Bounoure, Francois. Fernández, Javier.

Diseño de un sistema de trasmisión digital en banda base

Autores: Bounoure, Francois. Fernández, Javier.

Año 2004

Universidad Nacional de Mar del Plata

Facultad de Ingeniería

Dpto. Electrónica

ÍNDICE

RESUMEN

Se presenta en este trabajo el estudio, el desarrollo y la implementación de un sistema transmisor - receptor digital en banda base, con codificación y formato variable. La implementación se llevó a cabo en una FPGA programada con lenguaje de alto nivel VHDL. Una de las principales características que posee el sistema es la capacidad de trabajar con dos formatos distintos, Manchester y Bipolar con retorno a cero. La trama transmitida se compone de un encabezado, con fines de sincronismo, y un bloque de datos codificados utilizando códigos cíclicos.

En primer lugar se realizó un estudio de distintos métodos de sincronismo para el sistema. El mecanismo que fue implementado se basa en el algoritmo llamado Early Late, que resultó ser el más conveniente. La primera etapa de diseño se llevó a cabo utilizando Matlab; por medio de diferentes simulaciones se fueron adaptando los distintos parámetros del algoritmo hasta obtener el diseño más conveniente para su implementación en VHDL. La siguiente etapa consistió en llevar a VHDL el diseño obtenido primeramente en Matlab. La programación se hizo por medio del programa Max+Plus II de Altera. Tanto el transmisor como el receptor fueron divididos en distintos bloques funcionales para simplificar las etapas de diseño. En los bloques principales del transmisor se implementaron funciones como por ejemplo la codificación, la conformación de la trama y el formato, mientras que en el receptor las funciones principales son la detección de trama, el algoritmo de sincronismo y el bloque decodificador.

Como última etapa se armó un banco de prueba en el cual se implementó en forma experimental el sistema completo en dos FPGA FLEX10k20, utilizando un CDA y un CAD ambos de 8 bits. El sistema se probó en distintas condiciones, demostrando un funcionamiento satisfactorio para ambos tipos de formato y variando la frecuencia de transmisión en un 10%.

INTRODUCCION

Herramienta de diseño

Este trabajo surgió a partir de la idea de desarrollar e implementar un sistema transmisor receptor digital con codificación y formato variable. Se decidió realizar la implementación en una FPGA (Field Programable Logic Array, arreglos programables de campos de compuertas), utilizando tecnología de diseño de alto nivel. Para ello se recurrió al VHDL (Very high level Hardware Description Language), un lenguaje de descripción de hardware que permite la programación de dichas FPGA's. Cuando se aborda el diseño de un sistema electrónico y surge la necesidad de implementar un bloque con hardware dedicado son varias las posibilidades con las que se cuenta. En la siguiente figura se han representado las principales aproximaciones ordenadas en función de los parámetros costo, flexibilidad, prestaciones y complejidad. Como se puede ver, las mejores prestaciones las proporciona un diseño full-custom (completamente a medida), consiguiéndose a elevados costos y enorme complejidad de diseño. En el otro extremo del abanico de posibilidades se encuentra la implementación software, que es muy barata y flexible, pero que en determinados casos no es válida para alcanzar un nivel de prestaciones relativamente alto.

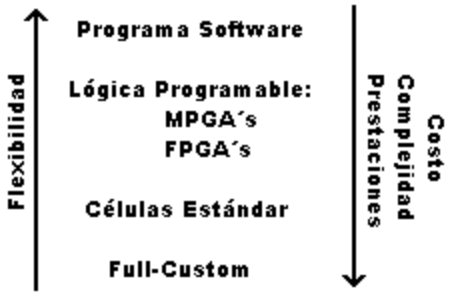

Figura 1. Prestaciones de los distintos dispositivos.

Entre estas dos opciones se puede elegir la fabricación de un circuito electrónico realizado mediante diseño semi-custom, utilizando células estándar, o recurrir a un circuito ya fabricado que se pueda "programar", como son las FPGA's. De estas dos opciones la primera proporciona mejores prestaciones,

aunque es más cara y exige un período de diseño relativamente largo. Por otro lado, los dispositivos lógicos programables constituyen una buena oferta para realizar diseños electrónicos digitales con un buen compromiso entre costo y prestaciones. Y lo que es mejor, permiten obtener una implementación en un tiempo de diseño asombrosamente corto.

Otro aspecto que se debe tener en cuenta para decidirse por este tipo de implementación es que el costo de realización es muy bajo, por lo que suele ser una buena opción para la realización de prototipos. Otra característica muy importante para remarcar es que las FPGA's pueden ser reprogramadas una gran cantidad de veces hasta que se decida su implementación final, que luego de ser "bajada" al dispositivo, éste no podrá ser programado nuevamente.

Las FPGA, introducidas por Xilinx en 1985, son el dispositivo programable por el usuario de espectro más general. También se denominan LCA (Logic Cell Array, arreglo de celdas lógico). Consisten en una matriz bidimensional de bloques configurables que se pueden conectar mediante recursos generales de interconexión. Estos recursos incluyen segmentos de pista de diferentes longitudes, más unos conmutadores programables para enlazar bloques a pistas o pistas entre sí. En realidad, lo que se programa en una FPGA son los conmutadores que sirven para realizar las conexiones entre los diferentes bloques, más la configuración de los bloques.

El proceso de diseño de un circuito digital utilizando una matriz lógica programable puede descomponerse en dos etapas básicas:

1. Dividir el circuito en bloques básicos, asignándolos a los bloque configurables del dispositivo.

2. Conectar los bloques de lógica mediante los conmutadores necesarios.

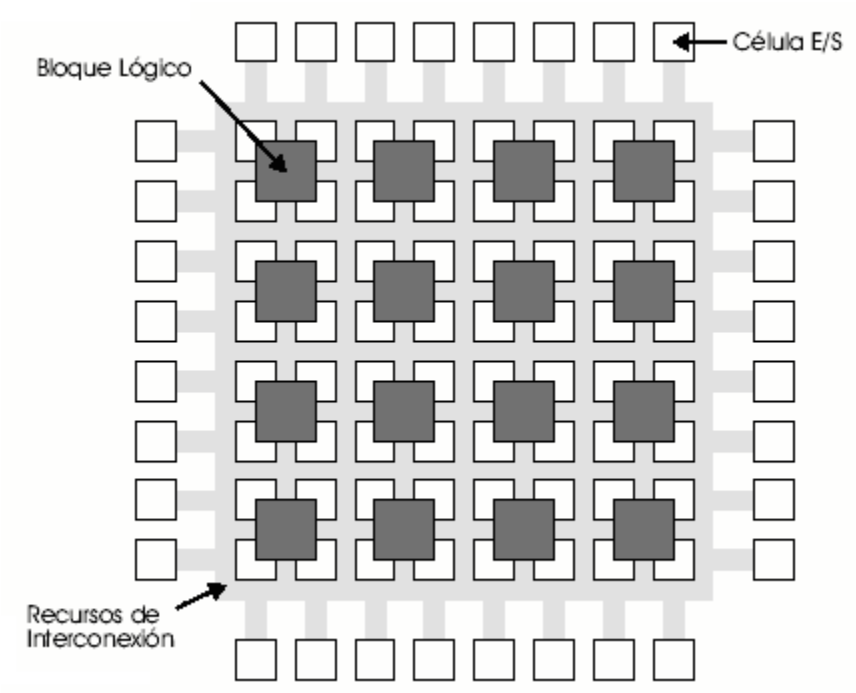

Figura 2. Estructura de una FPGA

Los elementos básicos constituyentes de una FPGA se pueden ver en la figura anterior y son los siguientes:

1. Bloques lógicos, cuya estructura y contenido se denomina arquitectura. Hay muchos tipos de arquitecturas, que varían principalmente en complejidad (desde una simple compuerta hasta módulos más complejos o estructuras tipo PLD, Programable Logia Device). Suelen incluir biestables para facilitar la implementación de circuitos secuenciales. Otros módulos de importancia son los bloques de Entrada/Salida.
2. Recursos de interconexión, cuya estructura y contenido se denomina arquitectura de ruteado.
3. Memoria RAM, que se carga durante el RESET para configurar bloques y conectarlos.

Entre las numerosas ventajas que proporciona el uso de FPGA's se destacan las siguientes:

- el bajo costo de prototipado
- el corto tiempo de producción

No todas son ventajas. Entre los inconvenientes de su utilización están su baja velocidad de operación y baja densidad lógica (poca lógica implementable en un solo chip). Su baja velocidad se debe a los retardos introducidos por los conmutadores y las largas pistas de conexión. Por supuesto, no todas las FPGA son iguales. Dependiendo del fabricante nos podemos encontrar con diferentes soluciones. Las FPGA's que existen en la actualidad en el mercado se pueden clasificar como pertenecientes a cuatro grandes familias, dependiendo de la estructura que adoptan los bloques lógicos que tengan definidos. Las cuatro estructuras se pueden ver en la siguiente figura, sin que aparezcan en la misma los bloques de entrada/salida.

1. Matriz simétrica, como son las de XILINX
2. Basada en canales, ACTEL
3. Mar de puertas, ORCA
4. PLD jerárquica, ALTERA o CPLD's de XILINX.

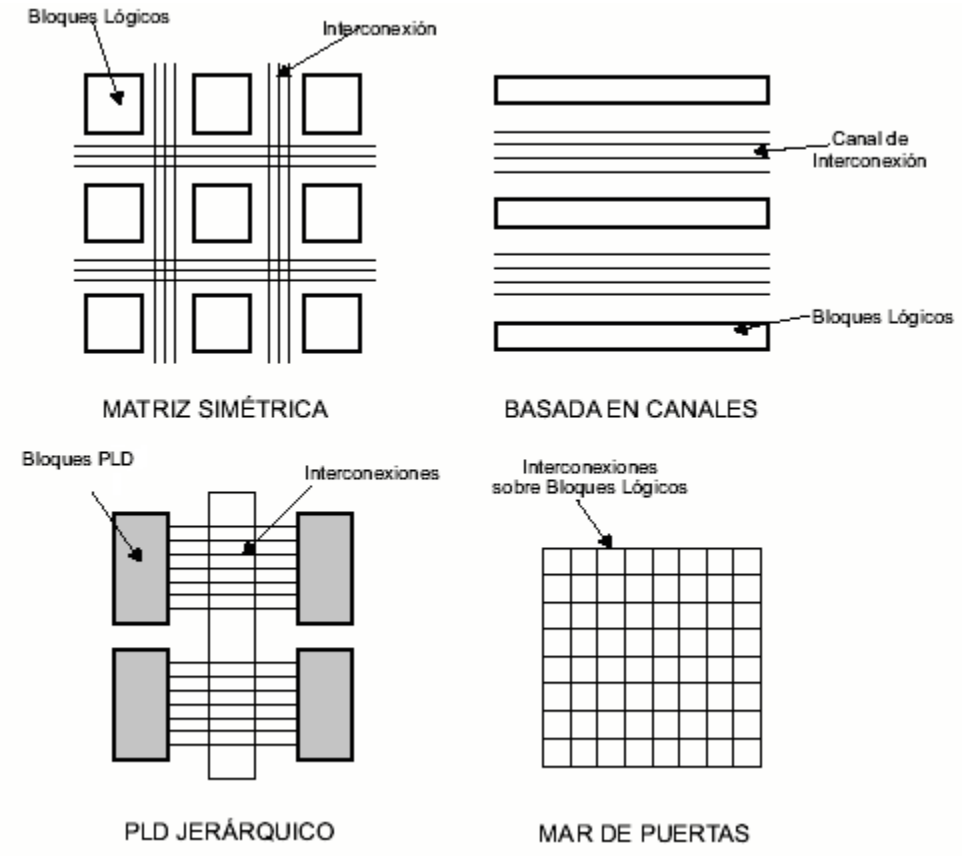

Figura 3. Distintas estructuras de FPGA.

En cuanto a la tecnología de programación hay que aclarar como se realiza el proceso (por ejemplo las conexiones necesarias entre bloques y pistas). En primer lugar, si se piensa que el número de dispositivos de conexión que hay en una FPGA es muy grande (típicamente superior a 100.000), es necesario que cumplan las siguientes propiedades:

- Ser lo más pequeños posible.
- Tener la resistencia ON lo más baja posible, mientras la OFF ha de ser lo más alta posible (para que funcione como conmutador).
- Se deben poder incorporar al proceso de fabricación de la FPGA.

El proceso de programación no es único, sino que se puede realizar mediante diferentes "tecnologías", como son células RAM estáticas, transistores EPROM y EEPROM, etc. En algunas FPGA's los elementos de programación se basan en células de memoria RAM que controlan transistores de paso, puertas de transmisión o multiplexores. En la figura a continuación se puede ver esquemáticamente como son, dependiendo del tipo de conexión requerida se elegirá un modelo u otro.

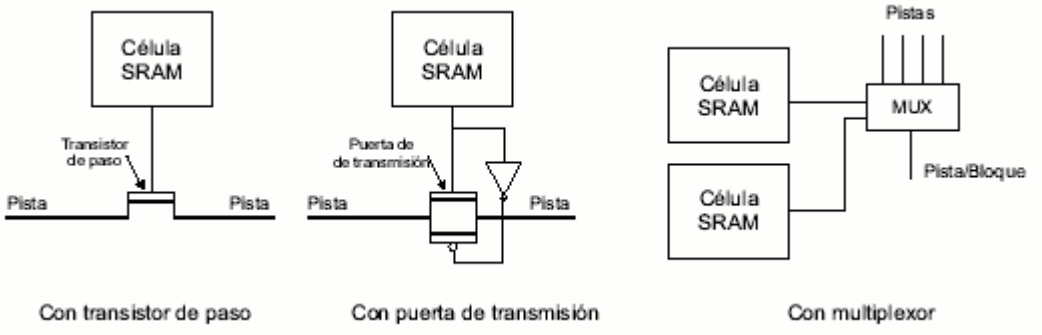

Figura 4. Diferentes células RAM.

Es importante destacar que si se utilizan células SRAM la configuración de la FPGA será válida únicamente mientras esté conectada la alimentación, pues es memoria volátil. En los sistemas finales está claro que hace falta algún mecanismo de almacenamiento no volátil que cargue las células de RAM. Esto se puede conseguir mediante EPROM's o disco. Este elemento de programación es relativamente grande (necesita por lo menos 5 transistores),

pero se puede implementar en el proceso normal de fabricación del circuito (es CMOS). Además, permite reconfigurar la FPGA de una forma muy rápida.

Diseño de un sistema de transmisión en banda base: Formato y sincronismo

En cuanto al formato de onda es posible utilizar formatos Manchester y Bipolar con Retorno a Cero. El receptor tiene la capacidad de autosincronizarse utilizando un algoritmo basado en el método Early-Late, con el cual se logra trabajar en condiciones de ruido y jitter considerables. La transmisión se efectúa en banda base, contemplando una posterior implementación, en un futuro proyecto, de lo relacionado a modulación y demodulación.

Aunque la frecuencia de transmisión es considerada constante, debido a ciertos factores, como las distintas desviaciones que pueden aparecer en los cristales del receptor y el transmisor, se debe trabajar con un receptor que tenga la capacidad de mantenerse sincronizado a pesar de los cambios en la frecuencia de transmisión anteriormente mencionados.

Cuando se dice que el receptor es autosincronizable se hace referencia al hecho de que es capaz de generar un clock de referencia, cuya fase es idéntica, excepto por un pequeño offset, a la fase de la señal enviada por el transmisor. La decisión sobre que valor lógico representa el símbolo recibido se toma a partir de un procedimiento que luego será analizado en detalle. Una vez tomada la decisión es necesario generar una cadena, que represente los datos recibidos, junto a un clock coherente en fase y frecuencia con ésta. Este tipo de sincronismo es llamado sincronización por símbolo.

Para poder generar esta señal de referencia, el receptor debe estar en sincronismo con la frecuencia de transmisión de la señal recibida. Ahora, utilizando los formatos unipolar y bipolar sin retorno a cero, si apareciera en la señal enviada una cadena grande de ceros o unos, se podría perder el sincronismo, para evitar esto se utilizaron formatos que siempre presentan al

menos un flanco por símbolo. En consecuencia, los formatos utilizados en el trabajo son el formato Manchester y el Bipolar con Retorno a Cero.

Detección y corrección

En cuanto a detección y corrección de errores el sistema cuenta con la posibilidad de detectar y corregir un error por palabra transmitida. La codificación está diseñada a partir de los códigos BCH que constituyen una clase de códigos de bloques lineales y cíclicos que aparecen como una generalización de los códigos de Hamming. Se incluye también un desarrollo de las técnicas de decodificación, como también sobre algunos aspectos teóricos relacionados.

Estudio de métodos de sincronismo

Para el diseño del mecanismo que provee el sincronismo primero se estudió las diferentes posibilidades y luego utilizando Matlab se logró desarrollar un nuevo método que basado en el llamado Early/Late proporciona un sincronismo con altos niveles de ruido y jitter. Este sincronizador trabaja por medio de correcciones en frecuencia y fase de acuerdo a un algoritmo que pondera una señal de error, símbolo a símbolo, y en base a estimaciones decide, independientemente en frecuencia y fase, las correcciones a realizar.

Otra característica importante del método ideado es que al procesar los símbolos entrantes para efectuar las correcciones anteriormente dichas, efectúa la lectura de los símbolos que van ingresando en el mismo procedimiento. Gracias a esto se obtiene un ahorro en el hardware utilizado, ya que lectura y sincronismo es un mismo procedimiento.

Por otra parte cabe destacar que la onda analógica que ingresa al receptor es primero muestreada por un CAD, debido a que su frecuencia de muestreo es constante, al variar la frecuencia de transmisión se ve modificada la cantidad de muestras que integran un símbolo. Es decir que por ejemplo si la

frecuencia de muestreo es 50 veces la frecuencia de transmisión, por cada símbolo enviado por el transmisor tendremos en el receptor un conjunto de 50 muestras que componen cada símbolo transmitido. Por lo tanto el algoritmo de sincronización trabaja con las muestras e intenta estimar el número de muestras que representa cada símbolo, lo cual es una forma de aproximarse a la frecuencia de transmisión.

Implementación en FPGA

Por limitaciones de espacio, ya que en principio se disponía de una Flex10k20 para el receptor y una Max9000 para el transmisor, hubo que trabajar con una trama compuesta por un patrón de 30 bits de unos y ceros alternados y 240 bits de datos (finalmente al armar el banco de pruebas, sucedió que no se disponía de una Max9000 sino de una Max7000, con lo cual hubo que conseguir otra FPGA, resultando ser ésta, otra Flex10k20). Estos 240 bits están compuestos por 16 palabras de 15 bits cada una. Esta configuración surge por conveniencia, ya que al codificador de errores le ingresan 7 bits en serie y produce 8 bits de codificación, con lo cual se genera la palabra de 15 bits. Inicialmente, el sistema transmisor-receptor, había sido diseñado para un tamaño de trama de 128 bits y con la posibilidad de escoger entre dos codificaciones distintas. Una agregaba 16 bits de control a continuación del bloque de bits de mensaje, y la otra 32 bits. Así, la trama final de transmisión quedaba compuesta por 30 bits de patrón más 128 bits de mensaje más 16 o 32 bits de control respectivamente. La ventaja de esta implementación radicaba en la posibilidad de variar la tasa del código en función de las condiciones de ruido de la comunicación. Esta implementación se presentará al final del informe.

Debido a que en este tipo de codificación la corrección de errores se hace muy compleja, se optó por fragmentar el paquete de datos en porciones que son codificadas generando los bits de control para cada una de ellas.

Decodificación

En cuanto a la decodificación, se diseñaron tres módulos distintos. El más importante es el decodificador que corrige un error, el cual se implementó adaptando un método que utiliza el algoritmo de Euclides y se desarrolla partiendo de los Campos de Galois, cuyo desarrollo teórico se incluirá previamente a su implementación en VHDL. Los otros dos solamente detectan errores y decodifican las tramas de 128 bits de mensaje y 16 o 32 bits de control.

Diseño

La modalidad de diseño se basó en dividir al sistema en distintos bloques, de acuerdo a sus características funcionales. El diseño de cada uno de estos bloques se desarrolló a nivel circuital, inicialmente se idearon y se bosquejaron a mano, componente a componente, para luego pasar a la programación en VHDL. El entorno VHDL presenta la posibilidad de hacer simulaciones, las cuales se realizaron en cada bloque por separado y luego en el sistema en todo su conjunto.

A continuación se describen brevemente los módulos que componen transmisor y receptor:

Transmisor:

- *DIVISORTX:* Simplemente divide el clock de entrada para obtener todas las relaciones de clock necesarias.

- *CODSERIEA15*: Este elemento recibe los datos desde la fuente en forma serial, los codifica y los presenta en su salida como palabras de 15 bits en paralelo. Estas palabras están compuestas por 7 bits de mensaje y 8 de codificación.

- *PARMEMBLOQUESERIE*: Recibe las palabras de 15 bits y las almacena en memoria hasta completar el número que compone una trama para luego agregar un encabezado de trama y enviarla en serie al próximo bloque. El encabezado esta compuesto por 30 unos y ceros alternados.

- *FORMAT*: Es el encargado de dar el formato de onda elegido, en forma digital.

- *CDA (Conversor Digital Analógico)*: Entrega en su salida una onda analógica relativa a la digital en su entrada.

Receptor:

- *CAD (Conversor Analógico Digital)*: Expresa en forma digital las ondas analógicas presentes en su entrada.

- *DIVISOR:* Igual función que el bloque *DIVISORTX* del transmisor.

- *COMPA2:* Convierte a complemento a2 la salida conversor.

- *HABILTRAMA*: Detecta el inicio y fin de la trama y genera una señal que habilita los procesos relacionados con la lectura de la trama.

- *MEM*: Almacena en memoria una cantidad variable de muestras (a partir de una estima de la duración de un bit) suministradas por el *CAD*. También permite el acceso, desde el bloque *ER*, a los datos, listos para ser procesados.

- *ER*: Es el encargado de procesar los datos que se encuentran en *MEM* y calcular la variable "e" (error del algoritmo de sincronismo). En el cálculo de "e" se obtiene la lectura del bit adquirido y su período correspondiente.

- *DPOSTOT*: A partir de "e" se calculan las variables "TOT" (número estimado de muestras que componen un bit, es relativo a la frecuencia de transmisión) y "DPOS" (corrección en fase).

- *COEFS*: Los bits que fueron leídos anteriormente por *ER* ingresan en serie para la decodificación y corrección de errores, palabra a palabra. Por último presenta en su salida los datos obtenidos.

