$.

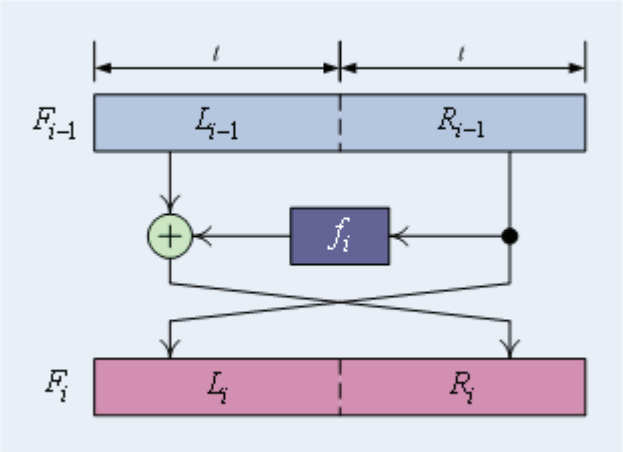

Figura 2.3 Esquema de una ronda de Feistel.

Formalmente, definimos una ronda (i-ésima) de la Red de Feistel como:

$$G_i : \{0,1\}^{2t} \to \{0,1\}^{2t}$$
$$L \,||\, R \mapsto G_i(L \,||\, R) = R \,||\, L \oplus f_i(R) \qquad \textbf{(2.7)}$$

Desde este punto de vista, podemos pensar en un cifrador de Feistel F como una composición de r funciones de ronda G_i, esto es $F(p) \equiv G_r(G_{r-1}(...G_1(p)...)) \equiv G_r \circ G_{r-1} \circ ... \circ G_1(p)$.

2.2.2.2 Función f

La función de ronda f es el corazón de la ronda de Feistel y cada algoritmo (DES, RC5, IDEA, etc.) tiene la suya. Por el esquema iterativo del Cifrador

Feistel, la función de ronda puede ser *unidireccional* (sin inversa); pudiéndose usar de *funciones de resumen* (sección **1.3.3.2.1**).

En muchos casos la función de ronda no depende solamente del segmento R, sino que además requiere de una *clave de ronda* $k_i \in \{0,1\}^u ; u \in \mathbb{N}$, es decir que $f_i(R) = f(R, k_i)$. Se redefine entonces la función de ronda como $f : \{0,1\}^t \times \{0,1\}^u \to \{0,1\}^t$; y por consiguiente, también el Cifrador de Feistel se redefine como:

$$\begin{aligned} F : \{0,1\}^{2t} \times \{0,1\}^{r.u} &\to \{0,1\}^{2t} \\ (p, K_s(k)) &\mapsto F(p, K_s(k)) \end{aligned} \qquad \textbf{(2.8)}$$

Donde K_s es un denominado Esquema de Claves que genera r claves de ronda k_i, una para cada ronda de la Red de Feistel, a partir de una *clave principal* $k \in \{0,1\}^l ; l \in \mathbb{N}$. Formalmente:

$$\begin{aligned} K_s : \{0,1\}^l &\to \{0,1\}^{r.u} \\ k &\mapsto K_s(k) = k_1 \,||\, k_2 \,||\, ... \,||\, k_i \,||\, ... \,||\, k_r \end{aligned} \qquad \textbf{(2.9)}$$

En el DES, la longitud de la clave principal es $l = 56$ y la de las claves de ronda es $u = 48$. La función de ronda f del DES consiste en la secuencia:

1. Se aplica una *permutación de expansión*, E, al segmento R.
2. Se suma en módulo 2 (XOR) el segmento expandido con la clave de ronda (cuya obtención se explica en la sección **2.2.4**).
3. Se toma ese valor y se lo hace pasar por una etapa de sustitución, S, que devuelve el tamaño original al segmento.
4. Se aplica una permutación P al resultado anterior.

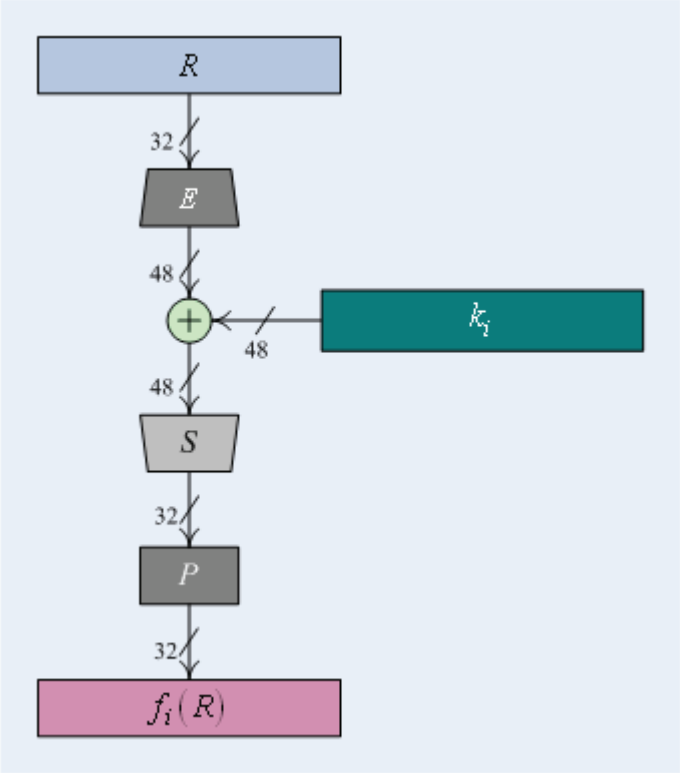

Figura 2.4 Estructura de la función de ronda f para el DES.

Nótese que, por la naturaleza de sus operaciones, esta función es irreversible, o sea, unidireccional. Formalmente la función de ronda f del DES es definida como:

$$f:\{0,1\}^{32}\times\{0,1\}^{48}\rightarrow\{0,1\}^{32}$$
$$(R,k)\mapsto f(R,k)=P\left(S\left(E(R)\oplus k\right)\right) \quad \textbf{(2.10)}$$

Cada fase se describe a continuación.

2.2.2.2.1 Expansión E

La expansión E es una función que consigue dos objetivos: transponer los bits del segmento de entrada y aumentar su longitud mediante la repetición de algunos bits. Es definida formalmente como:

$$E:\{0,1\}^{32}\rightarrow\{0,1\}^{48}$$
$$R\mapsto E(R) \quad \textbf{(2.11)}$$

El siguiente arreglo sintetiza la transposición operada por la expansión.

$$E=\begin{Bmatrix}32 & 1 & 2 & 3 & 4 & 5\\ 4 & 5 & 6 & 7 & 8 & 9\\ 8 & 9 & 10 & 11 & 12 & 13\\ 12 & 13 & 14 & 15 & 16 & 17\\ 16 & 17 & 18 & 19 & 20 & 21\\ 20 & 21 & 22 & 23 & 24 & 25\\ 24 & 25 & 26 & 27 & 28 & 29\\ 28 & 29 & 30 & 31 & 32 & 1\end{Bmatrix} \quad \textbf{(2.12)}$$

El efecto de esta operación se lo entiende mejor al observar su patrón ilustrado en la figura 2.5.

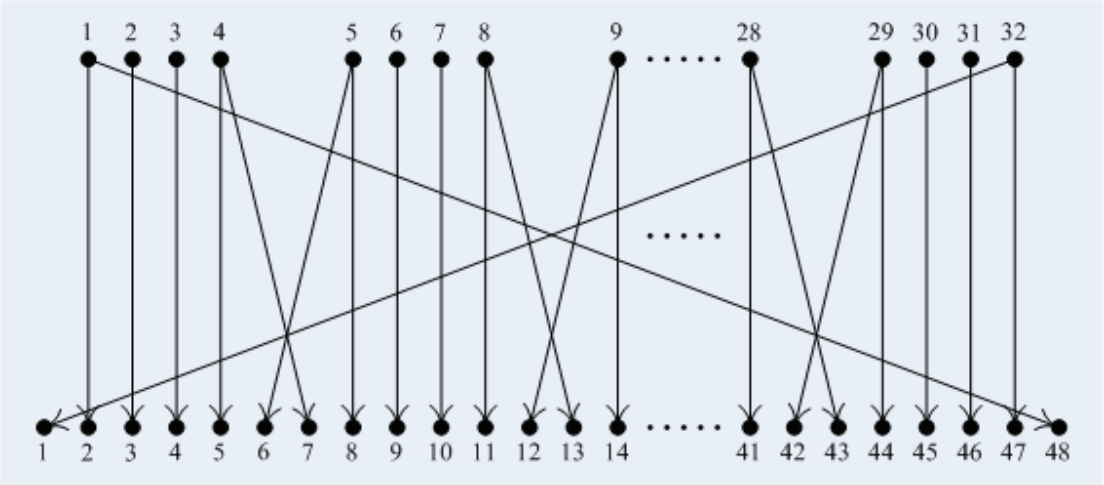

Figura 2.5 Patrón de transposición-expansión de la función E.

2.2.2.2.2 Disyunción Exclusiva

La suma en módulo 2 (XOR) es utilizada en el DES para dotar de *difusión* al cifrador. La suma con la clave de ronda $E\left(R_{i-1}\right) \oplus k_i$ (denominada *Key-mixing*) crea dependencia del texto cifrado a la clave principal k, y la suma $L_{i-1} \oplus f_i\left(R_{i-1}\right)$ propaga ese efecto y fortalece la dependencia al texto plano. Así se obtiene un proceso criptográfico en el cual el cambio de un solo bit de la clave o del texto plano producirá un cambio total en el texto cifrado.

Definiremos $B_i \in \{0,1\}^6$ como la salida de esta etapa:

$$B_i := E(R_{i-1}) \oplus k_i \qquad \textbf{(2.13)}$$

2.2.2.2.3 Sustitución *S*

La función de sustitución S se encarga, por un lado, de dotar de la propiedad de la *confusión* al cifrado DES, y por el otro hace las veces de *compresor*, al reducir la longitud del segmento de entrada de 48 a 32 bits.

La operación de sustitución es aplicada al segmento B_i por fragmentos. Se comienza por dividir a B_i en 8 fragmentos $B_i^j \in \{0,1\}^6$, $j = \overline{1,8}$:

$$B_i^j := \left(B_i / 2^{6(8-j)}\right) \bmod 2^6 \qquad \textbf{(2.14)}$$

$$B_i \equiv B_i^1 \,||\, B_i^2 \,||\, B_i^3 \,||\, B_i^4 \,||\, B_i^5 \,||\, B_i^6 \,||\, B_i^7 \,||\, B_i^8 \qquad \textbf{(2.15)}$$

A cada fragmento B_i^j se le aplica una de las funciones de sustitución $S^j : \{0,1\}^6 \to \{0,1\}^4$; denominadas *Cajas S*. (Ver figura 2.4.) Cada función S^j es una tabla o matriz de 4x16, que contiene 64 posibles

valores de sustitución $s_{n,m}^{j} \in \{0,1\}^4$ para la entrada B_i^j.

$$S^j\left(B_i^j\right) = s_{m,n}^j \tag{2.16}$$

Las filas de cada S^j están indizadas de 0 a 3, y las columnas de 0 a 15. Así pues, el índice de las filas puede ser descrito con 2 bits, y el índice de las columnas con 4 bits. Cada B_i^j es una cadena de bits $b_5b_4b_3b_2b_1b_0$; para identificar la fila y la columna usamos: $m := b_5b_0$ y $n := b_4b_3b_2b_1$.

$$\begin{aligned} S : \{0,1\}^{48} &\rightarrow \{0,1\}^{32} \\ B_i &\mapsto S(B_i) = S^1\left(B_i^1\right) \| S^2\left(B_i^2\right) \| \ldots \| S^8\left(B_i^8\right) \end{aligned} \tag{2.17}$$

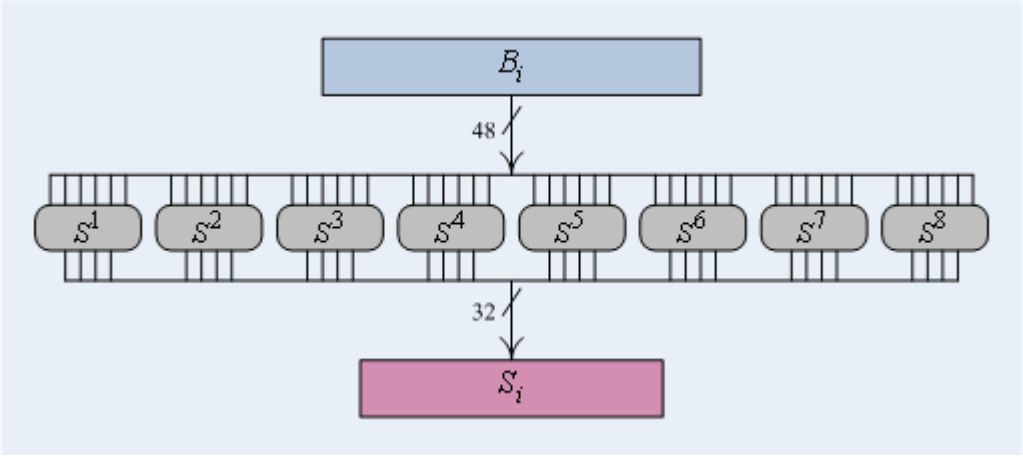

Figura 2.6 Ilustración de la etapa de Sustitución de la función f del DES.

La Cajas S tienen la propiedad de que para cualquier par $u, v \in \{0,1\}^6$

que difieran en un solo bit, lo correspondientes valores de la función $S^j(u)$ y $S^j(v)$ difieren en al menos dos bits. Cada fila de una tabla S^j es una permutación sobre $\mathbb{Z}_{16}$.

Las tablas de las Cajas S del DES se encuentran especificadas en el estándar FIPS 46-3, y las presentamos en el apéndice B.

2.2.2.2.4 Permutación *P*

Para culminar la función f, la permutación P, sobre el conjunto $\mathbb{N}_{32}$, es usada para transponer los bits del segmento S_i.

$$P = \begin{Bmatrix} 14 & 7 & 20 & 21 \\ 29 & 12 & 28 & 17 \\ 1 & 15 & 23 & 26 \\ 5 & 18 & 31 & 10 \\ 2 & 8 & 24 & 14 \\ 32 & 27 & 3 & 9 \\ 19 & 13 & 30 & 6 \\ 22 & 11 & 4 & 25 \end{Bmatrix} \quad \textbf{(2.18)}$$

Esta permutación entrecruza los resultados de las Cajas S, aumentando la confusión.

2.2.3 Permutación Final

P_F es una permutación sobre el conjunto $\mathbb{N}_{64}$ y es usada para transponer los bits del bloque de salida del Cifrador Feistel, F_{16}. Esta permutación es la inversa de P_I, esto es $P_F = P_I^{-1}$; esto quiere decir que $P_F \circ P_I = P_0$.

Su patrón de transposición está representado en el siguiente arreglo:

$$P_F = \begin{Bmatrix} 40 & 8 & 48 & 16 & 56 & 24 & 64 & 32 \\ 39 & 7 & 47 & 15 & 55 & 23 & 63 & 31 \\ 38 & 6 & 46 & 14 & 54 & 22 & 62 & 30 \\ 37 & 5 & 45 & 13 & 53 & 21 & 61 & 29 \\ 36 & 4 & 44 & 12 & 52 & 20 & 60 & 28 \\ 35 & 3 & 43 & 11 & 51 & 19 & 59 & 27 \\ 34 & 2 & 42 & 10 & 50 & 18 & 58 & 26 \\ 33 & 1 & 41 & 9 & 49 & 17 & 57 & 25 \end{Bmatrix} \quad \textbf{(2.19)}$$

2.2.4 El Esquema de Llaves

El Esquema de Claves del algoritmo DES recibe una clave principal (secreta) $k \in \{0,1\}^{56}$ como entrada y produce 16 claves de ronda $k_i \in \{0,1\}^{48}$, una por cada ronda del Cifrador Feistel:

$$K_s : \{0,1\}^{56} \to \{0,1\}^{16\times 48}$$
$$k \mapsto K_s(k) = k_1 \,||\, k_2 \,||\, ... \,||\, k_i \,||\, ... \,||\, k_{16} \quad \textbf{(2.20)}$$

El proceso de generación de claves de ronda del DES es mucho más simple que su proceso de cifrado:

4. Se recibe la clave del sistema $k \in \{0,1\}^{64}$ (que incluye 8 bits adicionales para la paridad) y se le aplica la *elección permutada* PC_1, obteniendo una clave efectiva $k_0 \in \{0,1\}^{56}$.

5. Se divide k_0 en dos segmentos $C_0, D_0 \in \{0,1\}^{28}$, siendo C_0 los 28 bits más significativos y D_0 los 28 bits menos significativos.

6. Se hace un desplazamiento a cada segmento C_i y D_i, de s_i bits (véase la sección **2.2.4.2**) hacia la izquierda:

$$\left\| \begin{array}{l} C_i := C_{i-1}.2^{s_{i-1}} \\ D_i := D_{i-1}.2^{s_{i-1}} \end{array} \right. \quad \textbf{(2.21)}$$

7. A cada par C_i, D_i se le aplica la elección permutada PC_2 que descarta otros 8 bits para obtener finalmente la clave de ronda $k_i \in \{0,1\}^{48}$.

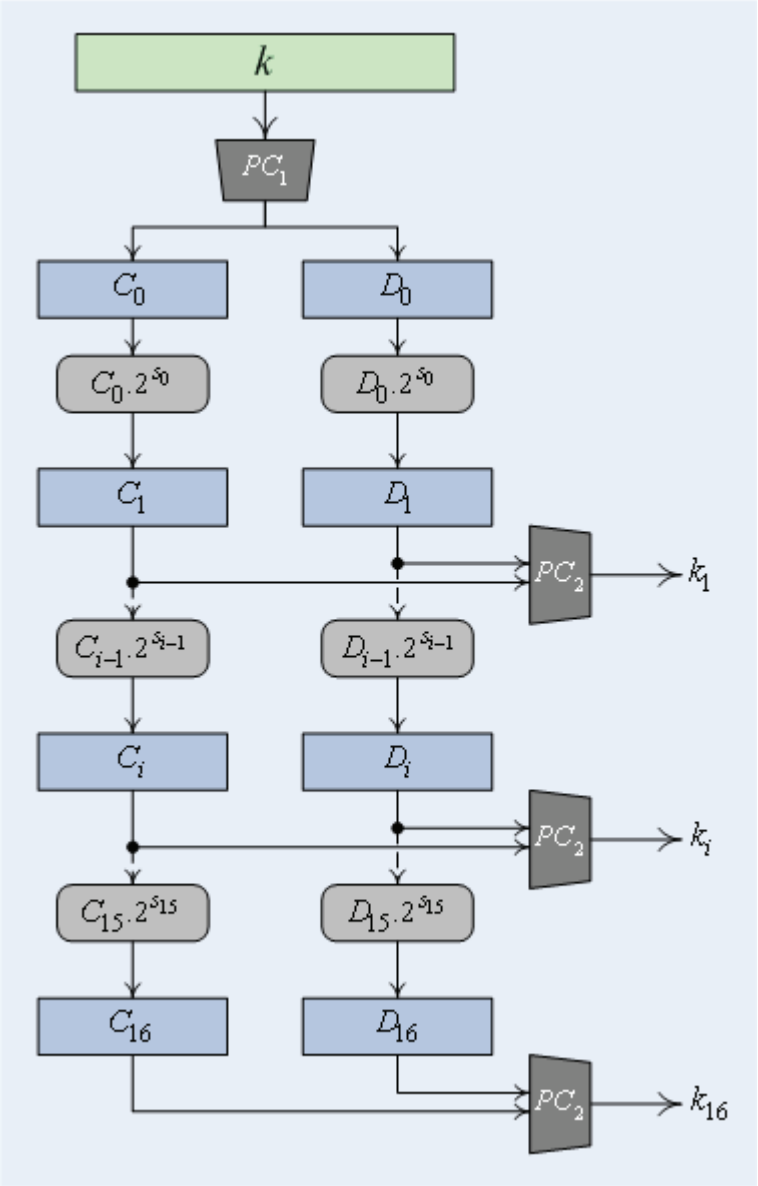

Figura 2.7 Esquema de generación de subclaves del DES.

2.2.4.1 Permutaciones PC1 y PC2

Las Elecciones Permutadas son funciones que permutan los bits de la clave y descartan 8 de los bits recibidos.

La *Primera Elección Permutada* se define como una función $PC_1 : \{0,1\}^{64} \rightarrow \{0,1\}^{56}$, que excluye los bits de paridad de la clave principal k.

$$PC_1 = \begin{Bmatrix} 57 & 49 & 41 & 33 & 25 & 17 & 9 \\ 1 & 41 & 50 & 42 & 34 & 26 & 18 \\ 10 & 50 & 59 & 51 & 43 & 35 & 27 \\ 19 & 59 & 3 & 60 & 52 & 44 & 36 \\ 63 & 55 & 47 & 39 & 31 & 23 & 15 \\ 7 & 62 & 54 & 46 & 38 & 30 & 22 \\ 14 & 6 & 61 & 53 & 45 & 37 & 29 \\ 21 & 13 & 5 & 28 & 20 & 12 & 4 \end{Bmatrix} \quad \textbf{(2.22)}$$

La *Segunda Elección Permutada* es una función $PC_2 : \{0,1\}^{56} \rightarrow \{0,1\}^{48}$ que recibe como entrada la unión $C_i \,||\, D_i$ y devuelve una clave de ronda k_i.

$$PC_2 = \begin{Bmatrix} 14 & 17 & 11 & 24 & 1 & 5 \\ 3 & 28 & 15 & 6 & 21 & 10 \\ 23 & 19 & 12 & 4 & 26 & 8 \\ 16 & 7 & 27 & 20 & 13 & 2 \\ 41 & 52 & 31 & 37 & 47 & 55 \\ 30 & 40 & 51 & 45 & 33 & 48 \\ 44 & 49 & 39 & 56 & 34 & 53 \\ 46 & 42 & 50 & 36 & 29 & 32 \end{Bmatrix} \quad \textbf{(2.23)}$$

2.2.4.2 Secuencia de Desplazamientos

La secuencia de desplazamientos que se aplica a cada par C_i, D_i del Esquema de Llaves se especifica en el estándar FIPS 46-3. La siguiente tabla muestra los valores de cada s_i:

Índice i	*Desplazamiento* s_i
0	1
1	1
2	2
3	2
4	2
5	2
6	2
7	2
8	1
9	2
10	2
11	2
12	2
13	2
14	2
15	1

Tabla 2.I Bits de desplazamiento para el DES.

Finalmente, la figura 2.7 nos muestra el esquema iterativo completo del algoritmo de cifrado del DES, logrado al integrar el Cifrador de Feistel y el Esquema de Claves, junto con las permutaciones Inicial y Final.

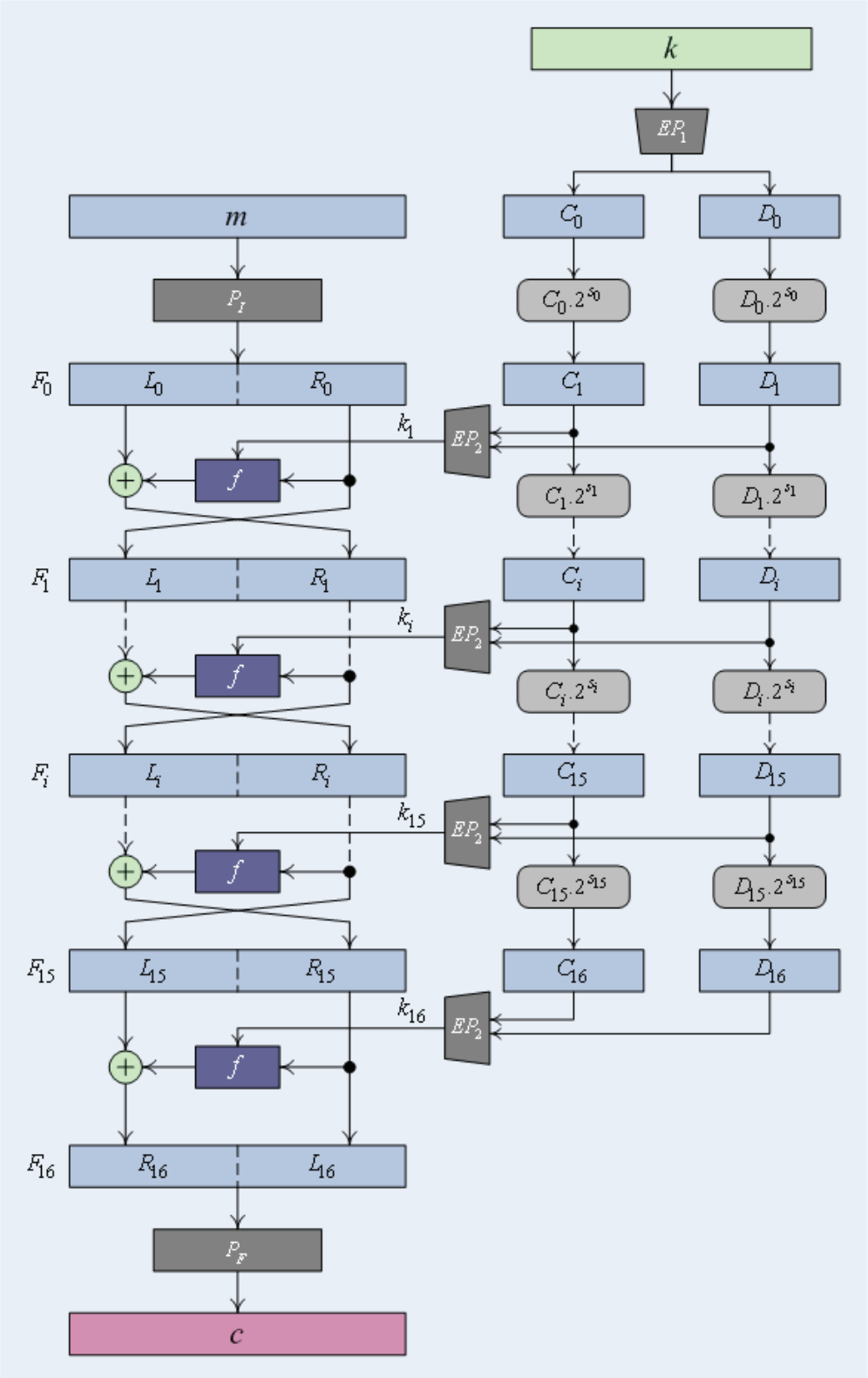

Figura 2.8 Esquema de iterativo global del DES

2.3 Proceso de Descifrado

El procedimiento de descifrado D_k del DES se obtiene a partir de dos propiedades intrínsecas de su estructura:

1. La permutaciones P_I y P_F se anulan entre sí; eliminando la una el efecto de la otra sobre el texto.
2. El proceso de descifrado para una Red de Feistel consiste en pasar el texto cifrado por el mismo cifrador, pero invirtiendo el orden de ingreso de las claves de ronda.

Expandiendo un poco más el punto 2, si definimos $\overline{K_s}$ como la secuencia del Esquema de Claves en orden inverso, $\overline{K_s}(k) = k_{16} \,||\, k_{15} \,||\, ... \,||\, k_i \,||\, ... \,||\, k_1$, entonces la función de descifrado de Feistel se define como:

$$F^{-1}\left(c, K_s(k)\right) = F\left(c, \overline{K_s}(k)\right) = p \qquad \textbf{(2.24)}$$

Esto quiere decir que si se colocan dos redes de Feistel en cascada, y se transpone el orden de ingreso de las llaves de ronda de la segunda, ambas se anulan. Para el DES este comportamiento se mantiene, salvo que se incluyen las permutaciones P_I y P_F. La figura 2.9 ilustra cómo se logra el descifrado DES.

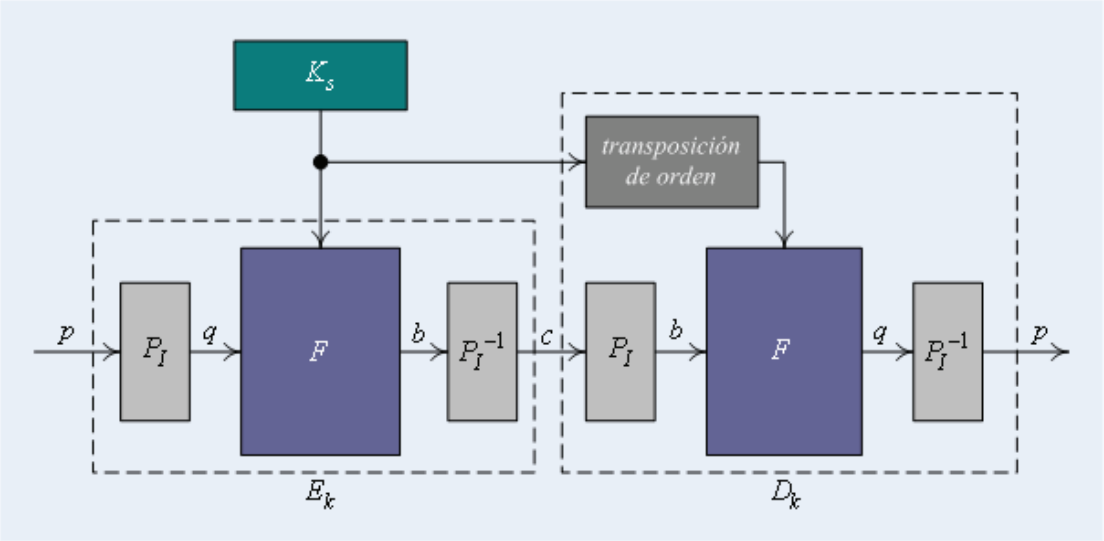

Figura 2.9 Ilustración del mecanismo de descifrado DES.

Tal como se puede apreciar, para descifrar un criptograma DES se utiliza la misma estructura de la función de cifrado, con la única salvedad de que se transpone el orden de ingreso de las llaves. Formalmente definimos a la función de descifrado DES como:

$$D_k : \{0,1\}^{64} \times \{0,1\}^{56} \to \{0,1\}^{64}$$
$$(c,k) \mapsto D_k(c) = P_F\left(F\left[P_I(c), \overline{K_s}(k)\right]\right) \qquad \textbf{(2.25)}$$

Gracias a esta característica del DES, un mismo circuito integrado que implemente este algoritmo fácilmente puede funcionar como cifrador o descifrador; tan sólo incluyendo una fase de transposición de orden en la estructura del integrado antes del ingreso de las llaves de ronda.

III CAPÍTULO

3 IMPLEMENTACIÓN DEL CIFRADO DE DATOS NORMALIZADO

En este capítulo se explica cómo se ha implementado el cifrador de datos empleando el algoritmo DES. Expondremos por qué escogimos el DES como algoritmo de cifrado; presentaremos el diseño, los materiales utilizados, la estrategia de implementación y el funcionamiento de cada componente.

3.1 Selección del Cifrado de Datos Normalizado como algoritmo de cifrado a implementarse

Los factores que influyeron en la selección del algoritmo del Cifrado de Datos Estándar (DES) como cifrador a ser implementado fueron los siguientes:

1. Sus especificaciones de diseño pertenecen al dominio público y estaban suficientemente detalladas para poder implementarlo.
2. Su vigencia en el medio de la seguridad de datos aún permanece en muchos entornos como las transacciones bancarias y la Telemetría.

3. En su diseño, las operaciones fundamentales que lo componen son muy viables de implementar en la tecnología de hardware (los FPGA) y con la herramienta de diseño (el lenguaje VHDL) propuestas.
4. Su diseño es muy útil (didácticas) para comprender las técnicas de cifrado modernas.

3.2 Implementación del Circuito

El objetivo es implementar un circuito cifrador de datos, utilizando el Cifrado de Datos Estándar (DES) como algoritmo de cifrado, en un circuito integrado de Arreglos de Puertas Programables por Campos (FPGA), valiéndonos del Lenguaje de Descripción de Hardware de Circuitos Integrados de Alta Velocidad (VHDL) como herramienta de diseño digital.

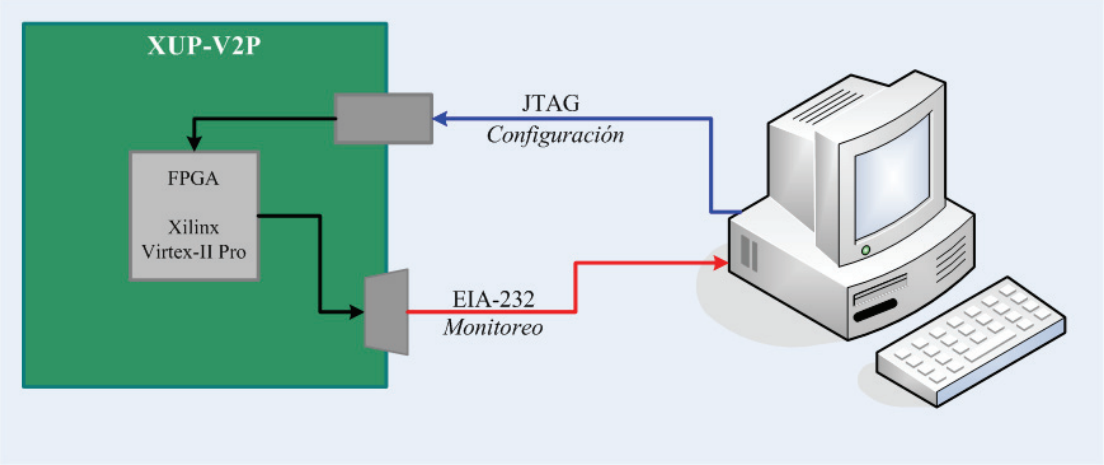

Figura 3.1 Representación del entrono de desarrollo.

El entorno de desarrollo (diseño y prueba) de la implementación del circuito consiste en dos componentes de hardware principales:

- Una tarjeta electrónica, la Plataforma de Desarrollo Xilinx University Program Virtex-II Pro (XUP-V2P); y
- Un ordenador, con el software de diseño para configurar el FPGA, y un software de monitoreo del puerto serial.

3.2.1 Planteamiento del Circuito Final

La tarjeta XUP-V2P va a contener el diseño del cifrador en el FPGA, mismo que es configurado mediante el computador haciendo uso del Xilinx ISE, a través de una interfaz JTAG. El cifrador tiene su salida conectada a un transmisor UART (embebido en el diseño) que enviará la secuencia de caracteres, a través de una interfaz EIA-232, al computador, donde se visualizarán los datos usando el software de monitoreo del puerto serial; así se podrá verificar el correcto funcionamiento del circuito.

DEL FUNCIONAMIENTO

Tal como se puede observar en la figura 3.2, el diseño consiste en una etapa de cifrado, una de descifrado y una etapa de transmisión de datos. La función de cifrado E_k recibe como entrada una cadena de 8 bytes (64 bits) como texto plano, y genera el texto cifrado. El texto cifrado es la entrada de la

función de descifrado D_k la cual devuelve el texto plano descifrado.