Finalmente vale aclarar que los componentes externos a las FPGA's utilizados fueron únicamente los conversores D/A y A/D. Para el primero se utilizó el DAC0801, el cual es un conversor digital-analógico de alta velocidad de 8 bits, compatible con niveles TTL, CMOS, etc. Este conversor posee un gran rango de voltajes de salida (entre –10 y +18 Volts) y un error a fondo de escala de $\pm$ 1LSB, el cual es despreciable en este diseño, dado que solamente se trabajó con los dos bits más significativos. El costo de este integrado es bajo y sus prestaciones están por encima de los requerimientos mínimos necesarios del diseño, por esa razón fue elegido. Este último es un componente de singular importancia en el transmisor puesto que es el encargado de dar forma a la onda electromagnética que será transmitida. En cuanto al conversor analógico-digital hubo que optar por el modelo ADC0804 por cuestiones de existencia en las casas de componentes electrónicos. Este integrado se trata de un conversor por aproximaciones sucesivas, de 8 bits compatible con microprocesadores, ya que con sus salidas TRI-STATE es capaz de controlar un bus de datos directamente. Entre sus características principales se puede mencionar que posee un clock on-chip (es decir integrado) y una entrada para un clock externo, que será la configuración que se utilizará. Este clock externo debe tener un ciclo de entre 40 y 60% y su frecuencia se tiene que encontrar en el rango de 100-1460 Khz, para garantizar el funcionamiento correcto del conversor. Lamentablemente este conversor no posee un tiempo de conversión adecuado para el diseño a implementar, esto se verá más en detalle a continuación, pero por la no disponibilidad de otros integrados se utilizó este integrado.

DESARROLLO

Selección de método de sincronismo

Para diseñar el receptor se realizó una investigación sobre distintos métodos de sincronismo entre los cuales se diferencian los de lazo abierto y de lazo cerrado. Para elegir el más conveniente se trabajó con MATLAB y se realizaron simulaciones para los dos tipos de formato y distintas condiciones de ruido.

Sincronizador de lazo abierto

Primero se puso a prueba el *sincronizador de filtro no lineal*, un método a lazo abierto, el cual, por medio de distintos filtrados, limitaciones, etc. da como resultado un clock de la frecuencia fundamental de la onda entrante. Por medio de varias simulaciones bajo diferentes condiciones de jitter y ruido blanco gaussiano, se analizó y se llegó a concluir que no tenía la suficiente inmunidad a las condiciones ruidosas. Además al ser un sistema a lazo abierto no tiene forma de adaptar sus parámetros a las variaciones de la entrada, es decir que no puede responder de la misma forma a diferentes frecuencias de transmisión. La ventaja de este método es que es invariante para ambos formatos, lo cual sólo requeriría una única implementación, a diferencia del que se describe posteriormente.

Diagrama en bloques:

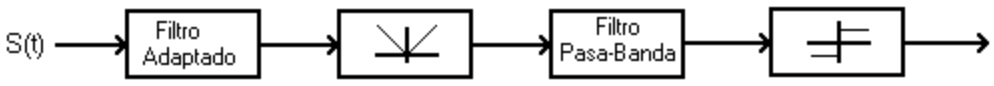

Figura 5. Sincronizador de filtro no lineal.

Los sincronizadores de símbolo a lazo abierto también suelen ser llamados *sincronizadores de filtro no lineal*. Esta clase de sincronizadores genera una componente de frecuencia similar a la frecuencia de transmisión de símbolo operando sobre la secuencia de entrada en banda base con una

combinación de filtrado y dispositivos alineales. En este caso la componente de frecuencia deseada, es aislada con un filtro pasabanda y luego transformada en una onda cuadrada, señal de clock de los datos, usando un amplificador saturador de alta ganancia.

La señal de entrada s(t) ingresa al filtro adaptado, la salida de este filtro será la función autocorrelación de la señal de entrada, ya que la respuesta al impulso del filtro tiene la forma del pulso de entrada. Por ejemplo para simbolización de onda cuadrada, la salida será de forma triangular. La secuencia de salida del filtro adaptado es luego rectificada por algún tipo de alinealidad de ley par sin memoria, por ejemplo un dispositivo de ley cuadrática. La forma de onda resultante tiene picos de amplitud positiva que corresponden a la transición de símbolo de entrada. Así la salida del dispositivo alineal posee una componente de Fourier en la frecuencia fundamental del clock de los datos. A continuación la forma de onda resultante de la alinealidad se ingresa a un filtro pasabanda cuya función es separar esta componente de sus frecuencias armónicas. Finalmente el último paso consiste en amplificar y saturar la onda senoidal para obtener una onda cuadrada de igual frecuencia.

Para el estudio de este método, utilizando Matlab, primero efectuamos la construcción de los filtros. Para el filtro adaptado no encontramos mayores problemas, dado que al trabajar con muestras, simplemente tabulamos un vector de igual cantidad de muestras que las que componen los bits a transmitir. Cada una de estas componentes tiene el valor “uno” lógico así conformando un filtro que tiene la misma forma que un pulso de datos. Para el filtro pasabanda se realizaron varias pruebas hasta que se llegó a la conclusión de que el filtro que mejor se ajusta a los requerimientos es un Chebyshev de orden 8, estos filtros se obtienen fácilmente en Matlab utilizando el comando “cheby1”.

Una vez elaborada la respuesta al impulso de los filtros, la salida se obtiene a partir de la operación convolución.

Luego para las alinealidades simplemente se utilizaron los comandos de Matlab “abs”, para rectificar (aplica valor absoluto) y “sign” (función signo) para generar la onda cuadrada. En el apéndice A se incluyen lo desarrollado en Matlab, programas, gráficos y conclusiones.

Para no extenderse demasiado en este método se dirá aquí que el sistema a lazo abierto tuvo un funcionamiento aceptable para el formato bipolar con retorno a cero, en situación de ruido medio, deteriorándose considerablemente a medida que este último aumentaba. Con el formato Manchester no se logró un sincronismo en condiciones ruidosas. Evaluando lo expuesto se arribó a la conclusión de que el método a lazo abierto no cumplió los requisitos que se esperaban.

Early-late

El siguiente método que se analizó fue el llamado Early-late que trabaja a lazo cerrado. Este es el esquema en su forma original:

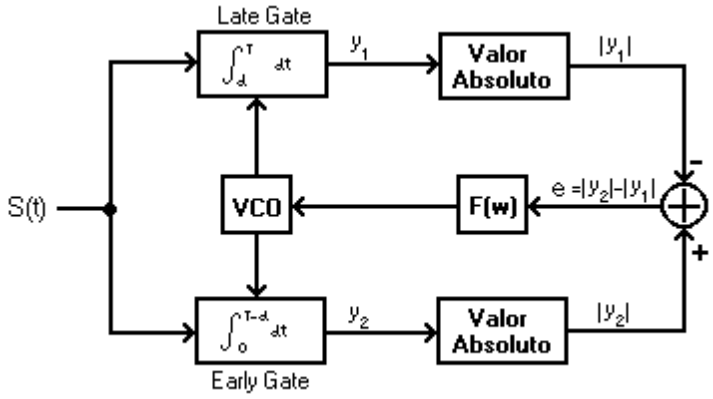

Figura 6. Sincronizador Early Late.

Los sincronizadores de símbolo a lazo cerrado utilizan mediciones comparativas entre la señal de entrada y una señal de clock, generada localmente, para sincronizar esta última con las transiciones de la primera. El sistema de sincronismo más popular entre los de lazo cerrado es el *sincronizador early/late-gate.* El sistema opera realizando dos integraciones separadas, de la energía de la señal entrante, sobre dos porciones diferentes de duración T-d segundos, siendo T la duración del símbolo. La

primera integración (la compuerta temprana, *Early gate*) empieza en la estimación del comienzo del símbolo (el tiempo nominal cero, t=0) e integra durante los siguientes T-d segundos. La segunda integral (la compuerta tardía, Late *gate*) retarda su comienzo hasta los d segundos y finaliza en el fin estimado del símbolo. La diferencia de los valores absolutos de los resultados de estas integraciones, y_1 e y_2, es una medida del error temporal del receptor, y puede ser realimentada, con el fin de controlar el sincronismo. La acción del sincronizador puede ser comprendida a partir de la siguiente figura:

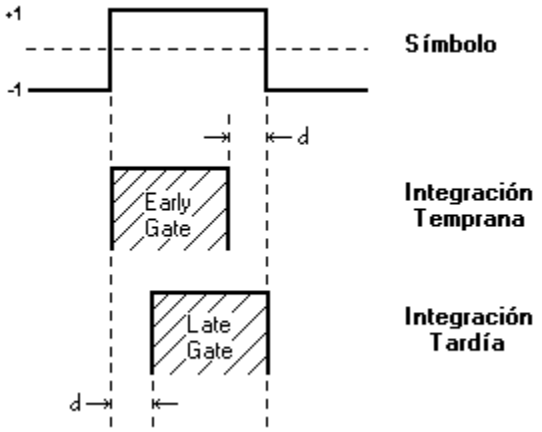

Figura 7. Intervalos de integración en sincronismo perfecto.

En el caso de sincronización perfecta la figura 7 muestra que ambas compuertas se encuentran completamente dentro de la duración del símbolo. En este caso ambos integradores acumularan las mismas áreas, y su diferencia (la señal de error, e) será cero. Así cuando el dispositivo está en sincronismo, es estable y no hay tendencia a perderlo. En el caso de la siguiente figura, el clock de datos del receptor se adelanta respecto del dato de entrada:

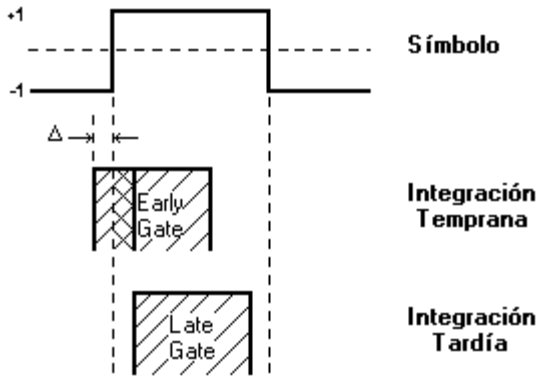

Figura 8. Integración con desfasaje.

Una porción de la compuerta temprana integra en el bit previo (el intervalo representado por Δ), mientras que la compuerta tardía está enteramente dentro del símbolo actual. El integrador de compuerta tardía acumulará señal sobre todo su intervalo de integración T-d como en el caso anterior. Por otra parte, el integrador de compuerta temprana finalizará con un área acumulada equivalente sólo al intervalo (T-d) -2Δ, dado que en este caso el bit previo es negativo. Así, para este caso la señal error será e = -2Δ, el cual va a bajar el voltaje de entrada al VCO. Esto reducirá la frecuencia de salida del VCO y retardará el inicio de la integración en el próximo bit. Análogamente si la señal de error tiene signo positivo, la frecuencia de salida del VCO aumentará y esto provocará un adelanto en el inicio del la próxima integración.

En el ejemplo anterior se asume que habrá cambios de estado en los datos, antes y después del símbolo de interés. Si no existen transiciones puede verse que ambas compuertas integrarán áreas iguales, por lo tanto la señal de error será cero mientras no haya cambios de estado en los datos de entrada. Esto debe considerarse a la hora de implementar este mecanismo en la práctica. La respuesta más obvia a este problema consiste en utilizar un formato a partir del cual se asegure que no existan intervalos sin transición lo suficientemente largos como para causar una pérdida en el sincronismo. Otra consideración en la implementación es el tamaño de las compuertas dado que esto afectará la magnitud de la señal de error resultante.

Nuevamente para el estudio sobre la implementación de este método se utilizó el Matlab, por lo tanto el primer paso fue diseñar el lazo compuesto por los integradores, el filtro y el VCO, básicamente un PLL. Dado que la señal de entrada es muestreada al ingresar al receptor se decidió trabajar con las muestras. Esto quiere decir que la duración de cada bit está representada por un número entero de muestras. Ya que la frecuencia de muestreo es constante, una variación en la frecuencia de transmisión repercutirá directamente en el número de muestras que representan un bit, por lo tanto esa cantidad de muestras representa la

frecuencia de transmisión. Entonces el tamaño de las compuertas de integración está definido simplemente por una cantidad de muestras. Esta cantidad es una función de la señal de error, en consecuencia la rama en que se encuentran el filtro y el VCO (que figura en el diagrama en bloques) en el diseño se remplazó por un algoritmo que en función de la magnitud del error, directamente proporciona a las compuertas sus intervalos de integración. Cabe mencionar que los dos formatos elegidos, Manchester y Bipolar con retorno a cero, aseguran al menos una variación de estado por símbolo, esto nos proporciona una solución al problema de la falta de transiciones en los símbolos, comentado anteriormente.

Primera implementación

Para implementar los integradores se utilizó la función de Matlab "cumsum", la cual efectúa una suma acumulativa del valor de las muestras que ingresan. La cantidad de muestras que componen un símbolo inicialmente es un valor arbitrario (50 muestras), identificado por una variable llamada "TOT", luego es actualizada sucesivamente a partir del valor del error obtenido para cada símbolo procesado. Por lo tanto en cada símbolo, al calcular "TOT", se obtiene una estima de la duración del siguiente. Luego para hallar los límites de integración simplemente se calcula un porcentaje del tamaño de la duración del bit. En este caso se tomó un área de integración del 80% de la duración estimada, es decir que la compuerta temprana actúa entre el comienzo estimado del bit, la primer muestra, y la muestra correspondiente al cálculo 0.8 TOT, mientras que la tardía comienza a actuar en la 0.2 TOT y finaliza en TOT. En la siguiente figura se puede apreciar mejor:

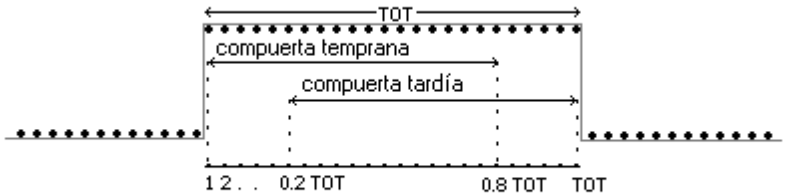

Figura 9. Intervalos de integración en relación a las muestras.

En el primer desarrollo que se efectuó para implementar este algoritmo, la primer muestra era la que estaba a continuación de la última utilizada para el proceso del símbolo previo. En el APENDICE B se estudia este método, y se presentan los programas empleados. Debido a que el valor de “TOT” es una función del error, se ideó un coeficiente de proporcionalidad entre ambos. Para determinar cual era el valor más apropiado para este coeficiente, hubo que hacer sucesivas simulaciones.

En forma similar al método de lazo abierto, se encontró que el sistema podía funcionar sin ruido ni jitter, pero en cuanto estas condiciones eran modificadas, se veía incapaz de mantener el sincronismo.

Como conclusión, se destacan dos aspectos:

- En primer lugar el sistema lograba obtener un valor de periodo de símbolo muy cercano al real, lo que representaría una buena aproximación a la frecuencia de transmisión. Pero ocurría que se necesitaba encontrar alguna forma de corregir la fase, que se acumulaba progresivamente, para evitar la pérdida de sincronismo. Para efectuar la corrección en fase, se precisaba extraer mayor información de la variable del error, “e”.

- En segundo lugar, se pudo apreciar que realizar la lectura en el centro estimado del bit no era lo más adecuado, ya que en ciertas ocasiones, si bien el sistema no perdía el sincronismo, se hacían lecturas erradas. Téngase en cuenta que a medida que las condiciones de ruido y jitter se extreman, es inevitable que el sistema pase momentáneamente por desfasajes considerables.

Implementación definitiva

Por último se desarrolla el método que finalmente logró satisfacer los requerimientos que se buscaban. Este está basado en el algoritmo Early-Late que se ha descrito anteriormente. Para lograr un buen funcionamiento,

hubo que hacer algunas modificaciones. Como principales diferencias se destacan las siguientes:

- La señal de entrada es convolucionada con la forma estimada de un pulso. Esta señal resultante es la que se integra a través de las compuertas. Al convolucionar, se logra una mayor inmunidad al ruido.

- Se desarrolló un control de fase (la variable “DPOS”) que mejoró notablemente el desempeño del sincronizador.

- La frecuencia ya no es controlada exclusivamente por la señal de error, sino por una combinación con la derivada discreta del error.

- La corrección en fase es proporcional a la señal de error.

- Se procesa un primer símbolo, de duración y comienzo arbitrario. Como resultado de ese proceso se obtiene la estimación de la duración y comienzo del siguiente símbolo.

- En términos generales, la lectura se realiza a partir de la convolución de la señal con el filtro adaptado.

Para abordar el análisis de este método, se debe tener en cuenta que la trama de entrada se va fragmentando en partes que son la estimación de lo que sería un símbolo (estas particiones se componen de un número de muestras, representado por “TOT”). El método, no sólo va variando el largo de esta estimación, sino que también cambia el inicio de esta fragmentación. Esto significa que el grupo de muestras asociado al próximo símbolo que será procesado, no comenzará necesariamente justo a continuación de la última muestra del grupo asociado al símbolo anterior. Se llega a utilizar muestras del símbolo previo estimado en caso de un “DPOS” negativo, o se pueden dejar algunas sin utilizar si “DPOS” es positivo. A continuación se puede observar mejor gráficamente:

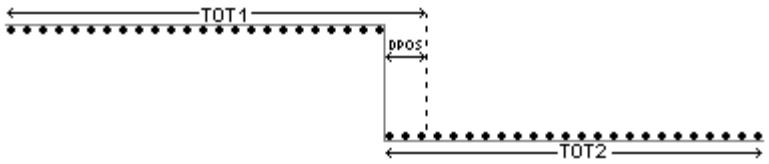

Figura 10. Corrección total en fase por medio de DPOS.

Aquí se ve como la estimación de la duración del primer símbolo fue mayor que la correcta, luego esto se corrigió adelantando el procesamiento del siguiente símbolo. Como se observa, el comienzo y el largo estimado del segundo símbolo coinciden con el real. Ahora se analizará otro caso:

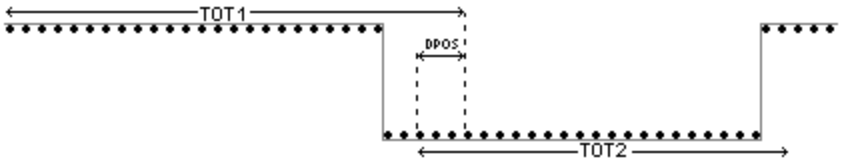

Figura 11. Corrección parcial en fase por medio de DPOS.

En la figura anterior se aprecia como si bien el origen del segundo símbolo se corrió en el sentido correcto, este corrimiento no fue de la magnitud suficiente. Así que, aunque la duración estimada del segundo símbolo es la correcta, el siguiente bit todavía deberá corregir un cierto desfasaje.

A continuación podemos ver un diagrama en bloques del sistema completo:

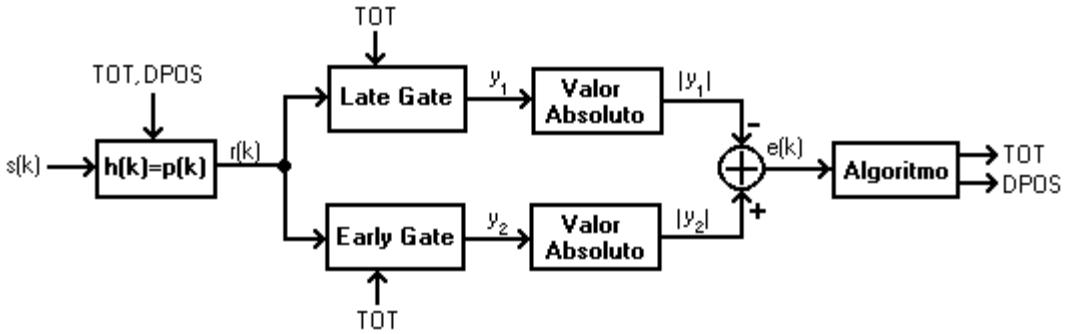

Figura 12. Diagrama en bloques del sincronizador desarrollado.

La señal de entrada ingresa al primer bloque h(k), que tiene como respuesta al impulso un pulso de amplitud unitaria y largo función de “TOT”.

La salida de este bloque, r(k), es la convolución discreta de la entrada y el pulso anteriormente descrito. Esta última es integrada por las compuertas temprana y tardía. El ancho de las compuertas fue adoptado como 0.8 TOT, ejemplificando esto es un valor de 40 muestras para un valor de TOT=50 (80%). Así, la primer compuerta integra entre la muestra 1 y la 40, y la segunda compuerta lo hace entre la muestra 11 y la 50. Luego las salidas de ambas compuertas ingresan a los bloques de valor absoluto, para luego obtener la señal de error, mediante la resta. Esta señal ingresa al bloque denominado “Algoritmo” donde se obtienen las señales que se realimentarán, las denominadas “TOT” y “DPOS”. Para la mejor comprensión del funcionamiento de este último bloque se adjunta el siguiente diagrama de flujo:

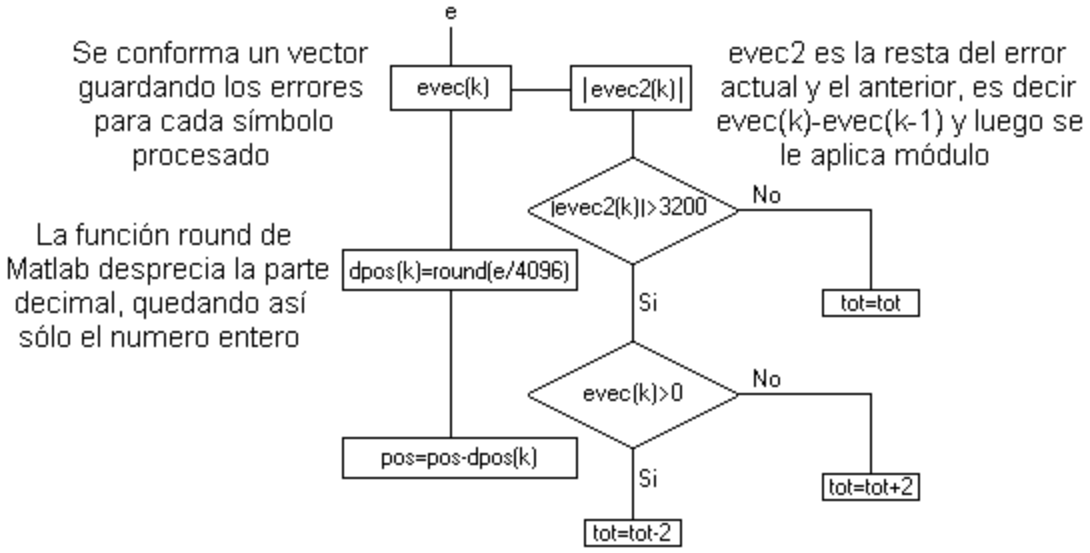

Figura 13. Diagrama de flujo del algoritmo de control.

Entonces a partir del ingreso de la señal de error se crean dos vectores, uno con todos los errores resultantes del procesamiento de los símbolos, y otro con el resultado de la resta del error del procesamiento del símbolo actual y el error del símbolo previo. Con el valor de cada componente de los vectores anteriores se toman decisiones en cuanto a las correcciones de fase (DPOS) y frecuencia (TOT).