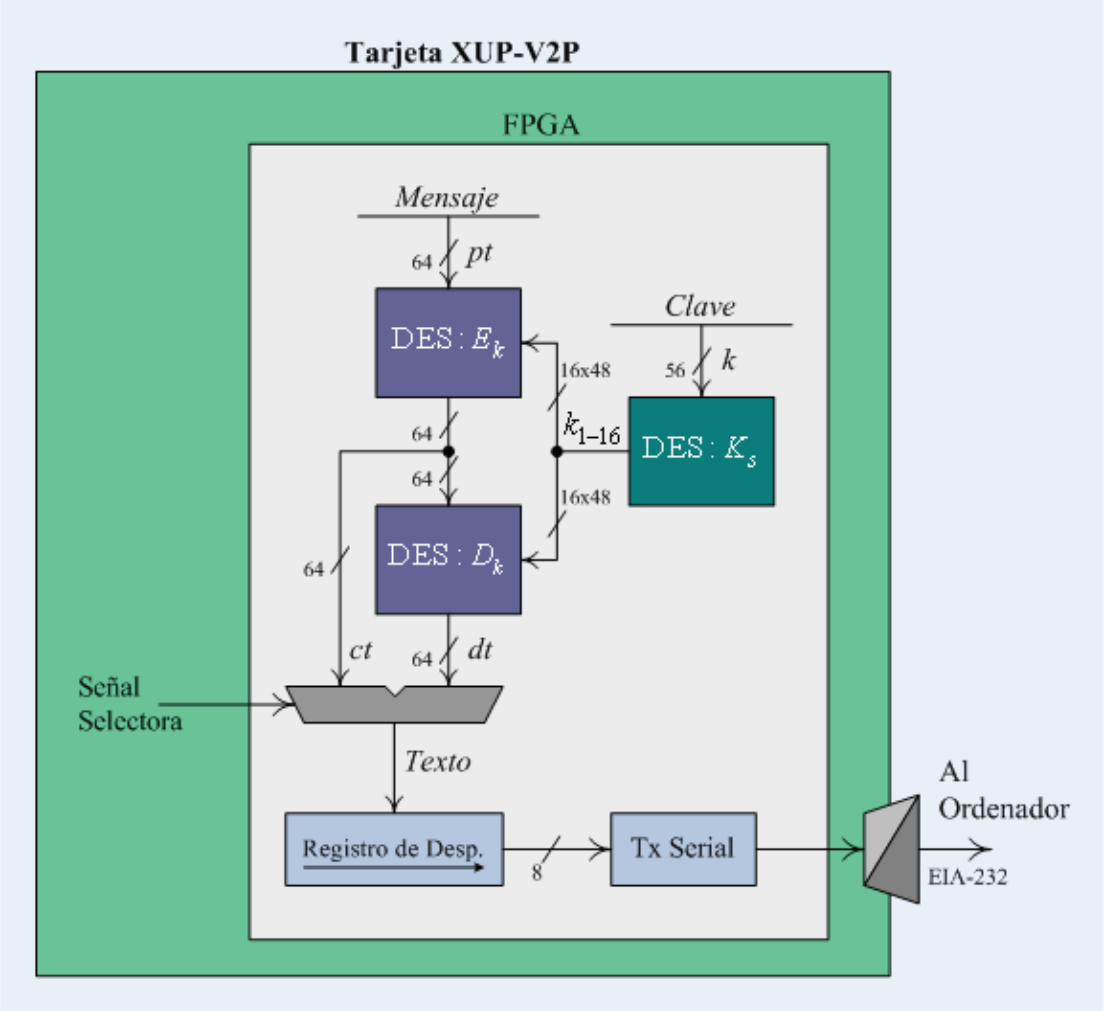

Figura 3.2 Diagrama de bloques simplificado del diseño digital.

Una etapa selectora permite seleccionar qué texto será cargado, el texto cifrado o el texto descifrado, en el registro de trasmisión el cual enviará, byte por byte, el texto al trasmisor serial UART, que enviará el mensaje al computador. (Los detalles de la implementación se explican en la sección **3.2.4.4**)

3.2.2 Descripción de la Tecnología utilizada

Esta implementación pertenece a la categoría de las *implementaciones en hardware reconfigurable*, y más específicamente al campo de las *implementaciones de algoritmos criptográficos en hardware reconfigurable*. En esta sección revisaremos de una manera sucinta las características de la tecnología y herramienta de diseño utilizadas para implementar el circuito propuesto.

3.2.2.1 Características principales de los FPGA

Un Arreglo de Puertas Programables por Campo (FPGA) es un circuito integrado que pertenece a una familia de los dispositivos programables denominada Dispositivos Lógicos Programables (PLDs). Un FPGA puede ser entendido como matriz de bloques de construcción conocidos como *Bloques Lógicos Configurables* (CLBs), interconectados por un intrincado arreglo de matrices de conmutación, ambos programables por el usuario. El diseño específico de los bloques CLB varía de fabricante a fabricante e inclusive de dispositivo a dispositivo.

La arquitectura del CLB define el nivel de *tamización* o *resolución* de la lógica reconfigurable y el tamaño de la unidad funcional más pequeña que puede ser registrada por las herramientas de programación. Las

interconexiones de los FPGA tienen un rol preponderante en el desempeño de un dispositivo FPGA debido a la necesidad de vías rápidas entre los diferentes bloques lógicos, los cuales están organizados en filas y columnas.

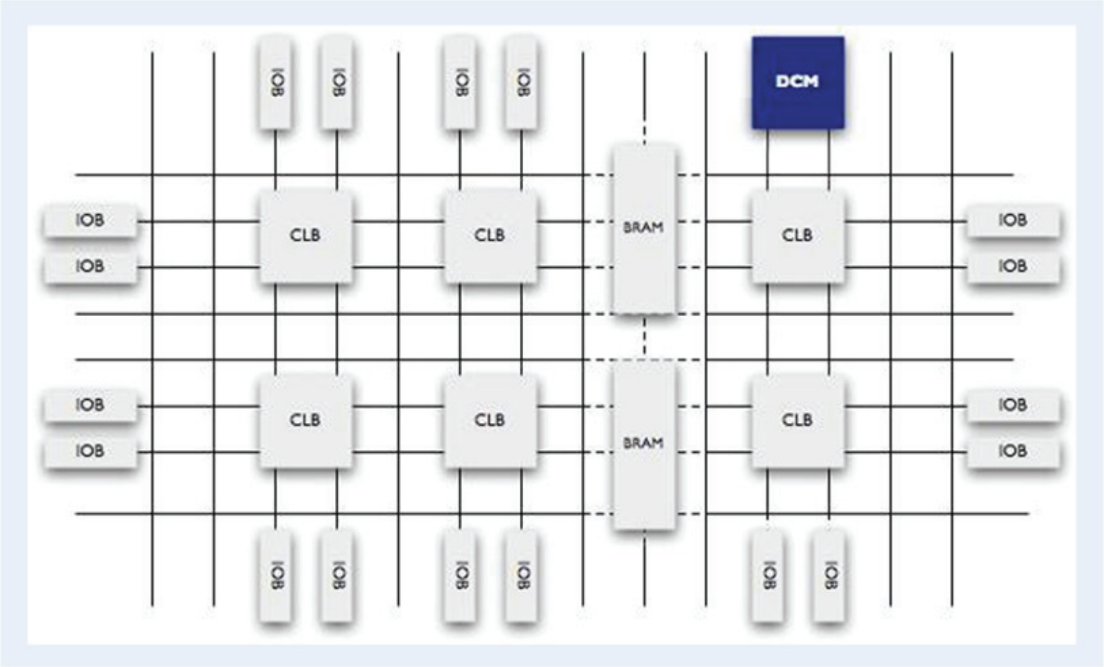

Figura 3.2 Estructura de bloques de un FPGA de Xilinx. Adaptado de [24].

Hay relativamente pocos fabricantes comerciales de FPGAs, y generalmente cada uno de ellos ha desarrollado una o más familias de dispositivos. Los dos más populares son Xilinx (líder del mercado) y Altera; que comparten el 70% del mercado. Siendo nuestro diseño basado en un FPGA de Xilinx nos referiremos a las características de los FPGA de este fabricante.

FPGAS DE XILINX

La arquitectura de las familias Virtex-5, Virtex-4, Virtex-II Pro y Spartan 3E de Xilinx consiste en cinco elementos funcionales fundamentales:

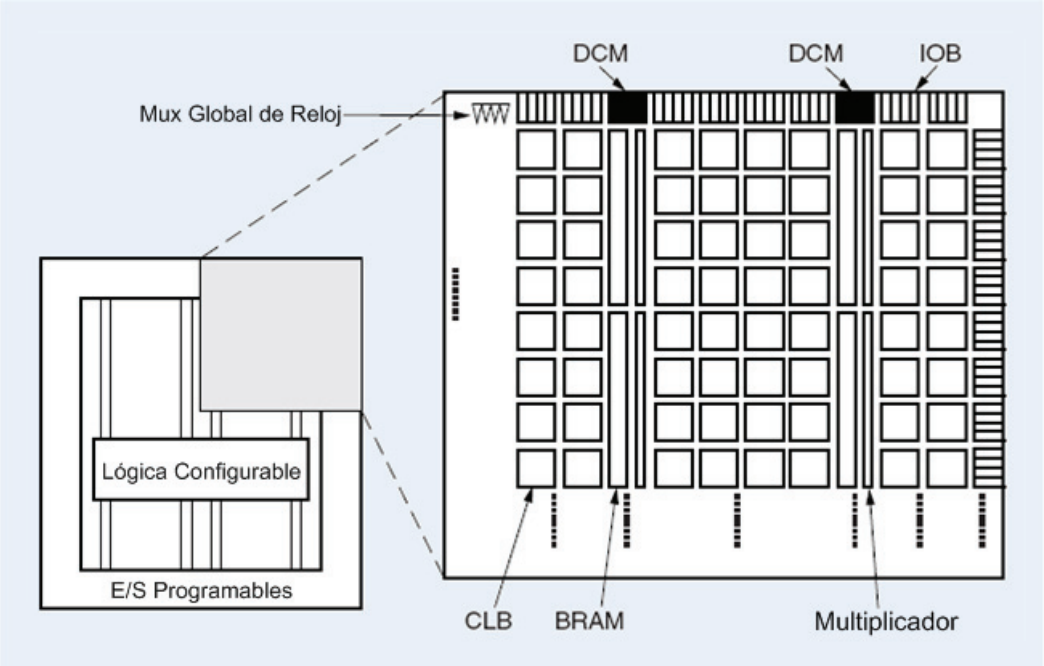

Figura 3.3 Vista general de la Arquitectura de un Virtex-II. Adaptado de Fig. 1 de [25].

- Bloques Lógicos Configurables (CLBs) y la arquitectura Slice†;
- Bloques de Entrada/Salida (IOBs);
- Bloques RAM (BRAMs);
- Multiplicadores Dedicados; y
- Administradores Digitales de Reloj (DCMs).

† Slice es un término introducido por Xilinx. Especifica una unidad básica de procesamiento en un FPGA de Xilinx.

Estos elementos están físicamente organizados en un arreglo regular tal como se muestra en la figura 3.3. A continuación vamos a explicar brevemente cada uno de estos cinco elementos:

BLOQUES LÓGICOS CONFIGURABLES (CLBS)

Los Bloques Lógicos Configurables (CLBs) son el recurso físico más importante y más abundante de un FPGA. Son utilizados generalmente tanto para diseños de lógica combinatorial como sincrónica. Cada CLB está compuesto por cuatro *slice*, los cuales están interconectados como muestra la figura 3.4. Los slice están agrupados en pares, y cada par está organizado en una columna con una cadena de acarreo independiente.

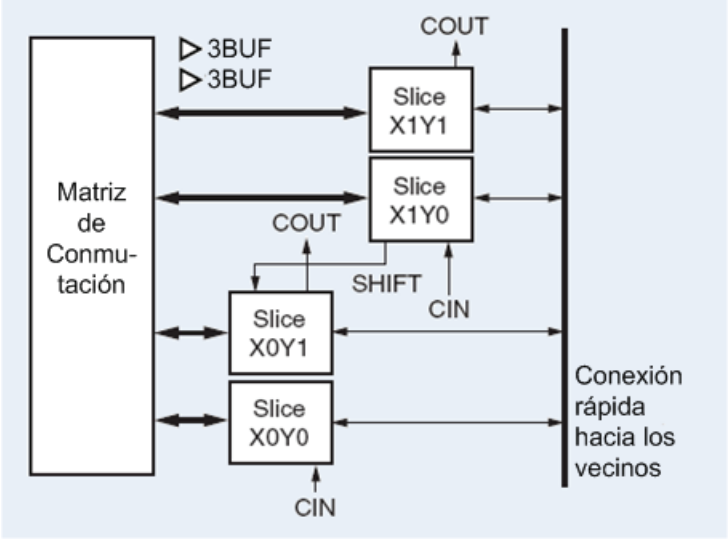

Figura 3.4 Elemento CLB de un Virtex-II Pro. Adaptado de Fig. 32 de [27].

Los 4 slice tiene los siguientes elementos comunes: dos Tablas de Verdad (LUTs, *Lookup Tables*), dos biestables tipo D (FF-D), multiplexores (MUXs), circuitos lógicos para el manejo del acarreo y puertas lógicas aritméticas.

Ambos pares utilizan estos elementos para proveer funciones lógicas, aritmética y ROM.

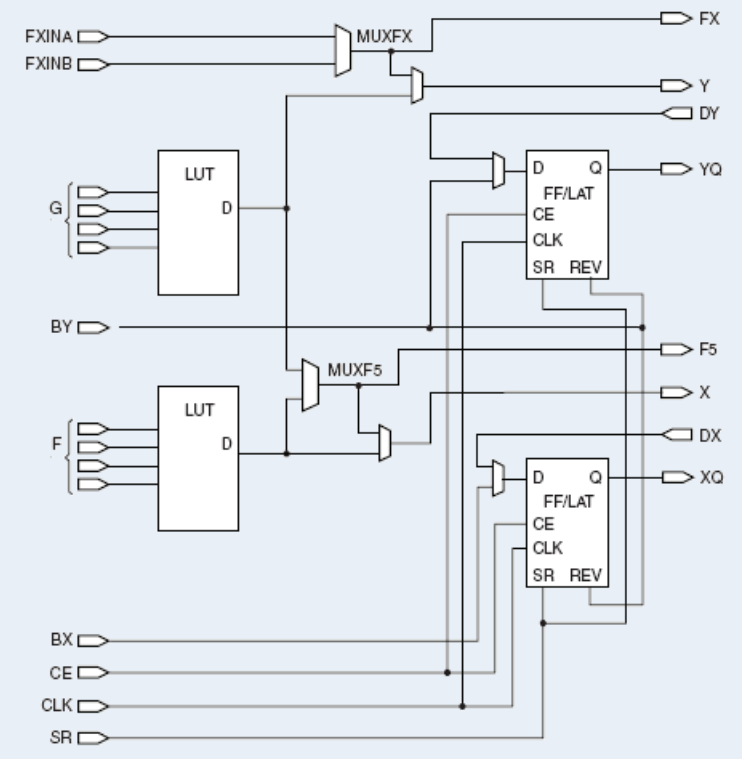

Figura 3.5 Slice de un Virtex-II Pro. Adaptado de Fig. 2-1 de [27].

Las funciones generadoras de 4 entradas, F y G, mostradas en la figura 3.5, pueden ser programadas cada una como una LUT $4-1$ (modo de lógica), una RAM 16x1 (modo de memoria), o un registro de desplazamiento.

BLOQUES DE ENTRADA/SALIDA (IOBS)

Los Bloques de Entrada/Salida (IOBs) proveen una interfaz bidireccional programable entre el mundo exterior y la estructura lógica interna del dispositivo FPGA.

Existen tres tipos de enrutamiento para un IOB: señal de salida, señal de entrada y alta impedancia. Cada una de las opciones de señal tienen su propio par de elementos de almacenamiento que pueden comportarse como registros o retenedores.

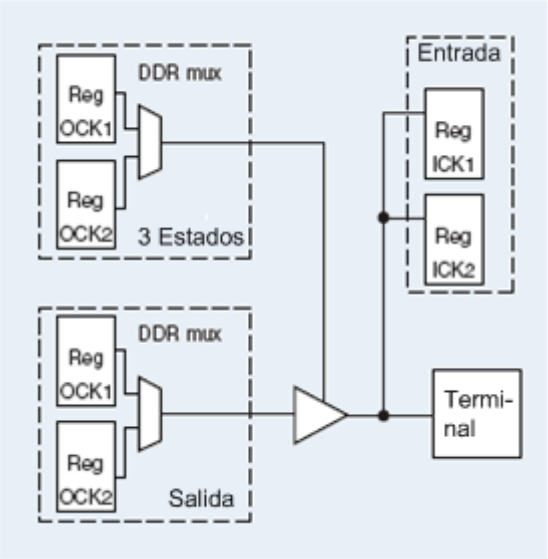

Figura 3.6 Bloque IOB de un Virtex-II. Adaptado de Fig. 19 de [26].

MULTIPLICADOR DE 18X18 BITS

Los FPGA de Xilinx tienen algunos bloques multiplicadores dedicados. Estos multiplicadores aceptan operandos en complemento a 2 para computar su producto también en complemento a 2. Tales multiplicadores han sido optimizados para desempeñarse a alta velocidad a la vez que su consumo de potencia se mantiene bajo en comparación a multiplicadores implementados directamente usando los recursos de los CLBs.

BLOQUES RAM (BRAMS)

Los dispositivos Virtex incluyen 18K-bit de memoria RAM interna, llamada BRAM. Las BRAMs están previstas para almacenar grandes cantidades de datos; son bloques polimórficos en el sentido de que su número de entradas y capacidad de almacenamiento puede ser configurada.

Configuración	*Capacidad*	*Bits de Salida*	*Bits de Paritdad*
16K x 1	16Kb	1	0
8K x 2	8Kb	2	0
4K x 4	4Kb	4	0
2K x 9	2Kb	8	1
1K x 18	1Kb	16	2
512 x 36	512	32	4

Tabla 3.I Posibles configuraciones de una BRAM

ADMINISTRADORES DIGITALES DE RELOJ (DCMS)

Los Administradores Digitales de Reloj (DCMs) proveen un control flexible sobre la frecuencia, el corrimiento de fase y el sesgo del reloj. Las tres funciones más importantes de los DCMs son: Mitigar el sesgo debido a la diferencia de tiempo de arribo entre las señales de reloj, generar un amplio rango de frecuencias de reloj derivadas de la señal maestra de reloj, y generar desplazamientos de fase de las señales de reloj.

3.2.2.2 Características principales del lenguaje VHDL

El ciclo de diseño para programar FPGAs inicia con una descripción del

comportamiento del diseño, usando ya sea lenguajes de descripción de hardware (HDLs) o un diseño esquemático. El Lenguaje de Descripción de Hardware de Circuitos Integrados de Alta Velocidad (VHDL) fue creado por el Departamento de Defensa de los EUA (DoD) a comienzos de los 80. En diciembre de 1987, VHDL fue adoptado como un estándar IEEE. VHDL es un *lenguaje funcional* que tomó mucho de su estructura del lenguaje de programación ADA junto con un conjunto de construcciones para soportar el paralelismo inherente de los diseños de hardware.

El lenguaje VHDL fue creado con el propósito de especificar y documentar circuitos y sistemas digitales utilizando un lenguaje formal. En la práctica se ha convertido, en un gran número de entornos de CAD, en el HDL de referencia para realizar modelos sintetizables automáticamente. Las principales características del lenguaje VHDL se explican en los siguientes puntos:

- **Descripción textual normalizada:** El lenguaje VHDL es un lenguaje de descripción que especifica los circuitos electrónicos en un formato adecuado para ser interpretado tanto por máquinas como por personas. Se trata además de un lenguaje formal, es decir, no resulta ambiguo a la hora de expresar el comportamiento o representar la estructura de un circuito. Está, como ya se ha dicho, normalizado, o sea, existe un único modelo para el lenguaje, cuya utilización está abierta a cualquier grupo que quiera desarrollar herramientas basadas en dicho modelo,

garantizando su compatibilidad con cualquier otra herramienta que respete las indicaciones especificadas en la norma oficial. Es, por último, un lenguaje ejecutable, lo que permite que la descripción textual del hardware se materialice en una representación del mismo utilizable por herramientas auxiliares tales como simuladores y sintetizadores lógicos, compiladores de silicio, simuladores de tiempo, de cobertura de fallos, herramientas de diseño físico, etc.

- **Amplio rango de capacidad descriptiva:** El lenguaje VHDL posibilita la descripción del hardware con distintos niveles de abstracción, pudiendo adaptarse a distintos propósitos y utilizarse en las sucesivas fases que se dan en el desarrollo de los diseños. Además es un lenguaje adaptable a distintas metodologías de diseño y es independiente de la tecnología, lo que permite, en el primer caso, cubrir el tipo de necesidades de los distintos géneros de instituciones, compañías y organizaciones relacionadas con el mundo de la electrónica digital; y, en el segundo, facilita la actualización y adaptación de los diseños a los avances de la tecnología en cada momento.

- **Otras ventajas:** Además de las ventajas ya reseñadas también es destacable la capacidad del lenguaje para el manejo de proyectos de grandes dimensiones, las garantías que brinda su uso cuando, durante el ciclo de mantenimiento del proyecto, si hay que sustituir componentes o realizar modificaciones en los circuitos, y el hecho de que, para muchas organizaciones contratantes, sea parte indispensable de la documentación

de los sistemas.

Además, VHDL provee un basto conjunto de construcciones. Con VHDL se pueden describir sistemas electrónicos discretos de complejidad variada (sistemas, tarjetas, chips. módulos) con un variado nivel de abstracción. Las construcciones del lenguaje VHDL están divididas en tres categorías por su nivel de abstracción:

- **de comportamiento.-** los aspectos funcionales o algorítmicos de un diseño se lo expresa en un proceso secuencial;
- **flujo de datos.-** los datos son vistos como un flujo a través del diseño, desde la entrada hasta la salida. Una operación es definida en términos de una colección de transformaciones de los datos, expresada como sentencias concurrentes; y
- **estructural.-** es un modelo donde los componentes del diseño están interconectados, y es expresado como la instanciación de componentes. Es la más cercana al hardware.

DISEÑO EN FPGAS

En general, el diseño en FPGAs consiste en seis pasos básicos:

1. **El Ingreso del Diseño.** Un diseñador puede describir un diseño para FPGAs en un lenguaje abstracto de alto nivel como VHDL o Verilog. Estos lenguajes son ideales para construir máquinas de estado, lógica combinatorial, diseños largos y complejos. La mayoría de las

herramientas de software tiene sofisticados compiladores que pueden traducir eficientemente las especificaciones HDL a los recursos del FPGA.

2. **Verificación Funcional y Simulación.** En este paso es validado el correcto funcionamiento lógico de un diseño en FPGA. Una vez que el diseño ha sido especificado es necesario verificar si tal descripción cumple con las especificaciones de diseño.
3. **Síntesis.** La síntesis convierte las especificaciones de un diseño ingresado en puertas/bloques de un dispositivo FPGA. Un listado de redes de puertas lógicas básicas es preparado a partir del diseño HDL ingresado, el cuál más adelante es optimizado a nivel de puertas. El siguiente paso es el mapeo de dicho listado en los recursos reales de un FPGA. Este es un paso importante basado en el diseño ingresado. Cuando se escribe código HDL un diseñador FPGA debería siempre tener en cuenta la estructura básica del dispositivo objetivo.
4. **Ubicación y enrutamiento (PAR).** En este proceso se seleccionan la ubicación física óptima de los bloques elementales del diseño y la mínima distancia de interconexión entre ellos. Las herramientas PAR normalmente usan las especificaciones del proveedor del dispositivo. Usualmente proveen la opción de ubicación manual y también opciones automáticas de optimización de las rutas críticas sea por velocidad o por área.
5. **Análisis del Circuito.** El análisis del circuito evalúa diferentes métricas de desempeño del diseño. La verificación en tiempo esta hecha de tal

manera que difiere de la simulación funcional, ya que provee una corrección lógica tomando en cuenta todos los retrasos del circuito, que ocurren en el dispositivo real. De forma similar una evaluación analítica de la potencia provee una estimación del consumo de potencia del diseño.

6. **Programación del dispositivo FPGA.** Programar un FPGA implica descargar códigos a manera de flujos de bits desde el último paso del diseño hacia el dispositivo FPGA objetivo. Herramientas universales de programación trabajan con FPGAs de diferentes proveedores. Sin embargo existen herramientas de programación dedicadas limitadas únicamente a una sola familia de dispositivos FPGA.

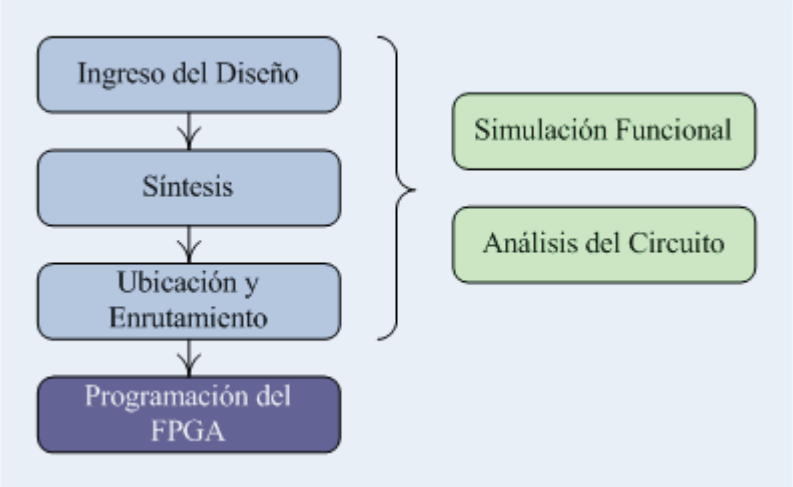

Figura 3.7 Procedimiento general de diseño en FPGAs.

Estos pasos no necesariamente deben ser ejecutados en un orden determinado, pero deben repetirse para mejorar el desempeño del diseño.

3.2.3 Equipo y Herramientas utilizadas

Se detalla a continuación los materiales, y recursos con los que contamos para la implementación, y su papel en el proceso de desarrollo del cifrador.

3.2.3.1 Hardware de Desarrollo

El Sistema de Desarrollo XUP-V2P provee una plataforma de hardware que consiste en un FPGA de alto desempeño Virtex-II Pro, rodeado por una completa colección de componentes periféricos (ver figura 3.8) que pueden ser usados para crear un sistema complejo y demostrar la capacidad del FPGA Virtex-II Pro.

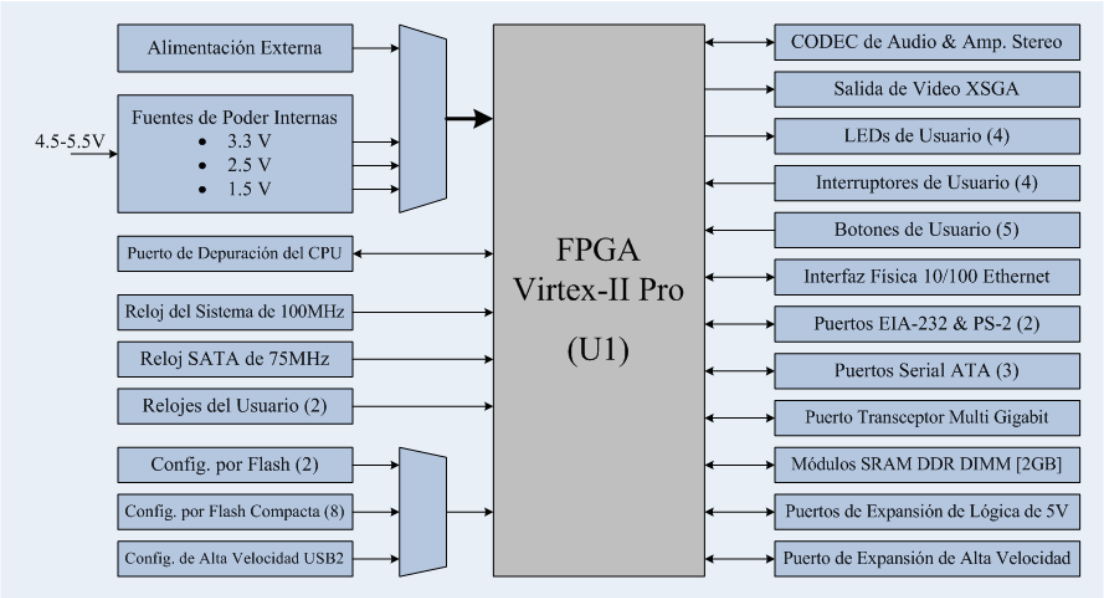

Figura 3.8 Diagrama de bloques del sistema XUP-V2P. Adaptado de Fig. 1-1 de [31].

El componente central, U1, es un FPGA Virtex-II Pro XC2VP30 que tiene un empaquetamiento FF896 BGA, de alta densidad de terminales. Las características del XC2VP30 se detallan en la tabla 3.II.

Componente	*Capacidad*
Cantidad de Slice	13969
Tamaño del Arreglo	80 x 46
RAM Distribuida	428 Kb
Multiplicadores	136
Bloques RAM	2448 Kb
DCMs	8
CPUs PowerPC	2
Tranceptores Multi Gb	8

Tabla 3.II Características de un dispositivo XC2VP30.

La gran cantidad de recursos que tiene el circuito integrado XC2VP30 es uno de los motivos por los que escogimos esta plataforma de desarrollo; tal como veremos más adelante la implementación del algoritmo DES demanda muchos recursos (puertas, memorias, decodificadores, señales, etc.) si se optimiza su velocidad. El requerimiento aumenta si tomamos en cuenta el hecho de que el DES es implementado dos veces en el integrado.

Las interfaces que han sido aprovechadas para el proyecto son: el puerto serial EIA-232, los LEDs, botones e interruptores de usuario, y los puertos de expansión.

EL PUERTO SERIAL EIA-232

El sistema XUP-V2P provee un puerto serial con interfaz EIA-232. Este

puerto tiene una configuración de señalización tipo DCE (Equipo de Comunicación de Datos) con control de flujo por hardware. La interfaz física está dado por un conector DB9 y la eléctrica por un transceptor EIA-232. Las señales del puerto serial que maneja el FPGA son: TXD (entrada), DSR (salida), CTS (salida), RXD (salida), RTS (entrada).

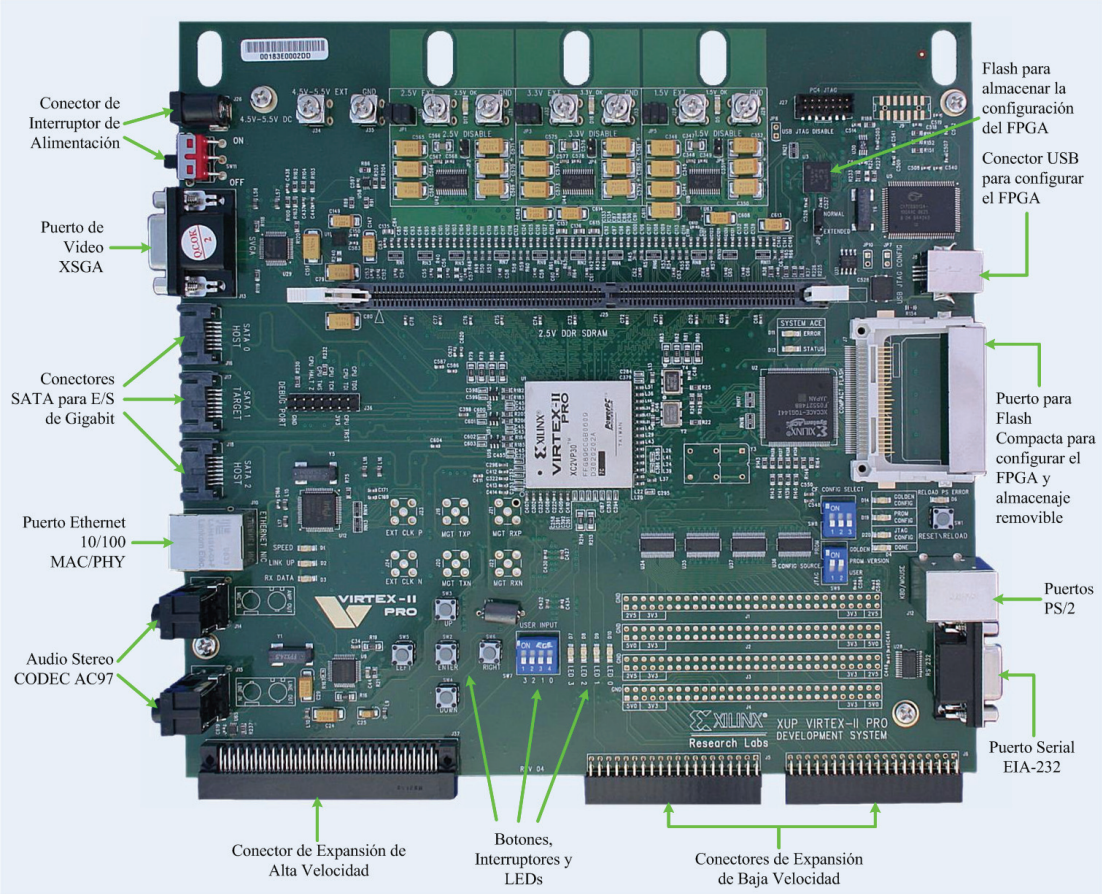

Figura 3.9 Vista superior de la tarjeta XUP-V2P. Cortesía de Digilent Inc.

LEDs, Interruptores y Botones de Usuario

Un total de 4 LEDs están disponibles para usos definidos por el usuario. Cuando el FPGA emite un nivel de voltaje bajo (0), los LEDs

correspondientes se encienden. Un interruptor modular de cuatro posiciones de empaque DIP (paquete en línea doble) y cinco botones de pulso están disponibles para entradas de usuario. Si el interruptor DIP es está *arriba*, *cerrado*, o *encendido*, o si el botón de pulso es presionado, un nivel de voltaje bajo (0) es recibido por el FPGA, caso contrario recibirá un nivel alto (1).

PUERTOS DE EXPANSIÓN

Un total de 80 terminales de entrada/salida de baja velocidad del Virtex-II Pro están disponibles como terminales de expansión. La mayoría de estas señales se encuentran compartidas entre los cabezales J1-4 y los conectores frontales J5-6. Todas estas señales están equipadas con dispositivos protección contra sobrevoltaje (J34-41) para proteger el FPGA. Algunas señales de alimentación están disponibles en los conectores de expansión, 2.5V, 3.3V, y 5.0V, dependiendo de tipo de conector. Para la distribución de señales en los terminales de cada conector, revisar [31].

3.2.3.2 Software de Desarrollo

Para el desarrollo del diseño digital en el FPGA se utilizó la versión 7 del Entorno de Software Integrado (ISE) de Xilinx. El ISE es una herramienta CAD (diseño asistido por computadora) de asistencia en el diseño en FPGA de Xilinx. El ISE incluye Diseño por Esquemáticos, Simulación,

Herramientas de Implementación y Programación, los cuales pueden ser iniciados desde el navegador, el Project Navigator. El Navegador muestra todos los archivos fuente, todas las herramientas CAD pueden ser usadas con los archivos fuentes y con cualquier salida, mensaje de estado o archivo que resulte de correr una herramienta dada.

El ISE trabaja con la herramienta de simulación ModelSim XE (Edición para Xilinx) de Mentor Graphics. Para la compilación VHDL y la síntesis es utilizada la herramienta Xilinx Synthesis Tool (XST). Las señales son asignadas a los terminales usando la herramienta Pinout and Area Constrains Editor (PACE). Para la traducción, mapeo, ubicación y enrutamiento (PAR) se tienen las herramientas de implementación de Xilinx.

Además, el ISE cuenta con dos herramientas especiales, el Xilinx CoreGenerator y el ChipScope Pro. El CoreGenerator es una herramienta de diseño que genera *núcleos*[†] parametrizables optimizados para FPGAs Xilinx. El software ChipScope Pro es un conjunto de herramientas (ChipScope Core Generator, ChipScope Core Inserter y ChipScope Analyzer) que integran componentes analizadores de hardware (en el computador) con el diseño dentro del dispositivo FPGA, proveyendo al diseñador un analizador lógico completo.

[†] Un núcleo, o *core*, es una pequeña porción de código compilado para una arquitectura específica, el cuál puede ser utilizado en un diseño más grande.

3.2.4 Estrategia de Implementación

Para la implementación exitosa del cifrador de datos se ha seguido una estrategia progresiva: Se inició probando los módulos de diseño fundamentales y se fueron añadiendo poco a poco los módulos de cada función, hasta concluir con el circuito propuesto.

Los módulos de diseño principales son:

- La función de cifrado DES, en modo EBC;
- El Sistema de Reloj, que genera las señales de reloj a las frecuencias necesarias para cada función;
- El Transmisor Serial Asincrónico, que transmite un byte en serie, en modo 8-N-1: 8 bits de datos sin bit de paridad, sin control de flujo por hardware, usando 1 bit de parada;
- El Registro de Desplazamiento de 8x8, que es usado como Registro de Alimentación de Datos (secuenciador de bytes) y como Registro de Estado del Cifrador DES EBC; y
- El Controlador, que coordina todo el proceso.