La variable “TOT” crece o decrece dos unidades, este valor de corrección está relacionado (en este caso es un 4%) con el valor de “TOT”

inicial que es de 50 muestras. Esta relación se eligió a partir de realizar distintas pruebas.

Por otro lado, DPOS es proporcional al error, la constante de proporcionalidad (1/4096) surge también de realizar distintas pruebas, pero además bajo la condición de tener que ser potencia de 2 (el denominador). Esto facilita la implementación de la división, que se efectúa haciendo desplazamientos en los registros. En el APENDICE C se adjunta el programa utilizado. Se muestran los dos tipos de filtro adaptado usados, en función del tipo de formato.

Habiéndose interiorizado en el método de sincronismo elegido, se comenzará ahora a explicar el funcionamiento de los distintos bloques que componen la implementación en VHDL.

Transmisor

Codificación

El primer bloque que compone el transmisor es “CODSERIEA15”. Este bloque recibe los datos a transmitir en serie (“Datain”), junto a un clock de transmisión serial (“Clock”) y una señal de habilitación (“habil”). Este módulo cuenta con un codificador cíclico (“codcicli”) que toma 7 bits de datos en serie y genera 8 bits de control. A partir de un contador interno (“cont1”), deja ingresar los 7 bits de datos y a una velocidad mucho mayor (dada por “Clockmax”) los procesa generando los 8 bits de control. Una vez generados los bits de control, coloca los 15 bits en paralelo (7 datos+8 control) en un arreglo de flip flops (“FF3”) a la salida. A continuación lee otros 7 bits, vuelve a procesar y renueva el contenido de los flip flops de salida con otros 15 bits. Junto a esta palabra, el bloque produce también un clock de velocidad de transmisión de palabra (“Clockout”).

Este procedimiento se reitera hasta que la señal de habilitación indique el fin de trama. Su diagrama en bloques es el siguiente:

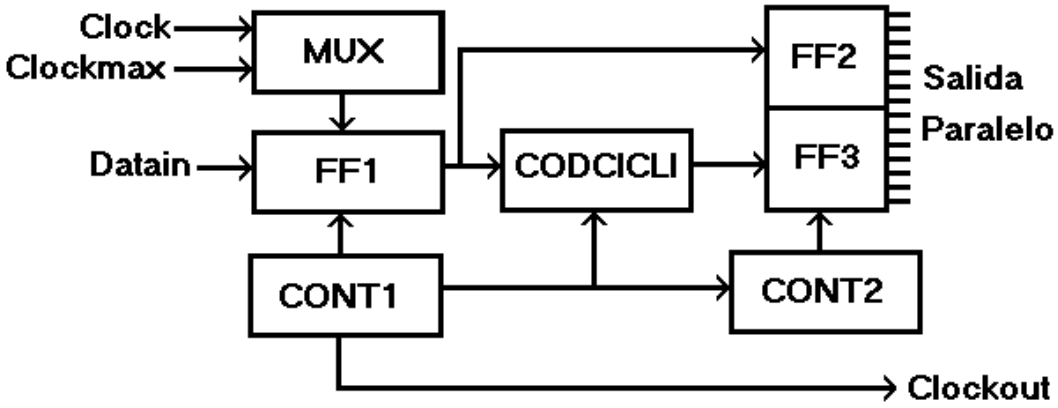

Figura 14. Diagrama en bloques de CODSERIEA15.

El multiplexor es el encargado de proveer el clock necesario a “FF1”. Hasta que no hayan entrado los 7 bits, “FF1” se maneja con la señal “Clock”, luego necesita un clock mucho más veloz. El contador “CONT1” cuenta estos 7 bits y entonces habilita a “CONT2” y al codificador, cambia la

selección del multiplexor para que ingrese “Clockmax” a “FF1” y vuelca el contenido de este último a “FF2”. Luego es “CONT2” el que determina el fin de codificación, a partir de contar el ingreso de los 7 bit contenidos en “FF1” en el codificador. Finalmente, “CONT2”, habiendo determinado el fin de codificación, genera una señal que permite el vuelco de los bits generados por el codificador (que se encuentran en un flip flop interno) en “FF3”. Esto se ve en la figura siguiente:

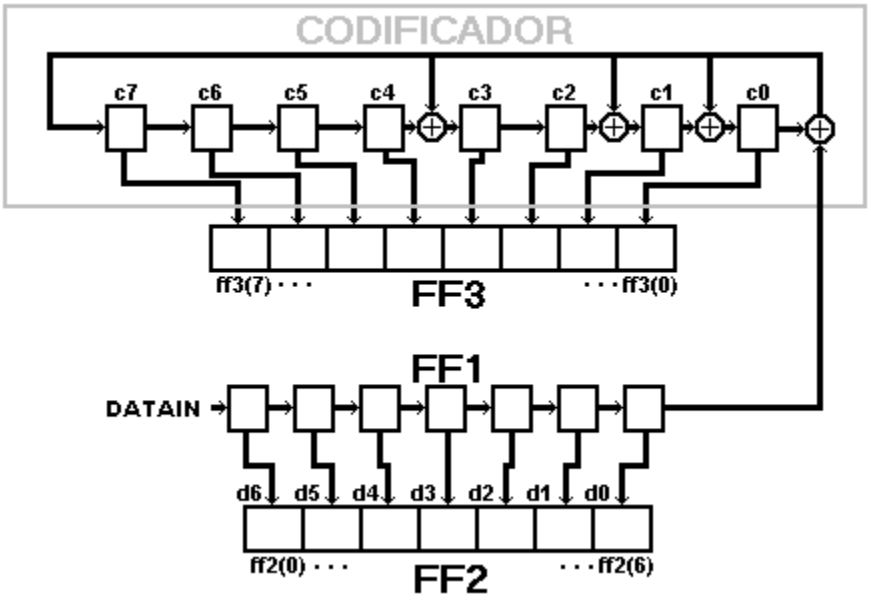

Figura 15. Diagrama del mecanismo de codificación.

La salida está compuesta por el contenido de “FF2” y el de “FF3” dispuestos en paralelo, en total 15 bits. Si bien “FF2” tiene los datos disponibles antes que “FF3”, no se utilizó un flip flop adicional para volcar los 15 bits, ya que en relación a la velocidad paralelo en que cambian los 15 bits ese retardo es despreciable. Siempre será despreciable si se cumple que fclockmax >> fclock.

Los 15 bits quedan agrupados de la siguiente forma:

```
Dataout(0)  <=d0
Dataout(1)  <=d1
Dataout(2)  <=d2
Dataout(3)  <=d3
Dataout(4)  <=d4
Dataout(5)  <=d5
Dataout(6)  <=d6
```

```
Dataout(7)  <=c0
Dataout(8)  <=c1
Dataout(9)  <=c2
Dataout(10) <=c3
Dataout(11) <=c4
Dataout(12) <=c5
Dataout(13) <=c6
Dataout(14) <=c7
```

A continuación se adjunta el programa de VHDL:

```
LIBRARY altera;                    --Librería de Altera
USE altera.maxplus2.ALL;           --Las liberarías permiten utilizar unidades almacenadas
LIBRARY ieee;                      --Librería de IEEE
USE ieee.std_logic_1164.ALL;
LIBRARY lpm;                       --Librería de mega funciones lógicas
USE lpm.lpm_components.all;


ENTITY codseriea15 IS                  --Aquí se declara la entidad, o sea el bloque codseriea15,
PORT                                   --con sus puertos de entrada y salida, y el tamaño de éstos
(Datain,clockmax,clock,clockff,habil       : IN  STD_LOGIC;                --Entradas
 clockout                                  : OUT STD_LOGIC;                --Salidas
 Dataout                                   : OUT STD_LOGIC_VECTOR(14 DOWNTO 0));

END codseriea15;

ARCHITECTURE arc OF codseriea15 IS     --En la arquitectura de la identidad, se declaran
                        --los distintos componentes utilizados (en forma genérica)  junto a sus puertos,

COMPONENT lpm_ff                                    --Se declaran los flip flops
GENERIC (LPM_WIDTH: POSITIVE);                      -- tamaño del arreglo de ffs
      PORT (data: IN STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0); -- entrada del ff
    clock: IN STD_LOGIC;                                      -- clock del ff
    aclr: IN STD_LOGIC := '0';                          -- clear asincrónico del ff
     q: OUT STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0)); -- salida del ff
END COMPONENT;

COMPONENT lpm_counter                               --Se declaran los contadores
 GENERIC (LPM_WIDTH: POSITIVE);                     -- tamaño del contador
 PORT ( clock: IN  STD_LOGIC;                       -- clock del contador
        aclr: IN STD_LOGIC := '0';                  -- clear asincrónico del contador
        clk_en: IN STD_LOGIC := '1';                -- habilitación de la entrada de clock
              q: OUT STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0)); -- salida del contador
END COMPONENT;
COMPONENT monoestable                  --Se declaran los monoestables, diseñados por
                          --nosotros, es otra identidad, que está en el mismo directorio
PORT (Datain,clock              : IN  STD_LOGIC;
          Dataout           : OUT STD_LOGIC);
END COMPONENT;

COMPONENT lpm_mux                      --Se declaran los multiplexores
 GENERIC (LPM_WIDTH: POSITIVE;         -- tamaño entrada
    LPM_WIDTHS: POSITIVE;              -- tamaño de selección de entradas
    LPM_SIZE: POSITIVE);               -- nro de entradas
PORT (data: IN STD_LOGIC_2D(LPM_SIZE-1 DOWNTO 0, LPM_WIDTH-1 DOWNTO 0);
    sel: IN STD_LOGIC_VECTOR(LPM_WIDTHS-1 DOWNTO 0);
    result: OUT STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0));END COMPONENT;

COMPONENT codeciclicirc8   --Se declara el codificador cíclico diseñado previamente por
                           --nosotros (se muestra mas adelante)
PORT (Datain,clockmax,habilserie    : IN    STD_LOGIC;
    Doutpar                 : OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
END COMPONENT;
```

```
                --en esta sección se declaran las señales, y sus tamaños, que actuarán en la
                – identidad, estas son entradas y salidas de los distintos componentes.
signal mux1in: STD_LOGIC_2D(1 DOWNTO 0, 0 DOWNTO 0);
signal codoutpar,dff3,qff3: STD_LOGIC_VECTOR(7 DOWNTO 0);
signal dff3,qff3,dff1,qff1,dff2,qff2: STD_LOGIC_VECTOR(6 DOWNTO 0);
signal sal1: STD_LOGIC_VECTOR(3 DOWNTO 0);
signal sal2: STD_LOGIC_VECTOR(2 DOWNTO 0);
signal mux1out,mux1sel,dff6,qff6,dff5,qff5: STD_LOGIC_VECTOR(0 DOWNTO 0);
signal clock1,stcod,fincod,codin,habilserie,clockff1,clockff2,clockff3,clockff6,
aclrff1,aclrff2,aclr1,clken1,clken2,aclr2,monin,monout: STD_LOGIC;


BEGIN      –aquí comienza el programa principal, que también esta contenido en la arquitectura

          --a continuación se especifican los componentes que se utilizarán, definiendo nombre de
--componente, tamaño, nombre de pines, etc., es decir, dando valores concretos a los campos que
--se habían expresado en forma genérica al comienzo de la arquitectura

cont1: lpm_counter GENERIC MAP (4)        --Contadores
PORT MAP (clock1,aclr1,clken1,sal1);      --Entradas y salidas
cont2: lpm_counter GENERIC MAP (3)        --véase que a partir de la declaración de un tipo de
PORT MAP (clockmax,aclr2,clken2,sal2);    --componente, luego se pueden declarar varios
                                          --componentes y de características independientes
ff1: lpm_ff GENERIC MAP (7)               --Flip Flops
PORT MAP (dff1,clockff1,aclrff1,qff1);    --Entradas y salidas
ff2: lpm_ff GENERIC MAP (7)
PORT MAP (dff2,clockff2,aclrff2,qff2);
ff3: lpm_ff GENERIC MAP (8)
PORT MAP (dff3,clockff3,aclrff1,qff3);
ff5: lpm_ff GENERIC MAP (1)
PORT MAP (dff5,clockff,aclrff1,qff5);
ff6: lpm_ff GENERIC MAP (1)
PORT MAP (dff6,clockff6,aclrff1,qff6);

mux1:lpm_mux GENERIC MAP (1,1,2)          --Multiplexor
PORT MAP (mux1in,mux1sel,mux1out);        --Entradas y salidas

cod8:codeciclicirc8                                  --Codificador
PORT MAP (codin,clockmax,habilserie,codoutpar);      --Entradas y salidas

mono1: monoestable                        --Monoestable
PORT MAP (monin,clockmax,monout);         --Entradas y salidas

proceso: process      --En esta sección se asocian todos los pines de los distintos componentes

begin

--cod8
codin<=qff1(6);
habilserie<=sal1(3);

--cont1
clken1<=habil and (not stcod);
aclr1<=qff6(0);
clock1<=clock and (not monout);

--cont2
clken2<=stcod;
aclr2<=qff6(0);

--ff1
dff1(0)<= Datain;
dff1(1)<= qff1(0);
dff1(2)<= qff1(1);
dff1(3)<= qff1(2);
dff1(4)<= qff1(3);
dff1(5)<= qff1(4);
```

```
dff1(6)<= qff1(5);
clockff1<=qff5(0);
dff5(0)<=mux1out(0) and habil;
aclrff1<=qff6(0);

--ff2
dff2<= qff1;
clockff2<=sal1(3);
aclrff2<='0';

--ff3
dff3(0)<= codoutpar(7);
dff3(1)<= codoutpar(6);
dff3(2)<= codoutpar(5);
dff3(3)<= codoutpar(4);
dff3(4)<= codoutpar(3);
dff3(5)<= codoutpar(2);
dff3(6)<= codoutpar(1);
dff3(7)<= codoutpar(0);
clockff3<=habil and (not sal1(3));

--ff6
dff6(0)<=sal2(2)and sal2(1);
clockff6<=clockmax;

--mono1
monin<=not (sal1(2) or sal1(3));

--mux1
mux1in(1,0) <= clockmax;
mux1in(0,0) <= clock;
mux1sel(0) <= stcod;

--Salidas
stcod<=sal1(3);
fincod<=qff6(0);
clockout<=sal1(2) or sal1(3);
Dataout(0) <= qff2(6);
Dataout(1) <= qff2(5);
Dataout(2) <= qff2(4);
Dataout(3) <= qff2(3);
Dataout(4) <= qff2(2);
Dataout(5) <= qff2(1);
Dataout(6) <= qff2(0);
Dataout(7) <= qff3(0);
Dataout(8) <= qff3(1);
Dataout(9) <= qff3(2);
Dataout(10) <= qff3(3);
Dataout(11) <= qff3(4);
Dataout(12) <= qff3(5);
Dataout(13) <= qff3(6);
Dataout(14) <= qff3(7);

end process;
END arc;
```

Como ya se mencionó, el codificador cíclico, es un componente dentro de la identidad “CODSERIEA15”. Para comprender mejor el funcionamiento del codificador cíclico vease la evolución de sus registros en función de un determinado vector de entrada E:

E								Despl		R							
1	1	0	0	1	0	1		0		0	0	0	0	0	0	0	0
	1	1	0	0	1	0		1		1	0	0	0	1	0	1	1
		1	1	0	0	1		2		1	1	0	0	1	1	1	0
			1	1	0	0		3		1	1	1	0	1	1	0	0
				1	1	0		4		0	1	1	1	0	1	1	0
					1	1		5		0	0	1	1	1	0	1	1
						1		6		0	0	0	1	1	1	0	1
						-		7		0	0	0	0	1	1	1	0

Tabla 1. Codificación de la palabra "1100101".

Entonces la palabra que el codificador presenta en su salida es "01110000", luego en el resto del bloque se ensambla la parte del mensaje, quedando así la palabra de salida final de este bloque de la siguiente manera: "011100001100101". Donde los 7 bits menos significativos son el mensaje y los siguientes 8 son los bits de control agregados por el codificador.

Ahora se analizará el programa del codificador cíclico:

```
--Entrada en serie(15,7)
--1 + x4 + x6 + x7 + x8      es el polinomio generador
--Aclaración: en adelante se obviarán las declaraciones de librerías, ya que son idénticas a la
--realizada en el programa anterior, de acá en más el resto de los programas se adjuntarán
--en el Apéndice F. El próximo se explicará brevemente.

ENTITY codeciclicirc8 IS              --Declaración de la identidad del codificador
        PORT
        (Datain,clockmax,habilserie     : IN  STD_LOGIC;
         Doutpar                        : OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
END codeciclicirc8;

ARCHITECTURE arc OF codeciclicirc8 IS
COMPONENT lpm_ff                                  --Declaración de FF's
        GENERIC (LPM_WIDTH: POSITIVE);
      PORT (data: IN STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0);
    clock: IN STD_LOGIC;
    aclr: IN STD_LOGIC := '0';
     q: OUT STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0));
END COMPONENT;

COMPONENT lpm_mux                                 --Declaración de MUX
 GENERIC (LPM_WIDTH: POSITIVE;
    LPM_WIDTHS: POSITIVE;
    LPM_SIZE: POSITIVE);
PORT (data: IN STD_LOGIC_2D(LPM_SIZE-1 DOWNTO 0, LPM_WIDTH-1 DOWNTO 0);
    sel: IN STD_LOGIC_VECTOR(LPM_WIDTHS-1 DOWNTO 0);
    result: OUT STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0));

--Declaración de las señales y su tamaño.
signal and1in,mux1in: STD_LOGIC_2D(1 DOWNTO 0, 0 DOWNTO 0);
signal dff1,qff1: STD_LOGIC_VECTOR(7 DOWNTO 0);
signal mux1out,dff2,qff2,mux1sel: STD_LOGIC_VECTOR(0 DOWNTO 0);
signal clkff1,aclrff1,aclrff2,clk2,aclr1,clken1,clken2,aclr2,g0: STD_LOGIC;
```

```
BEGIN
ff1: lpm_ff GENERIC MAP (8)
PORT MAP (dff1,clockmax,aclrff1,qff1);
ff2: lpm_ff GENERIC MAP (1)
PORT MAP (dff2,clockmax,aclrff2,qff2);
mux1:lpm_mux GENERIC MAP (1,1,2)
PORT MAP (mux1in,mux1sel,mux1out);
generar : process
begin
g0<=(datain xor qff1(7))and habilserie;

--ff1
dff1(0) <= g0;          --aquí vemos como se conectan los flip flops internos del codificador
dff1(1) <= qff1(0);
dff1(2) <= qff1(1);
dff1(3) <= qff1(2);
dff1(4) <= qff1(3) xor g0;
dff1(5) <= qff1(4);
dff1(6) <= qff1(5) xor g0;
dff1(7) <= qff1(6) xor g0;
aclrff1<=not habilserie;

--ff2
dff2(0)<=mux1out(0);
aclrff2<='0';

--mux1
mux1in(1,0) <= qff1(7);
mux1in(0,0) <= datain;
mux1sel(0) <= not habilserie;

--Salida
Doutpar <= qff1;
end process;
END arc;
```

Parmembloqueserie

El siguiente bloque es “PARMEMBLOQUESERIE”. Este consta de dos memorias. Cada memoria tiene una capacidad de 16x15, esto quiere decir que almacena 16 palabras de 15 bits cada una. El tamaño de la palabra está condicionado por el tipo de codificador cíclico utilizado. En este caso la palabra es de 15 bits ya que está compuesta por 8 bits de control y 7 bits de datos. Para interpretar su función, se describe el proceso desde el principio de trama:

A medida que se habilita el proceso, las palabras ingresan a la primera memoria, que es direccionada por un contador de escritura (“CONT1”), cuando ésta se completa, se empieza a llenar la segunda. En paralelo al llenado de la segunda memoria se envía, en serie, un patrón de 15 “1” y 15 “0” alternados seguido del contenido de la primera memoria.

Esto se realiza a una velocidad mucho mayor que la velocidad de transmisión paralelo (de palabra). Téngase en cuenta que en un tiempo menor o igual al tiempo que se tarda en recibir 16 palabras en paralelo, se deben enviar 30 bits de patrón más 240 bits (16x15) de mensaje, o sea 270 bits. Para leer los bits de las memorias se usa, un contador (“CONT6”) para direccionar las palabras y un multiplexor para sacar bit a bit cada palabra. Hay otro contador (“CONT5”) más rápido de módulo 15 (tamaño de la palabra) que actúa como selección del multiplexor (“MUX4”) y a su vez provee el clock para el otro contador (“CONT6”), ya que recién cuando todos los bits son leídos, se puede incrementar el contador que direcciona las palabras. Para comprender el funcionamiento de este bloque véase la siguiente figura:

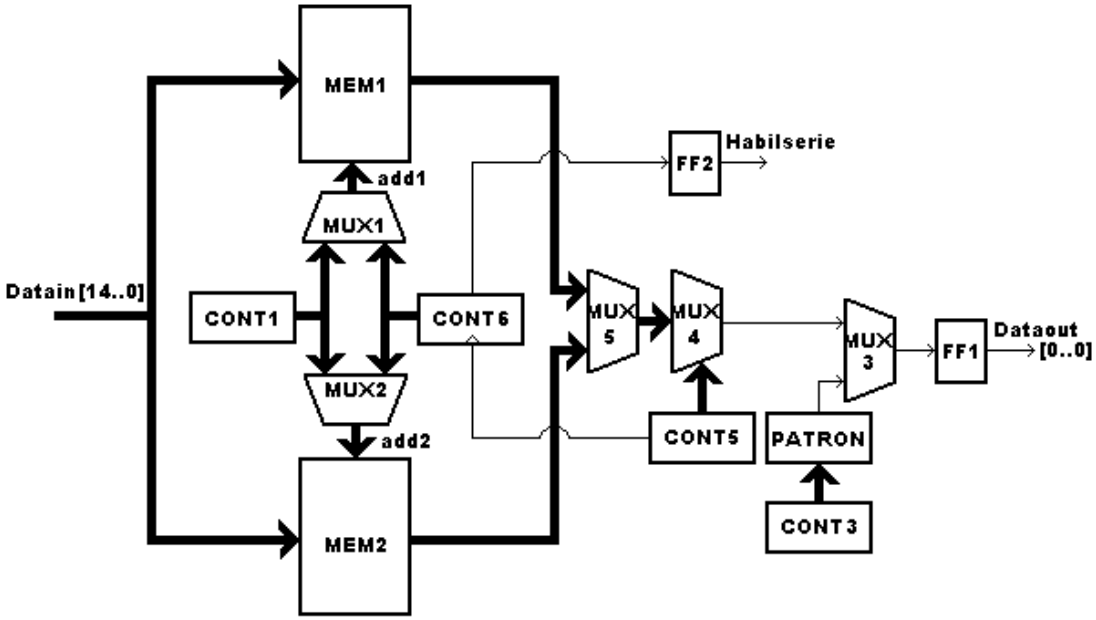

Figura 16. Esquema de Parmembloqueserie.