De todos los componentes, únicamente la función de cifrado DES es un módulo combinatorial, mientras que el resto son máquinas secuenciales algorítmicas sincrónicas. La implementación de cada módulo de diseño se explica en la sección **3.2.5.3**, y el código VHDL se muestra en el apéndice A. En seguida presentaremos la progresión de las pruebas.

3.2.4.1 Prueba del Sistema de Reloj y del Transmisor Serial Asincrónico

El Circuito de Prueba 1 sirve para validar el correcto funcionamiento de dos etapas, el Sistema de Reloj y el Transmisor Serial Asincrónico: El Sistema de Reloj debe generar apropiadamente la señal de reloj para la transmisión serial CLKtx.H (de 115200 Hz para nuestro caso), recibiendo como entrada la señal de reloj (diferencial) de 100 MHz que provee la tarjeta XUP-V2P; mientras que el Transmisor Serial Asíncrono debe transmitir hacia el computador los 8 bits de datos (en una trama de 10 bits: 1 bit de inicio, 8 bits de datos y 1 bit de parada) cuando reciba la señal de envío.

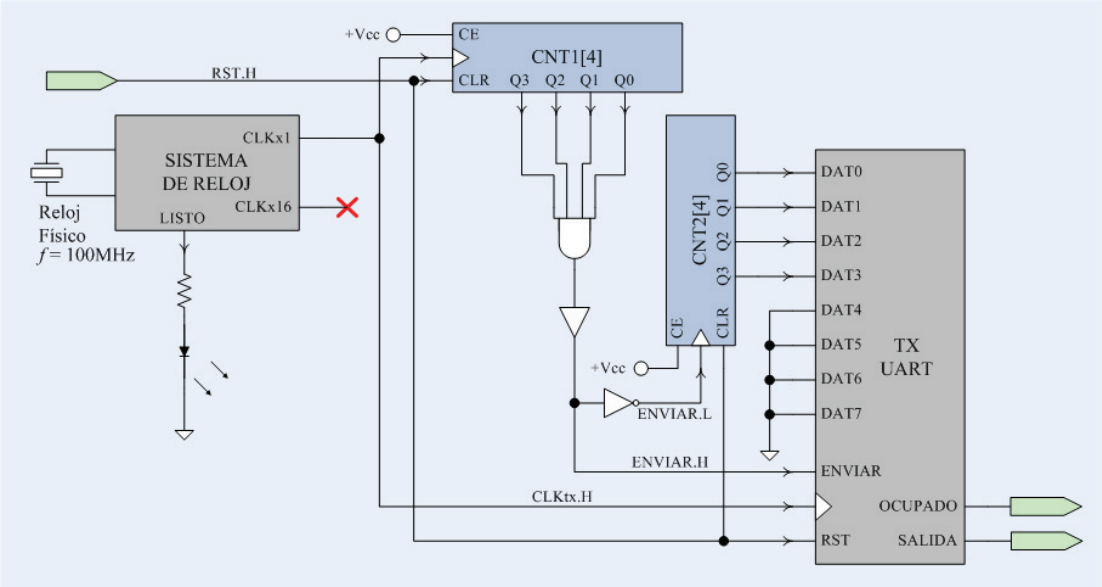

Figura 3.10 Diagrama Esquemático del Circuito de Prueba 1.

La prueba consiste en generar y transmitir al computador caracteres o cifras hexadecimales de dos dígitos, desde el 00_{16} hasta el $0F_{16}$. Los caracteres

se reciben en el computador y se los lee utilizando un monitor de puerto serial para verificar que se recibe la secuencia correcta. El contador CNT1 se usa para dividir la frecuencia del reloj de transmisión y generar una señal de envío con un tiempo intercaracter (TIC) suficientemente largo para que no interfirieran un caracter con el siguiente. El contador CNT2 es el que genera el nibble menos significativo de los caracteres. Mientras está siendo transmitida una trama, el Transmisor emite una señal OCUPADO.H. La descripción RTL (Lenguaje de Transferencia de Registros) es la siguiente:

$$
\begin{Vmatrix}
\mathrm{RST}: \mathrm{CNT1} \leftarrow 0, \mathrm{CNT2} \leftarrow 0, \\
\mathrm{CLK} \uparrow: \mathrm{CNT1} \leftarrow \mathrm{CNT1} + 1, \\
\qquad \mathrm{CNT1} = 15: \mathrm{ENVIAR} \leftarrow 1, \\
\qquad\qquad \mathrm{TxUART\_BUF} \leftarrow \mathrm{CNT2}, \\
\mathrm{ENVIAR} \downarrow: \mathrm{CNT2} \leftarrow \mathrm{CNT2} + 1
\end{Vmatrix}
$$

El circuito genera los caracteres de manera continua, no tiene control de inicio ni de parada. El contador CNT1 aumenta con cada ciclo de reloj. La señal RST.H (proveniente de un Botón de Usuario) se utiliza para reiniciar el proceso. El Sistema de Reloj devuelve una señal LISTO.H cuando sus salidas están estables; ésta se la envía a un LED de Usuario para visualizar el estado del Reloj. Si los caracteres son recibidos íntegros por el computador, se evidencia que tanto la frecuencia de transmisión (que depende del sistema de reloj) como la conformación de las tramas seriales (que depende del transmisor) están correctas.

3.2.4.2 Prueba del Registro de Desplazamiento de 8x8 y Transmisión Serial Asincrónica de 8 bytes

En el Circuito de Prueba 2, se evalúa el envío de caracteres usando un Registro de Desplazamiento de 8x8. El registro recibe 64 bits (8 bytes) como datos de entrada paralela y los envía de 8 bits en 8 bits (de byte en byte) por su salida serial; cada caracter pasa al Transmisor Serial Asincrónico y es enviado al computador.

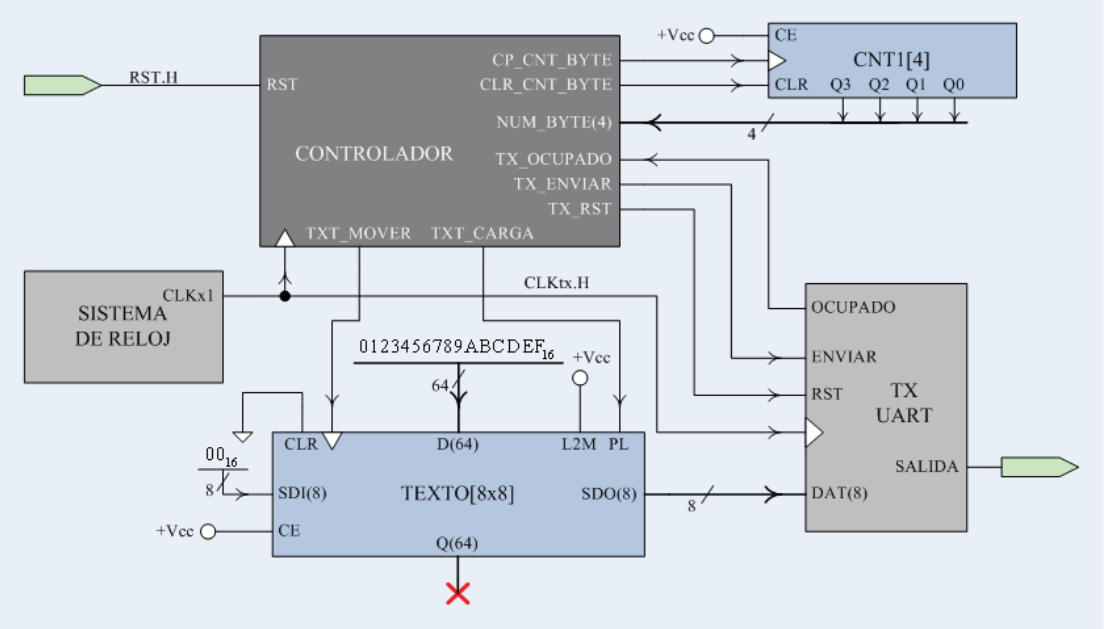

Figura 3.11 Partición funcional del Circuito de Prueba 2.

El Controlador coordina la carga del mensaje (insertado en el código) en el registro TEXTO y el envío de los ocho caracteres; usa CNT1 como contador de los bytes enviados. El desplazamiento del TEXTO es hecho en sentido

LSB (bit menos significativo) a MSB (bit más significativo), y rellenado con ceros (bytes 00_{16}). El proceso se detiene al ser enviados los ocho caracteres; la señal RST.H provoca que se vuelva a enviar la cadena. La descripción RTL es:

$$
\begin{Vmatrix}
\text{RST:} \rightarrow (T_0) \\
T_0 : \text{TEXTO} \leftarrow 0123456789\text{ABCDEF}_{16}, \\
\quad \text{CNT1} \leftarrow 0, \\
T_1 : \left(\text{OCUPADO}, \overline{\text{OCUPADO}}\right) \Big/ (T_1, T_2), \\
T_2 : \text{TxUART\_BUF} \leftarrow \text{TEXTO(63-56)}, \\
\quad \text{CNT1} \leftarrow \text{CNT1} + 1, \\
T_3 : \textbf{shl}\ \text{TEXTO}, \\
\quad \left(\text{CNT1} = 8, \overline{\text{CNT1} = 8}\right) \Big/ (T_3, T_1), \\
T_4 : \rightarrow (T_4)
\end{Vmatrix}
$$

En esta prueba, en el computador se debe recibir la cadena cargada en TEXTO. Si los ocho caracteres son recibidos íntegros por el computador, en ese preciso orden, se corrobora que el Registro de Desplazamiento funciona bien y se revalida el buen funcionamiento del Transmisor y el Reloj.

3.2.4.3 Prueba del Cifrado de 8 bytes y su Transmisión

El Circuito de Prueba 3 es idéntico al Circuito de Prueba 2, salvo que el

mensaje pasa primero por la función de cifrado E_k del DES antes de ser cargado en TEXTO. Los módulos E_k y K_s son combinatoriales, con retardos en el orden de las decenas de nanosegundos.

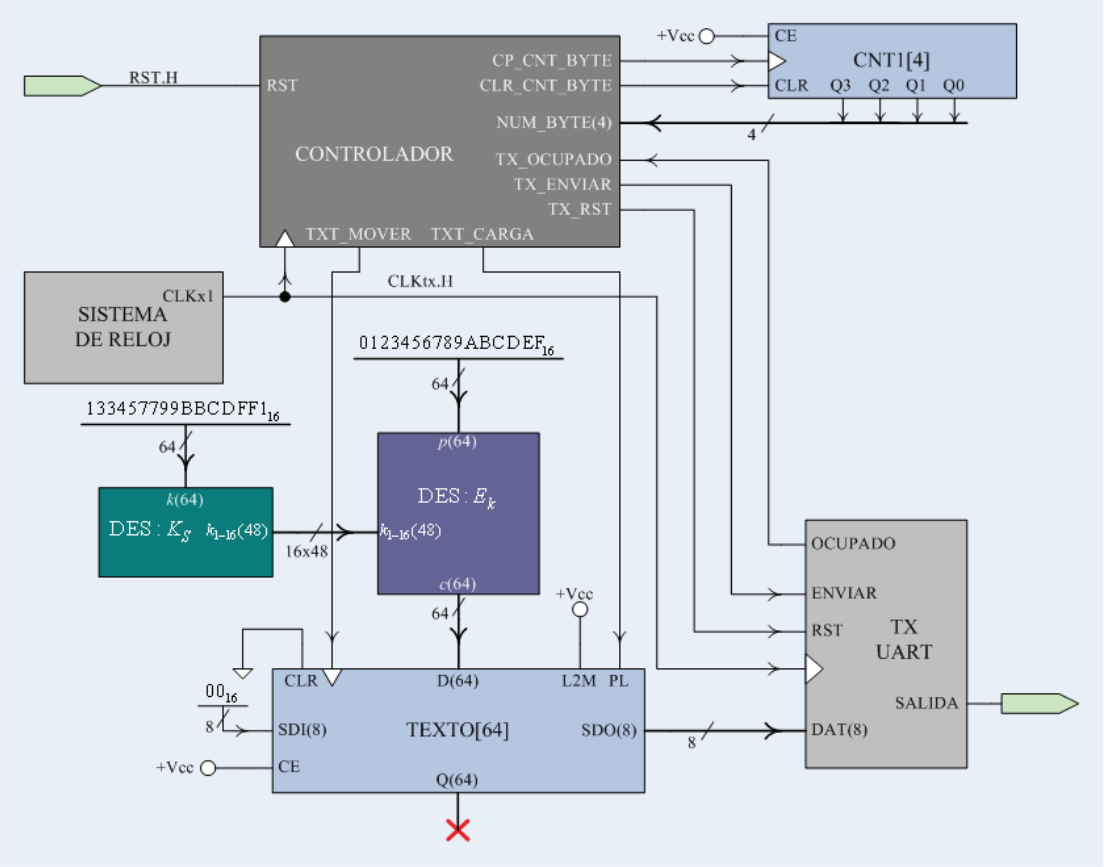

Figura 3.12 Partición funcional del Circuito de Prueba 3.

El Controlador coordina la carga del criptograma en el registro TEXTO y el envío de los ocho caracteres; con CNT1 se cuentan los bytes enviados. El desplazamiento del TEXTO es hecho en sentido LSB (bit menos significativo) a MSB (bit más significativo), y rellenado con ceros (bytes 00_{16}). El proceso se detiene al ser enviados los ocho caracteres; la señal

RST.H provoca que se vuelva a enviar la cadena. La descripción RTL es:

$$
\begin{aligned}
&\text{RST: } \rightarrow (T_0)\\
&T_0 : \text{CNT1} \leftarrow 0,\\
&T_1 : \text{TEXTO} \leftarrow E_k(pt),\\
&T_2 : \left(\text{OCUPADO}, \overline{\text{OCUPADO}}\right) / (T_2, T_3),\\
&T_3 : \text{TxUART\_BUF} \leftarrow \text{TEXTO(63-56)},\\
&\quad \text{CNT1} \leftarrow \text{CNT1} + 1,\\
&T_4 : \textbf{shl}\ \text{TEXTO},\\
&\quad \left(\text{CNT1} = 8, \overline{\text{CNT1} = 8}\right) / (T_5, T_2),\\
&T_5 : \rightarrow (T_5)
\end{aligned}
$$

Para comprobar el correcto funcionamiento de nuestra implementación del DES acudimos al documento [21], donde se implementa la misma versión del algoritmo de cifrado y se provee una triada (p,k,c) de prueba, la $\left(0123456789\text{ABCDEF}_{16}, 133457799\text{BBCDFF1}_{16}, 85\text{E}813540\text{F0AB}405_{16}\right)$; adicionalmente se desarrolló un programa en C++, de tipo consola, que ejecuta el DES FIPS 46-3 (y que cumple con la triada de prueba) para poder probar con diferentes textos planos y claves. El código C++ de este programa se muestra en el apéndice C. Si la cadena recibida en el computador coincide con el criptograma de la triada de prueba, comprobamos que los componentes del DES, E_k y K_s, funcionan correctamente.

3.2.4.4 Implementación del Circuito Final

El Circuito Final se construye tomando Circuito de Prueba 3 y añadiendo una etapa de descifrado a continuación del la etapa de cifrado, y una etapa de selección de qué bloque (cifrado o descifrado) se envía al computador.

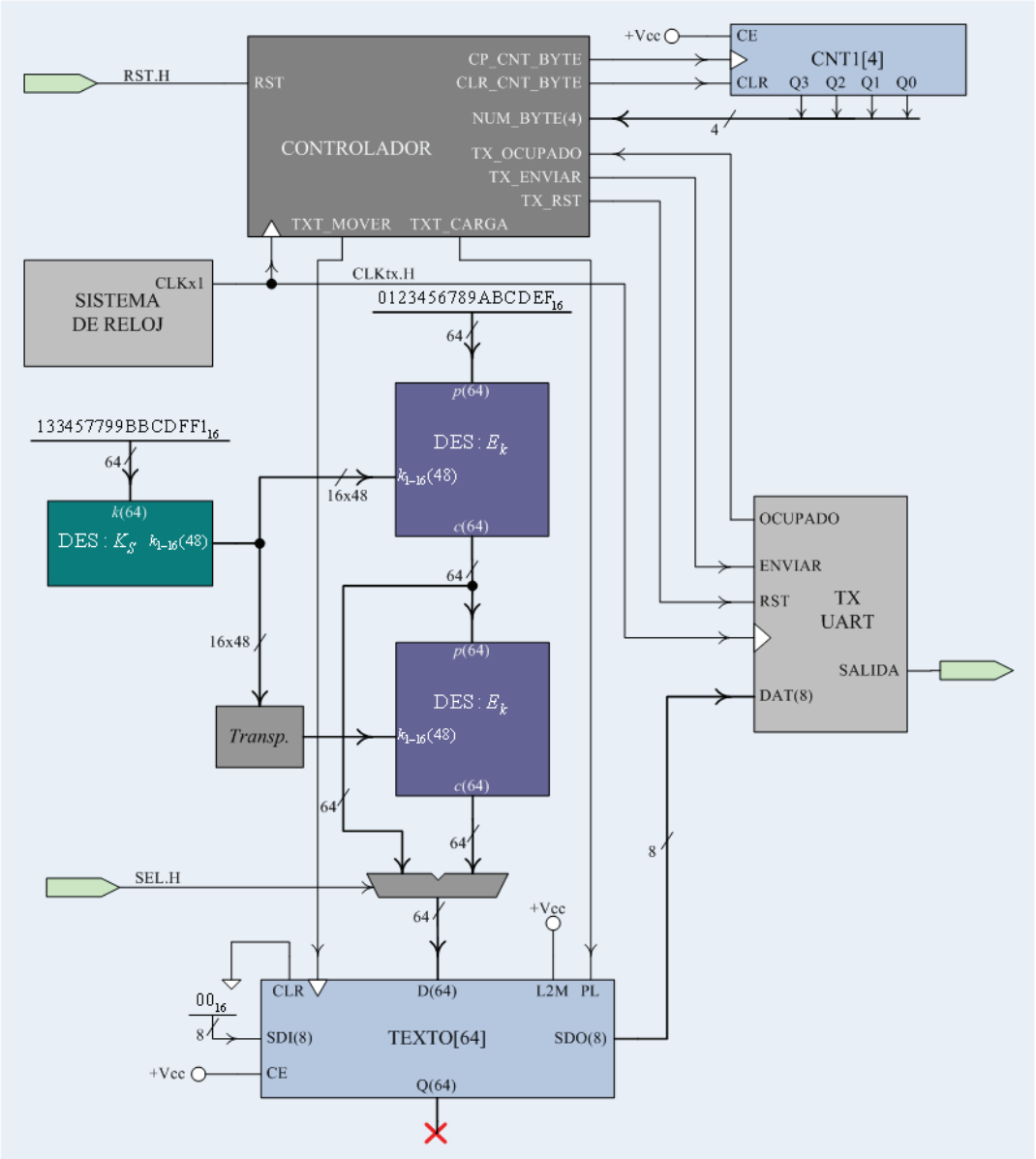

Figura 3.13 Partición funcional del Diseño Final.

La señal $pt_{1-64}.\mathrm{H}$ (con el valor $0123456789\mathrm{ABCDEF}_{16}$) representa el texto plano de 64 bits, $ct_{1-64}.\mathrm{H}$ representa el texto cifrado obtenido con E_k, y $dt_{1-64}.\mathrm{H}$ es el texto plano descifrado obtenido con D_k. La etapa de *Transposición* cambia el orden de las 16 claves de ronda k_i para efectos del descifrado. La descripción RTL es la siguiente:

$$
\begin{aligned}
&\mathrm{RST}: \rightarrow (T_0)\\
&T_0 : \mathrm{CNT1} \leftarrow 0,\\
&T_1 : \mathrm{SEL} = 0 : \mathrm{TEXTO} \leftarrow E_k(pt),\\
&\quad \mathrm{SEL} = 1 : \mathrm{TEXTO} \leftarrow D_k(ct),\\
&T_2 : \left(\mathrm{OCUPADO}, \overline{\mathrm{OCUPADO}}\right) \Big/ (T_2, T_3),\\
&T_3 : \mathrm{TxUART\_BUF} \leftarrow \mathrm{TEXTO(63\text{-}56)},\\
&\quad \mathrm{CNT1} \leftarrow \mathrm{CNT1} + 1,\\
&T_4 : \textbf{shl}\ \mathrm{TEXTO},\\
&\quad \left(\mathrm{CNT1} = 8, \overline{\mathrm{CNT1} = 8}\right) \Big/ (T_5, T_2),\\
&T_5 : \rightarrow (T_5)
\end{aligned}
$$

IMPLEMENTACIÓN DE LA FUNCIÓN DE CIFRADO DES

Tal como se ha indicado antes, decidimos que el diseño de la implementación del Data Encryption Standard sea de tipo combinatorial; con el fin de aumentar la velocidad de cálculo de los criptogramas, y así, reducir al mínimo la latencia, y simplificar las consideraciones por sincronización.

Para aumentar la velocidad de respuesta del circuito DES, la

implementación de la secuencia rondas, tanto de la Red de Feistel como del Esquema de Llaves, fue diseñada utilizando una arquitectura tipo *lazo desenvuelto* (*unrrolling*), en la que se replica el circuito de la ronda n veces (según el algoritmo); este diseño ofrece altas velocidades de respuesta, aunque también tiene una demanda de recursos muy alta. Otra opción de implementación es la arquitectura de *iteración por lazo*, en la que sólo se implementa físicamente una ronda, y las n iteraciones del algoritmo son llevadas a cabo realimentando el resultado de la ronda anterior; pero este tipo de implementación es más lenta e implica una recarga en el módulo de control.

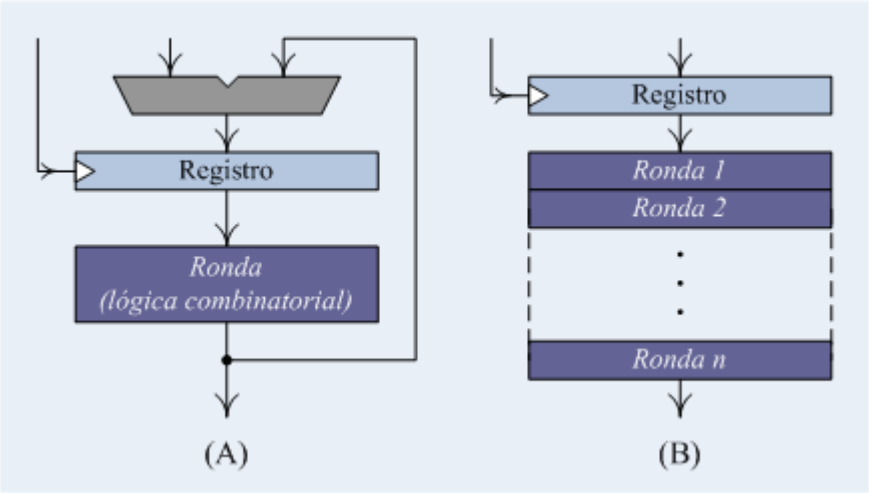

Figura 3.14 Arquitecturas de implementación: (A) Iteración por lazo; (B) Lazo desenvuelto.

Los circuitos de la función de cifrado E_k y del Esquema de llaves K_s, han sido diseñados como módulos separados, siguiendo el esquema de operación que se estudió en el capítulo 2.

Para la implementación de la etapa de Sustitución de la función de ronda f, se siguió como guía el diseño propuesto en la implementación del DES que hace Xilinx en el documento [33].

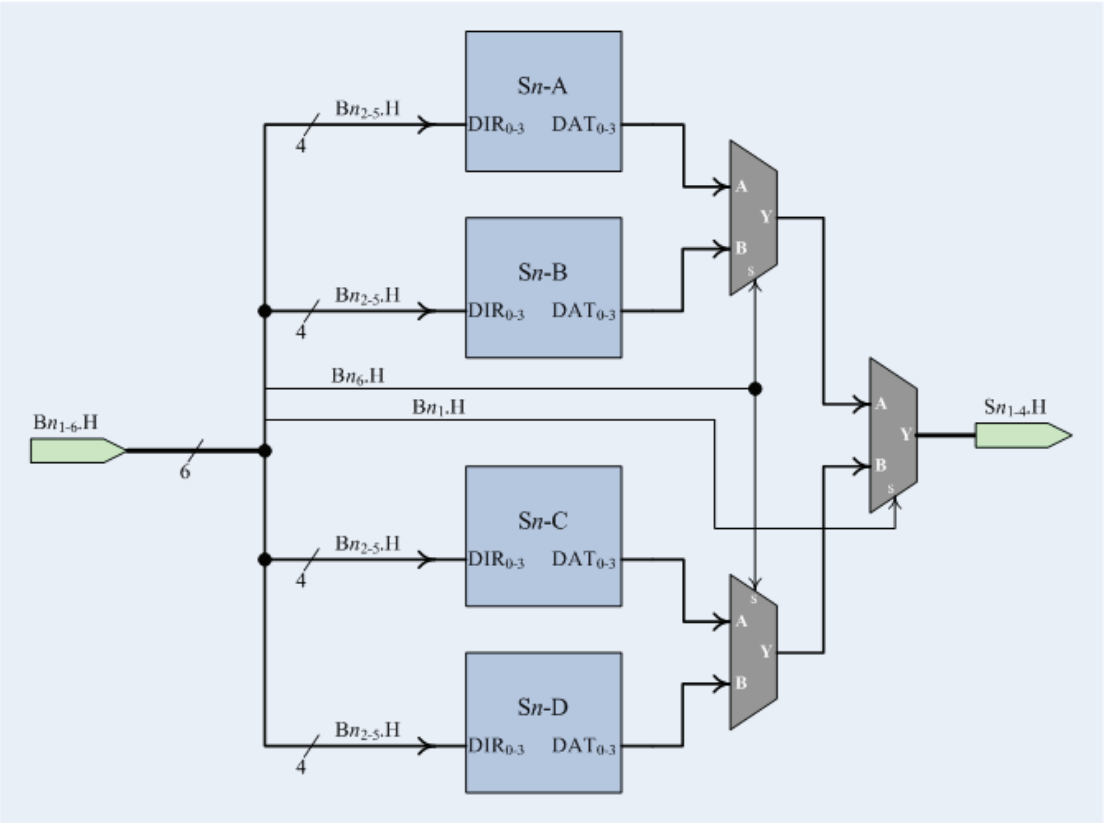

Figura 3.15 Esquema de la etapa de sustitución para un bloque B_i.

Esta implementación distribuye el contenido de cada Caja S en 4 tablas de verdad (A, B, C y D) de tamaño 16x4, construidas en base a 4 tablas LUT de 16x1. Las cuatro tablas de verdad contienen las salidas para las 64 combinaciones de Bn_{2-5}, mientras que $Bn_1.\mathrm{H}$ y $Bn_6.\mathrm{H}$ se usan en las dos etapas selectoras subsiguientes para determinar qué combinación es la

correspondiente. Los MUX 2-1 implementados se basan el MUX F5 de la arquitectura del Virtex. Este diseño ayuda a aprovechar de mejor manera los recursos nativos del FPGA. Para el resto de operaciones se utilizaron implementaciones estándares.

3.2.5 Pormenores Técnicos Relevantes

Para lograr la implementación de las distintas etapas del circuito se necesitaron de recursos auxiliares, provenientes tanto del Lenguaje utilizado (VHDL) como de las características que aporta el fabricante (Xilinx) tanto en el producto en sí como en las herramientas de desarrollo. Cabe recalcar que el diseño completo se ha descrito en código (texto).

A continuación referiremos cuáles y en dónde se utilizaron los recursos del Lenguaje y del Fabricante.

3.2.5.1 Ventajas del Lenguaje Aprovechadas

El siguiente cuadro ayuda a resumir las características que se explotaron del Lenguaje VHDL. De todas las etapas la del DES (función de cifrado y generador de clave) fue la que en particular utilizó la mayor parte de las características tabuladas.

Característica	*Implementación*	*Beneficio*
Paquetes (archivos package)	Agrupaciones de Definiciones de Tipos de Datos	Organización del código
	Agrupaciones de Definiciones de Componentes y Funciones más utilizadas	
Definiciones de Tipos y Subtipos de Datos (type y subtype)	Tipos de Datos usados en la Red de Feistel (bloques, medios bloques, llaves, etc.)	Versatilidad en el manejo de las variables
		Obtener una representación que vincule el manejo de los datos con los conceptos del algoritmo
	Vectores de Grupos de Datos (de llaves, de medias llaves y de medios bloques)	Obtener una representación más comprensible de agrupaciones complejas de datos, especialmente para el DES
	Variables de Estado	Versatilidad en la codificación de Máquinas de Estado
Funciones (módulos function)	Operaciones de Desplazamiento rol y srl del estándar IEEE 1164 (año 1993)	Disponibilidad (estas funciones no estaban implementadas en el sintetizador de Xilinx)
		Versatilidad en la codificación del componente del Esquema de Llaves Ks del DES
Procesos (módulos process)	Máquinas de Estado Finito (FSM)	Agilizar implementaciones de Máqinas de estado Finito (FSM) tipo *Mealy*
Arquitectura de Comportamieto (*behaivoral* architecture)	Operaciones sencillas (conjunciones [and], disyunciones [or], negaciones [not], concatenaciones [&], etc.)	Disminuir la carga en la implementación Estructural
Arquitectura Estructurada (*structural* architecture)	Módulos Digitales complejos	Diseñar una implementación basada en Partición Funcional

Tabla 3.III Propiedades del lenguaje VHDL aprovechadas en el diseño.

En general se puede decir que en el diseño ha prevalecido la arquitectura estructural; siendo la arquitectura de comportamiento aplicada básicamente

en la implementación de controladores, mediante máquinas de estado finito tipo Mealy; y las sentencias concurrentes para la interconexión de módulos.

3.2.5.2 Recursos del Fabricante Aprovechados

De la misma manera se ha compendiado los recursos del Fabricante aprovechados en la tabla 3.IV.

Recurso		***Implementación***	***Beneficio***
Módulos *Nativos*	Tablas de Verdad (*Look-up Tables,* LUT)	Bloques tipo ROM necesarios para las Cajas S del DES	Optimización de Recursos
	Multiplexadores (*Muliplexers,* MUX)	Multilexores para la implementación de las Cajas S del DES	
	Biestables tipo D (*Flip-Flops D,* FF-D)	Como Buffers de señales	
Núcleos de Propiedad Intelectual *(IP Cores)*	Registros de Sostenimiento *(Parallel Registers)*	Buffers Paralelos	Optimización de recursos.
	Registros de Desplazamiento *(Shift Registers)*	Registros para transmisión y realimentación	Aumento de la confiabilidad en la implementación.
	Puertas Lógicas para Bus de Datos (*Bus Gates*)	Operaciones binarias entre buses de datos (e.g. xor)	Disminución carga de implementación
	Contadores Binarios *(Binary Counters)*	Contadores binarios de distintas dimensiones	
	Manejadores Digitales de Reloj (*Digital Clok Manager,* DCM)	El Sistema de Reloj	Mayor precisión en la frecuencia de transmisión.

Tabla 3.IV Recursos del Fabricante aprovechados en el diseño.

3.2.5.3 Componentes Notables

En esta sección describiremos los bloques cardinales del diseño como lo son el Sistema de Reloj, el Trasmisor Serial Asincrónico y el Registro Universal 8x8. Se revisará el diseño y se explicará su funcionamiento.

3.2.5.3.1 Sistema de Reloj

El Sistema de Reloj recibe como entrada la señal proveída por el reloj diferencial de la tarjeta XUP-V2P y devuelve dos señales de reloj: el reloj de transmisión CLKtx.H y el reloj de recepción CLKrx.H (a una frecuencia 16 veces más rápida). Para nuestro caso la frecuencia de transmisión es de 115200 Hz.

Las velocidades de trasmisión serial estándar (1200, 2400, 4800, 9600, 19200, 38400, 115200 bps, etc.) se obtienen mediante el escalamiento en tiempo (división de frecuencia) de una señal de reloj con una frecuencia base de 73.7280 MHz (o un múltiplo).

$$f_G = 73.7280\,[\text{MHz}] = 100 \cdot \left(\frac{2304}{3125}\right)[\text{MHz}] \qquad \textbf{(2.26)}$$

La ecuación **(2.26)** muestra el factor de conversión necesario para obtener

la frecuencia base a partir de una señal de 100 MHz. Para implementar este escalamiento se utilizó 3 módulos DCM del Virtex-II Pro, en cascada, ya que estos módulos están especialmente hechos para manejar señales de reloj a tan altas frecuencias. La figura 3.16 especifica los factores de relación (R) configurados en cada DCM.

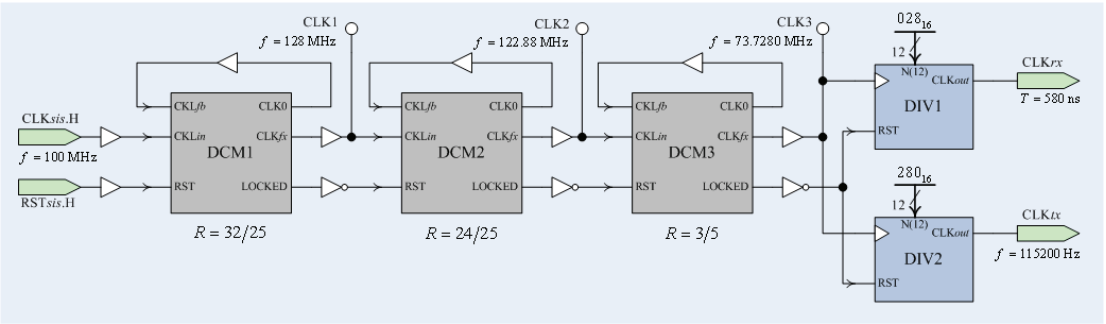

Figura 3.16 Diagrama de Bloques del Sistema de Reloj.

Dado que el rango de operación (de salida) de los DCM está entre 1 y 240 MHz, no es posible utilizarlos para bajar hasta la frecuencia de transmisión (en el orden de las centenas o miles de Hertz). Por esto se diseñó un bloque de división de frecuencia para lograr las frecuencias de transmisión serial. (Para mayor información sobre los DCM revisar el documento [34].)

Velocidad **[bps]**	***Relación*** f_G/f_{Tx}		***Relación*** f_G/f_{Rx}	
	Decimal	***Hexdecimal***	***Decimal***	***Hexdecimal***
115200	640	280	40	28
57600	1280	500	80	50
38400	1920	780	120	78
19200	3840	F00	240	F0
9600	7680	1E00	480	1E0

Tabla 3.V Factores de conversión de frecuencias.

Tal como se puede ver en la figura 3.16, se decidió colocar dos Divisores de Frecuencia en paralelo, que reciban como entrada la salida de la etapa de alta frecuencia (el reloj base) y que generen la señales de reloj de transmisión y de recepción respectivamente. Si bien una formación en cascada disminuiría la cantidad de recursos usados por el Divisor, la formación en paralelo incrementa la precisión.

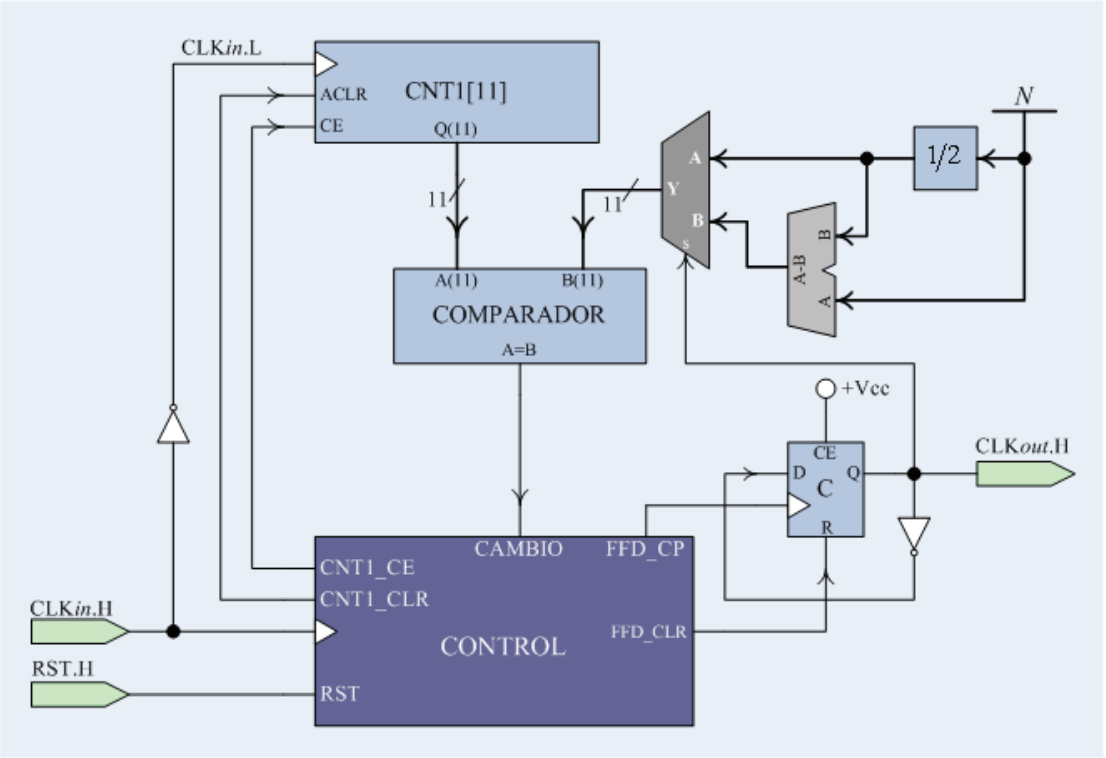

Figura 3.17 Partición funcional del Divisor de Frecuencia.

Tal como se puede apreciar en la figura 3.17, la partición funcional del Divisor de Frecuencia es muy sencilla. CNT1 cuenta la cantidad de pulsos del reloj entrada que han pasado y C cambia de estado dependiendo de si se ha alcanzado la relación configurada. El MUX sirve para dividir el

conteo, una mitad para el nivel alto, y la otra mitad para el nivel bajo; mientras que la ALU permite el uno de números impares como factor de conversión. La señal CAMBIO.H se activa cada vez que se ha llegado al término de cada mitad del ciclo y se debe conmutar la polaridad de la señal de reloj. La descripción RTL es la siguiente:

$$
\begin{Vmatrix}
\mathrm{RST}: \rightarrow (T_0) \\
T_0 : \mathrm{CNT1} \leftarrow 0, \mathrm{CLK}out \leftarrow 0, \\
T_1 : \overline{\mathrm{CAMBIO}} : \mathrm{CNT1} \leftarrow \mathrm{CNT1} + 1, \mathrm{C} \leftarrow \overline{\mathrm{C}} \\
\quad \mathrm{CAMBIO} : \mathrm{CNT1} \leftarrow 0, \\
\quad \rightarrow (T_1)
\end{Vmatrix}
$$

3.2.5.3.2 Transmisor Serial Asincrónico

El Trasmisor Serial Asincrónico recibe cuatro entradas, la señal de reloj a la frecuencia de transmisión que se desee, un byte de datos (8 bits paralelos), una señal de envío, y la señal de restauración; y emite dos salidas, la señal serial (de modo 8-N-1) y la una señal piloto que indica cuándo el módulo está ocupado en una transmisión.

La señal ENVIAR.H coordina la operación: con su flanco ascendente se cargan los datos de entrada en el registro DATOS y con su flanco descendente se inicia la transmisión. Cunado se ha cumplido el envío de

10 bits (1 bit de inicio, 8 bits de datos y 1 bit de parada) se detiene la transmisión y la señal OCUPADO.H se desactiva. Cuando se reinicia el proceso con la señal RST.H, se detiene el proceso y DATOS se carga (asincrónicamente) con el valor inicial $10\,0000\,0000_2$, para mantener el valor 1 (bit de parada) en la salida.

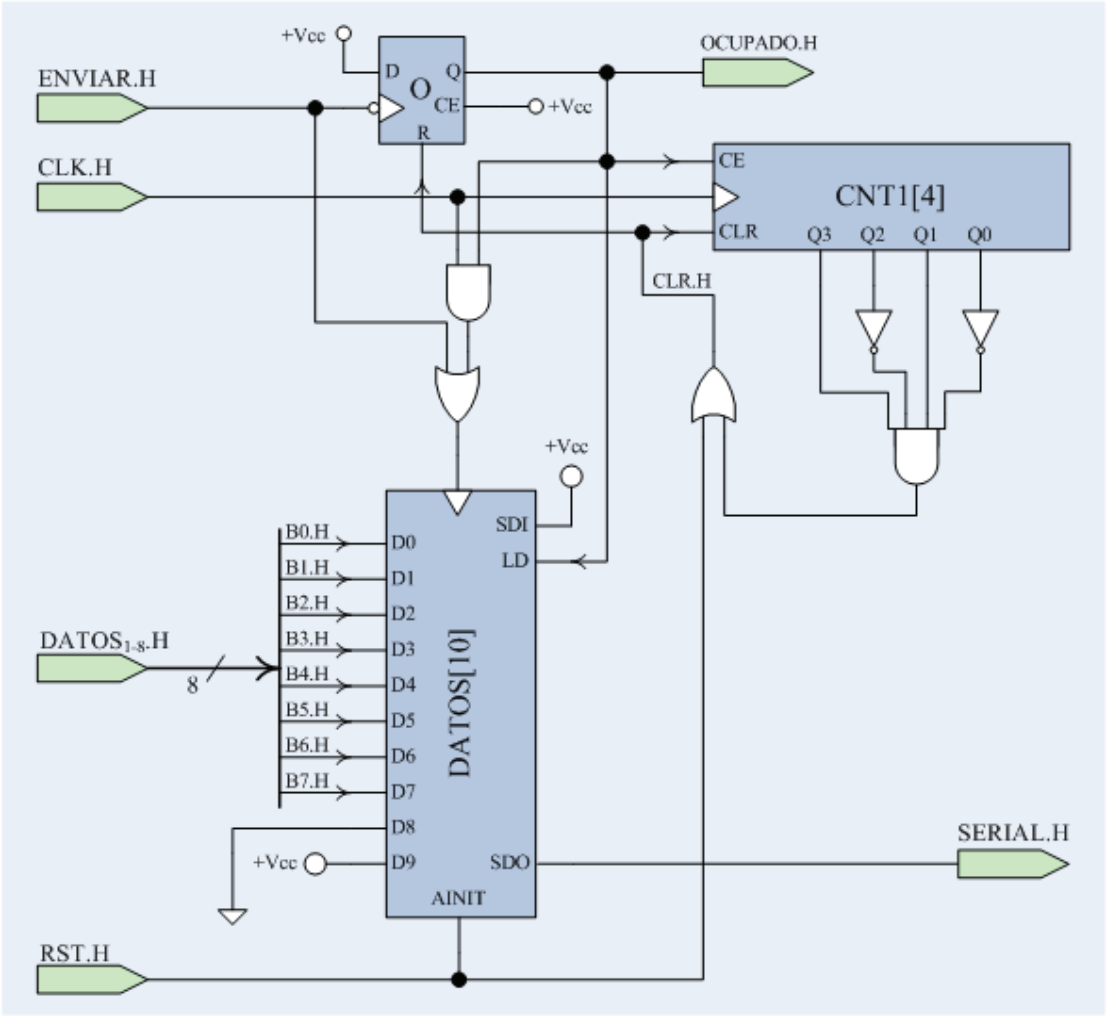

Figura 3.18 Diagrama esquemático del Transmisor Serial Asincrónico.

En este caso no se ha necesitado un controlador para coordinar el

proceso, gracias a la sencillez de operación del mismo.

3.2.5.3.3 Registro de Desplazamiento de 8x8

El Registro de Desplazamiento de 8x8 funciona de la misma manera y tiene las mismas señales que un registro universal de 8 bits de datos paralelos, con la diferencia de que en lugar de trabajar con bits trabaja con bytes. La entrada y la salida paralelas son de 8 caracteres (64 bits), y las entradas seriales de entrada y salida son de 8 bits. Las señales de control conservan su papel.

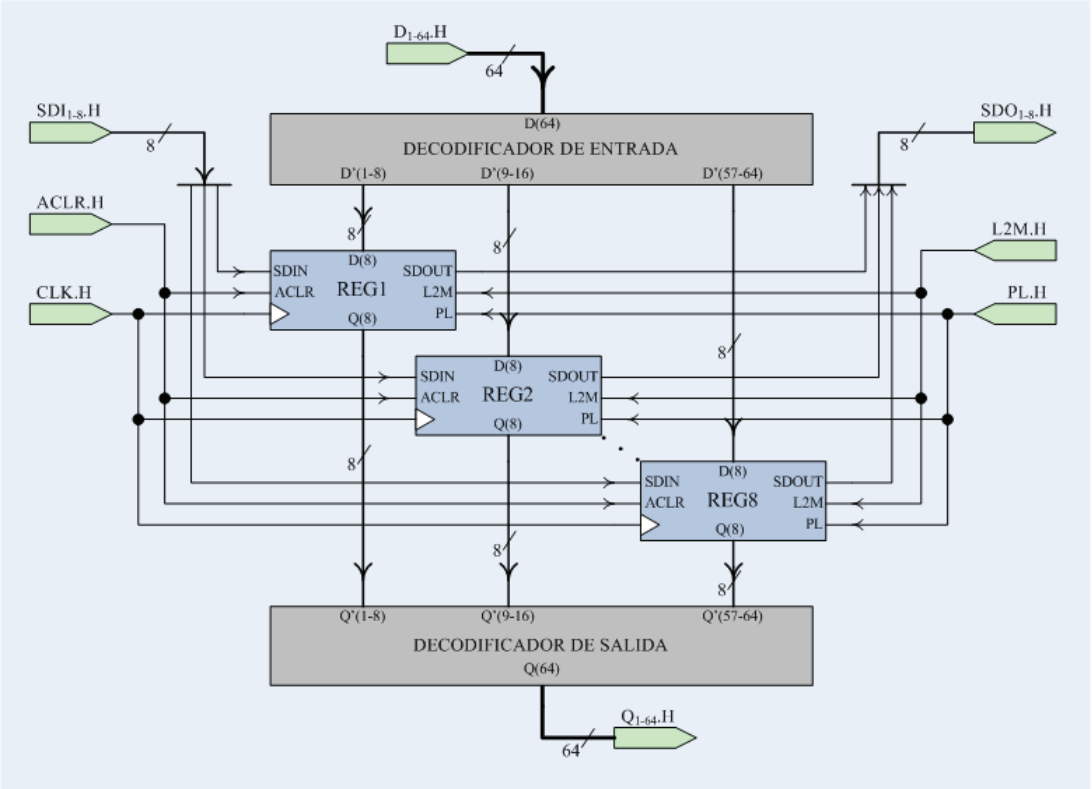

Figura 3.19 Diagrama esquemático del Registro Universal 8x8.

Para lograr este efecto fue necesario colocar dos decodificadores, uno para la entrada y otro para la salida paralelas. Estos decodificadores reubican cada bit de la entrada $D_{1-64}.H$ para que coincida con la entrada correspondiente de cada registro para lograr el efecto de desplazamiento de 8 bits en 8 bits. La misma idea se aplica a la salida paralela $Q_{1-64}.H$. (El código VHDL de este módulo aclara este artificio. Ver el apéndice A.)

3.2.5.4 Caso del Receptor Serial Asincrónico

Como un entorno de prueba suplementario se propuso incluir el circuito del cifrador DES en una línea de comunicación serial entre dos ordenadores; para esto era evidente que se necesitaría diseño de un receptor UART. En este apartado resumimos los diseños que propusimos para un Receptor UART y algunas consideraciones de elaboración.

3.2.5.4.1 Problemas encontrados

Entre las primeras dificultades que se enfrentaron estuvieron la determinación de la estrategia de lectura de los octetos recibidos y el tipo de arquitectura que se utilizaría.

Para la lectura de cada bit de los octetos se tiene como alternativas la

lectura sencilla (una sola muestra a la mitad del bit) o la lectura por sobre muestreo (*oversampling*) en la cual se toman algunas muestras del bit y se decide su valor por mayoría. La lectura sencilla ofrece como ventaja la simplicidad de tomar una sola muestra y la simplicidad del diseño, pero la fiabilidad es limitada; mientras que la lectura con sobre muestreo ofrece mayor fiabilidad, pero incrementa la complejidad sincrónica del sistema al necesitar relojes mucho más rápidos que la velocidad de transmisión.

Entre las arquitecturas de implementación tenemos la estructural, separando las funciones en módulos independientes, y la de comportamiento, haciendo todo el diseño como un único bloque de código. La arquitectura estructural permite la inclusión de mayor número de núcleos de propiedad intelectual, incrementando la eficiencia en el uso de recursos; mientras que la arquitectura por comportamiento da la facilidad de dejar a la herramienta de sintetización la administración de los recursos.

3.2.5.4.2 Diseños implementados

Tomando en cuenta las opciones disponibles decidimos probar dos diseños para el Receptor UART, uno que llamaremos Inédito, propuesto por nosotros de manera empírica, y otro que llamaremos Estándar, basado en normas bien conocidas para estos circuitos.

DISEÑO INÉDITO

Este circuito fue es nuestra propuesta empírica del diseño de un receptor UART 8-N-1, aplicando la técnica del sobre muestreo, y utilizando una arquitectura estructural (basada en partición funcional).

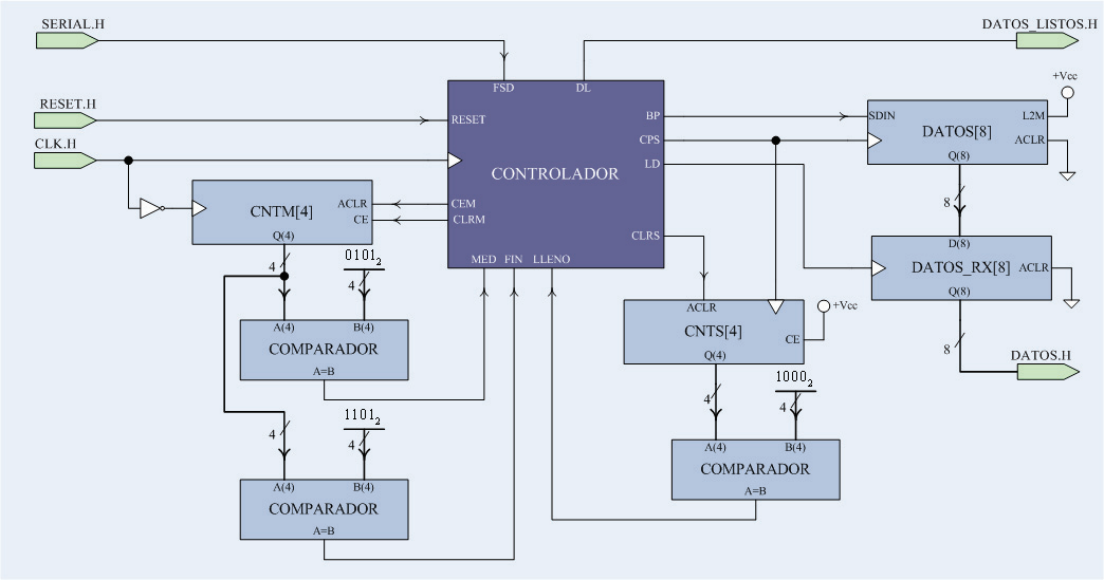

Figura 3.20 Partición funcional del Receptor Asincrónico propuesto.

Este circuito utiliza una señal de reloj 15 veces más rápida que la velocidad de transmisión que se esté usando. Toma 3 muestras, una en el pulso 7, una en el pulso 8 y una en el pulso 9; y con éstas decide por mayoría si el bit leído es 0 o 1. El contador CNTM se utiliza para el conteo de las muestras, mientras que el contador CNTS sirve para contar los bits que se han leído y han sido guardados en el registro DATOS. Una vez que se han completado los 8 bits, el contenido de DATOS se almacena en DATOS_RX, y se enciende la señal DATOS_LISTOS.H para que la

aplicación externa pueda acceder a los datos recibidos.

DISEÑO ESTÁNDAR

Este circuito fue diseñado basándonos en normas generalizadas de implementación de un Receptor UART, utilizando una arquitectura de comportamiento y aplicando la lectura de una sola muestra.

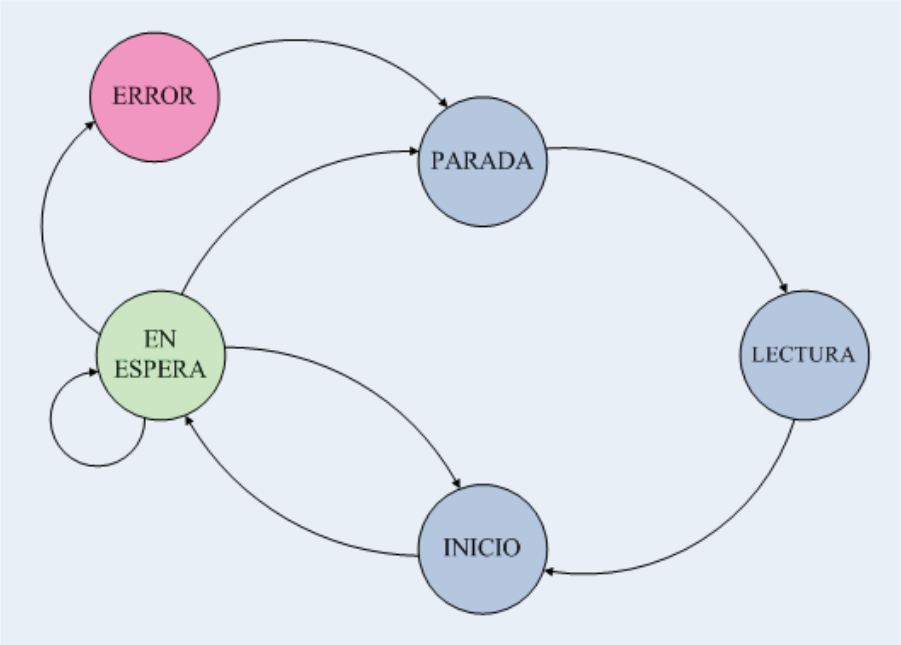

Figura 3.21 Diagrama de estados del Receptor UART estándar.

El diagrama de estados de la figura 3.21 resume el funcionamiento del circuito. El módulo inicia en estado de espera y permanece en él mientras siga recibiendo un 1 (bit de parada) como entrada. Cuando la señal serial cambia a 0 (bit de inicio) cambia al estado de inicio donde se toma una muestra a la mitad de la duración del bit para confirmar que se trata de un bit de inicio y no de una fluctuación. Verificado el bit de inicio, se pasa al

estado de lectura donde se lee una muestra de cada uno de los 8 bits de datos a la mitad de duración de cada uno. Al llegar al bit de parada (el noveno bit), se compara la lectura de ese bit; si es 1 se guarda el octeto leído, caso contrario se va a un estado de error y se descarta el byte recibido. Del estado de error se vuelve al estado de espera cuando se vuelve a recibir un 1 (bit de parada) en la entrada.

Siendo que estas son pruebas adicionales dejaremos las descripciones de estos circuitos limitada al código VHDL de las mismas, que presentamos en el apéndice A. Los resultados de simulación y de implementación en el FPGA se muestran en la sección **4.2**.

IV CAPÍTULO

4 RESULTADOS OBTENIDOS

Este capítulo comprende los resultados obtenidos con los diferentes circuitos que hemos diseñado para la implementación del cifrador. Mostraremos tanto los resultados *teóricos,* sustentados mediante la simulación de cada circuito, y los contrastaremos con los resultados *prácticos,* concretados en la plataforma de desarrollo (XUP-V2P) y observados mediante el monitor de puerto serial.

4.1 Resultados de la Implementación de los Circuitos

La revisión que prosigue se enfoca específicamente en los componentes utilizados para el circuito cifrador.

4.1.1 Del Sistema de Reloj y del Transmisor Serial Asincrónico

La figura 4.1 muestra la simulación del Circuito de Prueba 1 (sección **3.2.4.1**).

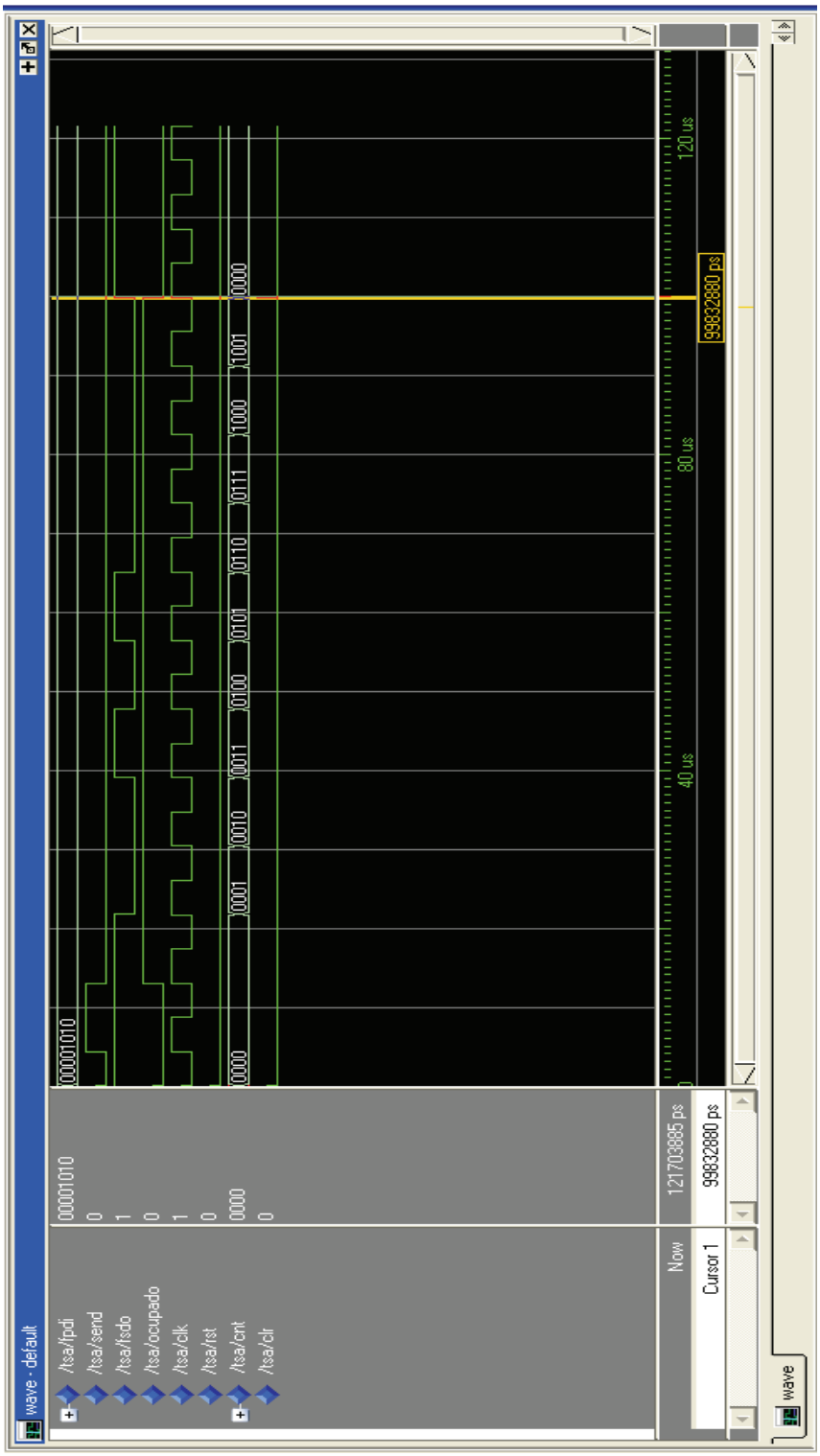

Figura 4.1 Simulación del Circuito de Prueba 1.

La señal $FPSI_{1-8}.H$ representa el flujo paralelo de datos de entrada a ser enviados; $SEND.H$, la señal de envío; y $FSDO.H$, el flujo serial de datos de salida. La simulación permite observar el diagrama de tiempo de la secuencia de bits de cada octeto enviado. Para una velocidad de transmisión de 115200 bps, el tiempo de duración de cada bit es de 8,6805555 µs.

Dado que el circuito tiene un comportamiento automático, sin uso de señales de control, lo que se genera es una secuencia repetitiva de 16 caracteres, desde el 00_{16} hasta el $0F_{16}$; la misma que debe ser recibida por el computador. La figura 4.1 muestra un caractér ($0A_{16}$) de la secuencia.

Con la ayuda del software de monitoreo del puerto serial se puede observar en el computador los datos emitidos por el circuito.

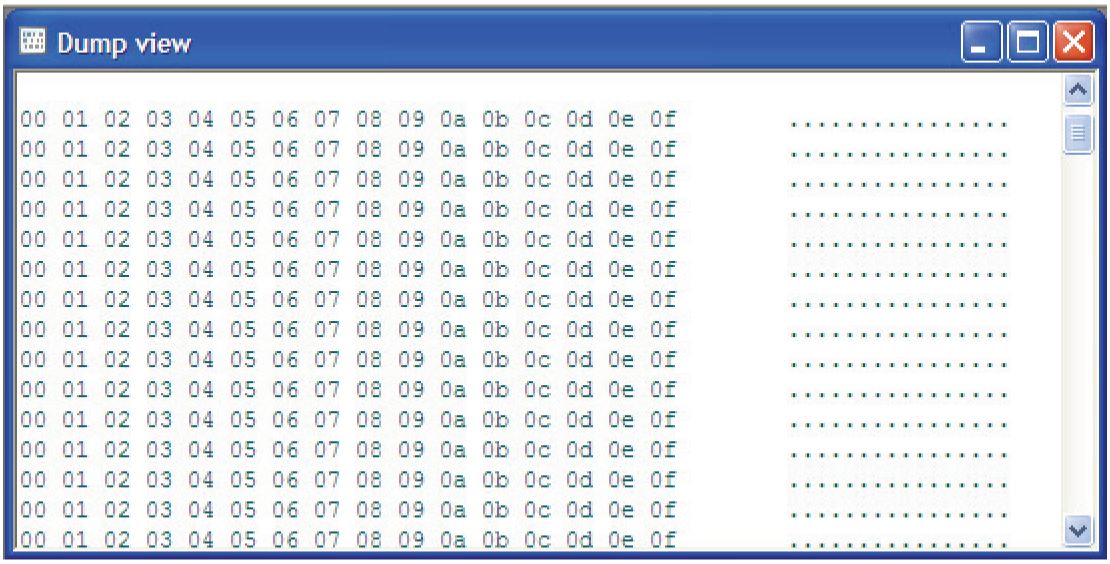

Figura 4.2 Salida del Circuito de Prueba 1.

La figura 4.2 muestra las tramas emitidas periódicamente por el circuito, tal como se describió en el diseño. Así queda comprobado entonces que los circuitos del Sistema de Reloj como el Transmisor funcionan correctamente.

4.1.2 Del Registro de Desplazamiento de 8x8 y su Transmisión Serial Asincrónica de 8 bytes

El Circuito de Prueba 2 fue simulado cargando $\mathrm{FEDCBA9876543210}_{16}$ en la entrada paralela $\mathrm{D}_{1-64}.\mathrm{H}$ del Registro de 8x8. Tal como se menciona en la sección **3.2.4.2**, el texto es desplazado en bloques de 8 bits (byte por byte), en el sentido de menos significativo a más significativo, y el caractér de salida es enviado al ordenador a través del Transmisor Serial Asincrónico. Así ocho veces.

El valor de la salida paralela $\mathrm{Q}_{1-64}.\mathrm{H}$ no sirve para observar el contenido del Registro y evaluar su comportamiento. Como entrada serial del Registro hemos utilizado un valor nulo (00_{16}) para rellenarlo a medida que se envía cada octeto.

La figura 4.3 muestra un comportamiento teórico exitoso del Registro de Desplazamiento.

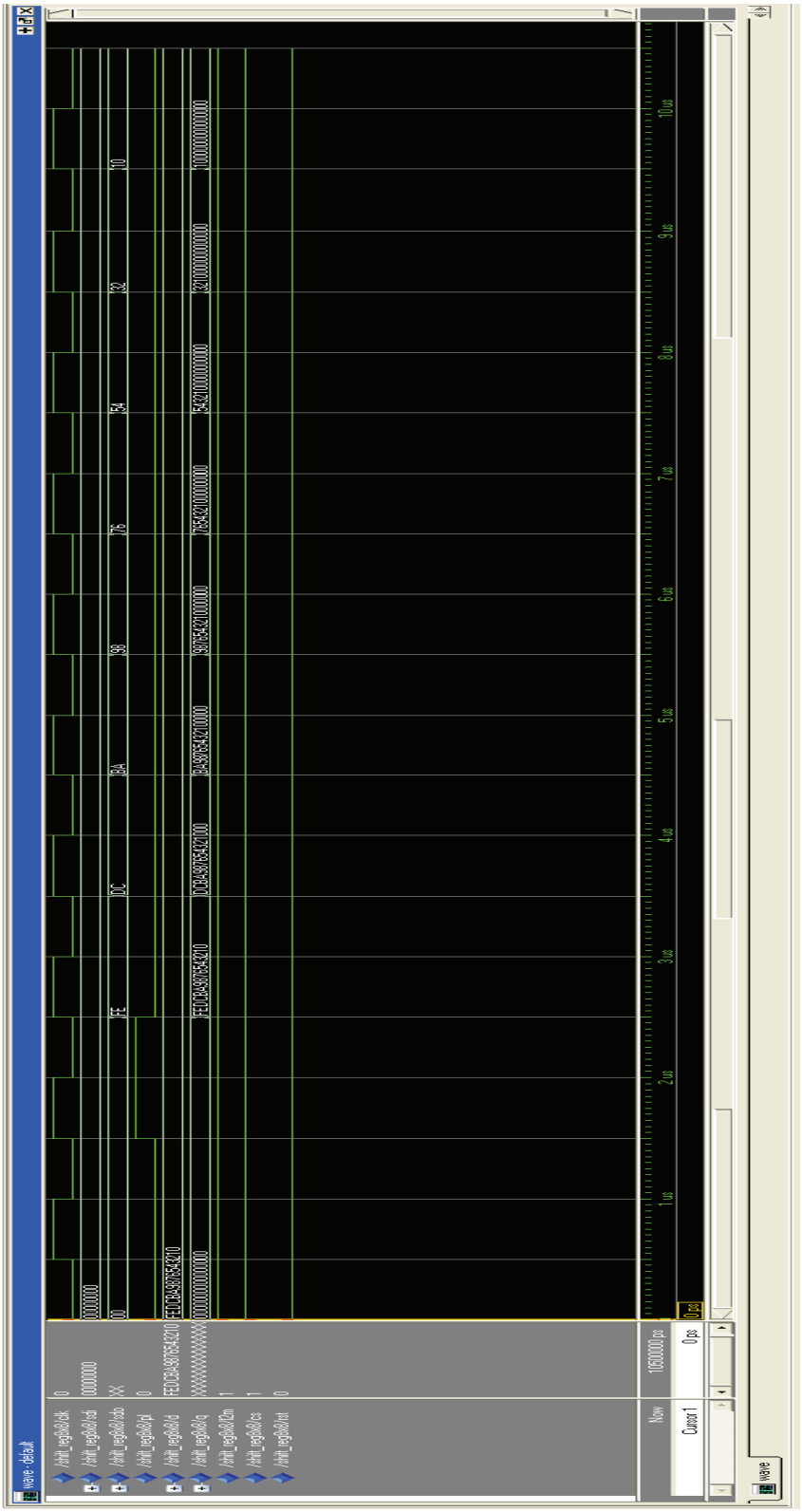

Figura 4.3 Simulación del Circuito de Prueba 2.

Para comprobar la correcta implementación del Circuito se espera recibir en el ordenador la misma secuencia cargada en el Registro, con el mismo orden significativo: $\mathrm{FEDCBA9876543210}_{16}$. La figura 4.4 muestra los datos recibidos en la prueba práctica.

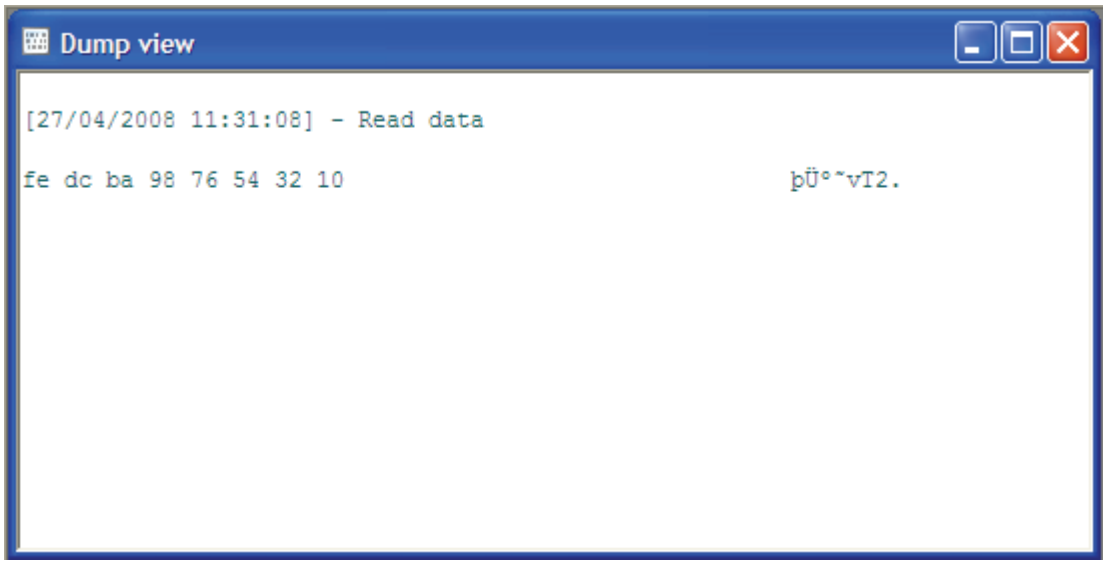

Figura 4.4 Salida del Circuito de Prueba 2.

Los resultados fueron correctos y se confirma que la implementación del Registro de Desplazamiento 8x8 funciona.

4.1.3 Del Cifrado de 8 bytes y su Transmisión

En la tercera etapa de prueba, simulamos independientemente los bloques E_k y K_s del DES (combinatoriales) para comparar sus resultados (teóricos) con la tritupla de prueba y la salida del programa de referencia citados en la

sección **3.2.4.3**. La figura 4.5 muestra la simulación del DES con la entrada (texto plano) $0123456789ABCDEF_{16}$ y con la calve $133457799BBCDFF1_{16}$.