Aquí se ve como cada memoria es direccionada por distintos contadores (CONT1 o CONT6) de acuerdo a su estado de escritura o lectura. La señal que define el estado de escritura o lectura es la misma que va a la selección de los “MUX1” y “MUX2”. Cuando en una memoria se escribe, se la direcciona con el contador de escritura (“CONT1”) y cuando se lee, con el de lectura (“CONT6”). A su vez “MUX5” es manejado por “CONT1” de forma tal que cuando se escribe en una memoria, “MUX5” presente a su salida la salida de la otra memoria. Cuando se está

escribiendo “MEM1”, la señal de selección de “MUX1” hace que sea direccionada con “CONT1”. “CONT3” cuenta la cantidad de bits de patrón, y es el encargado de manejar la selección de “MUX3”, el cual, una vez enviado el patrón, cambia de entrada conectando “MUX4” a la salida.

Es importante tener en cuenta que al mismo tiempo en que se inserta el patrón y se lee bit a bit una de las memorias, se pone en estado alto la señal “Habilserie” que es la encargada de avisar al próximo bloque de la presencia de una trama lista para ser transmitida. En el Apéndice F, Programa 1, se encuentra su implementación en VHDL.

Formato

Una vez que se tiene la trama en serie, se aplica el formato. A través de una entrada externa (“Manbip”) se selecciona el tipo de formato deseado. Las opciones disponibles son, formato Manchester y formato Bipolar con retorno a cero.

El conversor DAC que se utilizó es de 8 bits y trabaja entre los valores +Vcc, y –Vcc. En la figura se muestran los distintos niveles de tensión:

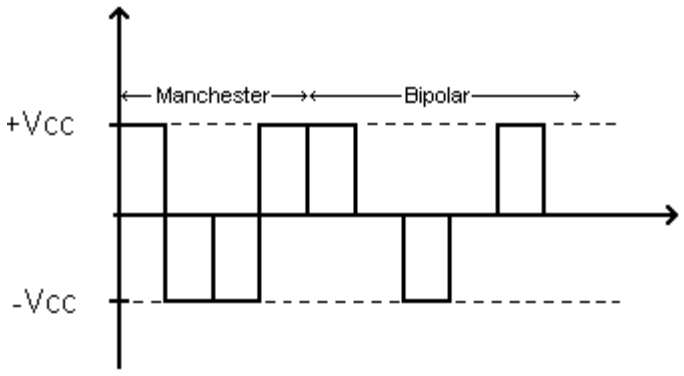

Figura 17. Niveles de tensión para los formatos Manchester y Bipolar.

En la tabla del conversor se observa que valores digitales, generan los valores analógicos buscados:

	B1	B2	B3	B4	B5	B6	B7	B8	E_o
Pos. Full Scale	1	1	1	1	1	1	1	1	9.96
Zero Scale	1	0	0	0	0	0	0	0	0.04

Neg. Full Scale	0	0	0	0	0	0	0	0	-9.96

Tabla 2. Valores de tensión correspondientes a los valores digitales.

Esta es la configuración bipolar del conversor, la salida se toma como $\mathbf{E}_o$. Se ve como para el valor 0V hay un pequeño offset de 0.04V. La relación entre valores digitales y análogicos resulta:

- +Vcc = 11111111
- 0V = 10000000
- –Vcc = 00000000

Para generar los formatos se utilizan 2 bits, que serían los más significativos del conversor digital analógico. El estado +Vcc se representa con “11”, el estado 0V con “10” y el estado –Vcc con “00”. Las 6 entradas restantes del DAC se completan con el bit menos significativo. Es importante tener en cuenta que el sistema trabaja manteniendo la línea en 0V cuando no hay transmisión de trama. El encargado de controlarlo es el bloque llamado FORMAT, para el cual se transcribe su programa en VHDL en el APENDICE F, programa 2.

El funcionamiento de este bloque es bastante simple, ya que está compuesto únicamente de dos multiplexores. Este es el circuito:

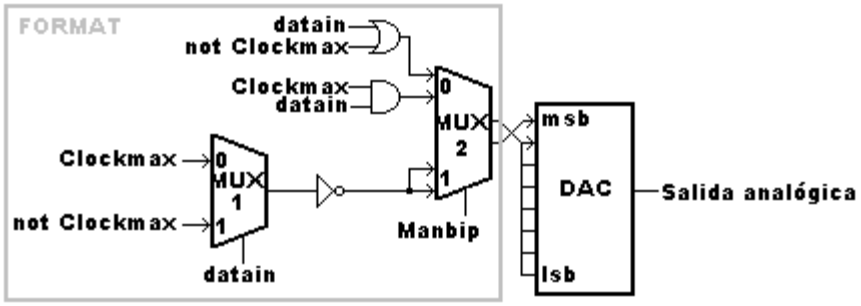

Figura 18. Bloque Format.

Manbip es la entrada externa que permite seleccionar el tipo de formato. El valor lógico ‘1’ corresponde al formato Manchester, y el ‘0’ al formato Bipolar con retorno a cero. En la siguiente tabla se muestran los valores que toma la salida de “MUX2”, en función del tipo de formato y el valor de “datain”:

Datain	Manchester		Bipolar	
	1er semiciclo	2do semiciclo	1er semiciclo	2do semiciclo
0	"00"	"11"	"00"	"10"
1	"11"	"00"	"11"	"10"

Tabla 3. Conformación digital de cada formato.

En el transmisor se integran los bloques detallados anteriormente. En el siguiente diagrama se puede apreciar cuales son las entradas y salidas del transmisor y como se conectan cada uno de estos bloques:

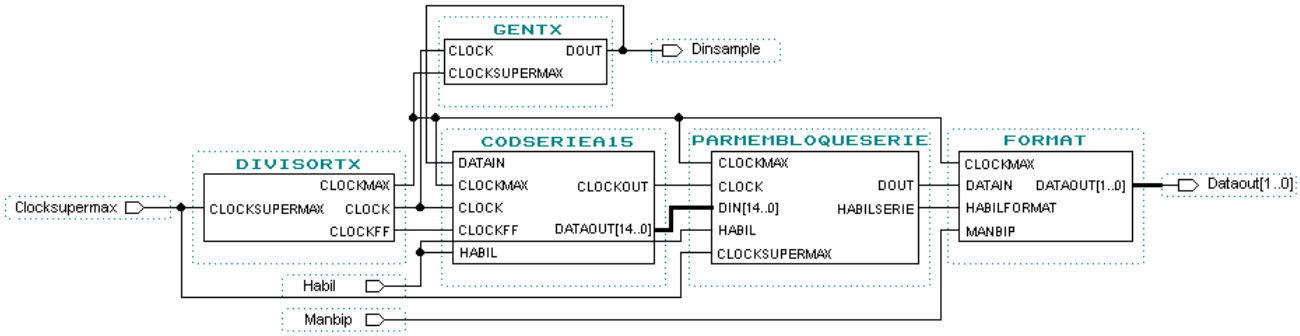

Figura 19. Diagrama en bloques de transmisor implementado.

El bloque DIVISORTX, esta compuesto simplemente por arreglos de flip flops que tienen como fin dividir el clock de entrada para proporcionar así los demás clocks a los demás bloques.

Por último se observa la trama que el transmisor coloca en la línea, y que deberá ser decodificada por el receptor (se debe tener en cuenta que en ausencia de trama la línea se mantiene en 0V):

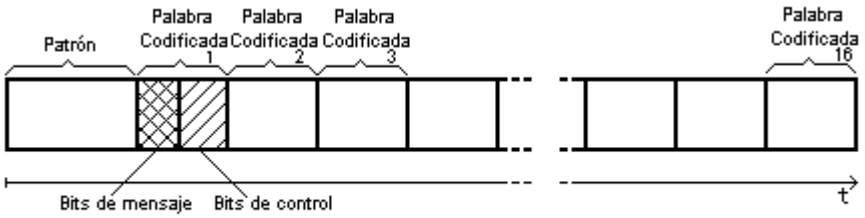

Figura 20. Composición de la trama.

Una vez estudiado en detalle el diseño y funcionamiento del Transmisor es posible analizar la implementación del Receptor.

Receptor

CAD

El receptor se compone inicialmente de un conversor analógico digital, el elemento externo a la FPGA que es el encargado de llevar a palabras digitales de 8 bits los voltajes presentes en la línea. Este conversor trabaja con un sobrerango, respecto de los valores entregados por el transmisor. En caso de que el CAD se maneje entre los valores -Vcc y +Vcc, las tensiones generadas por el transmisor se ajustan entre –Vcc/2 y +Vcc/2. Se aprecia en la figura:

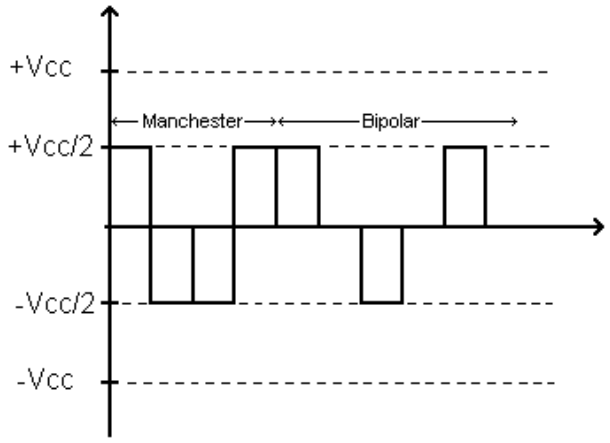

Figura 21. Niveles de tensión utilizados.

En consecuencia estos serían los valores digitales en complemento a 2 que corresponden a los distintos voltajes entregados por el transmisor:

- +Vcc/2 => “00111111”
- 0V=>”00000000”
- -Vcc/2 => “10111111”

Pero como la lógica del CAD es distinta a estos valores hubo que adaptarlos por medio del bloque “COMPA2”. El rango de trabajo del CAD comprende desde el valor –Vcc hasta +Vcc, que esta representado desde el valor digital “00000000” hasta el valor “11111111” respectivamente. Por lo

tanto la conversión (para llevarlo a complemento a 2) se realiza simplemente restando a la salida del CAD el valor “10000000”. Entonces ya se dispone de los valores analógicos de entrada expresados en palabras digitales de 8 bits en complemento a 2.

A continuación se presenta el diagrama en bloques del Receptor completo:

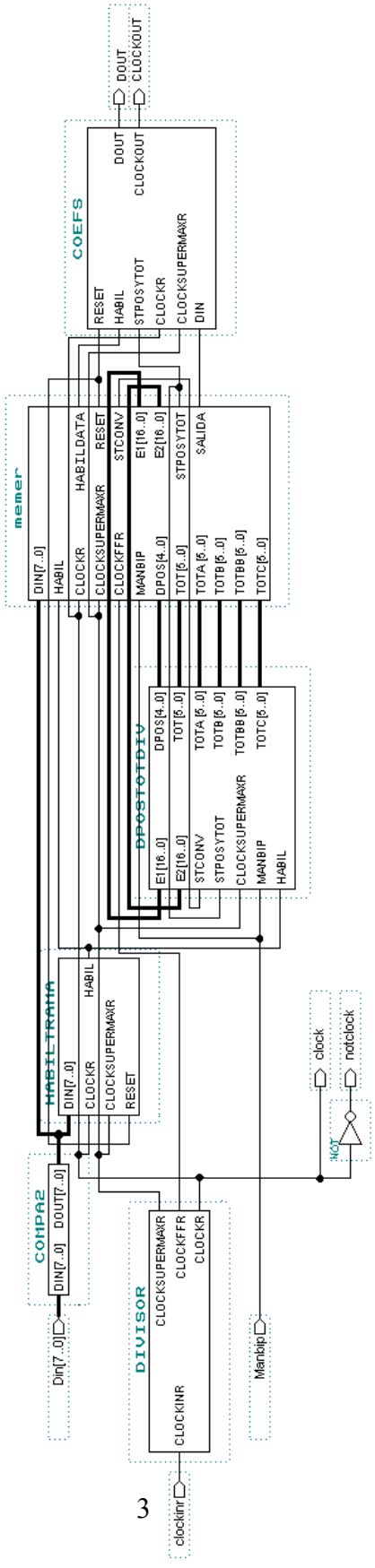

Figura 22. Diagrama en bloques del receptor completo.

Complemento a 2

Para comenzar se describe el bloque de conversión complemento a 2 (COMPA2 en el diagrama anterior). Este es el esquema:

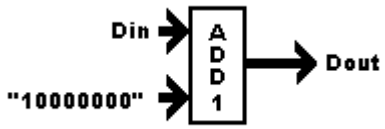

Figura 23. Bloque COMPA2.

Como se ve el funcionamiento es bastante sencillo. Solo cuenta con un restador digital, que sustrae a los datos de entrada el valor digital “10000000”. Su programa en VHDL se encuentra en el APENDICE F, programa 3.

Detección de trama-Habiltrama

El próximo bloque, “HABILTRAMA”, es el encargado de realizar ciertos análisis sobre los datos entregados por “COMPA2”, para detectar el comienzo de trama y así iniciar los demás procesos concernientes a la recepción. La forma de operar es la siguiente:

Se compara el valor del dato entrante para determinar si es positivo y mayor o igual a “00100000” (que representa el valor de tensión +Vcc/4), entonces como primer medida se pone en alto la señal de habilitación (“HABIL”). Luego se comparan los 2 datos siguientes y si cumplen también lo mencionado se inicia un conteo, en el cual se va a determinar si al menos 16 de los siguientes 24 datos también cumplen con este umbral. En todos

los casos si algo no se cumple, se resetean los componentes y se pone en bajo la señal “HABIL” reiniciándose el procedimiento. Para determinar cual es el período más corto con el cual se puede trabajar hay que tener en cuenta que los primeros 3 datos no entran en el conteo, por lo tanto el período mínimo esta dado por:

(3 + 16).2 = 38 muestras

Se debe notar que una señal de período compuesto por 38 muestras y en condiciones ideales tiene 19 muestras representando el valor Vcc/2. Se toma el umbral en Vcc/4 para determinar si corresponde a Vcc/2 o a la ausencia de señal en la línea.

Se aprecia ahora un diagrama más detallado del bloque:

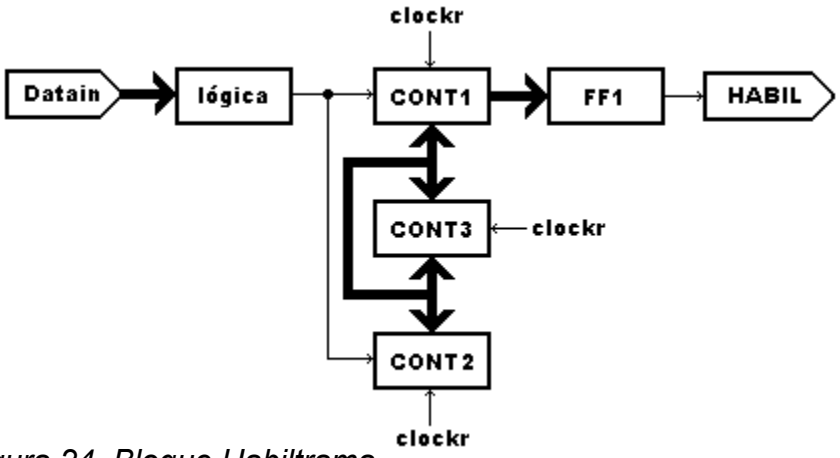

Figura 24. Bloque Habiltrama.

“CONT1” es el encargado de contar las primeras 3 muestras que son mayores a “00100000”, en cuanto una lo sea pone en alto la salida “HABIL”. “HABIL” se pone en bajo y se resetea todo si no hay una continuidad de 3 muestras seguidas mayores a “00100000”. Cuando este contador llega al valor “11” (3 en binario), se habilitan “CONT2” y “CONT3”. “CONT2” se incrementa en una unidad por cada muestra que cumple también el requisito de ser mayor a “00100000”. “CONT3” simplemente cuenta la cantidad de muestras. Cuando “CONT3” cuenta 24 muestras, si “CONT2” contó al menos 16 muestras, se considera que hay un inicio de trama y la señal

"HABIL" se mantendrá en alto hasta que se detecte el final de trama. En caso contrario se resetea todo y se reinicia el proceso.

El final de la trama se detecta en un bloque posterior y se utiliza la señal "RESET" (entrada perteneciente a este bloque) para finalizar todos los procesos. El mecanismo es similar al de la detección de comienzo de trama, sólo que en este caso se cuentan las muestras cuyo módulo está por debajo del umbral antes mencionado. Este mecanismo será analizado más en detalle cuando expliquemos el funcionamiento del bloque en el que se encuentra ("MEMER"). En conclusión este bloque, "HABILTRAMA", tiene la importante función del manejo de la señal "HABIL", esta señal es vital dado que su estado determina si los procesos subsiguientes se llevarán a cabo o no.

En la figura se aprecia como actúa el bloque en caso de un pico de ruido y en el caso de una trama inmersa en ruido:

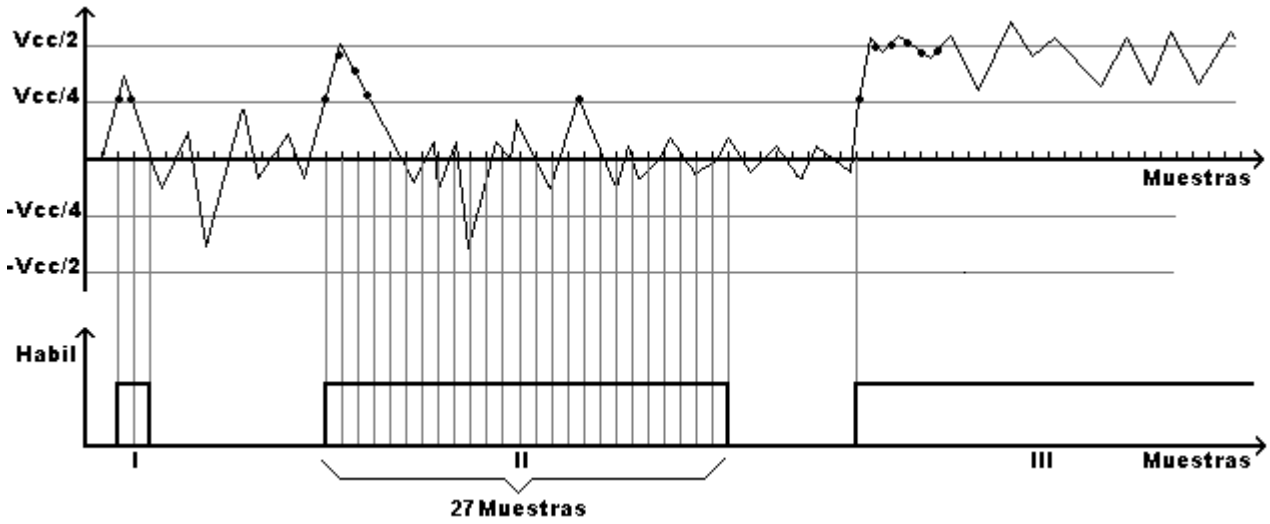

Figura 25. Mecanismo de detección de presencia de trama.

I: Dos muestras superan el umbral de Vcc/4, pero la tercera no, por lo que "HABIL" es reseteado.

II: Las primeras 3 muestras superan el umbral, por lo tanto se activan "CONT2" y "CONT3". Cuando "CONT3" llega a 26, "CONT2" llegó solo a 5, pero como debe llegar como mínimo a 16 "HABIL" es reseteado.

III: Todas las muestras superan el umbral por lo tanto la cuenta de “CONT2“ supera los 16 y “HABIL” queda en alto hasta que se detecte el fin de la trama.

Su implementación en VHDL es el programa 4 del APENDICE F.

Procesamiento de datos - Memer

Siguiendo el curso de los datos de entrada, se ve que también ingresan al bloque “MEMER”. Este es el bloque más importante y complejo en lo que refiere al procesamiento de las muestras. Aquí se generan la salida de datos (todavía codificada), el clock para estos datos y la señal “HABILDATA”, que es la encargada de informar la presencia de datos en la entrada del decodificador “COEFS”. También en este bloque se calculan las señales de error de compuerta temprana y tardía (vitales para el cálculo de “TOT” y “DPOS“) y como se mencionó anteriormente se detecta el fin de trama y se genera “RESET”. “MEMER” está únicamente integrado por dos bloques interconectados, es decir que en realidad es una entidad que sólo agrupa “MEM” y “ER”, dentro de los cuales se realizan todos los procesos. En la siguiente figura se aprecia más claramente:

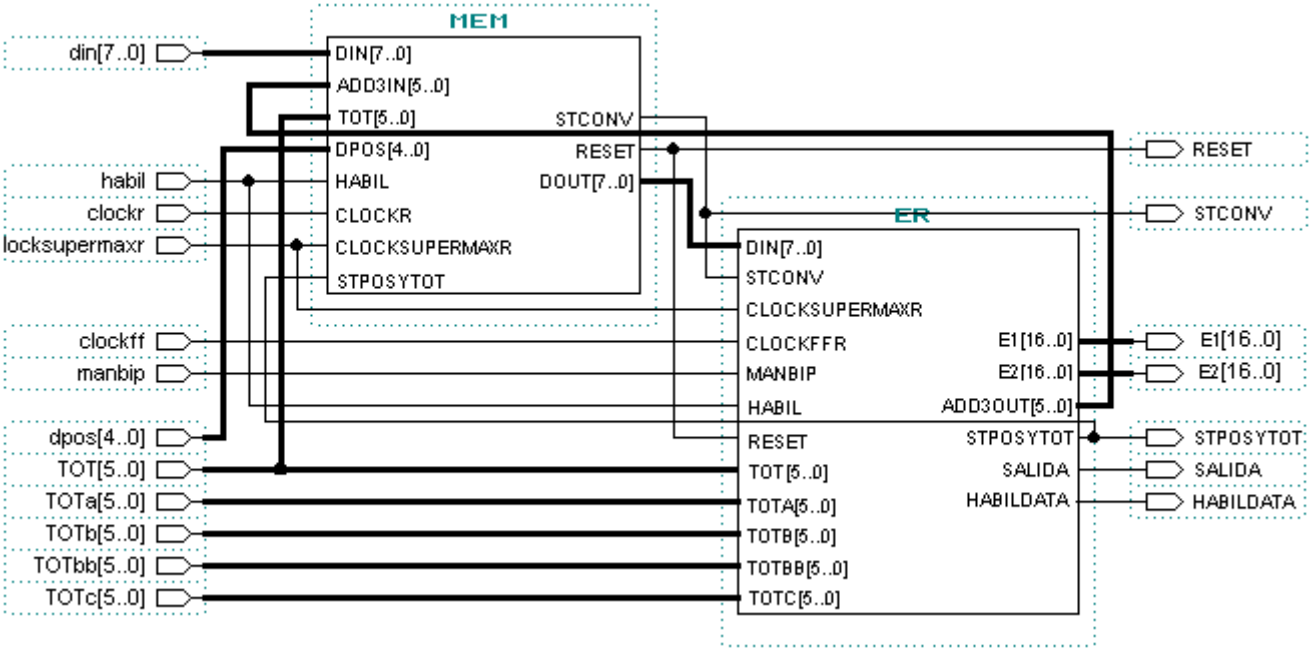

Figura 26. Bloques componentes de MEMER.