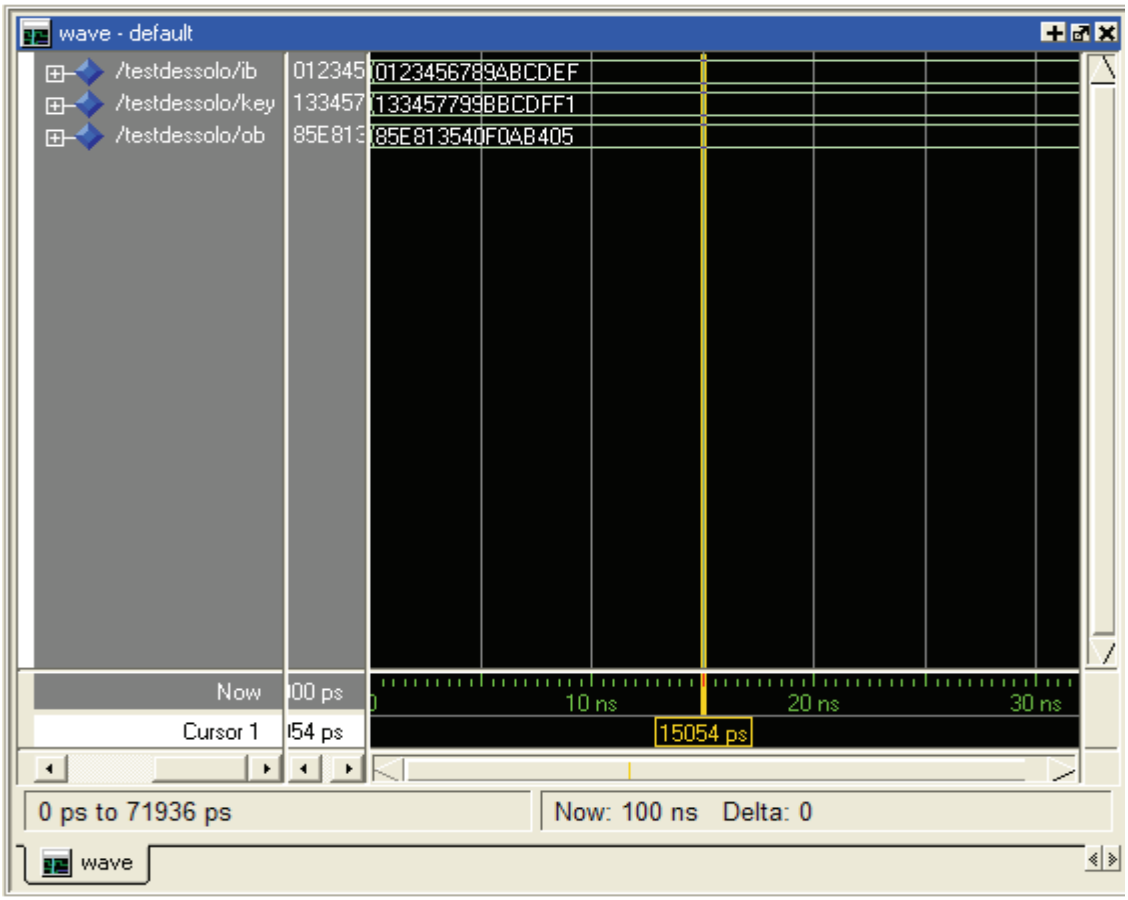

Figura 4.5 Simulación la Etapa DES: K_s y E_k.

En el gráfico se muestran, de arriba abajo, el texto plano, la clave y el texto cifrado. La respuesta $85E813540F0AB405_{16}$ coincide con el valor esperado según los valores de referencia. Se realizaron además algunas pruebas adicionales contrastando los valores de la simulación son los del programa en C++ que desarrollamos para este fin, como se muestra en la figura 4.6.

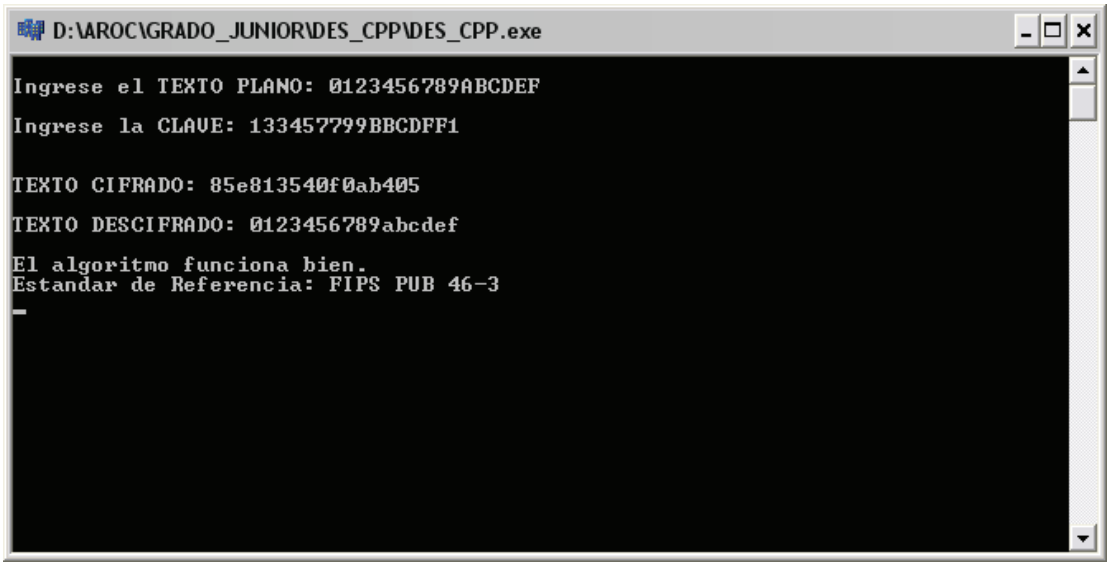

Figura 4.6 Programa de prueba del algoritmo DES.

En la parte práctica, el Circuito de Prueba 3 real debe retornar la secuencia mostrada en la pantalla anterior, y en el ordenador la misma debe aparecer en el monitor del puerto serial.

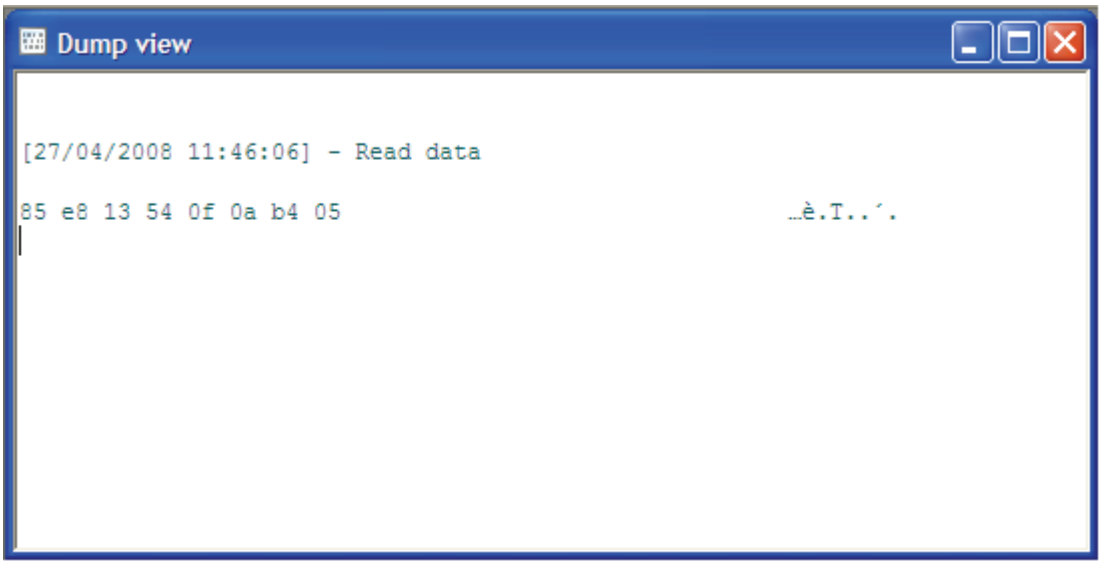

Figura 4.7 Salida del Circuito de Prueba 3.

La figura 4.6 dejar ver que el módulo DES funciona correctamente. Este mismo procedimiento se aplica al Circuito Final.

4.1.4 Del Circuito Final

Finalmente llegamos a la prueba del Circuito Final, que contiene tanto el cifrador como el descifrador DES. De la misma manera que con el Circuito de Prueba 3, se procedió a simular las etapas cifrador y descifradora para comprobar el diseño. (Ver sección **3.2.4.4**.)

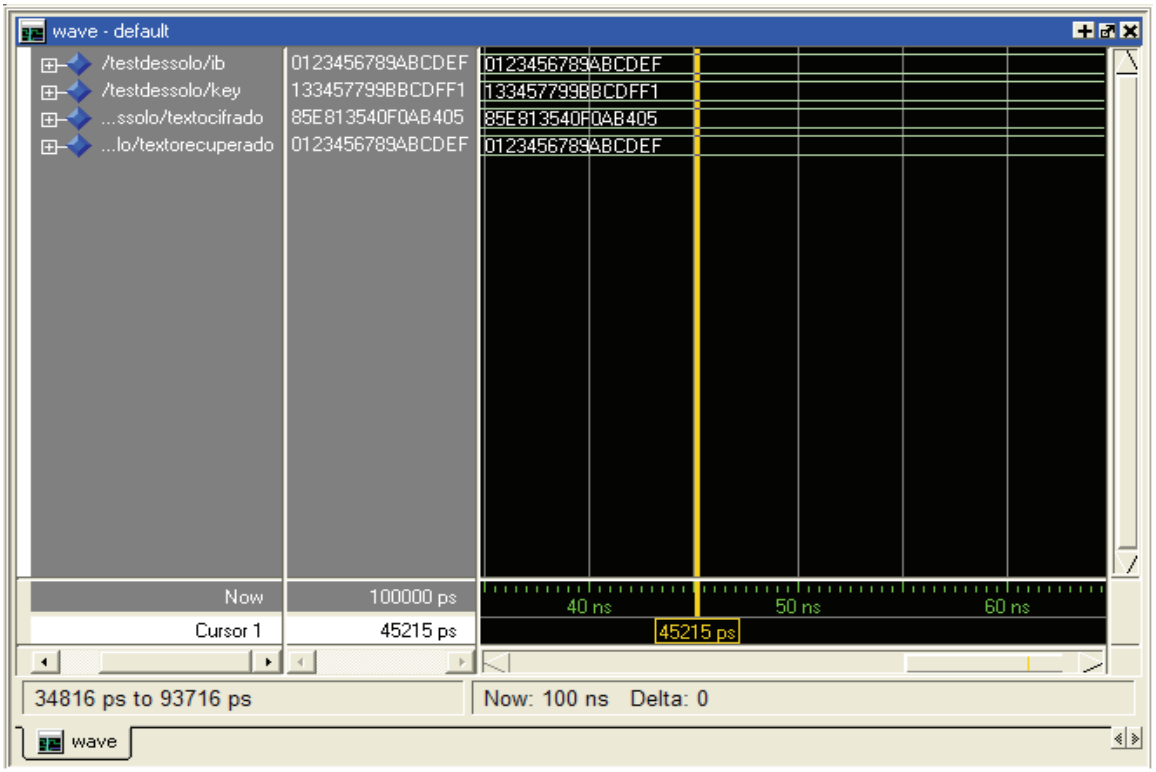

Figura 4.8 Simulación del Circuito Final.

Como se puede observar, el cifrador ha sido alimentado con el mismo texto plano ($0123456789ABCDEF_{16}$) y la misma clave ($133457799BBCDFF1_{16}$) de la prueba anterior; así mismo, la salida del cifrador ($85E813540F0AB405_{16}$) coincide con el texto cifrado ya verificado en la prueba 3, pero adicionalmente se puede ver que el texto descifrado coincide con el texto plano ingresado, lo que señala el correcto diseño de la función D_k del DES.

Para la prueba de funcionamiento del Circuito Meta se hizo uso de la señal selectora (conectada al interruptor SW0 del la XUP-V2P). Generamos dos secuencias: la primera seleccionando el texto cifrado y la segunda seleccionando el texto descifrado; así, el computador debería recibir dos cadenas consecutivas de 16 caracteres: la primera correspondiente al texto cifrado y la segunda al texto plano recuperado (descifrado).

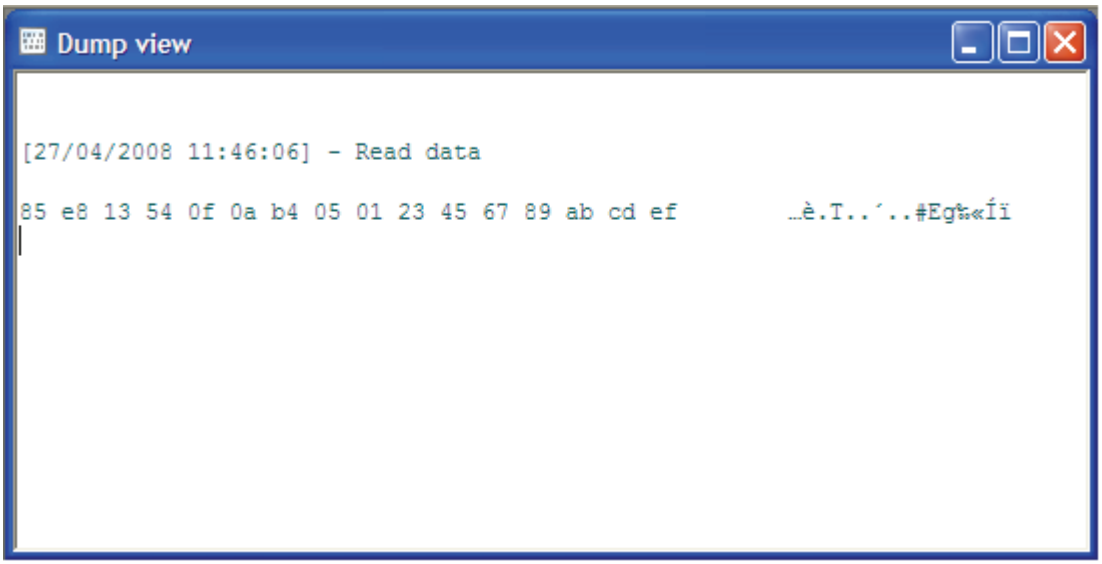

Figura 4.9 Salida del Circuito Final.

Con lo que se muestra en la figura concluye el desarrollo del circuito objetivo de nuestro proyecto; se verifica que el Circuito de cifrado y descifrado DES funcionan correctamente haciendo uso de los diseños digitales propuestos para su implementación en una arquitectura FPGA; y así mismo queda demostrado la utilidad de este tipo de entorno de desarrollo, y de las técnicas utilizadas, para aplicaciones de comunicación, en este caso serial, pero dejando abierta las posibilidad de implementaciones más complejas. (En las Conclusiones y Recomendaciones ampliamos estos puntos.)

La siguiente sección resume las experiencias de la implementación del componente adicional sugerido: el Receptor Serial Asincrónico; como un elemento *suplementario* al objetivo de nuestro proyecto.

4.2 El Caso del Receptor Serial Asincrónico

Tal como se ha planteado anteriormente, se introdujo, como un valor extra al proyecto, el diseño de un circuito Receptor Serial Asincrónico que, como aplicación adicional, permita incluir el cifrador en una línea de comunicación serial asincrónica entre dos ordenadores.

Los motivos para desarrollar un circuito como este fueron los siguientes:

1. Probar el alcance de las técnicas de diseño tradicional, en una plataforma

reconfigurable como lo son los FPGA, en un contexto de comunicaciones.

2. Dejar un punto inicial para futuras implementaciones de protocolos de seguridad de comunicación, más cercano a dicho contexto.

4.2.1 Diseños Propuestos

Tal como mencionamos en la sección **3.2.5.4.2**, desarrollamos dos diseños de Receptor Serial Asincrónico (RSA), uno empírico o inédito, que denominaremos RSA1, y uno siguiendo el esquema estándar, que llamaremos RSA2.

CIRCUITO RSA1

Para este circuito se necesita un reloj 15 veces más rápido que el reloj de transmisión, con el fin de hacer el muestreo de cada bit recibido. La figura 4.10 muestra el diagrama de tiempo con cada señal. Se puede apreciar la relación de frecuencia entre los bits y el reloj. A la mitad de cada bit se puede observar el pulso de muestreo.

Para esta prueba se simuló el circuito recibiendo el caractér AA_{16}. La señal $\mathrm{byte}_{1-8}.\mathrm{H}$ muestra los bits cargados en el octeto de salida; mientras el contador CNTB muestra la cantidad de bits recibidos; y $\mathrm{RxIn.H}$ la señal serial de entrada.

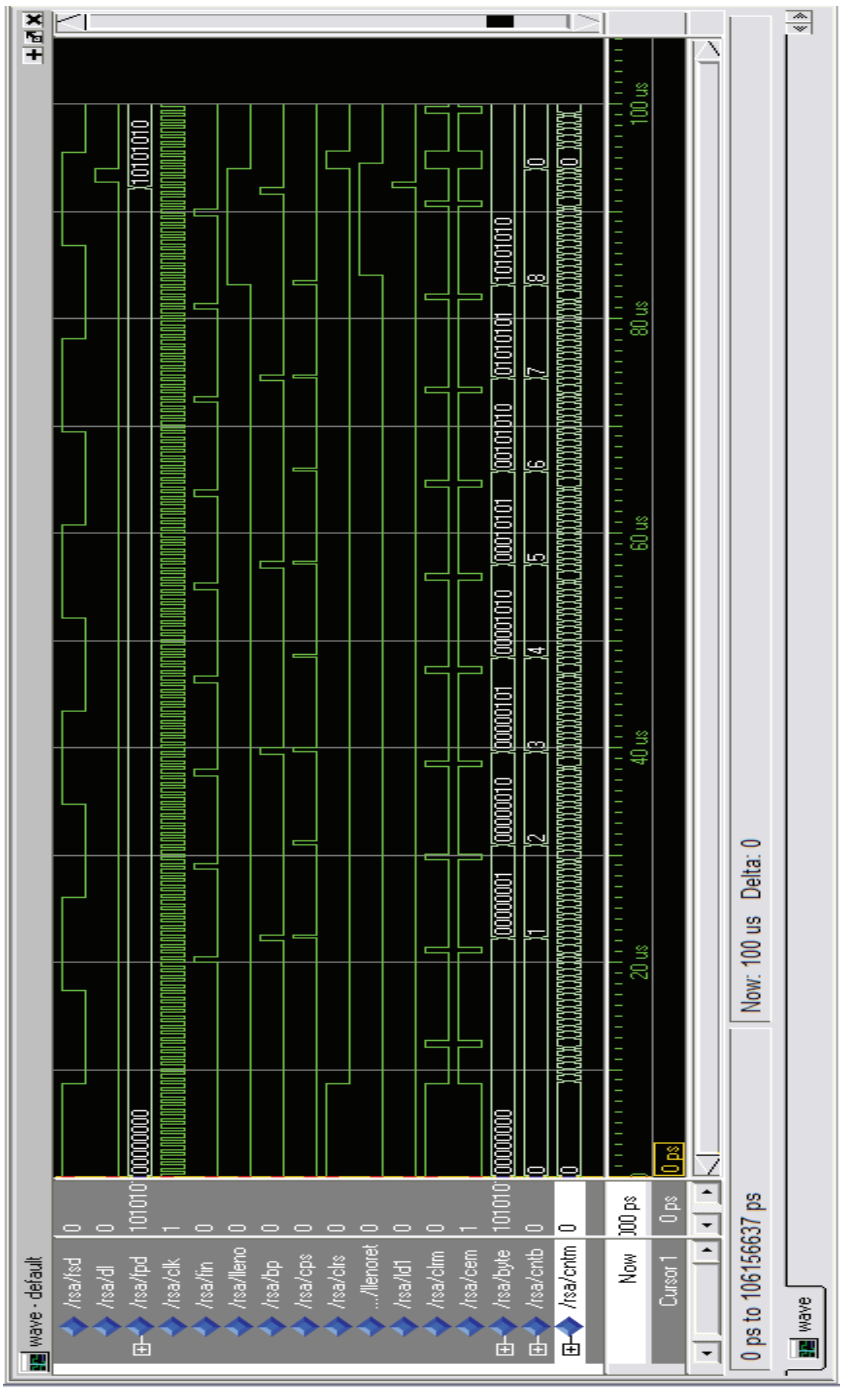

Figura 4.10 Simulación del Circuito RSA1.

La simulación muestra que el diseño está correctamente concebido y que puede funcionar.

CIRCUITO RSA2

En el circuito RSA2 se utilizó un diseño estándar (sección **3.2.5.4.2**): Se tienen 5 estados: *En Espera*, *Inicio*, *Lectura*, *Parada* y *Error*; el reloj utilizado es 16 veces más rápido que la razón de transferencia de bits; y se ha utilizado una arquitectura comportamental para la implementación.

Para la simulación del RSA2 recibimos el caractér $0D_{16}$. La señal RxPre muestra el estado presente del circuito; RxIn.H es la señal serial recibida; byte_cnt muestra los bits leídos acumulados; y $RxOut_{1-8}.H$ devuelve el octeto recibido.

De acuerdo con la simulación que muestra la figura 4.11, el circuito RSA2 funciona correctamente: RxIn.H empieza con un nivel alto, indicando que no hay transmisión en el canal, el estado presente es de inactividad, *En Espera*; a continuación, el bit de inicio (nivel bajo), el estado presente cambia a *Inicio*; luego vienen cada uno de los bits del byte a recibir, que en este caso son 00001101_2, el estado presente es *Lectura*; y finalmente RxIn.H se establece en el bit de parada, terminando la recepción del byte.

Esto demuestra que el diseño del RSA2 también está correctamente bosquejado y es funcional.

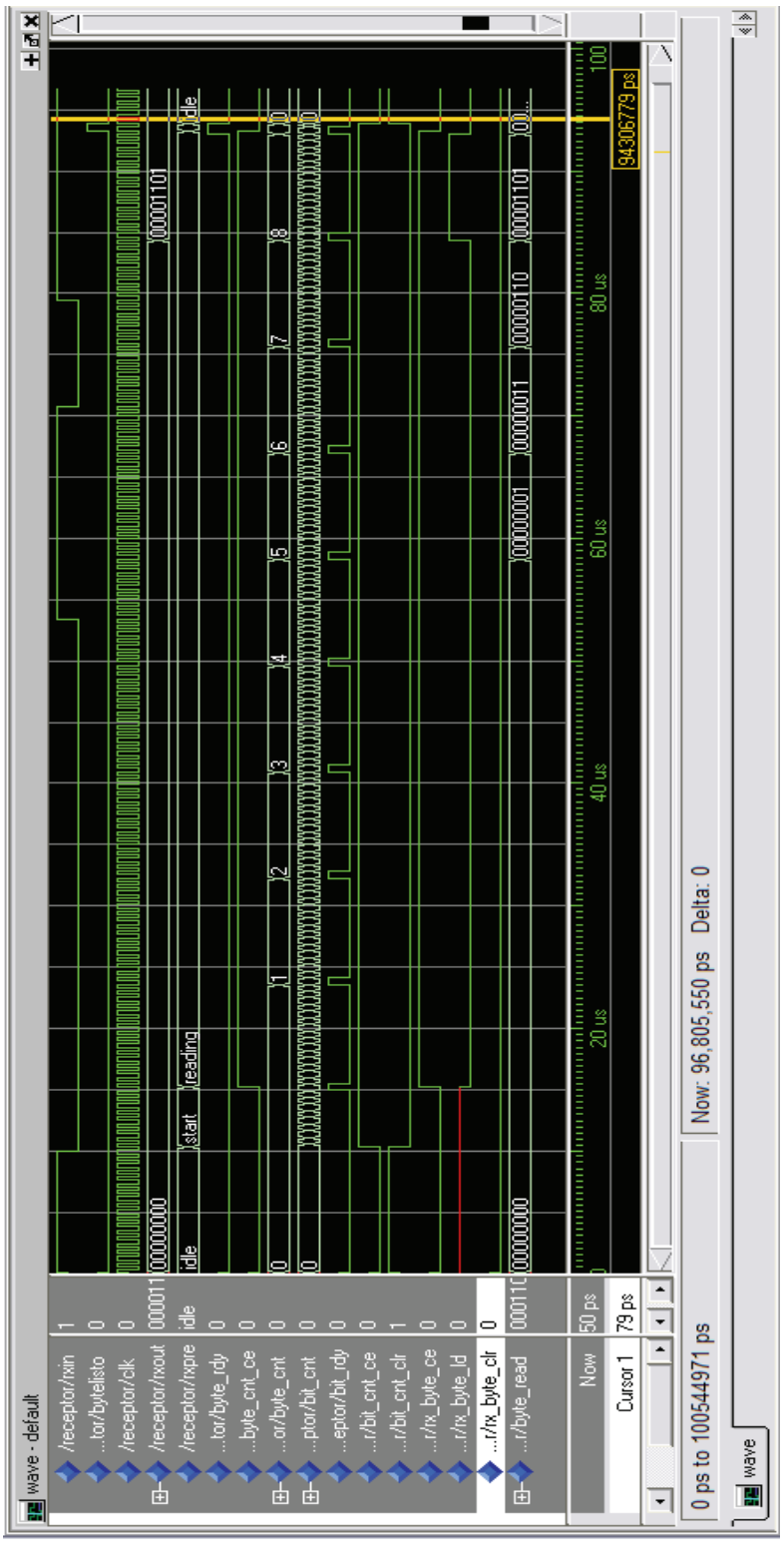

Figura 4.11 Simulación del Circuito RSA 2.

4.2.2 Resultados Obtenidos

Posterior a la etapa de prueba teórica mediante simulaciones, proseguimos con la etapa de prueba real implementando los RSA en la plataforma XUP-V2P.

Las pruebas fueron hechas haciendo un lazo (*loopback*) en el puerto serial del ordenador: los datos son enviado por el pin TXD a la tarjeta, recibidos por el módulo RSA de prueba, reenviado internamente al TSA y recibidos en el pin RXD del puerto serial el ordenador.

Los datos enviados como los recibidos serán observados por el software monitor del puerto serial en el ordenador.

CIRCUITO RSA1

Al momento de implementar el circuito RSA1, tal como se muestra en la figura 4.12, los datos sufrían una distorsión. Cabe mencionar que se presentaba un fenómeno inusual: los resultados variaban de sintetización a sintetización, aún cuando no se cambiaba el código.

Para la implementación se envió la cadena $30\ 31\ 32\ 33\ 34\ 35\ 36\ 37_{16}$ (o bien, del 0 al 7 en ASCII).

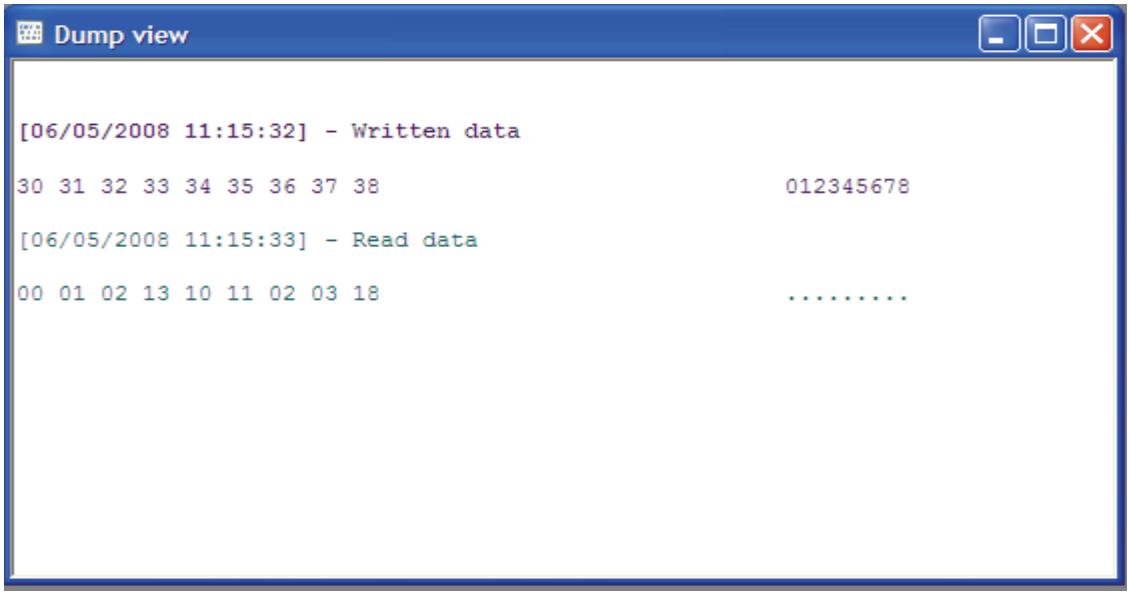

Figura 4.12 Salida del Circuito RSA 1.

Como se puede observar los datos se distorsionan al momento de pasar por el RSA1 implementado.

CIRCUITO RSA2

Para la prueba del circuito RSA2 se envió la cadena ASCII: AAAABBBB, en hexadecimal $41\ 41\ 41\ 41\ 42\ 42\ 42\ 42_{16}$. Para este circuito el comportamiento fue distinto. Tenía un comportamiento errático, pero algunas veces no devolvía ningún valor, aunque sí se observaban las señales piloto en la tarjeta.

La figura 4.13 muestra una de las observaciones del comportamiento del RSA2.

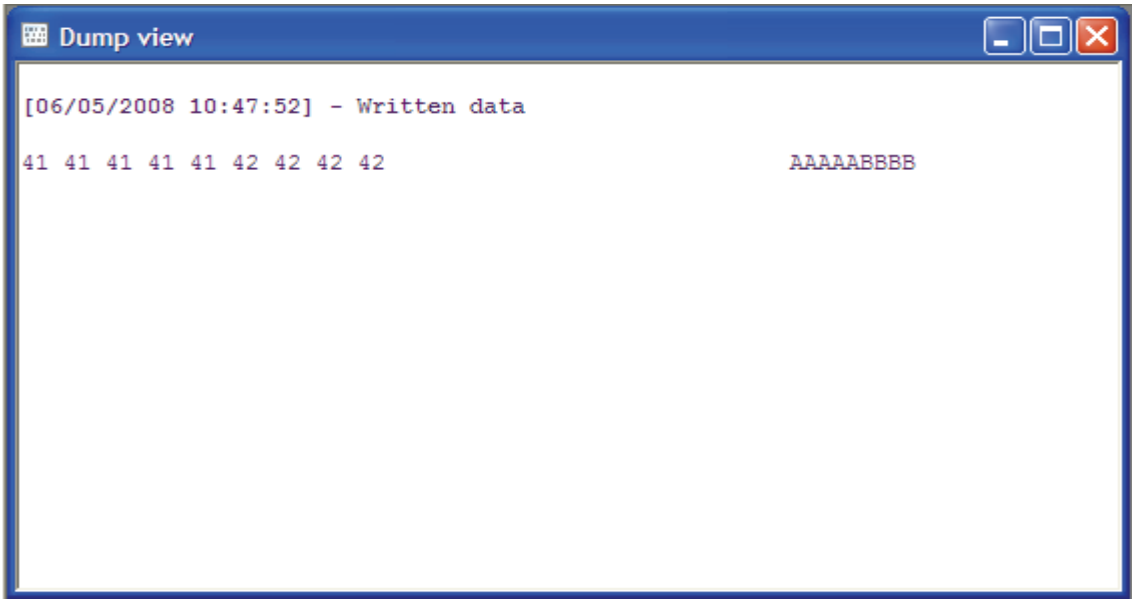

Figura 4.13 Salida del Circuito RSA 2.

Se analizaron ambos casos y se revisaron algunas referencias bibliográficas, académicas y comerciales, y pudimos concluir que el problema no yacía en el diseño funcional, sino más bien en el diseño de implementación (uso y ubicación de recursos nativos del FPGA) lo cual escapa del ámbito de este proyecto y queda como referencia para proyectos siguientes.

CONCLUSIONES Y RECOMENDACIONES

1. El correcto funcionamiento del Circuito Final demuestra que la aplicación de las técnicas de diseño aprendidas en los cursos de Sistemas Digitales son útiles para la implementación de algoritmos criptográficos simétricos de alta velocidad (como el DES) y módulos de comunicación (como el Transmisor UART). Esto muestra que el adiestramiento recibido en las aulas tiene un amplio alcance en las aplicaciones actuales.

2. La tecnología FPGA ha demostrado ser una plataforma de desarrollo compatible no sólo con las técnicas modernas de diseño digital, sino también con las tradicionales. Esto permite obtener lo mejor de ambos mundos, aprovechando la estructura (bien conocida) de las técnicas tradicionales y la flexibilidad de las modernas. De esta manera, se pueden generar diseños más confiables con gran versatilidad.

3. El lenguaje VHDL ha mostrado tener todas las propiedades y ventajas necesarias para hacer un diseño de estas dimensiones y características, sin añadir dificultad a la descripción del mismo. La traducción de las

operaciones propias de los procesos criptográficos a las instrucciones del Lenguaje VHDL fue limpia y consistente.

4. Con respecto al fabricante seleccionado, nosotros consideramos que la plataforma de desarrollo de Xilinx provee los recursos necesarios (de software y hardware) para hacer implementaciones complejas, versátilmente; pero esto demanda que la sintetización sea más elaborada y requiera de herramientas de gestión de los recursos físicos del FPGA, las cuales pueden llegar a ser muy complicadas de manejar. En general, podemos decir que hacer implementaciones con Xilinx es una opción de excelente calidad pero que requiere de conocimiento especializado en sus productos.

5. Como única recomendación, sugerimos que se incluyan más diseños de circuitos de comunicaciones (transmisión, recepción, procesamiento, buffers, etc.) en los Laboratorios de Sistemas Digitales. Esto ayudará a los estudiantes a desarrollar sus habilidades de diseño y especialmente a conocer las exigencias de las implementaciones y aplicaciones reales.

6. Finalmente, este trabajo es un precedente para futuros proyectos de desarrollo, sean de protocolos de comunicación, como de protocolos de seguridad, basados en hardware reconfigurable. Siendo el protocolo IPSec y los sistemas de Clave Pública opciones de gran valor y utilidad.

APÉNDICES

APÉNDICE A

CÓDIGO VHDL DE CADA MÓDULO FUNCIONAL

A.1 PRIMITIVOS.VHDL

```
------COMPONENTES PRIMITIVOS------

--DEPENDENCIAS

--      IEEE
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
--      UNISIM
library UNISIM;
use UNISIM.VComponents.ALL;
--      WORK
use WORK.TIPOS.ALL;

--DEFINICION DEL PAQUETE PRIMITIVOS

package PRIMITIVOS is

   ----------------Componentes Básicos Asincrónicos----------------

   component COMPARADOR11
        Port(A      : in  std_logic_vector(10 downto 0);
             B      : in  std_logic_vector(10 downto 0);
             A_EQ_B : out std_logic);
   end component;

   component mux11_2a1
         Port (MA: IN std_logic_vector(10 downto 0);
             MB: IN std_logic_vector(10 downto 0);
             S: IN std_logic_vector(0 downto 0);
             O: OUT std_logic_vector(10 downto 0));
        end component;

   component BUF8
        Port(I : in std_logic_vector(7 downto 0);
             O : out std_logic_vector(7 downto 0));
   end component;

   component MUX2a1
        Port( O   : out std_logic_vector(3 downto 0);
             I0  : in  std_logic_vector(3 downto 0);
             I1  : in  std_logic_vector(3 downto 0);
             SEL : in  std_logic);
   end component;
```

```
component ROM16x4
     Generic(    INIT3 : bit_vector:=x"FEDC";
                 INIT2 : bit_vector:=x"BA98";
                 INIT1 : bit_vector:=x"7654";
                 INIT0 : bit_vector:=x"3210");
     Port( DIR : in  std_logic_vector(3 downto 0);
           DAT : out std_logic_vector(3 downto 0));
end component;

----------------Componentes Básicos Sincrónicos----------------
component contador4
     Port (Q: OUT std_logic_vector(3 downto 0);
           CLK: IN std_logic;
           CE: IN std_logic;
           ACLR: IN std_logic);
end component;

component contador8
     Port (
            Q: OUT std_logic_vector(7 downto 0);
           CLK: IN std_logic;
            CE: IN std_logic;
           ACLR: IN std_logic);
end component;

component shift_reg10
     Port (
           CLK: IN std_logic;
           SDOUT: OUT std_logic;
           P_LOAD: IN std_logic;
           D: IN std_logic_vector(9 downto 0);
           LSB_2_MSB: IN std_logic;
           AINIT: IN std_logic);
end component;

component CONTADOR11
     Port(Q    : out std_logic_vector(10 downto 0);
           CLK  : in  std_logic;
           CE   : in  std_logic;
           ACLR : in  std_logic);
end component;

component full_shift_reg8
     Port (
           CLK: IN std_logic;
           SDIN: IN std_logic;
           SDOUT: OUT std_logic;
           P_LOAD: IN std_logic;
           D: IN std_logic_vector(7 downto 0);
           Q: OUT std_logic_vector(7 downto 0);
           LSB_2_MSB: IN std_logic;
           CE: IN std_logic;
           ACLR: IN std_logic);
     END component;

----------------Componentes Complejos----------------
component DIV
     Port(CLKin  : in  std_logic;
           N      : in  std_logic_vector(11 downto 0);
           CLKout : out std_logic;
```

```
            RESET  : in  std_logic);
   end component;

   --------------Funciones----------------
   function "rol" (A: std_logic_vector; N: integer) return std_logic_vector;
   function "srl" (A: std_logic_vector; N: integer) return std_logic_vector;
end PRIMITIVOS;

--CUERPO DEL PAQUETE DE PRIMITIVOS
package body PRIMITIVOS is
   function "rol" (A: std_logic_vector; N: integer) return std_logic_vector
is
        alias V : std_logic_vector(1 to A'length) is A;
        variable RESULT : std_logic_vector(1 to A'length):=(others=>'0');
   begin
        RESULT:=V;
        if N>0 then
             RESULT:=V(N+1 to V'length) & V(1 to N);
        end if;
        return RESULT;
   end "rol";
   -------------------------------------
   function "srl" (A: std_logic_vector; N: integer) return std_logic_vector
is
        alias V : std_logic_vector(1 to A'length) is A;
        variable RESULT : std_logic_vector(1 to A'length):=(others=>'0');
   begin
        if N>0 and N<V'length then
             RESULT(N+1 to V'length):=V(1 to V'length-N);
        end if;
        return RESULT;
   end "srl";
   -------------------------------------
end PRIMITIVOS;
```

A.2 TIPOS.VHDL

```
------TIPOS DE DATOS PARA EL DES FIPS46-3------

--DEPENDENCIAS
library ieee;
use ieee.std_logic_1164.ALL;
use ieee.numeric_std.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
library UNISIM;
use UNISIM.Vcomponents.ALL;

--DEFINICION
package TIPOS is
   --Tipos para llaves
   subtype llave is std_logic_vector(1 to 56);
   subtype media_llave is std_logic_vector(1 to 28);
   subtype llave_efectiva  is std_logic_vector(1 to 48);
   type llave_efectiva_vector is array  (natural range <>) of
        llave_efectiva;
   type media_llave_vector    is array  (natural range <>) of media_llave;
   --Tipos para bloque de datos
   subtype bloque is std_logic_vector(1 to 64);
   subtype medio_bloque is std_logic_vector(1 to 32);
   subtype medio_bloque_expandido is std_logic_vector(1 to 48);
   type medio_bloque_vector is array  (natural range <>) of medio_bloque;
   subtype nybble   is std_logic_vector(1 to 4);
   subtype nybble_expandido is std_logic_vector(1 to 6);
   --Tipos para variables de estado
   type VE1 is (t0,t1);
   type VE2 is (t0,t1,t2,t3);
   type VE3 is (t0,t1,t2,t3,t4,t5,t6,t7);
   type VE4 is (t0,t1,t2,t3,t4,t5,t6,t7,t8,t9,t10,t11,t12,t13,t14,t15);
   --Tipos para Comunicacion Serial
   subtype byte is std_logic_vector(7 downto 0);
end TIPOS;

--CUERPO
package body TIPOS is
end TIPOS;
```

A.3 CAJAS.VHDL

```
------CAJAS_S PARA EL DES FIPS46-3------

--DEPENDENCIAS
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use WORK.TIPOS.ALL;

--DEFINICION
package CAJAS_S is
   component S1
        Port(B1 : in  nybble_expandido;
             S1 : out nybble);
   end component;

   component S2
        Port(B2 : in  nybble_expandido;
             S2 : out nybble);
   end component;

   component S3
        Port(B3 : in  nybble_expandido;
             S3 : out nybble);
   end component;

   component S4
        Port(B4 : in  nybble_expandido;
             S4 : out nybble);
   end component;

   component S5
        Port(B5 : in  nybble_expandido;
             S5 : out nybble);
   end component;

   component S6
        Port(B6 : in  nybble_expandido;
             S6 : out nybble);
   end component;

   component S7
        Port(B7 : in  nybble_expandido;
             S7 : out nybble);
   end component;

   component S8
        Port(B8 : in  nybble_expandido;
             S8 : out nybble);
   end component;

end CAJAS_S;

--CUERPO
package body CAJAS_S is
end CAJAS_S;
```

A.4 COM_DES.VHDL

```
--Modulo circuito final / DES en modo de operacion directo

--DEPENDENCIAS
--      IEEE
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
--      UNISIM
library UNISIM;
use UNISIM.VComponents.ALL;
--      WORK
use WORK.TIPOS.ALL;
use WORK.PRIMITIVOS.ALL;

--ENTIDAD COM_DES
entity COM_DES is
   Port(SEL     : in  std_logic;
        PS1_DSR : out std_logic;
        PS1_TXD : in std_logic;
        PS1_RXD : out std_logic;
        PS1_RTS : in  std_logic;
        PS1_CTS : out std_logic;
        CLK     : in  std_logic;
        noRST   : in  std_logic;
        LED0    : out std_logic);
end COM_DES;

--ARQUITECTURA
architecture COM_DES of COM_DES is
   --Sistema de Reloj

   component RELOJ
        Port(CLK_580ns : out std_logic;
             CLK_115200Hz : out std_logic;
             CLK_RDY      : out std_logic;
             RST    : in  std_logic;
             CLK    : in  std_logic);
   end component;

   --Transmisor Serial Asincrónico
   component TSA
        Port(FPDI    : in  std_logic_vector(7 downto 0);
             SEND   : in  std_logic;
             FSDO    : out std_logic;
             OCUPADO : inout std_logic;
             CLK    : in  std_logic;
             RST : in  std_logic);
   end component;

   component SHIFT_REG8x8
        Port(CLK  : in  std_logic;
             SDI  : in  std_logic_vector(7 downto 0);
             SDO  : out std_logic_vector(7 downto 0);
             PL   : in  std_logic;
             D    : in  std_logic_vector(63 downto 0);
```

```
            Q    : out std_logic_vector(63 downto 0);
            L2M  : in  std_logic;
            CS   : in  std_logic;
            RST  : in  std_logic);
   end component;

   component DES_CORE
        Port(IB  : in  bloque;
            K   : in  llave_efectiva_vector(1 to 16);
            OB  : out bloque);
   end component;

   component KS
        Port(KEY : in  bloque;--llave
            K   : out llave_efectiva_vector(1 to 16));
   end component;

   --Señales Internas
   signal GND,VCC : std_logic;
   signal K,Kinv : llave_efectiva_vector(1 to 16);
   signal PT,CT,KEY,RT,TEXTOaMOSTRAR : bloque;
   signal OCTETO_Tx: byte:= x"00";
   signal RST,CLK_RST,SYS_RST : std_logic;
   signal CLK_RDY,noCLK_RDY,CLK_Tx : std_logic;
   signal SEND,Tx_BUSY : std_logic;
   signal RXD1,CARGAR,SHIFT_CP : std_logic;
   signal CP_CNT_BYTE,CLR_CNT_BYTE : std_logic;
   signal PRE,SIG : VE3;
   signal CNT_BYTE_Tx : std_logic_vector(3 downto 0);
begin
   GND<='0';
   VCC<='1';
RELOJES: RELOJ Port map(open,CLK_Tx,CLK_RDY,CLK_RST,CLK);

DEC_SIG: process(PRE,CNT_BYTE_Tx,Tx_BUSY)

begin
   case PRE is
        when t0 => SIG<=t1;
        when t1 => SIG<=t2;
        when t2 => if Tx_BUSY='0' then SIG<=t3; else SIG<=t2; end if;
        when t3 => SIG<=t4;
        when t4 => if CNT_BYTE_Tx=x"8" then SIG<=t5; else SIG<=t2; end if;
        when t5 => SIG<=t5;
        when others => SIG<=t5;
   end case;

end process;

MEM: process(CLK_Tx,RST)
begin
   if RST='1' then
        PRE<=t0;
   elsif rising_edge(CLK_Tx) then
        PRE<=SIG;
   end if;
end process;

DEC_SAL: process(PRE,CLK_Tx,noRST,CLK_RDY)
   variable T : std_logic_vector(7 downto 0);
```

```
begin
   --Decodificador de Estados
   case PRE is
        when t0 => T:=x"01";
        when t1 => T:=x"02";
        when t2 => T:=x"04";
        when t3 => T:=x"08";
        when t4 => T:=x"10";
        when t5 => T:=x"20";
        when t6 => T:=x"40";
        when t7 => T:=x"80";
   end case;
   --Decoder de Salida
   CLR_CNT_BYTE<=T(0);
   CARGAR<=T(1);
   SHIFT_CP<=(T(1) or T(4)) and not CLK_Tx;
   SEND<=T(3);
   CP_CNT_BYTE<=SEND;

   --Resets del sistema
   CLK_RST<=not noRST;
   noCLK_RDY<=not CLK_RDY;
   RST<=CLK_RST or noCLK_RDY;

   --Puerto Serial
   PS1_DSR<=GND;
   PS1_CTS<=PS1_RTS;
   PS1_RXD<=RXD1;

   --Señales DES
   PT<=x"0123456789ABCDEF";
   KEY<=x"133457799BBCDFF1";

   --Despliegue Visual
   LED0<=noCLK_RDY; --RELOJ LISTO

   case SEL is
        when '0' => TEXTOaMOSTRAR<=RT;
        when others => TEXTOaMOSTRAR<=CT;
   end case;
end process;

ListaDeLlaves: KS Port map(KEY,K);
Cifrador: DES_CORE Port map(PT,K,CT);
REGISTRO: SHIFT_REG8x8 Port map(SHIFT_CP,x"ff",OCTETO_Tx,CARGAR,
                                TEXTOaMOSTRAR,open,VCC,VCC,GND);

Kinv(16)<=K( 1); Kinv(15)<=K( 2); Kinv(14)<=K( 3); Kinv(13)<=K( 4);
Kinv(12)<=K( 5); Kinv(11)<=K( 6); Kinv(10)<=K( 7); Kinv( 9)<=K( 8);
Kinv( 8)<=K( 9); Kinv( 7)<=K(10); Kinv( 6)<=K(11); Kinv( 5)<=K(12);
Kinv( 4)<=K(13); Kinv( 3)<=K(14); Kinv( 2)<=K(15); Kinv( 1)<=K(16);

Decifrador: DES_CORE Port map(CT,Kinv,RT);

TX: TSA Port map(OCTETO_Tx,SEND,RXD1,Tx_BUSY,CLK_Tx,RST);
CONTADOR_BYTES: CONTADOR4 Port map(CNT_BYTE_Tx,CP_CNT_BYTE,
                                   VCC,CLR_CNT_BYTE);

end COM_DES;
```

A.5 COM_DES.UCF

```
#DESIGNACION DE PINES
#DISPOSITIVO: XUPV2P (VIRTEX 2 PRO)

#************************** SEÑALES DE CONTROL **************************

#SELECTOR DE MODO DE OPERACION (input)
NET "SEL" LOC = "AC11";           #Switch SW0 (.L)
NET "SEL" IOSTANDARD = LVCMOS25; #Nivel lógico

#RESET (input)
NET "noRST" LOC = "AG5";          #Enter (.L)
NET "noRST" IOSTANDARD = LVTTL;  #Nivel lógico

#RELOJ (input)
NET "CLK" LOC = "AJ15";           #RELOJ 100MHz
NET "CLK" IOSTANDARD = LVCMOS25; #Nivel lógico
NET "CLK" TNM_NET = "CLK";        #Asignación
TIMESPEC "TS_CLK" = PERIOD "CLK" 10 ns HIGH 50 % INPUT_JITTER 10 ns;

#************************** PUERTO SERIAL 1 (DCE) **************************

#NOTA: En la tarjeta TXD y RXD están cruzados, pero CTS y RTS no.

#UBICACION
NET "PS1_DSR" LOC = "AD10"; #Data Set Ready    (O)
NET "PS1_RTS" LOC = "AK8";  #Request To Send   (I)
NET "PS1_CTS" LOC = "AE8";  #Clear To Send     (O)
NET "PS1_TXD" LOC = "AJ8";  #Transmitted Data  (I)
NET "PS1_RXD" LOC = "AE7";  #Received Data     (O)

#Corriente manejada [mA]
NET "PS1_RXD" DRIVE = 8;
NET "PS1_CTS" DRIVE = 8;
NET "PS1_DSR" DRIVE = 8;

#Velocidad de cambio
NET "PS1_CTS" SLEW = SLOW;
NET "PS1_RXD" SLEW = SLOW;
NET "PS1_DSR" SLEW = SLOW;

##Nivel lógico
NET "PS1*" IOSTANDARD = LVCMOS25;

#************************** LEDS **************************
#NOTA: Ànodo Común

NET "LED0" LOC = "AC4";
NET "LED*" IOSTANDARD = LVTTL;
NET "LED*" DRIVE = 12;
NET "LED*" SLEW = SLOW;
```

A.6 DES_CORE.VHDL

```
------SISTEMA DES FIPS46-3------

--DEPENDENCIAS
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use WORK.TIPOS.ALL;

--DECLARACION DE LA ENTIDAD DES_CORE
entity DES_CORE is
   Port(IB : in  bloque;
        K  : in  llave_efectiva_vector(1 to 16);
        OB : out bloque);
end DES_CORE;

--ARQUITECTURA DE LA ENTIDAD DES_CORE
architecture DES_CORE of DES_CORE is
   --Componentes
   component IP
        Port(IB : in  bloque;
             L0 : out medio_bloque;
             R0 : out medio_bloque);
   end component;

   component FEISTEL
        Port(L0  : in  medio_bloque;
             R0  : in  medio_bloque;
             K   : in  llave_efectiva_vector(1 to 16);
             L16 : out medio_bloque;
             R16 : out medio_bloque);
   end component;

   component FP
        Port(L16 : in  medio_bloque;
             R16 : in  medio_bloque;
             OB  : out bloque);
   end component;

   --Conectores
   signal L0,R0,L16,R16 : medio_bloque;
begin

PermutacionInicial: IP Port map( IB, L0,R0);
RedDeFeistel: FEISTEL Port map(L0,R0,K,L16,R16);
PermutacionFinal: FP Port map(L16,R16,OB);

end DES_CORE;
```

A.7 FEISTEL.VHDL

```
------RED DE FEISTEL DE 16 RONDAS------

--DEPENDENCIAS
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use WORK.TIPOS.ALL;

--DECLARACION DE LA ENTIDAD FEISTEL
entity FEISTEL is
   Port(L0  : in  medio_bloque; --Left inicial
        R0  : in  medio_bloque; --Right inicial
        K   : in  llave_efectiva_vector(1 to 16); --Lista de llaves
        L16 : out medio_bloque; --Left 16
        R16 : out medio_bloque); --Right 16
end FEISTEL;

--ARQUITECTURA DE LA ENTIDAD FEISTEL
architecture FEISTEL of FEISTEL is
   --Componentes

   component F
        Port(Rn  : in  medio_bloque;
             Kn  : in  llave_efectiva;
             F   : out medio_bloque);
   end component;

   --Conectores
   signal L,R : medio_bloque_vector(1 to 15);
   signal FUNC : medio_bloque_vector(1 to 16);

begin
Rd1:  F Port map(  R0 ,K( 1),FUNC( 1));
   L( 1)<=R0;     R( 1)<=  L0  xor FUNC( 1);
Rd2:  F Port map(R( 1),K( 2),FUNC( 2));
   L( 2)<=R( 1);  R( 2)<=L( 1) xor FUNC( 2);
Rd3:  F Port map(R( 2),K( 3),FUNC( 3));
   L( 3)<=R( 2);  R( 3)<=L( 2) xor FUNC( 3);
Rd4:  F Port map(R( 3),K( 4),FUNC( 4));
   L( 4)<=R( 3);  R( 4)<=L( 3) xor FUNC( 4);
Rd5:  F Port map(R( 4),K( 5),FUNC( 5));
   L( 5)<=R( 4);  R( 5)<=L( 4) xor FUNC( 5);
Rd6:  F Port map(R( 5),K( 6),FUNC( 6));
   L( 6)<=R( 5);  R( 6)<=L( 5) xor FUNC( 6);
Rd7:  F Port map(R( 6),K( 7),FUNC( 7));
   L( 7)<=R( 6);  R( 7)<=L( 6) xor FUNC( 7);
Rd8:  F Port map(R( 7),K( 8),FUNC( 8));
   L( 8)<=R( 7);  R( 8)<=L( 7) xor FUNC( 8);
Rd9:  F Port map(R( 8),K( 9),FUNC( 9));
   L( 9)<=R( 8);  R( 9)<=L( 8) xor FUNC( 9);
Rd10: F Port map(R( 9),K(10),FUNC(10));
   L(10)<=R( 9);  R(10)<=L( 9) xor FUNC(10);
Rd11: F Port map(R(10),K(11),FUNC(11));
   L(11)<=R(10);  R(11)<=L(10) xor FUNC(11);
Rd12: F Port map(R(11),K(12),FUNC(12));
   L(12)<=R(11);  R(12)<=L(11) xor FUNC(12);
Rd13: F Port map(R(12),K(13),FUNC(13));
```

```
   L(13)<=R(12);  R(13)<=L(12) xor FUNC(13);
Rd14: F Port map(R(13),K(14),FUNC(14));
   L(14)<=R(13);  R(14)<=L(13) xor FUNC(14);
Rd15: F Port map(R(14),K(15),FUNC(15));
   L(15)<=R(14);  R(15)<=L(14) xor FUNC(15);
Rd16: F Port map(R(15),K(16),FUNC(16));
    L16 <=R(15);   R16 <=L(15) xor FUNC(16);
end FEISTEL;
```

A.8 F.VHDL

```
------FUNCION F------

--DEPENDENCIAS
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use WORK.TIPOS.ALL;

--DECLARACION DE LA ENTIDAD F
entity F is
   Port(Rn : in  medio_bloque;
        Kn : in  llave_efectiva;
        F  : out medio_bloque);
end F;

--ARQUITECTURA DE LA ENTIDAD F
architecture F of F is
   --Componentes

   component E
        Port(R : in  medio_bloque;
             E : out medio_bloque_expandido);
   end component;
   component S
        Port(B   : in  medio_bloque_expandido;
             S   : out medio_bloque);
   end component;

   component P
        Port(S : in  medio_bloque;
             P : out medio_bloque);
   end component;

   --Conectores
   signal EXP,B : medio_bloque_expandido;
   signal SUST: medio_bloque;

begin

Expansion: E Port map(Rn,EXP);
   B<=EXP xor Kn;
Sustitucion: S Port map(B,SUST);
Permutacion: P Port map(SUST,F);

end F;
```

A.9 E.VHDL

```
------PERMUTACION DE EXPANSION------
--DEPENDENCIAS
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use WORK.TIPOS.ALL;

--DECLARACION DE LA ENTIDAD E
   entity E is
        Port(R : in  medio_bloque; --Right
             E : out medio_bloque_expandido); --Expansion
   end E;

--ARQUITECTURA DE LA ENTIDAD E
architecture E of E is
begin
   E( 1)<=R(32); E( 2)<=R( 1); E( 3)<=R( 2); E( 4)<=R( 3); E( 5)<=R( 4);
          E(6)<=R( 5);
   E( 7)<=R( 4); E( 8)<=R( 5); E( 9)<=R( 6); E(10)<=R( 7); E(11)<=R( 8);
          E(12)<=R( 9);
   E(13)<=R( 8); E(14)<=R( 9); E(15)<=R(10); E(16)<=R(11); E(17)<=R(12);
          E(18)<=R(13);
   E(19)<=R(12); E(20)<=R(13); E(21)<=R(14); E(22)<=R(15); E(23)<=R(16);
          E(24)<=R(17);
   E(25)<=R(16); E(26)<=R(17); E(27)<=R(18); E(28)<=R(19); E(29)<=R(20);
          E(30)<=R(21);
   E(31)<=R(20); E(32)<=R(21); E(33)<=R(22); E(34)<=R(23); E(35)<=R(24);
          E(36)<=R(25);
   E(37)<=R(24); E(38)<=R(25); E(39)<=R(26); E(40)<=R(27); E(41)<=R(28);
          E(42)<=R(29);
   E(43)<=R(28); E(44)<=R(29); E(45)<=R(30); E(46)<=R(31); E(47)<=R(32);
          E(48)<=R( 1);
end E;
```

A10 S.VHDL

```
------FUNCION DE SUSTITUCION------

--DEPENDENCIAS
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use WORK.TIPOS.ALL;
use WORK.CAJAS_S.ALL;

--DECLARACION DE LA ENTIDAD S
   entity S is
        Port(B  : in  medio_bloque_expandido;
             S  : out medio_bloque);
   end S;

--ARQUITECTURA DE LA ENTIDAD S

architecture S of S is

begin

CAJA_S1: S1 Port map(B( 1 to  6),S( 1 to  4));
CAJA_S2: S2 Port map(B( 7 to 12),S( 5 to  8));
CAJA_S3: S3 Port map(B(13 to 18),S( 9 to 12));
CAJA_S4: S4 Port map(B(19 to 24),S(13 to 16));
CAJA_S5: S5 Port map(B(25 to 30),S(17 to 20));
CAJA_S6: S6 Port map(B(31 to 36),S(21 to 24));
CAJA_S7: S7 Port map(B(37 to 42),S(25 to 28));
CAJA_S8: S8 Port map(B(43 to 48),S(29 to 32));

end S;
```

A11 S1.VHDL

```
------CAJA S1------

--DEPENDENCIAS
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use WORK.PRIMITIVOS.ALL;
use WORK.TIPOS.ALL;
Library UNISIM;
use UNISIM.vcomponents.ALL;

--DECLARACION DE LA ENTIDAD S1
entity S1 is
   Port(B1  : in  nybble_expandido;
        S1  : out nybble);
end S1;

--ARQUITECTURA DE LA ENTIDAD S1
architecture S1 of S1 is

   --Conectores
   signal A,B,C,D,AB,CD : nybble;

begin
S1A: ROM16x4 generic map(x"2AE5",x"9C27",x"8771",x"B16C")
             Port map(B1(2 to 5),A);
S1B: ROM16x4 generic map(x"9D52",x"265E",x"4B36",x"78C6")
             Port map(B1(2 to 5),B);
S1C: ROM16x4 generic map(x"279C",x"4B35",x"39E4",x"5D92")
             Port map(B1(2 to 5),C);
S1D: ROM16x4 generic map(x"9A27",x"C993",x"5E89",x"87E1")
             Port map(B1(2 to 5),D);
MUX_AB: MUX2a1 Port map(AB,A,B,B1(6));
MUX_CD: MUX2a1 Port map(CD,C,D,B1(6));
MUX_ABCD: MUX2a1 Port map(S1,AB,CD,B1(1));
end S1;

--CONFIGURACION DE LA ENTIDAD S1

configuration CONFIG_S1 of S1 is
   for S1
        for S1A,S1B,S1C,S1D: ROM16x4
             use entity WORK.ROM16x4(ROM16x4);
        end for;
        for MUX_AB,MUX_CD,MUX_ABCD: MUX2a1
             use entity WORK.MUX2a1(MUX2a1);
        end for;
   end for;
end CONFIG_S1;
```

A12 S2.VHDL

```
------CAJA S2------

--DEPENDENCIAS

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use WORK.PRIMITIVOS.ALL;
use WORK.TIPOS.ALL;
Library UNISIM;
use UNISIM.vcomponents.ALL;

--DECLARACION DE LA ENTIDAD S2
entity S2 is
   Port(B2  : in  nybble_expandido;
        S2  : out nybble);
end S2;

--ARQUITECTURA DE LA ENTIDAD S2
architecture S2 of S2 is
   --Conectores
   signal A,B,C,D,AB,CD : nybble;
begin
S2A: ROM16x4 generic map(x"992D",x"5A99",x"8679",x"4B63")
             Port map(B2(2 to 5),A);
S2B: ROM16x4 generic map(x"69D2",x"919E",x"58B9",x"E41B")
             Port map(B2(2 to 5),B);
S2C: ROM16x4 generic map(x"965A",x"8D66",x"E81E",x"B2CC")
             Port map(B2(2 to 5),C);
S2D: ROM16x4 generic map(x"C927",x"6E61",x"47B4",x"A539")
             Port map(B2(2 to 5),D);

MUX_AB: MUX2a1 Port map(AB,A,B,B2(6));
MUX_CD: MUX2a1 Port map(CD,C,D,B2(6));
MUX_ABCD: MUX2a1 Port map(S2,AB,CD,B2(1));

end S2;
```

A13 S3.VHDL

```
------CAJA S3------

--DEPENDENCIAS

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use WORK.PRIMITIVOS.ALL;
use WORK.TIPOS.ALL;
Library UNISIM;
use UNISIM.vcomponents.ALL;

--DECLARACION DE LA ENTIDAD S3
entity S3 is
   Port(B3  : in  nybble_expandido;
        S3  : out nybble);
   end S3;

--ARQUITECTURA DE LA ENTIDAD S3
architecture S3 of S3 is

   --Conectores
   signal A,B,C,D,AB,CD : nybble;

begin
S3A: ROM16x4 generic map(x"964D",x"2ED8",x"5879",x"1BE4")
             Port map(B3(2 to 5),A);
S3B: ROM16x4 generic map(x"7A89",x"5C63",x"69D2",x"E41B")
             Port map(B3(2 to 5),B);
S3C: ROM16x4 generic map(x"6939",x"D827",x"E562",x"9369")
             Port map(B3(2 to 5),C);
S3D: ROM16x4 generic map(x"9666",x"A794",x"5E92",x"3AA5")
             Port map(B3(2 to 5),D);

MUX_AB: MUX2a1 Port map(AB,A,B,B3(6));
MUX_CD: MUX2a1 Port map(CD,C,D,B3(6));
MUX_ABCD: MUX2a1 Port map(S3,AB,CD,B3(1));

end S3;

--CONFIGURACION DE LA ENTIDAD S3
configuration CONFIG_S3 of S3 is
   for S3
        for S3A,S3B,S3C,S3D: ROM16x4
             use entity WORK.ROM16x4(ROM16x4);
        end for;
        for MUX_AB,MUX_CD,MUX_ABCD: MUX2a1
             use entity WORK.MUX2a1(MUX2a1);
        end for;
   end for;
end CONFIG_S3;
```

A14 S4.VHDL

```
-----CAJA S4------

--DEPENDENCIAS

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use WORK.PRIMITIVOS.ALL;
use WORK.TIPOS.ALL;
Library UNISIM;
use UNISIM.vcomponents.ALL;

--DECLARACION DE LA ENTIDAD S4
entity S4 is
   Port(B4  : in  nybble_expandido;
        S4  : out nybble);
   end S4;

--ARQUITECTURA DE LA ENTIDAD S4
architecture S4 of S4 is

   --Conectores
   signal A,B,C,D,AB,CD : nybble;
begin
S4A: ROM16x4 generic map(x"B4C6",x"E827",x"92AD",x"994B")
             Port map(B4(2 to 5),A);
S4B: ROM16x4 generic map(x"E827",x"4B39",x"66B4",x"92AD")
             Port map(B4(2 to 5),B);
S4C: ROM16x4 generic map(x"49B5",x"99D2",x"2D63",x"17E4")
             Port map(B4(2 to 5),C);
S4D: ROM16x4 generic map(x"99D2",x"B64A",x"E81B",x"2D63")
             Port map(B4(2 to 5),D);

MUX_AB: MUX2a1 Port map(AB,A,B,B4(6));
MUX_CD: MUX2a1 Port map(CD,C,D,B4(6));
MUX_ABCD: MUX2a1 Port map(S4,AB,CD,B4(1));

end S4;

--CONFIGURACION DE LA ENTIDAD S4
configuration CONFIG_S4 of S4 is
   for S4
        for S4A,S4B,S4C,S4D: ROM16x4
             use entity WORK.ROM16x4(ROM16x4);
        end for;
        for MUX_AB,MUX_CD,MUX_ABCD: MUX2a1
             use entity WORK.MUX2a1(MUX2a1);
        end for;
   end for;
end CONFIG_S4;
```

A15 S5.VHDL

```
------CAJA S5------

--DEPENDENCIAS

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use WORK.PRIMITIVOS.ALL;
use WORK.TIPOS.ALL;
Library UNISIM;
use UNISIM.vcomponents.ALL;

--DECLARACION DE LA ENTIDAD S5
entity S5 is
   Port(B5  : in  nybble_expandido;
        S5  : out nybble);
   End S5;

--ARQUITECTURA DE LA ENTIDAD S5
architecture S5 of S5 is

   --Conectores
   signal A,B,C,D,AB,CD : nybble;
begin
S5A: ROM16x4 generic map(x"D962",x"5A96",x"4CF1",x"9E58")
             Port map(B5(2 to 5),A);
S5B: ROM16x4 generic map(x"6C4B",x"8579",x"9C27",x"35E2")
             Port map(B5(2 to 5),B);
S5C: ROM16x4 generic map(x"87B8",x"9D61",x"B15A",x"2B6C")
             Port map(B5(2 to 5),C);
S5D: ROM16x4 generic map(x"1AA7",x"63AC",x"9369",x"CA99")
             Port map(B5(2 to 5),D);

MUX_AB: MUX2a1 Port map(AB,A,B,B5(6));
MUX_CD: MUX2a1 Port map(CD,C,D,B5(6));
MUX_ABCD: MUX2a1 Port map(S5,AB,CD,B5(1));

end S5;

--CONFIGURACION DE LA ENTIDAD S5
configuration CONFIG_S5 of S5 is
   for S5
        for S5A,S5B,S5C,S5D: ROM16x4
             use entity WORK.ROM16x4(ROM16x4);
        end for;
        for MUX_AB,MUX_CD,MUX_ABCD: MUX2a1
             use entity WORK.MUX2a1(MUX2a1);
        end for;
   end for;
end CONFIG_S5;
```

A16 S6.VHDL

```
-----CAJA S6------

--DEPENDENCIAS

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use WORK.PRIMITIVOS.ALL;
use WORK.TIPOS.ALL;
Library UNISIM;
use UNISIM.vcomponents.ALL;

--DECLARACION DE LA ENTIDAD S6
entity S6 is
   Port(B6  : in  nybble_expandido;
        S6  : out nybble);
   end S6;

--ARQUITECTURA DE LA ENTIDAD S6
architecture S6 of S6 is

   --Conectores
   signal A,B,C,D,AB,CD : nybble;
begin
S6A: ROM16x4 generic map(x"929D",x"7A49",x"B46C",x"E61A")
             Port map(B6(2 to 5),A);
S6B: ROM16x4 generic map(x"AC63",x"0DB6",x"691B",x"66D2")
             Port map(B6(2 to 5),B);
S6C: ROM16x4 generic map(x"6867",x"A54E",x"C996",x"718D")
             Port map(B6(2 to 5),C);
S6D: ROM16x4 generic map(x"C3D8",x"9A69",x"1BC6",x"8D72")
             Port map(B6(2 to 5),D);

MUX_AB: MUX2a1 Port map(AB,A,B,B6(6));
MUX_CD: MUX2a1 Port map(CD,C,D,B6(6));
MUX_ABCD: MUX2a1 Port map(S6,AB,CD,B6(1));

end S6;

--CONFIGURACION DE LA ENTIDAD S6
configuration CONFIG_S6 of S6 is
   for S6
        for S6A,S6B,S6C,S6D: ROM16x4
             use entity WORK.ROM16x4(ROM16x4);
        end for;
        for MUX_AB,MUX_CD,MUX_ABCD: MUX2a1
             use entity WORK.MUX2a1(MUX2a1);
        end for;
   end for;
end CONFIG_S6;
```

A17 S7.VHDL

```
------CAJA S7------

--DEPENDENCIAS

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use WORK.PRIMITIVOS.ALL;
use WORK.TIPOS.ALL;
Library UNISIM;
use UNISIM.vcomponents.ALL;

--DECLARACION DE LA ENTIDAD S7
entity S7 is
   Port(B7  : in  nybble_expandido;
        S7  : out nybble);
   end S7;

--ARQUITECTURA DE LA ENTIDAD S7
architecture S7 of S7 is
   --Conectores
   signal A,B,C,D,AB,CD : nybble;
begin
S7A: ROM16x4 generic map(x"26DA",x"5A99",x"691E",x"9D92")
             Port map(B7(2 to 5),A);
S7B: ROM16x4 generic map(x"69A5",x"AD19",x"B38C",x"266D")
             Port map(B7(2 to 5),B);
S7C: ROM16x4 generic map(x"4B9C",x"26DA",x"87E4",x"626D")
             Port map(B7(2 to 5),C);
S7D: ROM16x4 generic map(x"994E",x"9AA5",x"78C3",x"4B96")
             Port map(B7(2 to 5),D);

MUX_AB: MUX2a1 Port map(AB,A,B,B7(6));
MUX_CD: MUX2a1 Port map(CD,C,D,B7(6));
MUX_ABCD: MUX2a1 Port map(S7,AB,CD,B7(1));

end S7;

--CONFIGURACION DE LA ENTIDAD S7
configuration CONFIG_S7 of S7 is
   for S7
        for S7A,S7B,S7C,S7D: ROM16x4
             use entity WORK.ROM16x4(ROM16x4);
        end for;
        for MUX_AB,MUX_CD,MUX_ABCD: MUX2a1
             use entity WORK.MUX2a1(MUX2a1);
        end for;
   end for;
end CONFIG_S7;
```

A18 S8.VHDL

```
------CAJA S8------

--DEPENDENCIAS
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use WORK.PRIMITIVOS.ALL;
use WORK.TIPOS.ALL;
Library UNISIM;
use UNISIM.vcomponents.ALL;

--DECLARACION DE LA ENTIDAD S8
entity S8 is
   Port(B8  : in  nybble_expandido;
        S8  : out nybble);
   end S8;

--ARQUITECTURA DE LA ENTIDAD S8
architecture S8 of S8 is

   --Conectores
   signal A,B,C,D,AB,CD : nybble;
begin
S8A: ROM16x4 generic map(x"4B65",x"D839",x"8D72",x"96E1")
             Port map(B8(2 to 5),A);
S8B: ROM16x4 generic map(x"691E",x"27C6",x"AC72",x"4A67")
             Port map(B8(2 to 5),B);
S8C: ROM16x4 generic map(x"9C72",x"5A65",x"36C3",x"781B")
             Port map(B8(2 to 5),C);
S8D: ROM16x4 generic map(x"87E4",x"639C",x"D12D",x"B58A")
             Port map(B8(2 to 5),D);

MUX_AB: MUX2a1 Port map(AB,A,B,B8(6));
MUX_CD: MUX2a1 Port map(CD,C,D,B8(6));
MUX_ABCD: MUX2a1 Port map(S8,AB,CD,B8(1));

end S8;

--CONFIGURACION DE LA ENTIDAD S8
configuration CONFIG_S8 of S8 is
   for S8
        for S8A,S8B,S8C,S8D: ROM16x4
             use entity WORK.ROM16x4(ROM16x4);
        end for;
        for MUX_AB,MUX_CD,MUX_ABCD: MUX2a1
             use entity WORK.MUX2a1(MUX2a1);
        end for;
   end for;
end CONFIG_S8;
```

A19 MUX2A1.VHDL

```
------MUX2a1 DE 2 A 1 PARA 4 BITS------

--DEPENDENCIAS

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
Library UNISIM;
use UNISIM.vcomponents.ALL;

--DECLARACION DE LA ENTIDAD MUX2a1
   entity MUX2a1 is
        Port( O   : out std_logic_vector(3 downto 0);
              I0  : in  std_logic_vector(3 downto 0);
              I1  : in  std_logic_vector(3 downto 0);
              SEL : in  std_logic);
   end MUX2a1;

--ARQUITECTURA DE LA ENTIDAD MUX2a1
architecture MUX2a1 of MUX2a1 is

begin
bit3: MUXF5 Port map(O(3),I0(3),I1(3),SEL);
bit2: MUXF5 Port map(O(2),I0(2),I1(2),SEL);
bit1: MUXF5 Port map(O(1),I0(1),I1(1),SEL);
bit0: MUXF5 Port map(O(0),I0(0),I1(0),SEL);
end MUX2a1;
```