Primero se analizará el funcionamiento de “MEM”. Este bloque está compuesto principalmente por 3 memorias de las cuales dos están destinadas al almacenado directo de los datos que ingresan, mientras que en la tercera se vuelcan determinadas porciones del contenido de las dos memorias anteriores de acuerdo a las estimaciones del comienzo y de la cantidad de muestras que componen el período de cada bit. La escritura de las memorias 1 y 2 tiene la particularidad de que el contenido ingresado al final de una coincide con el del principio de la otra. La capacidad de ambas memorias es de 85 palabras (muestras) de 8 bits, la elección de este número se debió a que, para el correcto desempeño del algoritmo, era necesario contar con memorias cuyo tamaño permitiera asegurar con cierta precisión, el almacenamiento de un símbolo transmitido completo tomando un margen del anterior y del posterior. Para ejemplificar esto se tomará un símbolo transmitido estándar compuesto por 50 muestras. La memoria encargada de almacenar este símbolo comienza a escribirse 10 muestras antes del fin estimado del símbolo anterior (por ahora no se tiene en cuenta el factor de corrección de fase, “DPOS”) y se guardan hasta 10 muestras posteriores al fin estimado del símbolo actual. Entonces en este caso se tiene almacenado en memoria 70 muestras, 10 del símbolo anterior, 50 del símbolo actual y 10 del posterior. Este margen es necesario para efectuar la corrección en fase por medio de la variable “DPOS”. Como se mencionó anteriormente, parte del contenido de esta memoria se vuelca a otra memoria (MEM3), este fragmento abarca desde la posición de memoria 10+DPOS hasta la posición 10+TOT+DPOS, en este caso siendo TOT=50. Para mayor claridad obsérvese el siguiente gráfico:

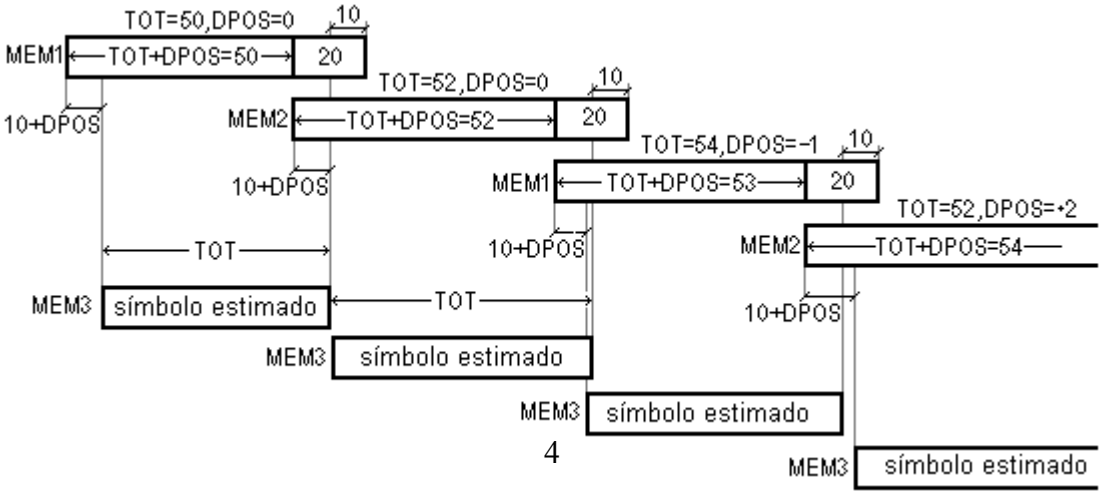

Figura 27. Vuelcos de las memorias 1y 2 en la memoria 3.

Se observa como el símbolo estimado tiene un tamaño de TOT muestras, y comienza en la posición 10+DPOS, y termina en la posición 10+DPOS+TOT. La importancia de comenzar a almacenar muestras en una memoria mientras todavía se siguen guardando en la otra, queda en evidencia cuando “DPOS” es negativo, ya que en ese caso el símbolo estimado actual se solapa con el anterior. Recuérdese que la variable “DPOS” es obtenida luego de haber volcado las muestras estimadas a la memoria 3 y de realizar ciertos cálculos con ellas. En consecuencia, si “DPOS” fuera negativo, las primeras muestras del símbolo estimado coincidirían con las del símbolo estimado anterior, y si no se las guardara previamente, se perderían, imposibilitando los procesos siguientes.

Resumiendo, en la memoria 3 quedan volcadas las muestras que van a representar el símbolo actual, y que serán procesadas en el próximo bloque (“ER”). Para permitir a dicho bloque el acceso a esta memoria, el bloque “MEM” tiene el puerto de entrada “ADD3IN” y el puerto de salida “DOUT”, el primero es el direccionamiento de “MEM3” y el segundo su salida. Para comprender mejor su funcionamiento general, se analizará la siguiente figura:

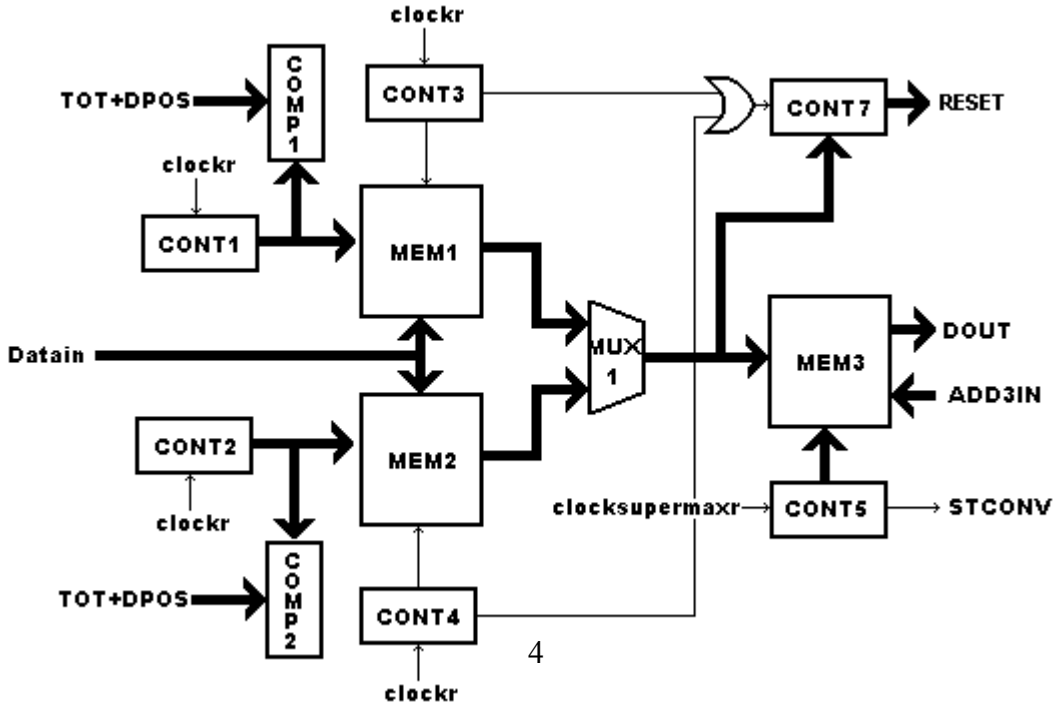

Figura 28. Bloque MEM.

A medida que los datos de entrada van llenando “MEM1”, “CONT1” se va incrementando. Cuando “COMP1” comprueba que la salida de “CONT1” alcanzó el valor TOT+DPOS, da inicio a “CONT2” y activa a “CONT3” para llenar 20 espacios más. Como “CONT1” es el que direcciona la memoria, debe seguir contando hasta TOT+DPOS+20.

“CONT2” es para “MEM2” lo que “CONT1” es para “MEM1”, por lo tanto, cuando “CONT1” llega al valor TOT+DPOS, se inicia el llenado de “MEM2”. Análogamente, cuando “CONT2” llega a TOT+DPOS, da inicio nuevamente a “CONT1” y activa a “CONT4”, para llenar 20 espacios más en “MEM2”. Una vez que “MEM1” termina de llenarse con TOT+DPOS+20 muestras, se vuelcan TOT muestras a “MEM3”. Mientras se van volcando estas muestras un contador (“CONT7”) incrementa su cuenta cada vez que detecta una muestra de módulo menor al de umbral. Si este contador llega al valor 32 quiere decir que se ha llegado al fin de la trama, entonces se pone en alto la señal “RESET”. Esta señal indica el fin de trama, por lo tanto frena los procesos siguientes y resetea algunos componentes.

Una vez finalizada la transferencia a “MEM3”, se genera una señal llamada “stconv”, que le indica al bloque “ER” que puede acceder a “MEM3”. El encargado de direccionar tanto a “MEM1” como a “MEM3” es “CONT5”. “MEM1” es direccionada desde la posición 10+DPOS hasta la posición 10+DPOS+TOT, y “MEM3” es direccionada entre la posición 1 y TOT. Como “CONT5” cuenta entre 0 y TOT, para direccionar “MEM1” se le suma 10+DPOS y se direcciona desde la salida del sumador. Una vez volcadas las TOT muestras en “MEM3”, se rellenan las posiciones restantes con ceros para limpiarla del contenido anterior. Esto se debe a que las memorias no disponen de “clear”. Finalmente, con “MEM3” lista, el bloque “ER” puede acceder al símbolo estimado para realizar los cálculos que van a permitir obtener nuevos valores de DPOS y TOT. Cuando éstos están listos, el proceso se repite para “MEM2”. A continuación, en la página siguiente se

presenta un esquema más completo del bloque. Mientras que su programa se transcribe en el APENDICE F, programa 5.

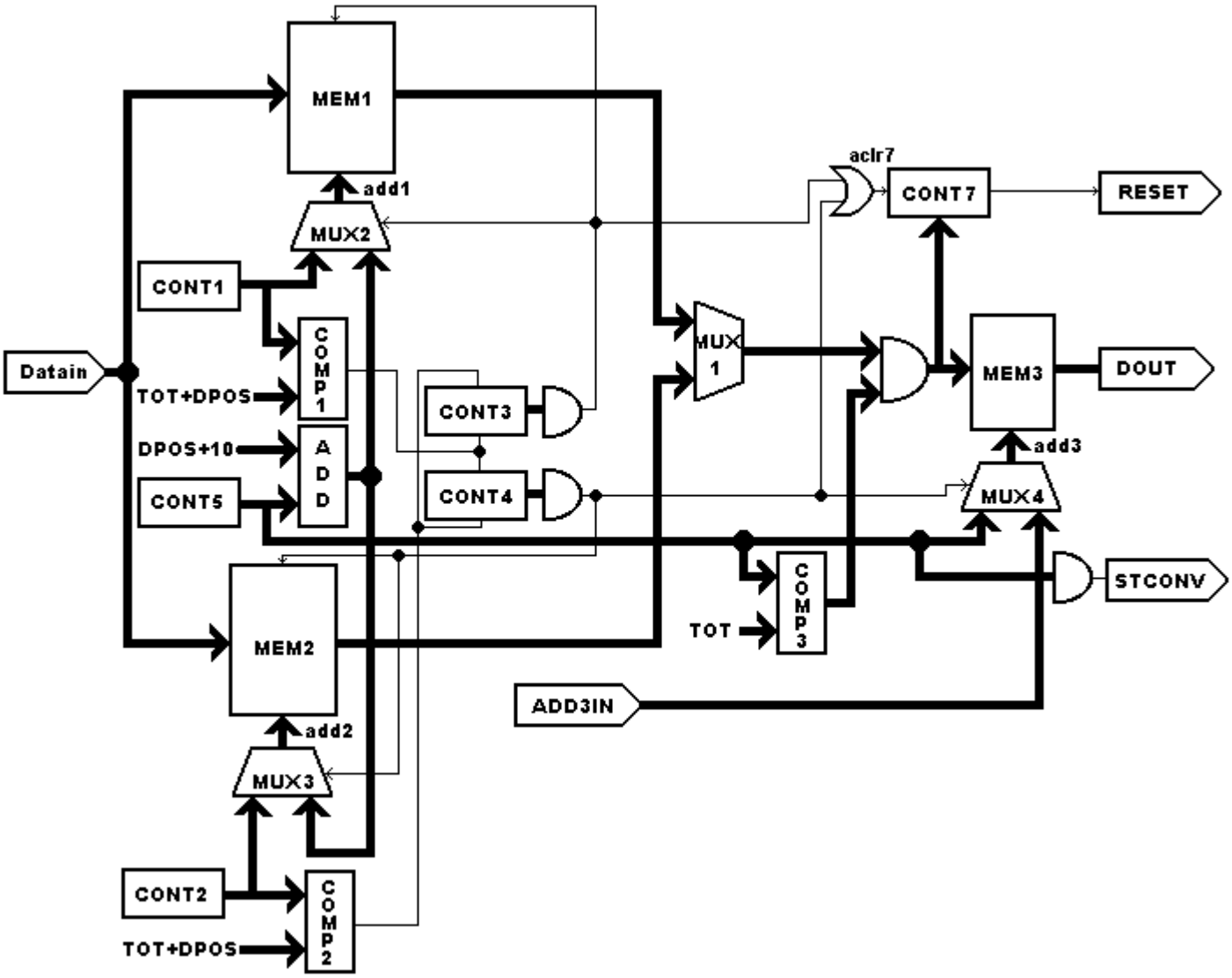

Figura 29. Esquema detallado de MEM.

Implementación del algoritmo de sincronismo

El bloque “ER”, es el encargado de implementar el filtro adaptado. Toma las muestras de “MEM3” y las convoluciona con un pulso de ancho TOT y características relativas al tipo de formato. Para acceder a “MEM3” genera “ADD3OUT”, señal de direccionamiento de la memoria. La puesta en alto de la señal “STCONV”, da comienzo a las funciones pertinentes a este bloque.

A la vez que efectúa la convolución, va realizando las integraciones pertinentes al cálculo de las variables e1 y e2 del algoritmo Early/Late (ver esquema página 21). También es el encargado de la lectura de los datos codificados, presentes en la trama, determinando el valor lógico del símbolo procesado.

Junto a esta señal de salida de datos en serie se genera un clock asociado a ella (“STPOSYTOT”) y una señal de habilitación (“HABILDATA”).

Primero se analizará el cálculo de e1 y e2. Como el método difiere en función del tipo de formato, se comenzará con el formato Bipolar con Retorno a Cero.

Como ejemplo se puede suponer un símbolo de ancho dado por 50 muestras. Esto implica realizar una convolución con un pulso de 25 muestras positivas y 25 ceros, en caso bipolar. La convolución se realiza en dos etapas. En la primera, se van sumando una a una las primeras 25 muestras, esto va a representar los distintos valores de la convolución en función del número de muestra. Se simbolizará con k el número de muestra y c(k) el valor de la convolución en función del número de muestra. Así, por lo dicho antes sería: $c(1)=m_1$, $c(2)=m_1+m_2$ y $c(n)=m_1+m_2+...+m_n$, esto es válido hasta n=25 en este ejemplo, ya que TOT=50. Obsérvese como es esta convolución gráficamente:

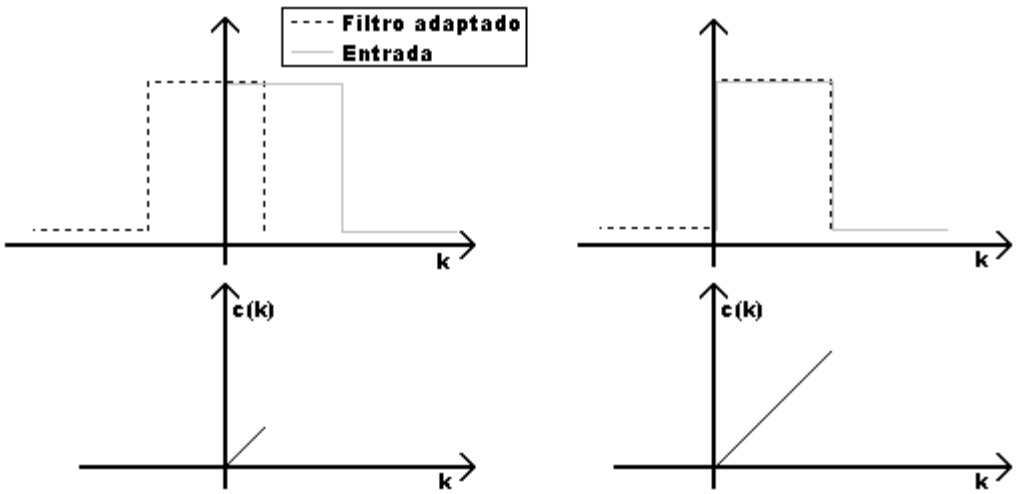

Figura 30. Convolución entre el filtro y la señal de entrada (Bipolar).

En la segunda etapa, para obtener cada valor de la convolución se debe sumar al valor acumulado el valor de la muestra siguiente (m_{26},m_{27},etc) y restar el valor de la muestra m_1,m_2, etc. Por ejemplo:

$c(26) = m_1+m_2+....+m_{25}+m_{26} -m_1$

$c(27) = m_1+m_2+....+m_{26}+m_{27} -m_1 -m_2 \quad = m_1+m_2+....+m_{26} +m_{27} -(m_1+m_2)$

.

.

$c(50) = m_1+m_2+....+m_{50} -m_1-m_2 -....-m_{25} = m_1+m_2+....+m_{50} -(m_1+m_2+....+m_{25})$

$$c(n) = \sum_{i=1}^{n} m_i - \sum_{i=1}^{n-25} m_i \qquad \text{para } n>25$$

Como se ve en la ecuación anterior, la convolución puede obtenerse como una diferencia de dos sumas acumulativas. Véase gráficamente, la convolución resultante:

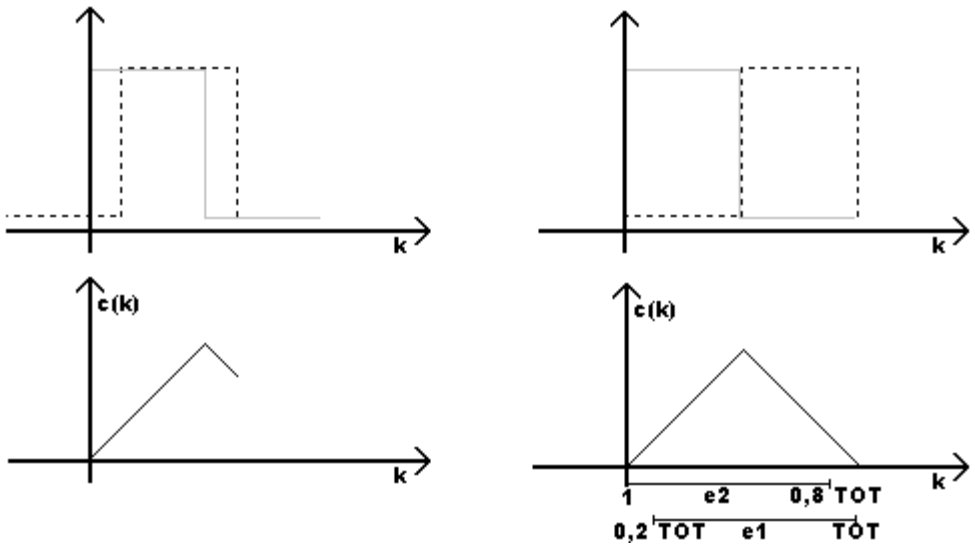

Figura 31. Convolución entre el filtro y la señal de entrada (Bipolar).

Idealmente, a partir del valor k=50 la convolución para un pulso de entrada bipolar con retorno vale cero. En la práctica cae a valores muy bajos, pero no se anula por la presencia de ruido en la segunda mitad del bit

de entrada. De todos modos, para el cálculo de e1 y e2, sólo interesan los primeros TOT valores de la convolución. Para calcular e2, se debe integrar la función convolución entre la muestra 1 y la muestra 0.8 TOT. En forma similar, e1 se obtiene integrando entre la muestra 0.2TOT y la muestra TOT.

Para el formato Manchester la implementación es similar, con la diferencia de que el pulso que representa al filtro adaptado estaría dado en este ejemplo por 25 (TOT/2) muestras positivas y 25 negativas. Para las primeras TOT/2 muestras se efectúan los mismos cálculos que en bipolar, entonces tomando nuevamente TOT=50 se obtendrían los siguientes valores para la convolución: $c(1)=m_1$, $c(2)=m_1+m_2$ y $c(n)=m_1+m_2+...+m_n$, esto es válido hasta n=25. Gráficamente sería:

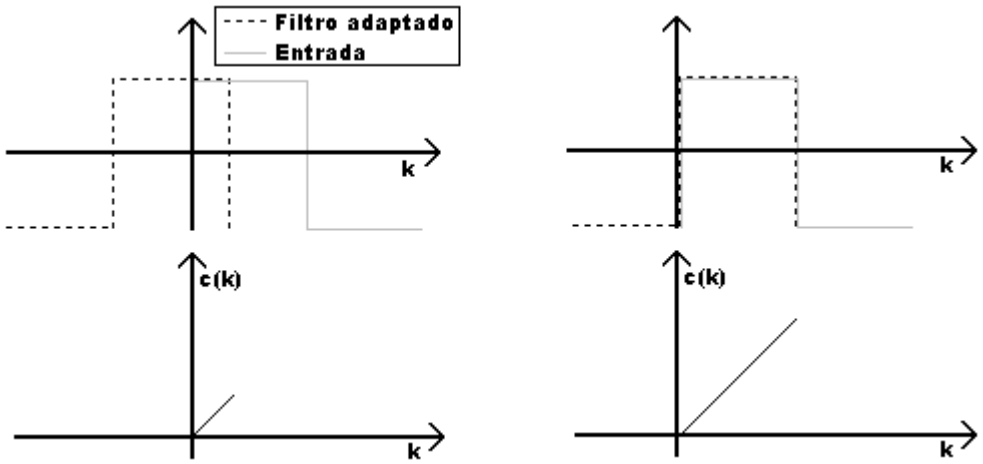

Figura 32. Convolución entre filtro y señal entrante (Manchester).