A20 ROM16X4.VHDL

```
------ROM ROM16x4------

--DEPENDENCIAS

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
Library UNISIM;
use UNISIM.vcomponents.ALL;
use WORK.TIPOS.ALL;

--DECLARACION DE LA ENTIDAD ROM16x4
   entity ROM16x4 is
   Generic(  INIT3 : bit_vector:=x"FEDC";
             INIT2 : bit_vector:=x"BA98";
             INIT1 : bit_vector:=x"7654";
             INIT0 : bit_vector:=x"3210");
         Port( DIR : in  std_logic_vector(3 downto 0);
               DAT : out std_logic_vector(3 downto 0));
   end ROM16x4;

--ARQUITECTURA DE LA ENTIDAD ROM16x4
architecture ROM16x4 of ROM16x4 is
begin
bit3: LUT4 generic map(INIT3) Port map(DAT(3),DIR(0),DIR(1),DIR(2),DIR(3));
bit2: LUT4 generic map(INIT2) Port map(DAT(2),DIR(0),DIR(1),DIR(2),DIR(3));
bit1: LUT4 generic map(INIT1) Port map(DAT(1),DIR(0),DIR(1),DIR(2),DIR(3));
bit0: LUT4 generic map(INIT0) Port map(DAT(0),DIR(0),DIR(1),DIR(2),DIR(3));
end ROM16x4;
```

A21 IP.VHDL

```
------PERMUTACION INICIAL------

--DEPENDENCIAS

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use WORK.TIPOS.ALL;

--DECLARACION DE LA ENTIDAD IP
   entity IP is
        Port(IB : in  bloque;        --Input Block
             L0 : out medio_bloque; --Left Inicial
             R0 : out medio_bloque);--Right Inicial
   end IP;

--ARQUITECTURA DE LA ENTIDAD IP
architecture IP of IP is
   signal OB : bloque;--Output Block
begin
   --Permutación inicial
   OB( 1)<=IB(58); OB( 2)<=IB(50); OB( 3)<=IB(42); OB( 4)<=IB(34);
   OB( 5)<=IB(26); OB( 6)<=IB(18); OB( 7)<=IB(10); OB( 8)<=IB( 2);
   OB( 9)<=IB(60); OB(10)<=IB(52); OB(11)<=IB(44); OB(12)<=IB(36);
   OB(13)<=IB(28); OB(14)<=IB(20); OB(15)<=IB(12); OB(16)<=IB( 4);
   OB(17)<=IB(62); OB(18)<=IB(54); OB(19)<=IB(46); OB(20)<=IB(38);
   OB(21)<=IB(30); OB(22)<=IB(22); OB(23)<=IB(14); OB(24)<=IB( 6);
   OB(25)<=IB(64); OB(26)<=IB(56); OB(27)<=IB(48); OB(28)<=IB(40);
   OB(29)<=IB(32); OB(30)<=IB(24); OB(31)<=IB(16); OB(32)<=IB( 8);
   OB(33)<=IB(57); OB(34)<=IB(49); OB(35)<=IB(41); OB(36)<=IB(33);
   OB(37)<=IB(25); OB(38)<=IB(17); OB(39)<=IB( 9); OB(40)<=IB( 1);
   OB(41)<=IB(59); OB(42)<=IB(51); OB(43)<=IB(43); OB(44)<=IB(35);
   OB(45)<=IB(27); OB(46)<=IB(19); OB(47)<=IB(11); OB(48)<=IB( 3);
   OB(49)<=IB(61); OB(50)<=IB(53); OB(51)<=IB(45); OB(52)<=IB(37);
   OB(53)<=IB(29); OB(54)<=IB(21); OB(55)<=IB(13); OB(56)<=IB( 5);
   OB(57)<=IB(63); OB(58)<=IB(55); OB(59)<=IB(47); OB(60)<=IB(39);
   OB(61)<=IB(31); OB(62)<=IB(23); OB(63)<=IB(15); OB(64)<=IB( 7);

   --Segmentación del bloque permutado
   L0<=OB(1 to 32); R0<=OB(33 to 64);

end IP;
```

A22 FP.VHDL

```
------PERMUTACION FINAL------

--DEPENDENCIAS

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use WORK.TIPOS.ALL;

--DECLARACION DE LA ENTIDAD FP
   entity FP is
        Port(L16 : in  medio_bloque;--Left 16
             R16 : in  medio_bloque;--Right 16
             OB  : out bloque); --Output Block
   end FP;

--ARQUITECTURA DE LA ENTIDAD FP
architecture FP of FP is
   signal IB : bloque; --Intermediate Block
begin
   --Cruce de 32 bits
   IB<=(R16 & L16);
   --Permutación final (inversa de la función inicial)
   OB( 1)<=IB(40); OB( 2)<=IB( 8); OB( 3)<=IB(48); OB( 4)<=IB(16);
   OB( 5)<=IB(56); OB( 6)<=IB(24); OB( 7)<=IB(64); OB( 8)<=IB(32);
   OB( 9)<=IB(39); OB(10)<=IB( 7); OB(11)<=IB(47); OB(12)<=IB(15);
   OB(13)<=IB(55); OB(14)<=IB(23); OB(15)<=IB(63); OB(16)<=IB(31);
   OB(17)<=IB(38); OB(18)<=IB( 6); OB(19)<=IB(46); OB(20)<=IB(14);
   OB(21)<=IB(54); OB(22)<=IB(22); OB(23)<=IB(62); OB(24)<=IB(30);
   OB(25)<=IB(37); OB(26)<=IB( 5); OB(27)<=IB(45); OB(28)<=IB(13);
   OB(29)<=IB(53); OB(30)<=IB(21); OB(31)<=IB(61); OB(32)<=IB(29);
   OB(33)<=IB(36); OB(34)<=IB( 4); OB(35)<=IB(44); OB(36)<=IB(12);
   OB(37)<=IB(52); OB(38)<=IB(20); OB(39)<=IB(60); OB(40)<=IB(28);
   OB(41)<=IB(35); OB(42)<=IB( 3); OB(43)<=IB(43); OB(44)<=IB(11);
   OB(45)<=IB(51); OB(46)<=IB(19); OB(47)<=IB(59); OB(48)<=IB(27);
   OB(49)<=IB(34); OB(50)<=IB( 2); OB(51)<=IB(42); OB(52)<=IB(10);
   OB(53)<=IB(50); OB(54)<=IB(18); OB(55)<=IB(58); OB(56)<=IB(26);
   OB(57)<=IB(33); OB(58)<=IB( 1); OB(59)<=IB(41); OB(60)<=IB( 9);
   OB(61)<=IB(49); OB(62)<=IB(17); OB(63)<=IB(57); OB(64)<=IB(25);

end FP;
```

A23 KS.VHDL

```
------LLAVE EFECTIVA------

--DEPENDENCIAS

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use WORK.TIPOS.ALL;
use WORK.PRIMITIVOS.ALL;

--DECLARACION DE LA ENTIDAD KS
   entity KS is
        Port(KEY : in  bloque;
             K   : out llave_efectiva_vector(1 to 16));
   end KS;

--ARQUITECTURA DE LA ENTIDAD KS
architecture KS of KS is
   --Componentes
   component PC1
        Port(KEY : in  bloque;
             C0  : out media_llave;
             D0  : out media_llave);
   end component;

   component PC2
        Port(Cn : in  media_llave;
             Dn : in  media_llave;
             Kn : out llave_efectiva);
   end component;

   --Conectores
   signal C,D : media_llave_vector(0 to 16);
begin
PCH: PC1 Port map( KEY,C( 0),D( 0));
   C( 1)<=C( 0) rol 1;           D( 1)<=D( 0) rol 1;
K1:  PC2 Port map(C( 1),D( 1),K( 1));
   C( 2)<=C( 1) rol 1;           D( 2)<=D( 1) rol 1;
K2:  PC2 Port map(C( 2),D( 2),K( 2));
   C( 3)<=C( 2) rol 2;           D( 3)<=D( 2) rol 2;
K3:  PC2 Port map(C( 3),D( 3),K( 3));
   C( 4)<=C( 3) rol 2;           D( 4)<=D( 3) rol 2;
K4:  PC2 Port map(C( 4),D( 4),K( 4));
   C( 5)<=C( 4) rol 2;           D( 5)<=D( 4) rol 2;
K5:  PC2 Port map(C( 5),D( 5),K( 5));
   C( 6)<=C( 5) rol 2;           D( 6)<=D( 5) rol 2;
K6:  PC2 Port map(C( 6),D( 6),K( 6));
   C( 7)<=C( 6) rol 2;           D( 7)<=D( 6) rol 2;
K7:  PC2 Port map(C( 7),D( 7),K( 7));
   C( 8)<=C( 7) rol 2;           D( 8)<=D( 7) rol 2;
K8:  PC2 Port map(C( 8),D( 8),K( 8));
   C( 9)<=C( 8) rol 1;           D( 9)<=D( 8) rol 1;
K9:  PC2 Port map(C( 9),D( 9),K( 9));
   C(10)<=C( 9) rol 2;           D(10)<=D( 9) rol 2;
K10: PC2 Port map(C(10),D(10),K(10));
   C(11)<=C(10) rol 2;           D(11)<=D(10) rol 2;
```

```
K11: PC2 Port map(C(11),D(11),K(11));
   C(12)<=C(11) rol 2;           D(12)<=D(11) rol 2;
K12: PC2 Port map(C(12),D(12),K(12));
   C(13)<=C(12) rol 2;           D(13)<=D(12) rol 2;
K13: PC2 Port map(C(13),D(13),K(13));
   C(14)<=C(13) rol 2;           D(14)<=D(13) rol 2;
K14: PC2 Port map(C(14),D(14),K(14));
   C(15)<=C(14) rol 2;           D(15)<=D(14) rol 2;
K15: PC2 Port map(C(15),D(15),K(15));
   C(16)<=C(15) rol 1;           D(16)<=D(15) rol 1;
K16: PC2 Port map(C(16),D(16),K(16));
end KS;
```

A24 PC1.VHDL

```
------PRIMERA SELECCION PERMUTADA------

--DEPENDENCIAS

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use WORK.TIPOS.ALL;

--DECLARACION DE LA ENTIDAD PC1

   entity PC1 is
        Port(KEY : in  bloque; --llave
             C0  : out media_llave;
             D0  : out media_llave);
   end PC1;

--ARQUITECTURA DE LA ENTIDAD PC1

architecture PC1 of PC1 is
   signal K : llave;

begin
   K( 1)<=KEY(57); K( 2)<=KEY(49); K( 3)<=KEY(41); K( 4)<=KEY(33);
   K( 5)<=KEY(25); K( 6)<=KEY(17); K( 7)<=KEY( 9); K( 8)<=KEY( 1);
   K( 9)<=KEY(58); K(10)<=KEY(50); K(11)<=KEY(42); K(12)<=KEY(34);
   K(13)<=KEY(26); K(14)<=KEY(18); K(15)<=KEY(10); K(16)<=KEY( 2);
   K(17)<=KEY(59); K(18)<=KEY(51); K(19)<=KEY(43); K(20)<=KEY(35);
   K(21)<=KEY(27); K(22)<=KEY(19); K(23)<=KEY(11); K(24)<=KEY( 3);
   K(25)<=KEY(60); K(26)<=KEY(52); K(27)<=KEY(44); K(28)<=KEY(36);
   K(29)<=KEY(63); K(30)<=KEY(55); K(31)<=KEY(47); K(32)<=KEY(39);
   K(33)<=KEY(31); K(34)<=KEY(23); K(35)<=KEY(15); K(36)<=KEY( 7);
   K(37)<=KEY(62); K(38)<=KEY(54); K(39)<=KEY(46); K(40)<=KEY(38);
   K(41)<=KEY(30); K(42)<=KEY(22); K(43)<=KEY(14); K(44)<=KEY( 6);
   K(45)<=KEY(61); K(46)<=KEY(53); K(47)<=KEY(45); K(48)<=KEY(37);
   K(49)<=KEY(29); K(50)<=KEY(21); K(51)<=KEY(13); K(52)<=KEY( 5);
   K(53)<=KEY(28); K(54)<=KEY(20); K(55)<=KEY(12); K(56)<=KEY( 4);

   C0<=K(1 to 28); D0<=K(29 to 56);

end PC1;
```

A25 PC2.VHDL

```
------SEGUNDA SELECCION PERMUTADA------

--DEPENDENCIAS

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use WORK.TIPOS.ALL;

--DECLARACION DE LA ENTIDAD PC2
   entity PC2 is
        Port(Cn : in  media_llave;--C
             Dn : in  media_llave;--D
             Kn : out llave_efectiva);
   end PC2;

--ARQUITECTURA DE LA ENTIDAD PC2

architecture PC2 of PC2 is
   signal K : llave;

begin
   K<=(Cn & Dn);
   --Segunda selección permutada
   Kn( 1)<=K(14); Kn( 2)<=K(17); Kn( 3)<=K(11); Kn( 4)<=K(24);
   Kn( 5)<=K( 1); Kn( 6)<=K( 5); Kn( 7)<=K( 3); Kn( 8)<=K(28);
   Kn( 9)<=K(15); Kn(10)<=K( 6); Kn(11)<=K(21); Kn(12)<=K(10);
   Kn(13)<=K(23); Kn(14)<=K(19); Kn(15)<=K(12); Kn(16)<=K( 4);
   Kn(17)<=K(26); Kn(18)<=K( 8); Kn(19)<=K(16); Kn(20)<=K( 7);
   Kn(21)<=K(27); Kn(22)<=K(20); Kn(23)<=K(13); Kn(24)<=K( 2);
   Kn(25)<=K(41); Kn(26)<=K(52); Kn(27)<=K(31); Kn(28)<=K(37);
   Kn(29)<=K(47); Kn(30)<=K(55); Kn(31)<=K(30); Kn(32)<=K(40);
   Kn(33)<=K(51); Kn(34)<=K(45); Kn(35)<=K(33); Kn(36)<=K(48);
   Kn(37)<=K(44); Kn(38)<=K(49); Kn(39)<=K(39); Kn(40)<=K(56);
   Kn(41)<=K(34); Kn(42)<=K(53); Kn(43)<=K(46); Kn(44)<=K(42);
   Kn(45)<=K(50); Kn(46)<=K(36); Kn(47)<=K(29); Kn(48)<=K(32);

end PC2;
```

A26 RELOJ.VHDL

```
------SISTEMA DE RELOJ-----

--DEPENDENCIAS

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

--      UNISIM
library UNISIM;
use UNISIM.VComponents.ALL;
--      WORK
use WORK.PRIMITIVOS.ALL;

--ENTIDAD RELOJ
   entity RELOJ is
        Port(CLK_580ns    : out std_logic; --Reloj de Recepción
             CLK_115200Hz : out std_logic; --Reloj de Transmisión
             CLK_RDY      : out std_logic; --Reloj Listo
             RST          : in  std_logic; --Restauración
             CLK          : in  std_logic); --Reloj de entrada [f=100MHz]
end RELOJ;

--ARQUITECTURA DE LA ENTIDAD RSA

architecture RELOJ of RELOJ is
   --Componentes

   component INV
      Port ( I : in    std_logic;
             O : out   std_logic);
   end component;

   component BUFG
      Port ( I : in    std_logic;
             O : out   std_logic);
   end component;

   component IBUFG
      Port ( I : in    std_logic;
             O : out   std_logic);
   end component;

   component DCM
      Generic( CLK_FEEDBACK : string :=  "1X";
               CLKDV_DIVIDE : real :=  2.000000;
               CLKFX_DIVIDE : integer :=  1;
               CLKFX_MULTIPLY : integer :=  4;
               CLKIN_DIVIDE_BY_2 : boolean :=  FALSE;
               CLKIN_PERIOD : real :=  0.000000;
               CLKOUT_PHASE_SHIFT : string :=  "NONE";
               DESKEW_ADJUST : string :=  "SYSTEM_SYNCHRONOUS";
               DFS_FREQUENCY_MODE : string :=  "LOW";
               DLL_FREQUENCY_MODE : string :=  "LOW";
```

```
                DUTY_CYCLE_CORRECTION : boolean :=  TRUE;
                FACTORY_JF : bit_vector :=  x"C080";
                PHASE_SHIFT : integer :=  0;
                STARTUP_WAIT : boolean :=  FALSE;
                DSS_MODE : string :=  "NONE");
        Port ( CLKIN    : in    std_logic;
               CLKFB    : in    std_logic;
               RST      : in    std_logic;
               PSEN     : in    std_logic;
               PSINCDEC : in    std_logic;
               PSCLK    : in    std_logic;
               DSSEN    : in    std_logic;
               CLK0     : out   std_logic;
               CLK90    : out   std_logic;
               CLK180   : out   std_logic;
               CLK270   : out   std_logic;
               CLKDV    : out   std_logic;
               CLK2X    : out   std_logic;
               CLK2X180 : out   std_logic;
               CLKFX    : out   std_logic;
               CLKFX180 : out   std_logic;
               STATUS   : out   std_logic_vector (7 downto 0);
               LOCKED   : out   std_logic;
               PSDONE   : out   std_logic);
    end component;

    --Señales
    signal GND : std_logic;
    signal CE_CNT,RST_CNT : std_logic;
    signal CNT1_NUM,CNT2_NUM : std_logic_vector(3 downto 0);
    signal noCLK_MATRIZ,noCLK_122_88MHz : std_logic;
    signal CLK_MATRIZ,CLK_122_88MHz : std_logic;
    signal DCM1_CLKin,DCM1_CLK0,DCM1_CLKfb,DCM1_CLKout,DCM1_LOCKED,DCM1_RST
           : std_logic;
    signal DCM2_CLKin,DCM2_CLK0,DCM2_CLKfb,DCM2_CLKout,DCM2_LOCKED,DCM2_RST
           : std_logic;
    signal DCM3_CLKin,DCM3_CLK0,DCM3_CLKfb,DCM3_CLKout,DCM3_LOCKED,DCM3_RST
           : std_logic;
    signal ESC_RST : std_logic;

begin
    GND<='0';
    ----------------- DCM1 -----------------

DCM1_RST<= RST;
DCM1_CLKin_BUF: IBUFG Port map(CLK,DCM1_CLKin); --CLK0: f=100MHz
DCM1_CLKfb_BUF: BUFG Port map (DCM1_CLK0,DCM1_CLKfb);
DCM1: DCM
    Generic map("1X",2.000000,25,32,FALSE,10.000000,"NONE",
                "SYSTEM_SYNCHRONOUS"LOW","LOW",TRUE,x"C080",0,TRUE,"NONE")
    Port map(DCM1_CLKin,DCM1_CLKfb,DCM1_RST,GND,GND,GND,GND,DCM1_CLK0,open,
            open,open,open,open,open,DCM1_CLKout,open,open,DCM1_LOCKED,open);
    ----------------- DCM2 -----------------
DCM2_RST_IN: INV Port map (DCM1_LOCKED,DCM2_RST);
DCM2_CLKin_BUF: BUFG Port map (DCM1_CLKout,DCM2_CLKin); --CLK1: f=128MHz
DCM2_CLKfb_BUF: BUFG Port map (DCM2_CLK0,DCM2_CLKfb);
DCM2: DCM
    generic map("1X",2.000000,25,24,FALSE, 7.812500,"NONE",
                "SYSTEM_SYNCHRONOUS","LOW","LOW",TRUE,x"C080",0,TRUE,"NONE")
    Port map(DCM2_CLKin,DCM2_CLKfb,DCM2_RST,GND,GND,GND,GND,DCM2_CLK0,open,
```

```
            open,open,open,open,open,DCM2_CLKout,open,open,DCM2_LOCKED,open);
   ----------------- DCM3 -----------------
DCM3_RST_IN: INV Port map (DCM2_LOCKED,DCM3_RST);
DCM3_CLKin_BUF: BUFG Port map (DCM2_CLKout,DCM3_CLKin); --CLK2: f=122.88MHz
DCM3_CLKfb_BUF: BUFG Port map (DCM3_CLK0,DCM3_CLKfb);
DCM3: DCM
   generic map("1X",2.000000, 5, 3,FALSE, 8.138020,"NONE",
               "SYSTEM_SYNCHRONOUS","LOW","LOW",TRUE,x"C080",0,TRUE,"NONE")
   Port map(DCM3_CLKin,DCM3_CLKfb,DCM3_RST,GND,GND,GND,GND,DCM3_CLK0,open,
            open,open,open,open,DCM3_CLKout,open,open,DCM3_LOCKED,open);
   ----------------- FRECUENCIA MATRIZ -----------------
CLK_FUENTE: BUFG Port map (DCM3_CLKout,CLK_MATRIZ);
--CLK_MATRIZ: f=73.7280MHz
ESC_RST_IN: INV Port map (DCM3_LOCKED,ESC_RST);
   CLK_RDY<=DCM3_LOCKED;

ESCALADOR1: DIV Port map(CLK_MATRIZ,x"028",CLK_580ns,ESC_RST);
ESCALADOR2: DIV Port map(CLK_MATRIZ,x"280",CLK_115200Hz,ESC_RST);

end RELOJ;
```

A27 DIV.VHDL

```
------DIVISOR DE FRECUENCIA PARA RELOJ-----

--DEPENDENCIAS

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

library UNISIM;
use UNISIM.vcomponents.ALL;
use WORK.PRIMITIVOS.ALL;

--DECLARACION DE LA ENTIDAD DIV
   entity DIV is
        Port(CLKin  : in  std_logic; --Reloj de entrada
             N      : in  std_logic_vector(11 downto 0);  --Relación
             CLKout : out std_logic; --Reloj de entrada
             RESET  : in  std_logic);  --Restauración
   end DIV;

--ARQUITECTURA DE LA ENTIDAD DIV
architecture DIV of DIV is
   --Componentes

   component DIV_CTRL
        Port(CAMBIO  : in  std_logic;
             CLRC   : out std_logic;
             CLRD   : out std_logic;
             CPD    : out std_logic;
             CE     : out std_logic;
             CLKin : in  std_logic;
             RESET : in  std_logic);
   end component;

   component BUFG
      Port ( I : in    std_logic;
             O : out   std_logic);
   end component;

   component INV
      Port(I : in    std_logic;
             O : out   std_logic);
   end component;

   --Conectores
   signal CNT,NH,NL,N0 : std_logic_vector(10 downto 0);
   signal N3,N1,N2 : std_logic_vector(11 downto 0);
   signal SEL : std_logic_vector(0 downto 0);
   signal CLRC,CLRD,CPD,CE,CAMBIO : std_logic;
   signal noSEL,noCLKin,CLKin_buf : std_logic;

begin
--RELOJ DE ENTRADA
CLKin1_BUF: INV Port map(CLKin,noCLKin);
--PARTICION FUNCIONAL
CONTEO: CONTADOR11 Port map(CNT,noCLKin,CE,CLRC);
```

```
   N3<=N;
   N1<=N3 srl 1;
   NL<=N1(10 downto 0);
   N2<=N-N1;
   NH<=N2(10 downto 0);

NUMERO: MUX11_2A1 Port map(NH,NL,SEL,N0);
COMP: COMPARADOR11 Port map(CNT,N0,CAMBIO);
CTRL: DIV_CTRL Port map(CAMBIO,CLRC,CLRD,CPD,CE,CLKin,RESET);
SALIDA: FDCE Port map(SEL(0),CPD,'1',CLRD,noSEL);
   noSEL<=not SEL(0);
CLKout_BUF: BUFG Port map(SEL(0),CLKout);
end DIV;
```

A28 DIV_CTRL.VHDL

```
------CONTROLADOR DEL DIVISOR DE FRECUENCIA-----

--DEPENDENCIAS

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

--DECLARACION DE LA ENTIDAD DIV_CTRL
   entity DIV_CTRL is
        Port(CAMBIO  : in  std_logic; --Señal para conmutar
             CLRC    : out std_logic;  --Limpiador del contador
             CLRD    : out std_logic;  --Limpiador del FFD
             CPD     : out std_logic;  --Cargador del FFD
             CE      : out std_logic;  --Habilitador del contador
             CLKin   : in  std_logic;  --Reloj de entrada
             RESET   : in  std_logic); --Restauración
   end DIV_CTRL;

--ARQUITECTURA DE LA ENTIDAD DIV_CTRL

architecture DIV_CTRL of DIV_CTRL is
   type VE1 is (t0,t1);
   signal PRE: VE1;

begin
MEN:
   process(CLKin,RESET)
   begin
        if RESET='1' then
             PRE<=t0;
        elsif rising_edge(CLKin) then
             PRE<=t1;
        end if;
   end process;
DEC_SAL:
   process(PRE,CAMBIO)
   begin
        case PRE is
             when t0 => CLRC<='1'; CLRD<='1'; CPD<='0'; CE<='0';
             when t1 =>
             CE<='1';
             CLRD<='0';
             if CAMBIO='1' then CLRC<='1'; CPD<='0';
             else   CLRC<='0'; CPD<='1'; end if;
        end case;
   end process;

end DIV_CTRL;
```

A29 SHIFT_REG8X8.VHDL

```
------SHIFT_REG8x8------

--DEPENDENCIAS

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use WORK.PRIMITIVOS.ALL;

--DECLARACION DE LA ENTIDAD SHIFT_REG8x8
   entity SHIFT_REG8x8 is
        Port(CLK  : in  std_logic;
             SDI  : in  std_logic_vector(7 downto 0);
             SDO  : out std_logic_vector(7 downto 0);
             PL   : in  std_logic;
             D    : in  std_logic_vector(63 downto 0);
             Q    : out std_logic_vector(63 downto 0);
             L2M  : in  std_logic;
             CS   : in  std_logic;
             RST  : in  std_logic);
   end SHIFT_REG8x8;

--ARQUITECTURA DE LA ENTIDAD SHIFT_REG8x8

architecture SHIFT_REG8x8 of SHIFT_REG8x8 is

   signal PDI,PDO : std_logic_vector(63 downto 0);

begin

--Decoder de Entrada
PDI( 7 downto  0)<=D(56)&D(48)&D(40)&D(32)&D(24)&D(16)&D( 8)&D(0);
PDI(15 downto  8)<=D(57)&D(49)&D(41)&D(33)&D(25)&D(17)&D( 9)&D(1);
PDI(23 downto 16)<=D(58)&D(50)&D(42)&D(34)&D(26)&D(18)&D(10)&D(2);
PDI(31 downto 24)<=D(59)&D(51)&D(43)&D(35)&D(27)&D(19)&D(11)&D(3);
PDI(39 downto 32)<=D(60)&D(52)&D(44)&D(36)&D(28)&D(20)&D(12)&D(4);
PDI(47 downto 40)<=D(61)&D(53)&D(45)&D(37)&D(29)&D(21)&D(13)&D(5);
PDI(55 downto 48)<=D(62)&D(54)&D(46)&D(38)&D(30)&D(22)&D(14)&D(6);
PDI(63 downto 56)<=D(63)&D(55)&D(47)&D(39)&D(31)&D(23)&D(15)&D(7);

Byte7: FULL_SHIFT_REG8 Port map(CLK,SDI(7),SDO(7),PL,PDI(63 downto 56),
                                PDO(63 downto 56),L2M,CS,RST);
Byte6: FULL_SHIFT_REG8 Port map(CLK,SDI(6),SDO(6),PL,PDI(55 downto 48),
                                PDO(55 downto 48),L2M,CS,RST);
Byte5: FULL_SHIFT_REG8 Port map(CLK,SDI(5),SDO(5),PL,PDI(47 downto 40),
                                PDO(47 downto 40),L2M,CS,RST);
Byte4: FULL_SHIFT_REG8 Port map(CLK,SDI(4),SDO(4),PL,PDI(39 downto 32),
                                PDO(39 downto 32),L2M,CS,RST);
Byte3: FULL_SHIFT_REG8 Port map(CLK,SDI(3),SDO(3),PL,PDI(31 downto 24),
                                PDO(31 downto 24),L2M,CS,RST);
Byte2: FULL_SHIFT_REG8 Port map(CLK,SDI(2),SDO(2),PL,PDI(23 downto 16),
                                PDO(23 downto 16),L2M,CS,RST);
Byte1: FULL_SHIFT_REG8 Port map(CLK,SDI(1),SDO(1),PL,PDI(15 downto  8),
                                PDO(15 downto  8),L2M,CS,RST);
Byte0: FULL_SHIFT_REG8 Port map(CLK,SDI(0),SDO(0),PL,PDI( 7 downto  0),
                                PDO( 7 downto  0),L2M,CS,RST);
```

```
--Decoder de Salida
Q( 7 downto  0)<=PDO(56)&PDO(48)&PDO(40)&PDO(32)&PDO(24)&PDO(16)&PDO( 8)
                 &PDO(0);
Q(15 downto  8)<=PDO(57)&PDO(49)&PDO(41)&PDO(33)&PDO(25)&PDO(17)&PDO( 9)
                 &PDO(1);
Q(23 downto 16)<=PDO(58)&PDO(50)&PDO(42)&PDO(34)&PDO(26)&PDO(18)&PDO(10)
                 &PDO(2);
Q(31 downto 24)<=PDO(59)&PDO(51)&PDO(43)&PDO(35)&PDO(27)&PDO(19)&PDO(11)
                 &PDO(3);
Q(39 downto 32)<=PDO(60)&PDO(52)&PDO(44)&PDO(36)&PDO(28)&PDO(20)&PDO(12)
                 &PDO(4);
Q(47 downto 40)<=PDO(61)&PDO(53)&PDO(45)&PDO(37)&PDO(29)&PDO(21)&PDO(13)
                 &PDO(5);
Q(55 downto 48)<=PDO(62)&PDO(54)&PDO(46)&PDO(38)&PDO(30)&PDO(22)&PDO(14)
                 &PDO(6);
Q(63 downto 56)<=PDO(63)&PDO(55)&PDO(47)&PDO(39)&PDO(31)&PDO(23)&PDO(15)
                 &PDO(7);


end SHIFT_REG8x8;
```

A.30 TSA.VHDL

```
------TRANSMISOR SERIAL ASINCRONICO-----

--DEPENDENCIAS

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
library UNISIM;
use UNISIM.vcomponents.ALL;
use WORK.PRIMITIVOS.ALL;

--DECLARACION DE LA ENTIDAD TSA

   entity TSA is
        Port(FPDI    : in  std_logic_vector(7 downto 0); --Datos de Entrada
             SEND    : in  std_logic;  --Datos listos
             FSDO    : out std_logic;  --Flujo Serial de Datos de Salida
             OCUPADO : inout std_logic; --Bandera
             CLK     : in  std_logic;  --Reloj
             RST  : in  std_logic); --Restauración
   end TSA;

--ARQUITECTURA DE LA ENTIDAD TSA

architecture TSA of TSA is
   --Señales

   signal CNT : std_logic_vector(3 downto 0);
   signal DAT_PARA : std_logic_vector(9 downto 0);
   signal CLR,noOCUPADO,CP : std_logic;

begin
   CLR<='1' when CNT="1010" or RST='1' else '0';
ENCENDIDO: FDCE_1 Port map(OCUPADO,SEND,'1',CLR,'1');
CONTEO: CONTADOR4 Port map(CNT,CLK,OCUPADO,CLR);
   noOCUPADO<=not OCUPADO;
   CP<=SEND or (CLK and OCUPADO);
   DAT_PARA<="10"&FPDI(0)&FPDI(1)&FPDI(2)&FPDI(3)
             &FPDI(4)&FPDI(5)&FPDI(6)&FPDI(7);
TRANSMISION: SHIFT_REG10 Port map(CP,FSDO,noOCUPADO,DAT_PARA,'1',RST);
--QI=0x200 --RELLENO <- 1

end TSA;
```

A.31 COM_DES.VHDL

```
---CIFRADO DE 8 BYTES Y SU TRANSMISION---

--DEPENDENCIAS

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

--      UNISIM
library UNISIM;
use UNISIM.VComponents.ALL;
--      WORK
use WORK.TIPOS.ALL;
use WORK.PRIMITIVOS.ALL;

--ENTIDAD COM_DES

   entity COM_DES is
        Port(SEL     : in  std_logic;
             PS1_DSR : out std_logic;
             PS1_TXD : in std_logic;
             PS1_RXD : out std_logic;
             PS1_RTS : in  std_logic;
             PS1_CTS : out std_logic;
             CLK     : in  std_logic;
             noRST   : in  std_logic;
             LED0    : out std_logic;
             PB_UP   : in  std_logic);
   end COM_DES;

--ARQUITECTURA

architecture COM_DES of COM_DES is
   --Sistema de Reloj
   component RELOJ
        Port(CLK_580ns : out std_logic;
             CLK_115200Hz : out std_logic;
             CLK_RDY      : out std_logic;
             RST    : in  std_logic;
             CLK    : in  std_logic);
   end component;

   --Transmisor Serial Asincrónico
   component TSA
        Port(FPDI    : in  std_logic_vector(7 downto 0);
             SEND   : in  std_logic;
             FSDO    : out std_logic;
             OCUPADO : inout std_logic;
             CLK    : in  std_logic;
             RST : in  std_logic);
   end component;

   component SHIFT_REG8x8
        Port(CLK  : in  std_logic;
```

```
            SDI   : in  std_logic_vector(7 downto 0);
            SDO   : out std_logic_vector(7 downto 0);
            PL    : in  std_logic;
            D     : in  std_logic_vector(63 downto 0);
            Q     : out std_logic_vector(63 downto 0);
            L2M   : in  std_logic;
            CS    : in  std_logic;
            RST   : in  std_logic);
   end component;

   component DES_CORE
        Port(IB  : in  bloque;
             K   : in  llave_efectiva_vector(1 to 16);
             OB  : out bloque);
   end component;

   component KS
        Port(KEY : in  bloque; --llave
             K   : out llave_efectiva_vector(1 to 16));
   end component;

   --Señales Internas
   signal GND,VCC : std_logic;
   signal K : llave_efectiva_vector(1 to 16);
   signal PT,CT,KEY : bloque;
   signal OCTETO_Tx: byte:= x"00";
   signal RST,CLK_RST,SYS_RST : std_logic;
   signal CLK_RDY,noCLK_RDY,CLK_Tx : std_logic;
   signal SEND,Tx_BUSY : std_logic;
   signal RXD1,CARGAR,SHIFT_CP : std_logic;
   signal CP_CNT_BYTE,CLR_CNT_BYTE : std_logic;
   signal PRE,SIG : VE3;
   signal CNT_BYTE_Tx : std_logic_vector(3 downto 0);

begin
   GND<='0';
   VCC<='1';
RELOJES: RELOJ Port map(open,CLK_Tx,CLK_RDY,CLK_RST,CLK);

DEC_SIG: process(PRE,CNT_BYTE_Tx,Tx_BUSY)

begin
   case PRE is
        when t0 => SIG<=t1;
        when t1 => SIG<=t2;
        when t2 => if Tx_BUSY='0' then SIG<=t3; else SIG<=t2; end if;
        when t3 => SIG<=t4;
        when t4 => if CNT_BYTE_Tx=x"8" then SIG<=t5; else SIG<=t2; end if;
        when t5 => SIG<=t5;
        when others => SIG<=t5;
   end case;
end process;

MEM: process(CLK_Tx,RST)

begin
   if RST='1' then
        PRE<=t0;
   elsif rising_edge(CLK_Tx) then
        PRE<=SIG;
```

```
   end if;
end process;

DEC_SAL: process(PRE,CLK_Tx,noRST,CLK_RDY)
   variable T : std_logic_vector(7 downto 0);
begin
   --Decodificador de Estados
   case PRE is
        when t0 => T:=x"01";
        when t1 => T:=x"02";
        when t2 => T:=x"04";
        when t3 => T:=x"08";
        when t4 => T:=x"10";
        when t5 => T:=x"20";
        when t6 => T:=x"40";
        when t7 => T:=x"80";
   end case;

   --Decoder de Salida
   CLR_CNT_BYTE<=T(0);
   CARGAR<=T(1);
   SHIFT_CP<=(T(1) or T(4)) and not CLK_Tx;
   SEND<=T(3);
   CP_CNT_BYTE<=SEND;

   --Resets del sistema
   CLK_RST<=not noRST;
   noCLK_RDY<=not CLK_RDY;
   RST<=CLK_RST or noCLK_RDY;

   --Puerto Serial
   PS1_DSR<=GND;
   PS1_CTS<=PS1_RTS;
   PS1_RXD<=RXD1;

   --Señales DES
   PT<=x"0123456789ABCDEF";
   KEY<=x"133457799BBCDFF1";
   --Despliegue Visual
   LED0<=noCLK_RDY; --RELOJ LISTO
end process;
ListaDeLlaves: KS Port map(KEY,K);
Cifrador: DES_CORE Port map(PT,K,CT);
REGISTRO: SHIFT_REG8x8 Port map(SHIFT_CP,x"ff",OCTETO_Tx,CARGAR,CT,
                                open,VCC,VCC,GND);
TX: TSA Port map(OCTETO_Tx,SEND,RXD1,Tx_BUSY,CLK_Tx,RST);
CONTADOR_BYTES: CONTADOR4 Port map(CNT_BYTE_Tx,CP_CNT_BYTE,VCC,
                                   CLR_CNT_BYTE);

end COM_DES;
```

A.32 COM_DES.VHDL

```
---CIRCUITO SHITF REGISTER---

--DEPENDENCIAS

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

--      UNISIM
library UNISIM;
use UNISIM.VComponents.ALL;
--      WORK
use WORK.TIPOS.ALL;
use WORK.PRIMITIVOS.ALL;

--ENTIDAD COM_DES
   entity COM_DES is
        Port(SEL     : in  std_logic;
             PS1_DSR : out std_logic;
             PS1_TXD : in std_logic;
             PS1_RXD : out std_logic;
             PS1_RTS : in  std_logic;
             PS1_CTS : out std_logic;
             CLK     : in  std_logic;
             noRST   : in  std_logic;
             LED0    : out std_logic;
             PB_UP   : in  std_logic);
   end COM_DES;

--ARQUITECTURA
architecture COM_DES of COM_DES is

   --Sistema de Reloj
   component RELOJ
        Port(CLK_580ns : out std_logic;
             CLK_115200Hz : out std_logic;
             CLK_RDY      : out std_logic;
             RST    : in  std_logic;
             CLK    : in  std_logic);
   end component;

   --Transmisor Serial Asincrónico
   component TSA
        Port(FPDI   : in  std_logic_vector(7 downto 0);
             SEND   : in  std_logic;
             FSDO   : out std_logic;
             OCUPADO : inout std_logic;
             CLK    : in  std_logic;
             RST : in  std_logic);
   end component;

   component SHIFT_REG8x8
        Port(CLK  : in  std_logic;
             SDI  : in  std_logic_vector(7 downto 0);
```

```
            SDO   : out std_logic_vector(7 downto 0);
            PL    : in  std_logic;
            D     : in  std_logic_vector(63 downto 0);
            Q     : out std_logic_vector(63 downto 0);
            L2M   : in  std_logic;
            CS    : in  std_logic;
            RST   : in  std_logic);
   end component;

   --Señales Internas
   signal GND,VCC : std_logic;
   signal OCTETO_Tx: byte:= x"00";
   signal RST,CLK_RST,SYS_RST : std_logic;
   signal CLK_RDY,noCLK_RDY,CLK_Tx : std_logic;
   signal SEND,Tx_BUSY : std_logic;
   signal RXD1,CARGAR,SHIFT_CP : std_logic;
   signal CP_CNT_BYTE,CLR_CNT_BYTE : std_logic;
   signal PRE,SIG : VE3;
   signal CNT_BYTE_Tx : std_logic_vector(3 downto 0);
   signal D : std_logic_vector(63 downto 0);

begin
   GND<='0';
   VCC<='1';
RELOJES: RELOJ Port map(open,CLK_Tx,CLK_RDY,CLK_RST,CLK);

DEC_SIG: process(PRE,CNT_BYTE_Tx,Tx_BUSY)

begin
   case PRE is
        when t0 => SIG<=t1;
        when t1 => SIG<=t2;
        when t2 => if Tx_BUSY='0' then SIG<=t3; else SIG<=t2; end if;
        when t3 => SIG<=t4;
        when t4 => if CNT_BYTE_Tx=x"8" then SIG<=t5; else SIG<=t2; end if;
        when t5 => SIG<=t5;
        when others => SIG<=t5;
   end case;
end process;

MEM: process(CLK_Tx,RST)
begin
   if RST='1' then
        PRE<=t0;
   elsif rising_edge(CLK_Tx) then
        PRE<=SIG;
   end if;
end process;

DEC_SAL: process(PRE,CLK_Tx,noRST,CLK_RDY)
   variable T : std_logic_vector(7 downto 0);
begin
   --Decodificador de Estados
   case PRE is
        when t0 => T:=x"01";
        when t1 => T:=x"02";
        when t2 => T:=x"04";
        when t3 => T:=x"08";
        when t4 => T:=x"10";
        when t5 => T:=x"20";
```

```
            when t6 => T:=x"40";
            when t7 => T:=x"80";
        end case;

        --Decoder de Salida
        CLR_CNT_BYTE<=T(0);
        CARGAR<=T(1);
        SHIFT_CP<=(T(1) or T(4)) and not CLK_Tx;
        SEND<=T(3);
        CP_CNT_BYTE<=SEND;
        D<=x"FEDCBA9876543210";

        --Resets del sistema
        CLK_RST<=not noRST;
        noCLK_RDY<=not CLK_RDY;
        RST<=CLK_RST or noCLK_RDY;

        --Puerto Serial
        PS1_DSR<=GND;
        PS1_CTS<=PS1_RTS;
        PS1_RXD<=RXD1;

        --Despliegue Visual
        LED0<=noCLK_RDY; --RELOJ LISTO
    end process;

    REGISTRO: SHIFT_REG8x8 Port
    map(SHIFT_CP,x"ff",OCTETO_Tx,CARGAR,D,open,VCC,VCC,GND);
    TX: TSA Port map(OCTETO_Tx,SEND,RXD1,Tx_BUSY,CLK_Tx,RST);
    CONTADOR_BYTES: CONTADOR4 Port map(CNT_BYTE_Tx,CP_CNT_BYTE,
                                       VCC,CLR_CNT_BYTE);
    end COM_DES;
```

A.33 COM_DES.VHDL

```
---CIRCUITO DE RELOJ Y TRANSMISOR

--DEPENDENCIAS

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

--      UNISIM
library UNISIM;
use UNISIM.VComponents.ALL;

--      WORK
use WORK.TIPOS.ALL;
use WORK.PRIMITIVOS.ALL;

--ENTIDAD COM_DES
   entity COM_DES is
        Port(SEL     : in  std_logic;
             PS1_DSR : out std_logic;
             PS1_TXD : in std_logic;
             PS1_RXD : out std_logic;
             PS1_RTS : in  std_logic;
             PS1_CTS : out std_logic;
             CLK     : in  std_logic;
             noRST   : in  std_logic;
             LED0    : out std_logic;
             PB_UP   : in  std_logic);
   end COM_DES;

--ARQUITECTURA
architecture COM_DES of COM_DES is

   --Sistema de Reloj
   component RELOJ
        Port(CLK_580ns : out std_logic;
             CLK_115200Hz : out std_logic;
             CLK_RDY      : out std_logic;
             RST    : in  std_logic;
             CLK    : in  std_logic);
   end component;

   --Transmisor Serial Asincrónico
   component TSA
        Port(FPDI    : in  std_logic_vector(7 downto 0);
             SEND   : in  std_logic;
             FSDO    : out std_logic;
             OCUPADO : inout std_logic;
             CLK    : in  std_logic;
             RST : in  std_logic);
   end component;

   component SHIFT_REG8x8
        Port(CLK  : in  std_logic;
```

```
            SDI   : in  std_logic_vector(7 downto 0);
            SDO   : out std_logic_vector(7 downto 0);
            PL    : in  std_logic;
            D     : in  std_logic_vector(63 downto 0);
            Q     : out std_logic_vector(63 downto 0);
            L2M   : in  std_logic;
            CS    : in  std_logic;
            RST   : in  std_logic);
   end component;

   component INV
      Port(I : in  std_logic;
           O : out std_logic);
   end component;

   component BUFG
      Port(I : in  std_logic;
           O : out std_logic);
   end component;

   component BUF
      Port(I : in  std_logic;
           O : out std_logic);
   end component;

   component IBUF
      Port(I : in  std_logic;
           O : out std_logic);
   end component;

   component OBUF
      Port(I : in  std_logic;
           O : out std_logic);
   end component;

   component DIV
        Port(CLKin  : in  std_logic;
             N      : in  std_logic_vector(11 downto 0);
             CLKout : out std_logic;
             RESET  : in  std_logic);
   end component;

   --Señales Internas

   signal GND,VCC : std_logic;
   signal OCTETO_Tx: byte:= x"00";
   signal CNT_NUM : nybble;
   signal RST,CLK_RST,SYS_RST : std_logic;
   signal CLK_RDY,noCLK_RDY,CLK_Rx,CLK_Tx,CLK_Tx12 : std_logic;
   signal SEND,noSEND,Tx_BUSY : std_logic;
   signal RXD1,PLOAD,SHIFT_CP : std_logic;
   signal ESTADO : VE2;
   signal CNT_aux,CNT_DAT : std_logic_vector(3 downto 0);
   signal CLK_aux,noCLK_aux : std_logic;
   signal D,Q : std_logic_vector(63 downto 0);

begin
   GND<='0';
   VCC<='1';
```

```
--RESET DEL SISTEMA
RESET0_BUF: INV Port map(noRST,CLK_RST);
RESET1_BUF: INV Port map(CLK_RDY,noCLK_RDY);
   SYS_RST<=CLK_RST or noCLK_RDY;
RESET3_BUF: BUF Port map(SYS_RST,RST);

--SEÑALES DEL PUERTO SERIAL
PS1_DSR_BUF: OBUF Port map(GND,PS1_DSR);
PS1_CTS_BUF: OBUF Port map(PS1_RTS,PS1_CTS);
PS1_RXD_BUF: OBUF Port map(RXD1,PS1_RXD);

--PARTICION FUNCIONAL
RELOJES: RELOJ Port map(CLK_Rx,CLK_Tx,CLK_RDY,CLK_RST,CLK);
CONT_AUX: CONTADOR4 Port map(CNT_aux,CLK_Tx,'1',RST);
   CLK_aux<='1' when CNT_aux=x"F" else '0';
CP_AUX: BUFG Port map(CLK_aux,SEND);
   noSEND<=not SEND;
CONT_DAT: CONTADOR4 Port map(OCTETO_Tx(3 DOWNTO 0),noSEND,VCC,RST);
TX: TSA Port map(OCTETO_Tx,SEND,RXD1,Tx_BUSY,CLK_Tx,RST);

RELOJ_LISTO: OBUF Port map(noCLK_RDY,LED0);

end COM_DES;
```

APÉNDICE B

TABLAS DE SUSTITUSIÓN DE LA FUNCIÓN DE RONDA

DEL ALGORITMO DES (FIPS PUB 46-3)

$$S^1 = \begin{pmatrix} 14 & 4 & 13 & 1 & 2 & 15 & 11 & 8 & 3 & 10 & 6 & 12 & 5 & 9 & 0 & 7 \\ 0 & 15 & 7 & 4 & 14 & 2 & 13 & 1 & 10 & 6 & 12 & 11 & 9 & 5 & 3 & 8 \\ 4 & 1 & 14 & 8 & 13 & 6 & 2 & 11 & 15 & 12 & 9 & 7 & 3 & 10 & 5 & 0 \\ 15 & 12 & 8 & 2 & 4 & 9 & 1 & 7 & 5 & 11 & 3 & 14 & 10 & 0 & 6 & 13 \end{pmatrix}$$

Tabla A.I Tabla de sustitución 1.

$$S^2 = \begin{pmatrix} 15 & 1 & 8 & 14 & 6 & 11 & 3 & 4 & 9 & 7 & 2 & 13 & 12 & 0 & 5 & 10 \\ 3 & 13 & 4 & 7 & 15 & 2 & 8 & 14 & 12 & 0 & 1 & 10 & 6 & 9 & 11 & 5 \\ 0 & 14 & 7 & 11 & 10 & 4 & 13 & 1 & 5 & 8 & 12 & 6 & 9 & 3 & 2 & 15 \\ 13 & 8 & 10 & 1 & 3 & 15 & 4 & 2 & 11 & 6 & 7 & 12 & 0 & 5 & 14 & 9 \end{pmatrix}$$

Tabla A.II Tabla de sustitución 2.

$$S^3 = \begin{pmatrix} 10 & 0 & 9 & 14 & 6 & 3 & 15 & 5 & 1 & 13 & 12 & 7 & 11 & 4 & 2 & 8 \\ 13 & 7 & 0 & 9 & 3 & 4 & 6 & 10 & 2 & 8 & 5 & 14 & 12 & 11 & 15 & 1 \\ 13 & 6 & 4 & 9 & 8 & 15 & 3 & 0 & 11 & 1 & 2 & 12 & 5 & 10 & 14 & 7 \\ 1 & 10 & 13 & 0 & 6 & 9 & 8 & 7 & 4 & 15 & 14 & 3 & 11 & 5 & 2 & 12 \end{pmatrix}$$

Tabla A.III Tabla de sustitución 3.

$$S^4 = \begin{pmatrix} 7 & 13 & 14 & 3 & 0 & 6 & 9 & 10 & 1 & 2 & 8 & 5 & 11 & 12 & 4 & 15 \\ 13 & 8 & 11 & 5 & 6 & 15 & 0 & 3 & 4 & 7 & 2 & 12 & 1 & 10 & 14 & 9 \\ 10 & 6 & 9 & 0 & 12 & 11 & 7 & 13 & 15 & 1 & 3 & 14 & 5 & 2 & 8 & 4 \\ 3 & 15 & 0 & 6 & 10 & 1 & 13 & 8 & 9 & 4 & 5 & 11 & 12 & 7 & 2 & 14 \end{pmatrix}$$

Tabla A.IV Tabla de sustitución 4.

$$S^5 = \begin{pmatrix} 2 & 12 & 4 & 1 & 7 & 10 & 11 & 6 & 8 & 5 & 3 & 15 & 13 & 0 & 14 & 9 \\ 14 & 11 & 2 & 12 & 4 & 7 & 13 & 1 & 5 & 0 & 15 & 10 & 3 & 9 & 8 & 6 \\ 4 & 2 & 1 & 11 & 10 & 13 & 7 & 8 & 15 & 9 & 12 & 5 & 6 & 3 & 0 & 14 \\ 11 & 8 & 12 & 7 & 1 & 14 & 2 & 13 & 6 & 15 & 0 & 9 & 10 & 4 & 5 & 3 \end{pmatrix}$$

Tabla A.V Tabla de sustitución 5.

$$S^6 = \begin{pmatrix} 12 & 1 & 10 & 15 & 9 & 2 & 6 & 8 & 0 & 13 & 3 & 4 & 14 & 7 & 5 & 11 \\ 10 & 15 & 4 & 2 & 7 & 12 & 9 & 5 & 6 & 1 & 13 & 14 & 0 & 11 & 3 & 8 \\ 9 & 14 & 15 & 5 & 2 & 8 & 12 & 3 & 7 & 0 & 4 & 10 & 1 & 13 & 11 & 6 \\ 4 & 3 & 2 & 12 & 9 & 5 & 15 & 10 & 11 & 14 & 1 & 7 & 6 & 0 & 8 & 13 \end{pmatrix}$$

Tabla A.VI Tabla de sustitución 6.

$$S^7 = \begin{pmatrix} 4 & 11 & 2 & 14 & 15 & 0 & 8 & 13 & 3 & 12 & 9 & 7 & 5 & 10 & 6 & 1 \\ 13 & 0 & 11 & 7 & 4 & 9 & 1 & 14 & 14 & 3 & 5 & 12 & 2 & 15 & 8 & 6 \\ 1 & 4 & 11 & 13 & 12 & 3 & 7 & 10 & 10 & 15 & 6 & 8 & 0 & 5 & 9 & 2 \\ 6 & 11 & 13 & 8 & 1 & 4 & 10 & 9 & 9 & 5 & 0 & 15 & 14 & 2 & 3 & 12 \end{pmatrix}$$

Tabla A.VII Tabla de sustitución 7.

$$S^8 = \begin{pmatrix} 13 & 2 & 8 & 4 & 6 & 15 & 11 & 1 & 10 & 9 & 3 & 14 & 5 & 0 & 12 & 7 \\ 1 & 15 & 13 & 8 & 10 & 3 & 7 & 4 & 12 & 5 & 6 & 11 & 0 & 14 & 9 & 2 \\ 7 & 11 & 4 & 1 & 9 & 12 & 14 & 2 & 0 & 6 & 10 & 13 & 15 & 3 & 5 & 8 \\ 2 & 1 & 14 & 7 & 4 & 10 & 8 & 13 & 15 & 12 & 9 & 0 & 3 & 5 & 6 & 11 \end{pmatrix}$$

Tabla A.VIII Tabla de sustitución 8.

APÉNDICE C

CÓDIGO C++ DEL PROGRAMA DE PRUEBA DES (FIPS PUB 46-3)

```
/*************************************************************************/
/*                              PROGRAMA PRINCIPAL                       */
/*************************************************************************/

#include "DES.H"


void main(){
        bloque pt,ct,dt;
        llave k;
        string PTstr,CTstr,DTstr,Kstr;
        register uint64_t N=0;

        DES_Ek(hex2bin("cdeff31fda070114"),hex2bin("0123456789abcdef"));

        /////////// ENTRADAS /////////

        //Texto Plano
        cout<<endl<<"Ingrese el TEXTO PLANO: ";
        cin>>PTstr; //0123456789ABCDEF

        //Clave
        cout<<endl<<"Ingrese la CLAVE: ";
        cin>>Kstr;  //133457799BBCDFF1

        //Conversión de datos
        pt=hex2bin(PTstr);
        k=hex2bin(Kstr);


        /////////// CIFRADOR /////////

        ct=DES_Ek(pt,k); //85E813540F0AB405
        cout<<endl<<"TEXTO CIFRADO: ";
        mostrar(ct);


        //////// DESCIFRADOR ////////

        dt=DES_Dk(ct,kinv);  //0123456789ABCDEF
        cout<<endl<<"TEXTO DESCIFRADO: ";
        mostrar(dt);

        /////// COMPROBACION ////////


        if(dt==pt) cout<<endl<<"El texto fue correctamente descifrado.\n";
        else cout<<endl<<"El algortimo tiene errores.\n";
        cout<<"Documento de Referencia: FinalProyect.pdf"<<endl;

        /////// FIN ////////
*/
```

```
        getch();
}

/*****************************************************************************/
/*                                FIN DEL ARCHIVO                            */
*/
/*****************************************************************************/
```

```
/*************************************************************************/
/*                          DES FIPS46-3(CODIGO FUENTE)                  */
/*************************************************************************/

#include "DES.H"


bloque DES_Ek(bloque pt, llave k){
        bloque F0,F16,ct;
        permutacion Pi(PI,PI+64),Pf(PF,PF+64);

        F0=permutar(pt,Pi);
        F16=Feistel_Ek(F0,k);
        ct=permutar(F16,Pf);

        return ct;
}

bloque DES_Dk(bloque ct, llave k){
        bloque F0,F16,dt;
        permutacion Pi(PI,PI+64),Pf(PF,PF+64);

        F0=permutar(ct,Pi);
        F16=Feistel_Dk(F0,k);
        dt=permutar(F16,Pf);

        return dt;
}

/*ShiftReg8x8::ShiftReg8x8(natural n){
        Q=bloque(n);
}

ShiftReg8x8::cargar(bloque D){
        natural n=D.size();
        Q.resize(n);
        Q=D;
}

ShiftReg8x8::desplazar(bool sentido,bloque dato_serial){
        natural m=Q.size(),n=dato_serial.size(),i=0;

        if(sentido==M2L){
                //SHR
                for(i=m-1;i>=n;i--) Q[i]=Q[i-n];
                for(i=0;i<n;i++) Q[i]=dato_serial[i];
        }
        if(sentido==L2M){
                //SHL
                for(i=0;i<m-n-1;i++) Q[i]=Q[i+n];
                for(i=0;i<n;i++) Q[m-n+i]=dato_serial[i];
        }
}*/

bloque rango(bloque b,natural inicio,natural fin){
        bloque y;
        natural i=0;
        for(i=inicio;i<=fin;i++){
                y.push_back(b[i-inicio]);
        }
```

```
        return y;
}

registro bloque2registro(bloque BITS){
        natural n=BITS.size()/8,i=0,j=0,inicio_byte=0;
        registro REGISTRO;
        byte N=0;

        for(i=0;i<n;i++){
                N=0;
                inicio_byte=i*8;
                for(j=0;j<8;j++){
                        N+=BITS[inicio_byte+j]*IntPower(2,7-j);
                }
                REGISTRO.push_back(N);
        }
        return REGISTRO;
}

uint64_t bloque2qword(bloque BITS){
        natural n=BITS.size()/8,i=0,j=0,inicio_byte=0;
        register uint64_t REGISTRO=0;
        byte N=0;

        for(i=0;i<n;i++){
                N=0;
                inicio_byte=i*8;
                for(j=0;j<8;j++){
                        N+=BITS[inicio_byte+j]*IntPower(2,7-j);
                }
                if(i) REGISTRO<<=8;
                REGISTRO+=N;
        }
        return REGISTRO;
}

bloque qword2bloque(uint64_t REGISTRO){
        natural n=sizeof(REGISTRO)*8,i=0;
        bloque BITS;
        bit b=0;

        for(i=0;i<n;i++){
                b=(REGISTRO>>n-i-1)&1;
                BITS.push_back(b);
        }

        return BITS;
}
/*
bloque DES_OFB(bloque P,uint64_t VI,llave k){
        natural n=P.size(),i=0,d=n/8;
        bloque C(n),Ek(64);
        registro p(d),o(d),c(d);
        register uint64_t z=0,s=0;

        p=bloque2registro(P);
        s=VI;

        cout<<"TIME | DES INPUT BLOCK  | DES OUTPUT BLOCK  | P | O |
C"<<endl;
```

```
        for(i=0;i<d;i++){
                Ek=DES_Ek(qword2bloque(s),k);
                //mostrar(Ek); cout<<endl;
                z=bloque2qword(Ek);
                o[i]=(z>>56)&0xFF; //Oi<-8MSB(Z)
                c[i]=o[i]^p[i];
                cout<<" "<<i+1;
                cout<<"      "; printf("%x%x ",s>>4,s&0xF);
                cout<<" "; mostrar(Ek);
                cout<<" "; printf("%x%x ",p[i]>>4,p[i]&0xF);
                cout<<" "; printf("%x%x ",o[i]>>4,o[i]&0xF);
                cout<<" "; printf("%x%x ",c[i]>>4,c[i]&0xF);
                cout<<endl;
                s<<=8;
                s+=o[i];
        }
        C=registro2bloque(c);
        return C;
}

/*************************************************************************/
/*                              FIN DEL ARCHIVO
*/
/*************************************************************************/
```

```
/*************************************************************************/
/*                          FEISTEL (CODIGO FUENTE)                      */
/*************************************************************************/

#include "FEISTEL.H"

bloque segmento_I(bloque B){
        natural i,d=B.size()/2;
        bloque I(d);
        for(i=0;i<d;i++) I[i]=B[i];
        return I;
}

bloque segmento_II(bloque B){
        natural i,d=B.size()/2;
        bloque II(d);
        for(i=0;i<d;i++) II[i]=B[i+d];
        return II;
}

bloque concatenar(bloque B1,bloque B2){
        natural i,D1=B1.size(),D2=B2.size();
        bloque CONCAT(D1+D2);
        //for(i=0;i<D1;i++) CONCAT[i]=B1[i];
        //for(i=0;i<D2;i++) CONCAT[i+D1]=B2[i];
        for(i=0;i<D2;i++) B1.push_back(B2[i]);
        return B1;
}

bloque CLS(bloque B,natural shift){
        natural i,d=B.size();
        bloque S1(shift),S2(d-shift),SH(d);
        for(i=0;i<shift;i++) S1[i]=B[i];
        for(i=shift;i<d;i++) S2[i-shift]=B[i];
        SH=concatenar(S2,S1);
        return SH;
}

particion KS(llave KEY){
        natural n,shift=2;
        llave C(28),D(28),CD(56);
        permutacion PermutedChoice1(EP1,EP1+56),PermutedChoice2(EP2,EP2+48);
        particion K(17);
        //C0 Y D0
        K[0]=permutar(KEY,PermutedChoice1);
        C=segmento_I(K[0]);
        D=segmento_II(K[0]);
        for(n=1;n<=16;n++){
                //Desplazamientos
                if(n<=2 || n==9 || n==16) shift=1;
                else shift=2;
                //Cn Y Dn
                C=CLS(C,shift);
                D=CLS(D,shift);
                //Kn
                CD=concatenar(C,D);
                K[n]=permutar(CD,PermutedChoice2);
                cout<<"K"<<n<<"\t"; mostrar(K[n]); cout<<endl;
        }
        cout<<endl;
```

```
        return K;
}

bloque Feistel_Ek(bloque M,llave KEY){
        natural n;
        bloque FSTL(64),L(32),R(32),TEMP(32);
        particion K(17);

        //KS
        K=KS(KEY);

        //L0 Y R0
        L=segmento_I(M);
        R=segmento_II(M);
        cout<<"F0\t"; mostrar(L); cout<<"\t"; mostrar(R); cout<<endl;
        for(n=1;n<=16;n++){
                //Ln Y Rn
                TEMP=L;
                L=R;
                R=EXOR(TEMP,f(R,K[n]));
                cout<<"F"<<n<<"\t"; mostrar(L); cout<<"\t"; mostrar(R);
cout<<endl;
        }
        //CRUCE DE 32 BITS
        FSTL=concatenar(R,L);
        return FSTL;
}

bloque Feistel_Dk(bloque M,llave KEY){
        natural n;
        bloque FSTL(64),L(32),R(32),TEMP(32);
        particion K(17);
        //L0 Y R0
        L=segmento_I(M);
        R=segmento_II(M);
        cout<<"F0\t"; mostrar(L); cout<<"\t"; mostrar(R); cout<<endl;
        //KS
        K=KS(KEY);
        for(n=1;n<=16;n++){
                //Ln Y Rn
                TEMP=L;
                L=R;
                R=EXOR(TEMP,f(R,K[17-n]));
                cout<<"F"<<n<<"\t"; mostrar(L); cout<<"\t"; mostrar(R);
cout<<endl;
        }
        //CRUCE DE 32 BITS
        FSTL=concatenar(R,L);
        return FSTL;
}

/*************************************************************************/
/*                              FIN DEL ARCHIVO                          */
/*************************************************************************/
```

```
/*************************************************************************/
/*                          FUNCION F (CODIGO FUENTE)
*/
/*************************************************************************/

#include "FUNCION_F.H"

bloque EXOR(bloque B1,bloque B2){
        natural i,d=B1.size();
        bloque EXOR(d);
        for(i=0;i<d;i++)
                EXOR[i]=((!B1[i] && B2[i]) || (B1[i] && !B2[i]));
        return EXOR;
}

particion particionar(bloque B){
        particion PART(8);
        bloque TEMP(6);
        natural i,j,k;
        for(k=i=0;k<8;k++,i+=6){
                for(j=0;j<6;j++) TEMP[j]=B[i+j];
                PART[k]=TEMP;
        }
        return PART;
}

bloque NaturalToWord(natural N,natural Long){
        bloque WORD;
        natural i,d,aux;
        if(!N) return (WORD=bloque(Long));
        else{
                d=max(Long,(natural)(Floor(Log2(N))+1));
                WORD=bloque(d);
        }
        for(i=0,aux=N;i<d;i++,aux/=2) WORD[d-1-i]=aux%2;
        return WORD;
}

natural WordToNatural(bloque B){
        natural i,d=B.size()-1,N=0;
        for(i=0;i<=d;i++) N+=B[d-i]*IntPower(2,i);
        return N;
}

natural generar_i(bloque B){
        bloque AUX;
        AUX.push_back(B[0]);
        AUX.push_back(B[B.size()-1]);
        return WordToNatural(AUX);
}

natural generar_j(bloque B){
        natural i,d=B.size()-1;
        bloque AUX;
        for(i=1;i<d;i++) AUX.push_back(B[i]);
        return WordToNatural(AUX);
}

bloque sustitucion(bloque B){
        bloque S,AUX(4);
```

```
        natural i,j,n,m,sust;
        particion PART=particionar(B);
        for(n=0;n<8;n++){
                i=generar_i(PART[n]);
                j=generar_j(PART[n]);
                sust=caja_S[n][i][j];
                AUX=NaturalToWord(sust,4);
                for(m=0;m<4;m++) S.push_back(AUX[m]);
                AUX.clear();
        }
        return S;
}

bloque f(bloque B,llave K){
        bloque F,B1,B2;
        permutacion EXPANSION(E,E+48),PERMUTACION(P,P+32);
        B1=permutar(B,EXPANSION);
        B2=sustitucion(EXOR(B1,K));
        F=permutar(B2,PERMUTACION);
        return F;
}

/*************************************************************************/
/*                              FIN DEL ARCHIVO                          */
/*************************************************************************/
```

```
/*************************************************************************/
/*                        PERMUTACIONES (CODIGO FUENTE)                  */
/*************************************************************************/

#include "PERMUTACIONES.H"

bloque permutar(bloque B,permutacion P){
        natural i,d=P.size();
        bloque BP(d);
        for(i=0;i<d;i++) BP[i]=B[P[i]-1];
        return BP;
}

natural pot2(natural n){
        natural POT=1;
        return POT<<n;
}

string bin2hex(bloque B){
        int i=0,j=0,num;
        string hex="";
        char *digito=" ";

        for(i=0;i<B.size();i+=4){
                num=0;
                for(j=0;j<4;j++) num+=B[i+j]*pot2(3-j);
                sprintf(digito,"%x",num);
                hex+=digito;
        }

        return hex;
}

natural hexchar2hexdig(char c){
        natural i=0;
        for(i=0;i<16;i++){
                if(toupper(c)==HEX[i]){
                        return i;
                }
        }
        return 0;
}

bloque hex2bin(string hexstr){
        natural i=0;
        byte hexdig=0,aux=0;
        bloque B;

        for(i=0;i<hexstr.length();i++){
                hexdig=hexchar2hexdig(hexstr[i]);
                //printf("%x ",hexdig);
                aux=(hexdig & 0x08)>>3; B.push_back(aux);
                aux=(hexdig & 0x04)>>2; B.push_back(aux);
                aux=(hexdig & 0x02)>>1; B.push_back(aux);
                aux=(hexdig & 0x01);    B.push_back(aux);
        }
        return B;
}

void mostrar(bloque B){
```

```
        string str=bin2hex(B);
        cout<<str;//<<endl;
}

void mostrarbin(bloque B){
        natural i=0;
        for(i=0;i<B.size();i++){
                if(i%4==0 && i>0) cout<<" ";
                cout<<B[i];
        }
        cout<<endl;
}

/*****************************************************************************/
/*                                FIN DEL ARCHIVO                            */
/*****************************************************************************/
```

BIBLIOGRAFÍA

[1] Gran Diccionario de la Lengua Española LAROUSSE; SPES Editorial, S.L.

[2] GRANADOS PAREDES, GIBRÁN; Introducción a la Criptografía; Revista Digital Universitaria, Universidad Autónoma de México; 10 de Julio, 2006; Vol. 7, No. 7.

[3] SHANNON, CLAUDE ELWOOD; A Mathematical Theory of Communication; The Bell System Technical Journal; Julio, Octubre, 1948; Vol. 27, pp. 379–423, 623–656.

[4] O'CALLAGHAN, PATRICK; Criptografía y Seguridad de Datos; Universidad Simón Bolívar de Venezuela; Septiembre–Diciembre, 2003.

[5] SCHNEIER, BRUCE; Applied Cryptography, Second Edition; John Wiley & Sons, Inc.; Enero 1996.

[6] FORNÉ, JORDI; Criptografía y Seguridad en Comunicaciones; Departamento Matemática Aplicada y Telemática, Universitat Politècnica de Catalunya; Febrero, 2001.

[7] LEDUC, GUY; Ingénierie des systèmes informatiques répartis; Université de Liège Institut Montefiore; Septiembre, 2003.

[8] MENGUAL GALÁN, LUIS; Arquitecturas de Seguridad; Departamento de Lenguajes y Sistemas e Ingeniería del Software, Facultad de Informática, Universidad Politécnica de Madrid; Diciembre, 2005.

[9] GÓMEZ PARETS, GERARDO; Normas De Seguridad De Las Tecnologías Para La Gestión Del Conocimiento Serie De Normas ISO-IEC 27000; Oficina de Seguridad para las Redes Informáticas, Gobierno de Cuba; Febrero, 2007.

[10] PENALVA IVARS, CRISTÓBAL; Seguridad:Criptografía; Universitat de València; Septiembre 2004.

[11] LUCENA LÓPEZ, MANUEL JOSÉ; Criptografía y Seguridad en Computadores; Universidad de Jaén; Marzo, 2002.

[12] SHANNON, CLAUDE ELWOOD; Communication Theory of Secrecy Systems; The Bell System Technical Journal; Octubre, 1949; Vol. 28, pp. 656-715.

[13] DOMÍNGUEZ ESPINOZA, EDGAR URIEL & PACHECO GÓMEZ, LEONARDO; Algoritmos de Criptografía Clásica; Facultad de Ingeniería, Universidad Nacional Autónoma de México; Abril, 2007.

[14] MORAGA DE LA RUBIA, MARÍA DE LOS ÁNGELES; Protección y Seguridad de la Información; Grupo Alarcos, Escuela Superior de Informática de Ciudad Real; Febrero, 2007.

[15] RAMIO AGUIRRE, JORGE; Aplicaciones Criptográficas; Facultad de Informática, Universidad Politécnica de Madrid; Junio, 1999.

[16] SIMARI, GERARDO I.; Una introducción a la Criptografía Algoritmos y Complejidad; Departamento de Ciencias e Ingeniería de la Computación, Universidad Nacional del Sur, Argentina; Abril, 2003.

[17] B. PRENEEL; Modes of Operation of a Block Cipher; Marzo, 2004.

[18] CRÉPEAU, CLAUDE; Cryptography and Data Security; Computer Science, McGill University; Noviembre, 2003.

[19] U.S. DEPARTMENT OF COMMERCE; NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY; FIPS PUB 46-3: Data Encryption Standard (DES); Federal Information Processing Standards Publication; Octubre, 1999.

[20] SÁNCHEZ ARRIAZU, JORGE; Descripción del algoritmo DES; Diciembre, 1999.

[21] JORDAN, PATRICK; WEIRATHER, JASON; Pipelined DES Cryptologic Processor; Iowa State University; Febrero, 2003.

[22] BLÖMER, JOHANNES; Introduction to cryptography; University of Paderborn; Julio, 2006.

[23] RODRIGUEZ-HENRIQUEZ, FRANCISCO; Cryptographic Algorithms on Reconfigurable Hardware; Springer Series on Signals and Communication Technology; 2006.

[24] XILINX, INC; Getting Started with FPGAs; Xilinx Web; disponible en: http://www.xilinx.com/company/gettingstarted/

[25] XILINX, INC; Virtex™-II Platform FPGAs: Detailed Description; Xilinx Support; Octubre, 2003.

[26] XILINX, INC; Virtex-II Pro and Virtex-II Pro X FPGA: Complete Data Sheet; Xilinx Support; Noviembre, 2007.

[27] XILINX, INC; Virtex-II Pro and Virtex-II Pro X FPGA User Guide; Xilinx Support; Marzo, 2007.

[28] FREIRE RUBIO, MIGUEL ANGEL; Introducción al Lenguaje VHDL; Departamento de Sistemas Electrónicos y Control, Universidad Politécnica de Madrid; Septiembre, 2001.

[29] PERRY, DOUGLAS L.; VHDL Programming by Example 4th Ed; McGraw-Hill; Mayo, 1999.

[30] VERIBEST, INC; VeriBest FPGA Synthesis VHDL Reference Manual; Octubre, 1997.

[31] XILINX, INC; Xilinx University Program Virtex-II Pro Development System Hardware Reference Manual; Xilinx Support; Marzo, 2005.

[32] ANGERER, CHRISTOPH; Testbed Implementation for a Low Density Parity Check Decoder; Telecommunications Research Center Vienna; Enero, 2007.

[33] PASHAM, VIKRAM; High-Speed DES and Triple DES Encryptor/Decryptor; Application Note: Virtex-E Family and Virtex-II Series; Xilinx Inc.; Agosto, 2001

[34] XILINX, INC; Libraries Guide; Xilinx Support; Abril, 2005