Se observa que para k = 25 las mitades positivas se han solapado por completo. Luego comenzarán a superponerse las partes negativas con las positivas. Entonces para obtener el valor de la convolución se debe sumar, al valor anterior, el valor de la muestra siguiente (m_{26}, m_{27}, etc) y restar el doble del valor de la muestra m_1,m_2, etc. Por ejemplo:

$c(26)=c(25)+m_{26}-2.m_1 = m_1+m_2+....+m_{25}+m_{26}-2.m_1$
$c(27)=m_1+m_2+....+m_{26}+m_{27}-2.m_1-2.m_2= m_1+m_2+....+m_{27}-2(m_1+m_2)$
.
.
$c(50)=m_1+m_2+....+m_{50}-2.m_1-2.m_2-....-2.m_{25}= -m_1+m_2+....+m_{50}-2(m_1+m_2+....+m_{25})$

$$\sum_{i=1}^{n} m_i - 2\sum_{i=1}^{n-25} m_i$$

c(n) = para n>25

Gráficamente, la convolución resultante es:

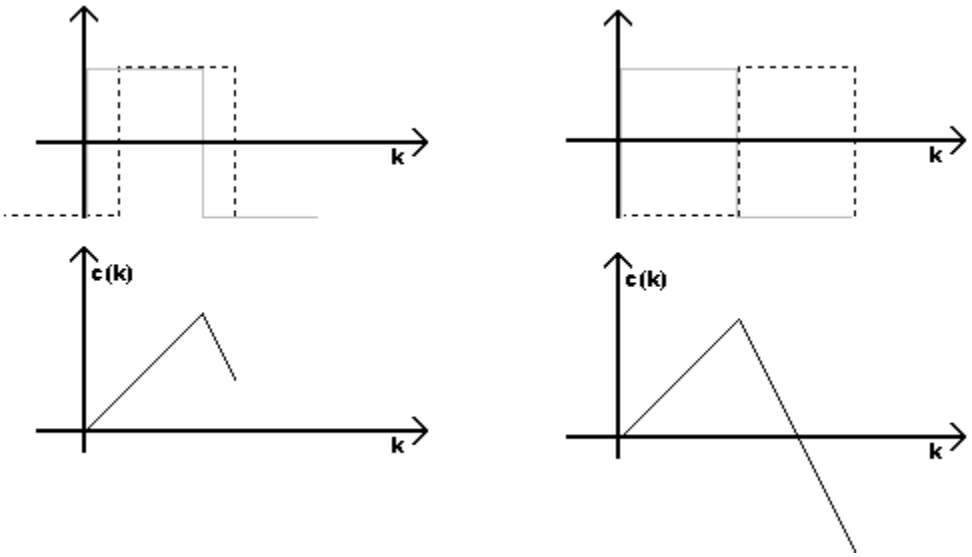

Figura 33. Convolución entre filtro y señal entrante (Manchester).

Cuando k supera el valor TOT, la convolución empieza a incrementarse con la misma pendiente anterior, esto se debe a que ahora se superponen las regiones negativas de los pulsos:

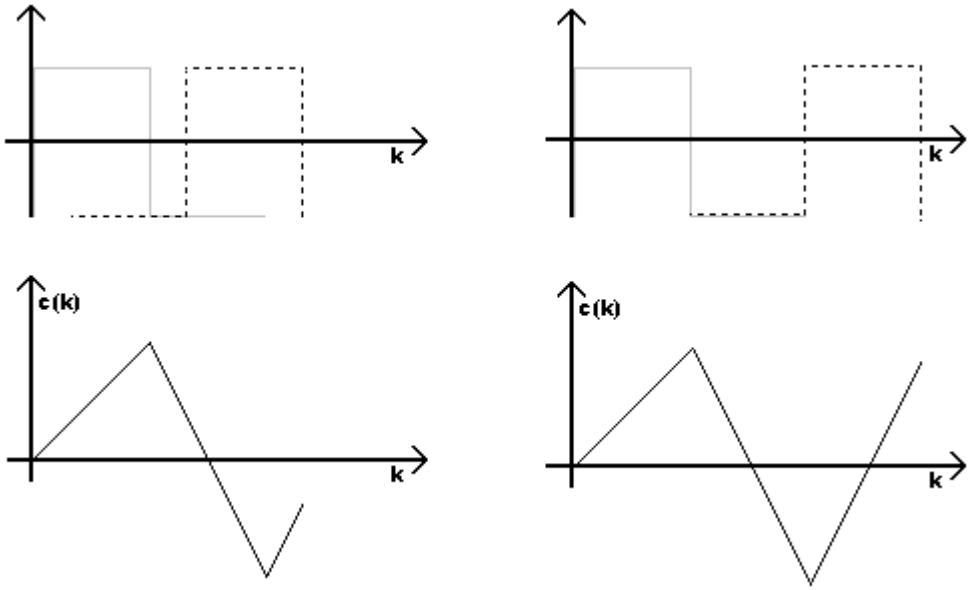

Figura 34. Convolución entre filtro y señal entrante (Manchester).

Finalmente, la convolución cae progresivamente a cero, ya que la zona de solapamiento no nulo empieza a reducirse en forma proporcional al aumento de k:

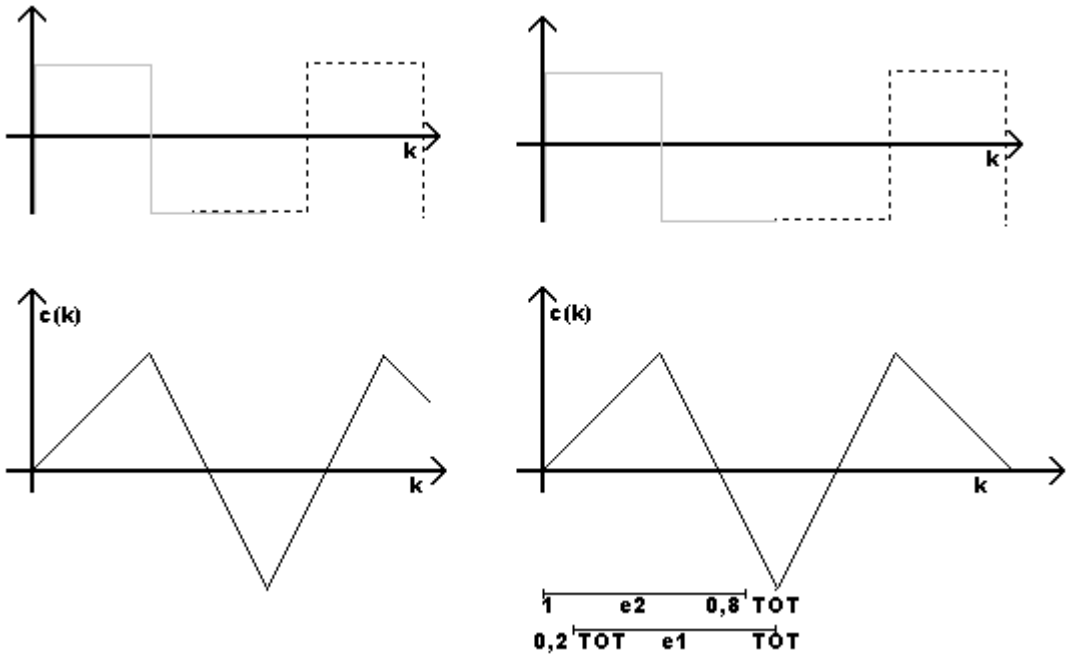

Figura 35. Convolución entre filtro y señal entrante (Manchester).

En el gráfico se puede apreciar que la convolución es simétrica alrededor de k=TOT. Aunque esto es válido para una señal ideal, en la implementación se calcula la convolución sólo hasta este valor. En la práctica, la convolución se deforma debido al ruido.

Viendo los intervalos de integración se puede notar que en sincronismo ideal, |e2| > |e1|. Esto significa que el sistema no permanece estable, y que va a oscilar en un determinado rango. Tras varias simulaciones, con distintas condiciones de ruido y jitter, se pudo comprobar que el sistema lograba mantener el sincronismo. Si bien el error no se anula, el sistema va realizando correcciones en frecuencia y fase, de forma tal que admite cierto margen de variación, sin desengancharse. Recuérdese que la corrección de fase (DPOS) se obtiene en forma proporcional al error, mientras que para modificar la frecuencia (TOT), se analiza el módulo de la derivada del error y si éste es superior a un umbral, se aumenta la frecuencia (se disminuye TOT) si el error es positivo o se disminuye si es negativo. Esto se ve más claramente en el APENDICE C.

El bloque que lleva a cabo todos los cálculos anteriormente mencionados es “ER”, en éste se encuentra la mayor parte de los dispositivos que efectúan funciones matemáticas como por ejemplo sumadores, multiplicadores, integradores, etc. Al implementar los integradores se encontró con el inconveniente de no contar con diseños genéricos de VHDL. Debido a esto, hubo que sintetizar estos integradores a partir de otros componentes.

Para los multiplicadores, si bien se contaba con algunos modelos prediseñados, estos requerían demasiado hardware para el tipo elemental de multiplicaciones que se debían realizar, entonces se decidió hacer un diseño a medida de las necesidades.

Las integraciones discretas se implementaron por medio de la suma acumulativa en complemento a 2 del valor de cada muestra. Para llevarlo a nivel componentes se decidió utilizar un sumador en cascada con un flip flop y realimentar su salida. Así se obtiene, a la salida del registro, la acumulación de los valores de entrada. Este es su diagrama:

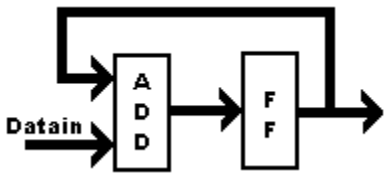

Figura 36. Integrador discreto implementado.

Con respecto a los multiplicadores, en este bloque sólo se utilizó un multiplicador x2 que fue implementado haciendo un desplazamiento de todos los bits hacia la izquierda. En otro bloque posterior hay una división que también se implementó haciendo desplazamientos, pero se tuvo que adaptar el diseño de forma tal que el divisor fuera múltiplo de 2.

Para realizar la convolución y obtener e1 y e2, se debe conocer además de “TOT”, los valores enteros de 0.8TOT, 0.5TOT y 0.2TOT. El

encargado de proveer estos valores es el bloque “TOTDIV”. A continuación, en la página siguiente, se muestra el diagrama de “ER”.

Se puede ver como el bloque está compuesto por dos elementos principales. El primero es el que recibe los datos de “MEM3” y realiza tanto la convolución como las integraciones que dan por resultado e1 y e2. Los límites de las integraciones están dados por la comparación de la salida de “CONT1” con los valores dados por “TOTDIV”.

El segundo elemento es el encargado de direccionar a “MEM3”. Para obtener los primeros TOT/2 valores se direcciona “MEM3” en forma creciente desde la posición 1 a la TOT/2, pero luego el cálculo se torna más complejo ya que hay que direccionar dos posiciones distintas para cada nuevo valor de convolución. “MUX2” se encarga de proveer las dos direcciones necesarias para hacerlo. Para la selección del multiplexor se usa el mismo clock que alimenta a “CONT1”, permitiendo que para cada valor de salida de “CONT1”, el multiplexor presente las dos direcciones, una en cada semiciclo.

Una entrada del multiplexor es simplemente la salida de “CONT1”. La otra entrada es la salida de “ADD6”, el cual es un restador que sustrae el valor TOT/2 a la salida de “CONT1”.

En “FF1” se acumulan los valores de “MEM3”. Como se explicó anteriormente, esta acumulación coincide con la convolución hasta la muestra TOT/2. A partir de allí hay que sustraerle una nueva acumulación, que se realiza en “FF4”. Esto se ejemplifica en las páginas 47 y 48. Luego en “FF6” se van presentando uno a uno los valores de la convolución. La salida de “FF6” ingresa a los integradores que representan a las compuertas temprana y tardía.

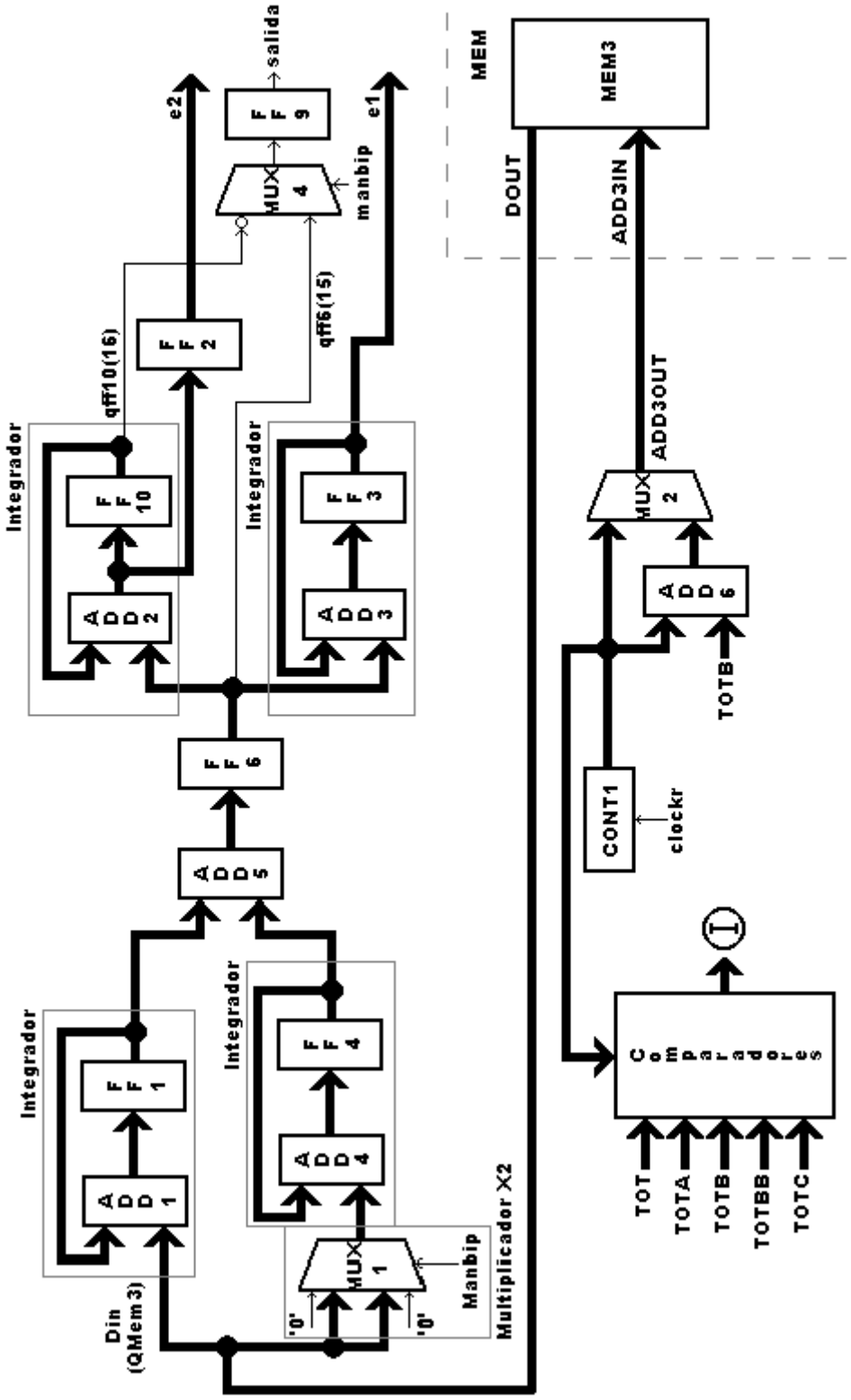

Figura 36. Bloque "ER"

En cuanto a la lectura del símbolo procesado, se debe diferenciar como es el procedimiento para cada formato:

En el caso de Manchester, la lectura se realiza en base al signo del último valor de la convolución, por lo tanto, el bit más significativo de “FF6”, cuando éste contiene el último valor de la convolución, es el valor de salida leído. Esto es así ya que si el bit enviado es un ‘1’, dicho valor de la convolución resulta negativo, y al trabajar en complemento a 2, el bit más significativo de “FF6” será también un ‘1’. Análogamente, si el bit enviado es un ‘0’, el valor de la convolución es positivo, y el bit más significativo de “FF6” es también un ‘0’.

Si se está trabajando con el formato Bipolar con Retorno a Cero, “FF10” acumula toda el área de la convolución y se analiza si esta acumulación es positiva o negativa. En caso de ser positiva, el valor leído es un ‘1’, en caso contrario es un ‘0’. Así, el bit de signo de esta acumulación representa el inverso del valor lógico del bit procesado.

Como se ve, la salida de este bloque depende del formato con el cual se este trabajando. Por esto se usa a “MUX4” el cual es el encargado de suministrar a “FF9” el valor de lectura correcto en función del tipo de formato usado.

Para nombrar las señales que provienen de “DPOSTOTDIV” se utilizó esta nomenclatura:

- TOTA=0.2TOT
- TOTB=0.5TOT
- TOTBB=0.5TOT + 1
- TOTC=0.8TOT

Estas señales son utilizadas por los distintos flip flops para delimitar ciertos eventos. La comparación de la salida de “CONT1” con “TOTA” marca el comienzo del intervalo de integración de la compuerta tardía. El integrador compuesto por “FF3” y “ADD3” espera la habilitación de dicha comparación, para comenzar a actuar. Así ocurre también con las demás señales. “TOTB”

y “TOTBB” dan cuenta del comienzo de la segunda etapa de la convolución ya mencionada, “TOTC” señala el fin del intervalo de integración de la compuerta temprana y TOT el fin del intervalo de integración de la compuerta tardía.

El programa de “ER” en VHDL, se encuentra en el APENDICE F, programa 6.

Ahora, se va a profundizar en la descripción del bloque “DPOSTOTDIV”; para ello se muestra a continuación su diagrama general en bloques:

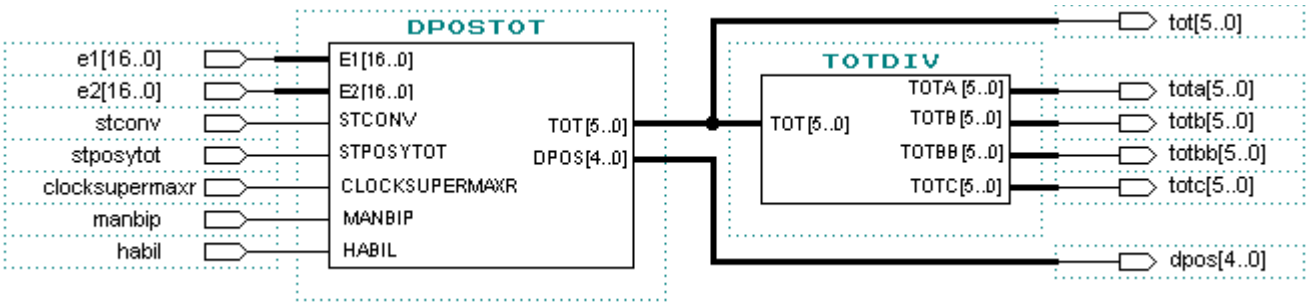

Figura 37. Bloque DPOSTOTDIV.

En este bloque se calcula “TOT” y “DPOS” en base a las señales e1 y e2 que le proporciona “ER”, el bloque anteriormente descrito. La señal “TOT”, originada en el sub-bloque “DPOSTOT”, ingresa posteriormente al sub-bloque “TOTDIV” donde se calculan “TOTA”, “TOTB”, “TOTBB” y “TOTC”. Estas son las señales que anteriormente fueron mencionadas como entradas de “ER” y que intervienen en los cálculos de los errores. Se comenzará ahora con la descripción del funcionamiento de “DPOSTOT”:

Con el ingreso de las señales e1 y e2 se obtiene la señal de error e, que está dada por la diferencia de los módulos de e1 y e2. Luego al error correspondiente al símbolo que actualmente está siendo procesado se le resta el error calculado para el símbolo anterior, obteniendo así la derivada discreta del error EVEC2(k). A continuación, con estas dos variables se

calcularán las señales “DPOS” y “TOT” que serán utilizadas para el procesamiento del próximo símbolo que ingrese.

Para hallar “DPOS” se divide el error. Dependiendo del formato elegido la división se hace por 4096, en el formato Manchester, y por 1024 en Bipolar. Para realizar la división se efectúan desplazamientos hacia la derecha de los bits en los registros, en función del tamaño del divisor. En el primer caso son 12 los desplazamientos, y en el segundo 10.

La razón por la cual el divisor cambia en función del tipo de formato tiene que ver con el hecho de que en sincronismo la señal de error del formato Bipolar toma valores menores que la del Manchester. Como se ve en la figura de la página 47 la convolución en el formato Manchester no es simétrica en el intervalo de trabajo (entre 0 y TOT), dando en general un error considerable en comparación al Bipolar, cuya convolución sí es simétrica. Esta simetría da lugar a que e1 y e2, en funcionamiento, tomen valores cercanos y por lo tanto el error sea pequeño. En Manchester al no haber simetría, e1 y e2 toman valores más distantes y esto genera un error apreciablemente mayor.

Si bien en la implementación el error es una palabra de 17 bits (16..0), en formato Bipolar, la magnitud del error permite expresarlo solo por los 15 bits menos significativos. Debido a esto es que para realizar la división por 1024 conservamos los bits 14,13, 12, 11, 10 (parte entera del cociente). Los bits 16 y 17 se descartan, y los bits comprendidos entre el bit 9 y 0 representarían la parte decimal del cociente. El bit 9 es utilizado para redondear el “DPOS” al valor entero más cercano.

En el caso del formato Manchester, la división por 4096 se realiza asignando a “DPOS” los bits 16,15,14,13,12 y el redondeo está dado por el bit 11. Para la implementación de esta división utilizamos simplemente un multiplexor con dos entradas de 5 bits. En la primer entrada ingresan los bits 14,13,12,11,10 de la señal de error, y en la segunda los bits 16,15,14,13,12 de la misma señal. La selección de este multiplexor está manejada por la

entrada “Manbip”, por lo tanto a la salida se obtiene el error dividido en función del tipo de formato.

Esta señal ingresa en un sumador que realiza el redondeo a partir del bit 11 en Manchester o 9 en Bipolar, que representa el bit más significativo de la parte decimal del resultado de la división; así finalmente se obtienen los 5 bits que componen “DPOS”.

Debe hacerse una aclaración referida al algoritmo que se utilizó para simular el sistema de sincronismo hecho en Matlab. En Matlab se trabaja en un intervalo de ancho dado por “TOT” y luego se modifica el tamaño de “TOT” hacia arriba o hacia abajo en dos muestras. Por otra parte, la posición donde comienza el símbolo siguiente está dada por la posición donde empezó el actual, más el valor de “TOT” (ya afectado con ±2) + “CFASE”.

En la implementación de VHDL la única señal que afecta el comienzo del siguiente símbolo es “DPOS” (“DPOS” en VHDL es el equivalente a “CFASE” en Matlab). Por eso para imitar el funcionamiento del algoritmo de Matlab, se afecta el valor de “DPOS” por la corrección que sufre “TOT” de ±2. “DPOS” resulta ser DPOS = e/4096 ±2, por lo tanto el comienzo del símbolo siguiente resulta ser la posición de comienzo del símbolo actual más “TOT” (sin ser todavía afectado por ±2) + “DPOS”. Así, este comienzo coincide con el de la simulación en Matlab.

En la página siguiente se encuentra el diagrama en bloques de “DPOSTOT”. “MUX1” y “MUX2” se encargan de invertir el orden de entrada de los módulos de e1 y e2 al restador “ADD1”, en función del tipo de formato. A partir de “MUX6”, “MUX7” y “ADD4” se efectúa la división de la señal de error, y su resultado se redondea al valor entero más cercano. “MUX6” provee los bits 9 y 11, que como ya se ha dicho, representan el bit más significativo de la parte decimal del resultado de la división. A “MUX7” ingresan los bits 16,15, 14, 13, 12 por un lado y los bits 14, 13, 12, 11, 10 por otro. Los primeros representan la división en Manchester y los otros en Bipolar. En “ADD4” se suma la parte entera de la división con el bit más

significativo de la parte decimal, dando como resultado el valor redondeado.
(Continua en la pág. siguiente)

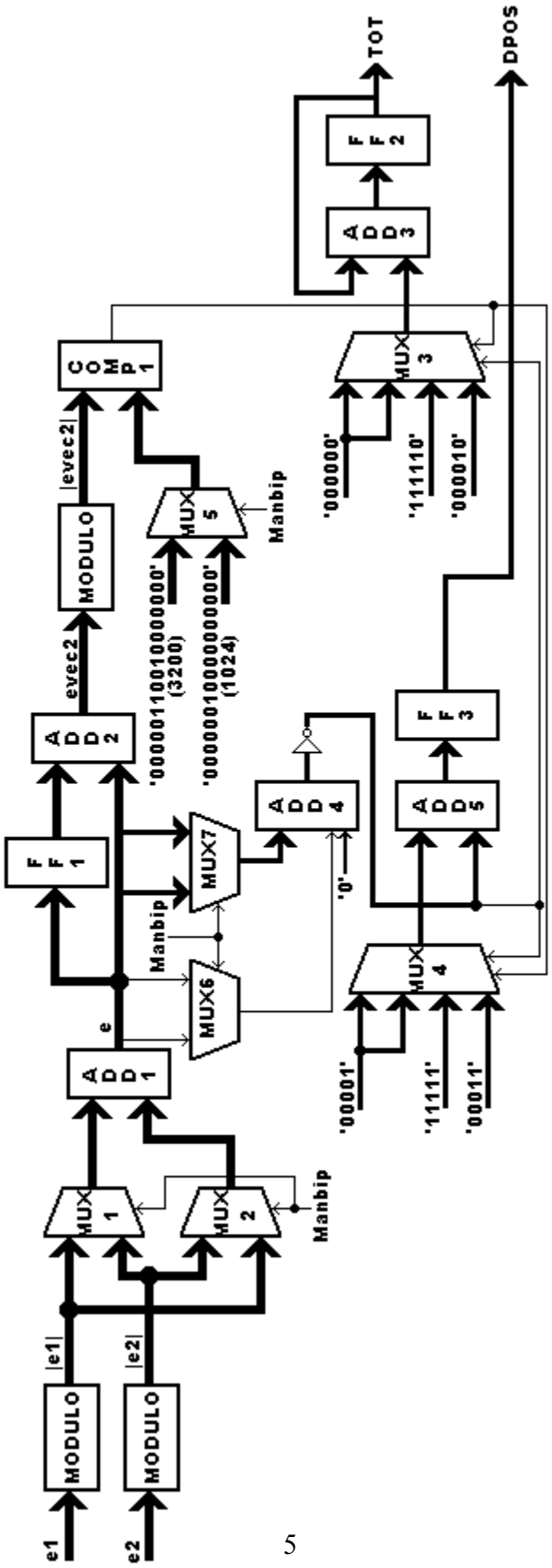

Figura 38. Diagrama en bloques de DPOSTOT.

El signo de este valor debe ser invertido a continuación. Para hacerlo se invierten todos los bits y se suma ‘1’. En este caso, además de sumarle ‘1’ a la salida de “ADD4” invertida, se le suma ±2 con “ADD5” si el módulo de “EVEC2” supera el umbral ya mencionado. Por eso “MUX4” tiene como entradas los valores ‘00001’, ‘00001’,’11111’ y ‘00011’. Las primeras 2 entradas son para el caso en que no se supera el umbral y por lo tanto “DPOS” no es modificado en ±2, y sólo se suma uno para completar la inversión de su valor.

La tercer entrada es el valor ‘–1’ que surge de sumar ‘1’ y restar ‘2’ eso se da cuando se supera el umbral de “EVEC2” y cuando el error es mayor que cero. La última entrada corresponde al valor ‘3’, que es la suma de ‘1’ más ‘2’; es el caso en que se supera el umbral y el error es menor que cero.

A través de “ADD2” y “FF1”, se obtiene “EVEC2” que es el valor del error actual menos el error del símbolo anterior. En “COMP1” se determina si el valor del módulo de “EVEC2” supera un determinado umbral, que también es función del tipo de formato. “MUX5” es el encargado de suministrar ese umbral a “COMP1”.

Para obtener el nuevo valor de “TOT” se tiene almacenado el valor anterior en “FF2” y a través de “ADD3” se hace una corrección que al igual que en “DPOS” depende del valor de “EVEC2” y del signo del error. Si “EVEC2” no supera el umbral, “TOT” no se modifica, en caso contrario, “TOT” se disminuye en dos unidades para un error mayor que cero o se incrementa en dos unidades para un error menor que cero.

Su programa es el número 7 que se encuentra en el APENDICE F.

Volviendo ahora al diagrama en bloques de “DPOSTOTDIV” que se encuentra en la página 54, se observa que el segundo bloque que lo compone es “TOTDIV”. Este último por medio de divisiones de la señal “TOT” proporciona las señales que controlan los intervalos de integración y la convolución en el bloque “ER”. Para que los intervalos de integración de las compuertas temprana y tardía sean iguales se implementó una tabla con la cual para cada valor de “TOT” posible dentro del rango de trabajo, se obtiene un conjunto de valores que indican el límite inferior de integración de la compuerta tardía, la mitad y la mitad + 1 de la convolución y el límite superior de integración de la compuerta temprana. Por lo tanto este bloque solamente precisa del ingreso de la señal ”TOT”, y presenta a su salida las señales mencionadas. “TOTA” es el valor 0,2TOT, “TOTB” es 0,5TOT, “TOTBB” es TOTB+1 y “TOTC” es 0,8TOT. En los casos que el número no sea entero, se hará un redondeo que contemplará el ancho de los intervalos de integración. Para que los intervalos de integración de los errores e1 y e2, sean iguales y en consecuencia se consiga mayor exactitud, el valor de “TOTC” se obtiene como TOTC=TOT–TOTA. Como este bloque es simplemente una tabla no tiene sentido hablar de un diagrama en bloques, por lo tanto solo se presenta su implementación de VHDL en el APENDICE F, programa 8.

Ahora sólo resta el bloque de decodificación, detección y corrección de errores. Este elemento llamado “COEFS” utiliza un algoritmo basado en los códigos BCH y los campos de Galois (Ver Apéndice D). A través de este algoritmo se calcula un coeficiente en función de la palabra recibida y los bits de codificación. Luego a partir del coeficiente se hace la corrección de un bit. Como ya se comentó, este método puede implementarse para corregir más errores, pero por razones de simplicidad y limitaciones de hardware, fue implementado para uno solo.

Antes de profundizar en la implementación del decodificador se van a desarrollar ciertos aspectos teóricos necesarios para la comprensión de su funcionamiento, y la posible generalización a patrones de más de un error.

Códigos BCH

Los códigos BCH (Bose, Chaudhuri, Hocquenghem) constituyen una clase de códigos de bloques lineales y cíclicos que aparecen como una generalización de los códigos de Hamming, de manera de poder diseñar un código de este tipo determinando cualquier valor deseado de la capacidad de corrección de errores t.

El polinomio mínimo

Los códigos cíclicos pertenecen a la familia de los códigos de bloques y poseen la particularidad de que las rotaciones cíclicas de las palabras de código pertenecen también al mismo código. Una de las características de los códigos cíclicos es que sus palabras o vectores código son generados multiplicando un determinado polinomio por el polinomio g(X), de tal manera que en el formato no sistemático el polinomio que multiplica a g(X) es el polinomio mensaje m(X) mientras que en el formato sistemático, el polinomio que multiplica a g(X) es un polinomio equivalente q(X). Como todo polinomio, g(X) ha de poseer raíces, que son en número igual al grado del polinomio. Estando definido con coeficientes en el campo binario GF(2), es posible que el polinomio no posea todas sus raíces dentro de ese campo, por lo cual deberá encontrar raíces en un campo extendido, $GF(2^m)$. Se hace referencia a este punto también en el apéndice D, donde se describe al campo de Galois extendido $GF(2^3)=GF(8)$.

Teniendo en cuenta al código de Hamming C_b (7,4), que es cíclico y de bloques, se podrá analizar cuales son las raíces del polinomio que lo genera. De esta forma, y siendo cada palabra del código un múltiplo de ese polinomio generador, se podrá asegurar que cada polinomio código posea entonces las mismas raíces que el polinomio generador $g(X)=g_1(X)$. De esta manera podrá entonces plantearse un conjunto de ecuaciones de síndrome, dado que los polinomios código tienen raíces o ceros conocidos.

Recurriendo a la tabla del campo de Galois extendido GF(8) que se muestra a continuación, puede verse que α es un elemento primitivo(ver Apéndice D) de ese campo, y también una raíz del polinomio primitivo $p(X)=1 + X + X^3$, que no debiera confundirse con el polinomio generador del código $g_1(X)$, a pesar de que en este caso y por coincidencia son iguales.

Representación Exp.	Representación Polinomica	Representación vectorial
0	0	0 0 0
1	1	1 0 0
α	α	0 1 0
α^2	α^2	0 0 1
α^3	$1 + \alpha$	1 1 0
α^4	$+\alpha + \alpha^2$	0 1 1
α^5	$1 + \alpha + \alpha^2$	1 1 1
α^6	$1 + \alpha^2$	1 0 1

Figura 39. Campo de Galois GF(2^3) generado por $p(x)=1+X+X^3$.

De esta forma se puede asegurar que α es raíz de $g_1(X)$ dado que:

$p(\alpha)=1 + \alpha + \alpha^3=0$ ➔ $1 + \alpha = \alpha^3$

Por lo tanto:

$g_1(\alpha) = 1 + \alpha + \alpha^3 = 1 + \alpha + 1 + \alpha = 0$

Este hecho permite aseverar que cualquier polinomio código del código de Hamming C_b (7,4) tiene a α como raíz o cero, generando una primera ecuación de síndrome $s_1 = r(\alpha)$. Restan por lo tanto encontrar dos raíces más del polinomio generador. Si se efectúa el reemplazo del elemento α^2 del campo extendido GF(8) se tiene:

$g_1(\alpha^2)=1 + \alpha^2 + \alpha^6 =1 + \alpha^2 + 1 + \alpha^2 = 0$

Se verifica entonces que α^2 es también raíz del polinomio generador del código de Hamming C_b (7,4), y por lo tanto también de cada uno de sus polinomios código, generando así una segunda ecuación de síndrome $s_2 = r(\alpha^2)$. Por este método de reemplazo puede verificarse que la raíz restante

de $g_1(X)$ es α^4, y que los elementos del campo extendido 1, α^3, α^5 y α^6 no son raíces de $g_1(X)$. Ahora, por ser α, α^2 y α^4 raíces del polinomio generador $g_1(X)$ se cumple que:

$g_1(X) = (X + \alpha)(X + \alpha^2)(X + \alpha^4) = X^3 + (\alpha + \alpha^2 + \alpha^4)X^2 + (\alpha^3 + \alpha^5 + \alpha^6)X + 1 = X^3 + X + 1$

Dado que el polinomio generador $g_1(X)$ tiene como raíces a α y α^2, se tiene un sistema de dos ecuaciones que permite resolver dos incógnitas, que son la posición y el valor de un error por palabra o polinomio código. Esto coincide con el hecho de que por tratarse de un código de Hamming, la distancia mínima es $d_{min}=3$ y la capacidad de corrección es de patrones de t=1 error. Si se deseara una capacidad de corrección mayor, es decir por ejemplo corregir todos los patrones de t=2 errores, se precisarían cuatro ecuaciones. El polinomio generador sólo tiene una raíz adicional, que es α^4 y por lo tanto solo una ecuación de síndrome adicional $S_4=r(\alpha^4)$. Por lo tanto no es posible con este código corregir todos los patrones de t=2 errores.

Descripción de los códigos cíclicos BCH

Para construir un código que pueda ser diseñado partiendo de un dado valor de la capacidad de corrección por palabra t, se definen los códigos BCH, que surgen como una generalización de los códigos de Hamming, donde se extiende la capacidad de corrección de t = 1 a cualquier valor genérico de t. El método está basado en la composición de polinomios mínimos, los que describimos anteriormente.

Para cualquier entero positivo $m \geq 3$ y $t < 2^{m-1}$ existe un código binario BCH con las siguientes propiedades:

Longitud de palabra:	$n = 2^m - 1$
Nº dígitos control de paridad:	$n - k \leq mt$
Distancia mínima:	$d_{min} \geq 2t + 1$

Capacidad de corrección de error: t errores por palabra de código

Estos códigos son capaces de corregir patrones de t o menos errores en palabras de n dígitos, siendo $n = 2^m–1$. El polinomio generador de estos códigos se describe en términos de sus raíces tomadas del campo de Galois $GF(2^m)$.

Si α es un elemento primitivo en $GF(2^m)$ el polinomio generador g(X) de un código BCH corrector de t errores de longitud de palabra $n = 2^m - 1$ es el polinomio de menor grado sobre GF(2) que tiene a α, α^2,..., α^{2t} como sus raíces:

$g(\alpha^i) = 0$; $i = 1,2,....,2t$

De esta manera g(X) tiene a α^i y su conjugado como raíces (ver APENDICE D).

Decodificación de códigos BCH

El polinomio código, el polinomio error y el polinomio recibido tienen la siguiente relación:

$r(X) = c(X) + e(X)$

Siendo códigos cíclicos y de bloques, los códigos BCH se pueden decodificar empleando la decodificación por síndrome recordando que si un polinomio es polinomio código entonces $c(\alpha^i) = 0$.

El producto interno del vector de código $(c_0, c_1,....,c_{n-1})$ con el vector $(1,\alpha^i, \alpha^{2i},...., \alpha^{(n-1)i})$ es cero. Se puede dar forma entonces a la siguiente matriz:

$$\boldsymbol{H} = \begin{bmatrix} 1 & \alpha & \alpha^2 & \alpha^3 & \cdots & \alpha^{n-1} \\ 1 & \alpha^3 & (\alpha^3)^2 & (\alpha^3)^3 & \cdots & (\alpha^3)^{n-1} \\ 1 & \alpha^5 & (\alpha^5)^2 & (\alpha^5)^3 & \cdots & (\alpha^5)^{n-1} \\ \vdots & \vdots & \vdots & \vdots & & \vdots \\ 1 & \alpha^{2t-1} & (\alpha^{2t-1})^2 & (\alpha^{2t-1})^3 & \cdots & (\alpha^{2t-1})^{n-1} \end{bmatrix}$$

De manera tal que si **c** es una palabra de código tendrá que cumplirse entonces que:

$$\mathbf{c} \cdot \mathbf{H}^T = 0$$

Cada elemento de la matriz **H** es un elemento del campo GF(2^m) que puede representarse como un vector de m componentes tomadas del campo GF(2) ordenada en forma de columna, de forma de construir la misma matriz en formato binario.

Si la expresión anterior se emplea junto a la de evaluación del síndrome:
$\mathbf{S} = (s_0, s_1, ..., s_{2t}) = \mathbf{r} \cdot \mathbf{H}^T$

Se obtiene el conjunto de condiciones:

$$s_i = r(\alpha^i) = e(\alpha^i) = r_0 + r_1(\alpha^i) + ... + r_{n-1}(\alpha^i)^{n-1}$$

con $1 \leq i \leq 2t$

Que permite la evaluación de la componente i–ésima del vector síndrome reemplazando la raíz α^i en el polinomio recibido r(X). Los elementos componentes del vector síndrome son elementos del campo GF(2^m).

Ejemplo:

Para el código de bloques, binario y cíclico BCH (15,7) con t=2, si el vector recibido
es **r** = 100000001000000 que en forma polinómica es $r(X)=1+X^8$, el vector de síndrome es;

$$s_1 = r(\alpha) = 1 + \alpha^8 = \alpha^2$$

$s_2=r(\alpha^2)=1+ \alpha = \alpha^4$

$s_3=r(\alpha^3)=1+ \alpha^9 = 1+ \alpha + \alpha^3 = \alpha^7$

$s_4=r(\alpha^4)=1+ \alpha^2 = \alpha^8$

Polinomios de localización y evaluación de error

Los coeficientes de los polinomios que conforman la relación r(X) = c(X) + e(X) pertenecen al campo GF(2). Se supone que el vector error posee τ elementos no nulos representado un patrón de error de τ errores ubicados en posiciones X^{j1}, X^{j2}, ..., $X^{j\tau}$, siendo $0 \leq j1<j2<...<j\tau \leq n-1$.

Se define entonces el número de localización de error:

$\beta_l=\alpha^{jl}$ donde l=1, 2, 3,..., τ

El cálculo de los síndromes es el mismo que el ya planteado y se realiza evaluando r(X) en las raíces α^i con i =1, 2, ...,2t. Se cumple una vez más que:

$s_i= r(\alpha^i) = c(\alpha^i) + e(\alpha^i) = e(\alpha^i)$

Surge así un sistema de 2t ecuaciones:

$s_1= r(\alpha) = e(\alpha) = e_{j1}\beta_1 + e_{j2}\beta_2 +...+ e_{j\tau}\beta_\tau$

$s_2= r(\alpha^2) = e(\alpha^2) = e_{j1}\beta_1^2 + e_{j2}\beta_2^2 +...+ e_{j\tau}\beta_\tau^2$

.

.

.

$s_{2t}= r(\alpha^{2t}) = e(\alpha^{2t}) = e_{j1}\beta_1^{2t} + e_{j2}\beta_2^{2t} +...+ e_{j\tau}\beta_\tau^{2t}$

Las variables β_1, β_2, ..., β_τ son desconocidas. Un algoritmo que resuelva este conjunto de ecuaciones es un algoritmo decodificador de un código BCH. Las variables desconocidas son las posiciones de error. En

general existen 2^k diferentes soluciones para este sistema pero aquella solución de patrón de error que tenga el peso mínimo será la verdadera solución, cuando los errores son aleatorios.

Para la decodificación de códigos cíclicos BCH se definen los siguientes dos polinomios:

El polinomio localizador de errores:

$$\sigma(X) = (X - \alpha^{-j_1}).(X - \alpha^{-j_2})....(X - \alpha^{-j_\tau}) = \prod_{l=1}^{\tau}(X - \alpha^{-j_l})$$

y el polinomio evaluador de errores:

$$W(X) = \sum_{l=1}^{\tau} e_{j_l} . \prod_{\substack{l=1 \\ l \neq l}}^{\tau}(X - \alpha^{-j_l})$$

Para este trabajo, este último polinomio no es necesario ya que solo se utiliza en el caso no binario ($q \geq 3$). En el caso binario el valor de error es siempre 1.

El conocimiento de los polinomios $\sigma(X)$ y W(X) permite evaluar las posiciones y el valor del error por empleo de la ecuación anterior. Adicionalmente se define el polinomio síndrome de grado gr (S(X)) $\leq$ 2t-1 como:

$$S(X) = s_1 + s_2.X + s_3.X^2 + ... + s_{2t}.X^{2t-1} = \sum_{j=0}^{2t-1} s_{j+1} X^j$$

La condición S(X)=0 indica que el polinomio recibido pertenece al código. Existe una relación entre los polinomios $\sigma(X)$, S(X) y W(X) que se denomina la ecuación clave, y cuya resolución es un método de decodificación de un código BCH. La relación entre los citados polinomios la determina el siguiente teorema:

Teorema: Existe un polinomio μ(X) tal que los polinomios $\sigma(X)$, S(X) y W(X) satisfacen la ecuación clave:

$\sigma(X).S(X) = -W(X) + \mu(X).X^{2t}$

Por simplicidad se obviará la demostración. La ecuación clave ofrece un método de decodificación para códigos BCH. Se emplea en este caso el algoritmo Euclidiano, que no sólo es aplicable a números, sino también a polinomios.

Decodificación de códigos BCH utilizando el algoritmo de Euclides

Dados dos números o polinomios A y B el algoritmo Euclidiano permite conocer el máximo común divisor entre ellos, C = MCD(A,B). El algoritmo también permite encontrar dos polinomios o números enteros S y T tales que:
C = S.A + T.B

Se pretende emplear este algoritmo para resolver la ecuación clave que involucra a los polinomios anteriormente definidos:

$-\mu(X).X^{2t} + \sigma(X).S(X) = -W(X)$

donde X^{2t} hace las veces de A y el polinomio síndrome S(X) de B.

Algoritmo de Euclides

Sean A y B dos números enteros o polinomios de forma que $A \geq B$ o equivalentemente $gr(A) \geq gr(B)$. La condición inicial es que $r_{-1} = A$ y $r_0 = B$. Si se realiza un cálculo recursivo se tendrá en la recursión i-ésima que r_i se obtiene como residuo de la división de r_{i-2} por r_{i-1} , es decir que:

$r_{i-2} = q_i.r_{i-1} + r_i$

siendo $r_i < r_{i-1}$ o bien para los polinomios $gr(r_i) \geq gr(r_{i-1})$.
La ecuación recursiva es entonces:

$$r_i = r_{i-2} - q_i.r_{i-1}$$

Se obtienen también expresiones para s_i y t_i que hacen que

$$r_i = s_i .A + t_i .B$$

Siendo la ley recursiva igualmente valida para ellos:

$$s_i = s_{i-2} - q_i.s_{i-1}$$
$$t_i = t_{i-2} - q_i.t_{i-1}$$

Luego:

$$r_{-1} = A =(1).A + (0).B$$
$$r_0 = B =(0).A + (1).B$$

establecen como condiciones iniciales:

$$s_{-1} = 1, \quad t_{-1} = 0$$

Ejemplo: aplíquese el algoritmo Euclidiano a los números A =112 y B = 54.
112/ 54 = 2 con resto 4. Luego:
$4 = 112 + (-2).54$
$r_1 = r_{-1} - q_1.r_0$
$r_{-1} = 112$, $r_0 = 54$, $r_1 = 4$
54/4 = 13 con resto 2.

$2 = 54 + (-13).4$
$r_2 = r_0 - q_2.r_1$
$r_2 = 2$
4/2 = 2 con resto 0.

Luego 2 es el MCD entre 112 y 54.

Este algoritmo se puede implementar por medio de una tabla:

i	$r_i = r_{i-2} - q_i . r_{i-1}$	q_i	$s_i = s_{i-2} - q_i . s_{i-1}$	$t_i = t_{i-2} - q_i . t_{i-1}$
-1	112		1	0
0	54		0	1
1	4	2	1	-2
2	2	13	-13	27

Tabla 4. Algoritmo de Euclides para calcular el MCD entre dos números enteros.

El MCD es 2 porque el próximo resto en el proceso de la tabla es 0. En cada paso de la recursión sucede que:

112 = (1).112 + (0).54
54 = (0).112 + (1).54
4 = (1).112 + (-2).54
2 = (-13).112 + (27).54

es decir:

$C = S.A + T.B = r_i = s_i .A + t_i .B$

Se aplica ahora este procedimiento a la ecuación clave:

$-\mu(X).X^{2t} + \sigma(X).S(X) = -W(X)$

El método opera sobre los polinomios X^{2t} y S(X) y la recursión i-ésima tiene la forma:

$r_i(X) = S_i(X).X^{2t} + t_i(X).S(X)$

Multiplicando por una constante λ la ecuación anterior se obtiene:

$\lambda.r_i(X) = \lambda.S_i(X).X^{2t} + \lambda.t_i(X).S(X) = -W(X) = -\mu(X).X^{2t} + \sigma(X).S(X)$

siendo $gr(r_i(X)) \leq t+1$

donde λ es una constante que hace que el polinomio sea mónico, es decir que el coeficiente de la mayor potencia sea unitario.

Véase otro ejemplo: para el código de bloques, cíclico y binario BCH (15,7) con t = 2, si el vector recibido es **r** = 100000001000000 que en forma polinómica es $r(X) = 1 + X^8$, que es el caso presentado en el ejemplo anterior, se va a determinar por el algoritmo de Euclides la palabra decodificada correspondiente.

De acuerdo al patrón recibido:

$r(X) = 1 + X^8$

Las componentes del vector síndrome que ya fueron calculadas son:

$s_1 = r(\alpha) = \alpha^2$

$s_2 = r(\alpha^2) = \alpha^4$

$s_3 = r(\alpha^3) = \alpha^7$

$s_4 = r(\alpha^4) = \alpha^8$

Luego el polinomio síndrome adopta la forma:

$S(X) = \alpha^8 X^3 + \alpha^7 X^2 + \alpha^4 X + \alpha^2$

Ha de tenerse en cuenta que al operar sobre un campo de Galois, el inverso aditivo de un elemento es ese mismo elemento (ver Apéndice D), con lo que los signos "-" en las ecuaciones anteriores pasan a ser signos "+". Se aplica entonces el método de Euclides conformando la siguiente tabla:

i	$r_i = r_{i-2} - q_i . r_{i-1}$	q_i	$t_i = t_{i-2} - q_i . t_{i-1}$
-1	$X^{n-k} = X^4$		0
0	$S(X) = \alpha^8 X^3 + \alpha^7 X^2 + \alpha^4 X + \alpha^2$		1
1	$\alpha^4 X^2 + \alpha^{13} X + \alpha^8$	$\alpha^7 X + \alpha^6$	$\alpha^7 X + \alpha^6$
2	α^5	$\alpha^4 X + \alpha^8$	$\alpha^{11} X^2 + \alpha^5 X + \alpha^3$

Tabla 5. Algoritmo de Euclides para la ecuación clave.

Si la columna $r_i(X)$ tiene grado menor que la columna $t_i(X)$ se detiene el proceso recursivo. Se tiene entonces:

$r_i(X) = \alpha^5$

$t_i(X) = \alpha^{11} X^2 + \alpha^5 X + \alpha^3$

Luego se procede a multiplicar por un factor $\lambda \in GF(2^4)$ para convertir a $t_i(X)$ en un polinomio mónico. Este valor de λ es $\lambda = \alpha^4$. Así se tiene:

$W(X) = -\lambda r_i(X) = \lambda r_i(X) = \alpha^4\alpha^5 = \alpha^9$

y

$\sigma(X) = \lambda t_i(X) = \alpha^4(\alpha^{11}X^2 + \alpha^5 X + \alpha^3) = X^2 + \alpha^9 X + \alpha^7$

Se efectúa la búsqueda de las raíces del polinomio localizador de errores por reemplazo de los elementos del campo en ese polinomio, algoritmo denominado de Chien. Una forma simple de determinar las raíces del polinomio $\sigma(X)$ es reemplazando la variable X por 1, α, α^2 ,..., α^{n-1} siendo $n = 2^m - 1$. Como $\alpha^n = 1$ y $\alpha^{-h} = \alpha^{n-h}$ entonces si α^h es raíz del polinomio $\sigma(X)$, α^{n-h} es el correspondiente número de localización de error, que define al dígito r_{n-h} como un dígito con error. En el caso binario esta información es suficiente para realizar la corrección ya que habiendo sólo dos valores posibles, r_{n-h} será cambiado en valor para corregirlo.

Aplicando el algoritmo descripto para el caso del ejemplo analizado se encuentra que las raíces del polinomio son $\alpha^{-j1} = 1$ y $\alpha^{-j2} = \alpha^7$. Luego como:

$\alpha^0 = \alpha^{-j1} = \alpha^0$

j1=0 y j2=8

Los errores se encuentran en posiciones j1=0 y j2=8. Para establecer el valor de los errores se necesita conocer el polinomio derivado:

$\sigma'(X) = \alpha^9$

En este caso, el valor de los errores es previamente conocido, dado que operando con coeficientes en el campo binario GF(2), dicho valor de error debe ser 1.

El polinomio error es entonces:
$e(X) = X^0 + X^8 = 1 + X^8$

Se corrige el vector recibido resultando como consecuencia que el mensaje enviado fue la palabra todos ceros.

Implementación en FPGA

Ahora, a partir de lo desarrollado en forma teórica se está en condiciones de analizar el bloque que se ha diseñado. Como se vio, el primer paso es calcular los coeficientes del síndrome. La cantidad de coeficientes está dada por el valor 2t, que en este caso da como resultado 2 coeficientes (más adelante se verá como en realidad sólo es necesario calcular s_1). Como ya se mostró en la introducción teórica, el cálculo de los coeficientes está dado por las siguientes ecuaciones:

$s_1 = r(\alpha) = r_0 + r_1\alpha^1 + r_2\alpha^2 + r_3\alpha^3 + ... + r_{14}\alpha^{14} = \alpha^m$

$s_2 = r(\alpha^2) = r_0 + r_1\alpha^2 + r_2\alpha^4 + r_3\alpha^6 + ... + r_{14}\alpha^{28} = \alpha^n$

En consecuencia, el polinomio síndrome adopta la forma:
$S(X) = \alpha^n X + \alpha^m$

El siguiente paso consiste en aplicar el algoritmo de Euclides:

$$
\begin{array}{l|l}
X^{n-k} = X^2 & \alpha^n X + \alpha^m \\
\hline
X^2 + \alpha^{15-n+m} X & \alpha^{15-n} X + \alpha^{15-2n+m} \\
\hline
\alpha^{15-n+m} X & \\
\alpha^{15-n+m} X + \alpha^{15-2n+2m} & \\
\hline
r(X) = \alpha^{15-2n+2m} &
\end{array}
$$

Se conforma la siguiente tabla:

i	$r_i = r_{i-2} - q_i \cdot r_{i-1}$	q_i	$t_i = t_{i-2} - q_i \cdot t_{i-1}$
-1	$X^{n-k} = X^2$		0
0	$S(X) = \alpha^n X + \alpha^m$		1
1	$\alpha^{15-2n+2m}$	$\alpha^{15-n} X + \alpha^{15-2n+m}$	$\alpha^{15-n} X + \alpha^{15-2n+m}$

Tabla 6. Aplicación del algoritmo de Euclides.

Como la columna $r_i(X)$ tiene grado menor que la columna $t_i(X)$ se detiene el proceso recursivo. Se tiene entonces:

$r_i(X) = \alpha^{15-2n+2m}$

$t_i(X) = \alpha^{15-n}X + \alpha^{15-2n+m}$

Ahora se debe obtener el polinomio localizador:

$\sigma(X) = \lambda t_i(X) = \alpha^n(\alpha^{15-n}X + \alpha^{15-2n+m}) = X + \alpha^{15-n+m}$

La raiz del polinomio es α^{15-n+m}, por lo que el error se localiza en la posición:

15 - (15-n+m) = n – m

En la siguiente tabla se muestran los valores que toman los coeficientes del vector síndrome en función de la posición del error:

POSICION	$s_1 = \alpha^m$	$s_2 = \alpha^n$	n - m
0	1	1	0
1	α	α^2	1
2	α^2	α^4	2
3	α^3	α^6	3

4	α^4	α^8	4
5	α^5	α^{10}	5
6	α^6	α^{12}	6

Tabla 7. Coeficientes del vector síndrome para cada posición.

Como se ve, si bien el valor de n-m coincide con el valor de la posición del error, este valor también puede obtenerse calculando solo s_1, ahorrando de esta manera, cálculos innecesarios. En otras palabras, calculando s_1 como una potencia de α coincide que el valor de dicha potencia (m) es igual a la posición del error.

En el caso de que ambos coeficientes valieran cero, no habría que realizar corrección alguna. Este sistema puede detectar y corregir un solo error, ya que siempre ocurre que n es el doble de m, siendo n y m las potencias de α en s_2 y s_1 respectivamente. El cálculo de s_1 es efectuado de la siguiente forma:

Se van leyendo uno a uno los bits recibidos, r_t. Si r_0 es '1' se guarda el valor que corresponde al '1' en el campo extendido GF(2^4) (generado por $p(X) = 1 + X + X^4$) que es "0001", en caso contrario se guarda el "0000". Luego si r_1 es '1', se suma en forma de or exclusiva el valor correspondiente a α que es "0010", si r_1 = '0' se suma "0000". Continuando, si r_2 es '1', se suma el valor correspondiente a α^2, "0100" y nuevamente, si r_2 = '0' se suma "0000". Este procedimiento se extiende hasta llegar a r_{14}, los valores que se deben ir sumando son suministrados por un multiplexor.

Una vez calculado s_1, con el valor obtenido se maneja la selección de otro multiplexor que tiene almacenadas las correcciones que deben realizarse en función del síndrome. Por ejemplo, si $S_1 = \alpha^2$, la palabra de corrección es "0000100". El bit a corregir es el tercero empezando desde la derecha. Véase el siguiente esquema:

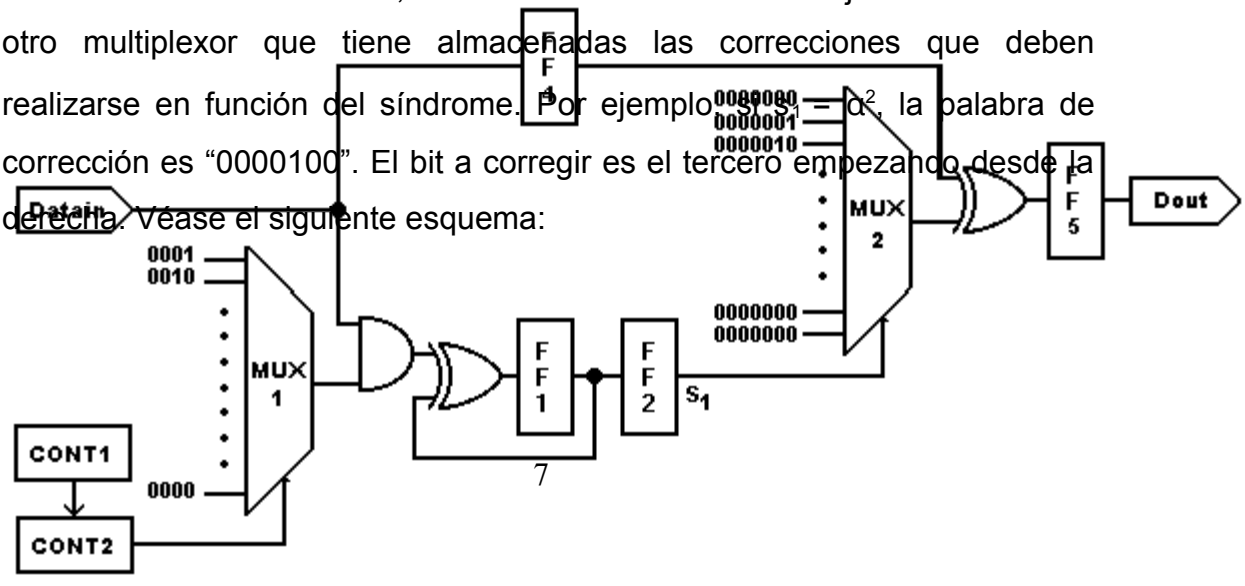

Figura 40. Diagrama del bloque Coefs.

"FF4" es un registro de desplazamiento, los datos entran por FF4(6) y se desplazan en cadena hasta FF4(0), esta disposición se debe a que así queda el bit menos significativo del vector recibido en el menos significativo de "FF4" y "FF5". "MUX1" es quien aporta los valores digitales de 1, α, α^2, etc. "CONT1" espera que se hayan leído los 30 bits del patrón, para luego habilitar a "CONT2" iniciando el proceso. "CONT2" es el que determina con que bit del vector recibido se está trabajando. Una vez calculado s_1 se decide que bit se debe corregir, si ese fuera el caso. En función del valor de s_1 se almacenan en "MUX2" los distintos vectores de corrección. El programa VHDL se encuentra en el APENDICE F, programa 9.

El clock que acompaña a los datos es obtenido a partir del bit menos significativo de "CONT2". Como el clock debe aparecer recién cuando se encuentra lista la primera decodificación, se utiliza un flip flop para retardar su aparición y un contador para extenderlo cuando ya se recibió la última palabra. Así se obtienen finalmente los datos junto a un clock coherente como salida del receptor. A continuación se describe el desempeño del banco experimental implementado.

BANCO DE PRUEBAS

Para comprobar el funcionamiento del sistema bajo ciertas condiciones, se decidió armar un banco de pruebas que contaba con los siguientes elementos:

- Una PC con puerto paralelo y el programa Max+Plus II instalado.
- Cable ByteBlaster.
- Dos FPGA's de Altera, Flex 10K20.
- Un Conversor Digital Analógico, DAC0801.
- Un Amplificador Operacional, TL081.
- Un Conversor Analógico Digital, ADC 0804.
- Un generador de Funciones.
- Dos osciloscopios.
- Fuentes de alimentación.

Conexiones

La PC se utiliza únicamente para la programación de las FPGA's, por eso es necesario disponer del cable ByteBlaster, el cual se debe conectar al LPT de la computadora, para realizar dicha programación.

Por medio del generador de funciones se entrega el clock al transmisor, la frecuencia de este último tiene una relación con la frecuencia de salida (frecuencia de transmisión serie) de 0,127, es decir que $f_{transm} = 0.127\ f_{clock}$ Como se utiliza el ADC con una frecuencia de muestreo de 3 KHz, la frecuencia de transmisión debe estar alrededor de 60 Hz para tener 50 muestras por bit. Por esto, la frecuencia del clock entregado al dispositivo programable es de 470 Hz, ya que 470 Hz x 0,127 ≈ 60 Hz. La utilidad de utilizar un generador de funciones como clock es que al tener la posibilidad de variar su frecuencia, se puede verificar el funcionamiento del sistema a distintas frecuencias de transmisión y encontrar el límite de variación para el cual el sistema mantiene un desempeño óptimo.

Por otro lado para simplificar el análisis del funcionamiento del sistema se implementó un pequeño programa que proporciona al transmisor los datos a transmitir, estos datos son la salida de un contador. El bloque de datos a transmitir está compuesto de 16 palabras de 7 bits de información (más los respectivos bits de codificación), esto significa que se transmiten:

7x16 = 112 bits

Para que la trama enviada sea siempre la misma, y se facilite así la comprobación del funcionamiento del sistema, se decidió hacer que el contador cuente desde el valor 0 al 13, en palabras de 8 bits ya que:

8x14 = 112 bits

De esta manera el contador, programado en la misma FPGA que el transmisor, entrega siempre los mismos valores al transmisor. El contador es un bloque externo al transmisor y funciona en forma independiente a este último.

Una vez programada la FPGA que será el transmisor y conectado el generador que hará las veces de clock, se debe conectar al circuito que transmitirá y dará el formato analógico a la onda. Este circuito se compone del DAC en su configuración bipolar y con offset simétrico, en el siguiente esquemático se muestran las conexiones:

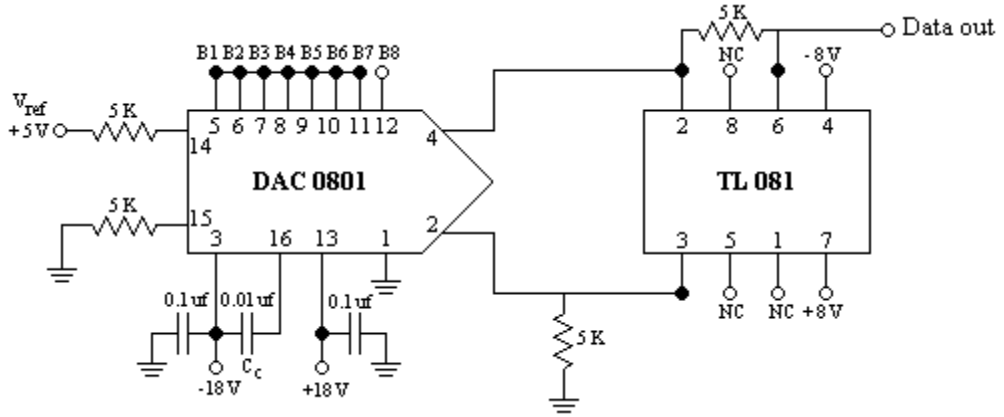

Figura 41.Configuración del conversor digital-analógico.

Como se ve, las entradas de los 7 bits menos significativos están conectadas entre si, dado que la FPGA proporciona al DAC únicamente 2 bits

ya que con 4 estados es suficiente para conformar la onda en los formatos utilizados. Por medio del V_{ref} se puede variar la tensión de salida. Este fue ajustado en 5V debido a que el conversor analógico digital del receptor fue preparado para recibir la señal en el rango de ±5V. La tensión de alimentación del amplificador operacional fue ajustada al mínimo que asevera un comportamiento óptimo. Si bien en la hoja de datos figura que el voltaje de alimentación máximo permitido es ±18V (que coincide con la tensión de alimentación de DAC), en el laboratorio no fue posible utilizar dicho valor porque con esa tensión de entrada se quemó en dos ocasiones. Aunque inicialmente se comenzó trabajando con un amplificador operacional LM741, porque así lo indicaba la hoja de datos del DAC, luego fue reemplazado por el TL081 que es un amplificador operacional similar pero con un diseño más moderno y de mayores prestancias.

La línea de transmisión, *Data out* en el gráfico anterior, se conecta a la entrada del conversor analógico digital, es decir la entrada al sistema receptor completo. También es conveniente visualizar la trama de salida del transmisor, por medio de un osciloscopio, para verificar que posea la cantidad de símbolos correcta. El receptor está compuesto por el ADC y una FPGA, el ADC se configura de la siguiente manera:

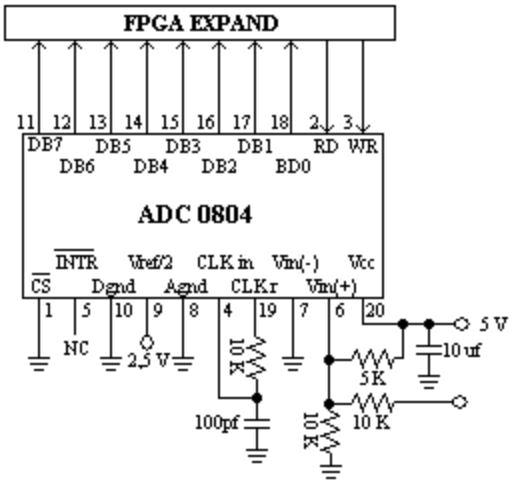

Figura 42. Configuración del conversor analógico-digital.

Con la anterior configuración el conversor es capaz de manejar voltajes de entre ±10V, con esto se cumple con los niveles de tensión de entrada

anteriormente mencionados en el informe. Otra observación importante es que las entradas RD (Read) y WR (Write), que controlan la toma de muestras, son manejadas por la misma FPGA, que fue programada para llevar a cabo esa función también. En esta configuración del ADC se aprovecha su característica de clock on board, por lo tanto con la combinación de la resistencia y el capacitor, conectados en el pin 19 del integrado, se obtiene la frecuencia deseada. Por medio del gráfico f_{clk} vs clock capacitor disponible en la hoja de datos del integrado, se obtienen los valores necesarios para lograr la frecuencia deseada.

Ya con los datos entregados por el ADC, la FPGA programada como receptor procesa símbolo a símbolo y va entregando en su salida los datos junto con un clock coherente a estos últimos. Con un osciloscopio con un canal conectado a la salida de datos y otro al clock de salida, es posible corroborar el buen funcionamiento del equipo. Entonces finalmente el banco de pruebas completo es el siguiente:

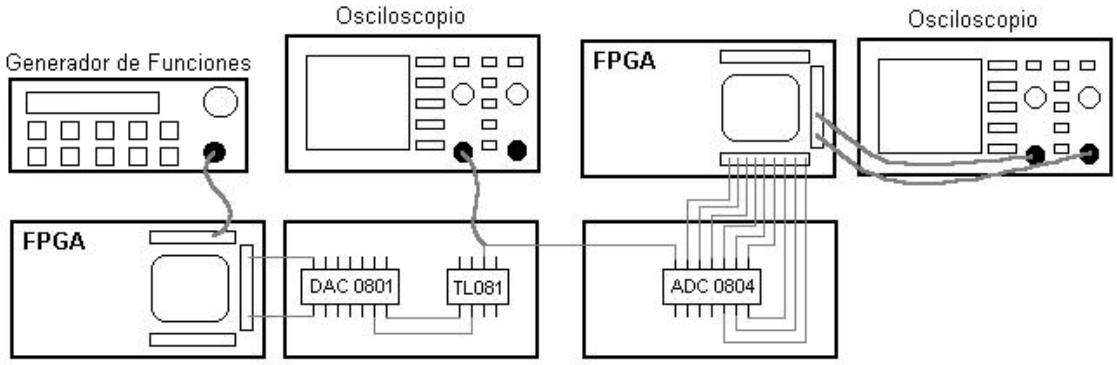

Figura 43. Banco de pruebas.

Mediciones

El generador de funciones se ajustó a una frecuencia de 470 Hz, onda cuadrada, 3 Vpp de amplitud y con un offset de 1,5 V. En el osciloscopio se ve de la siguiente manera:

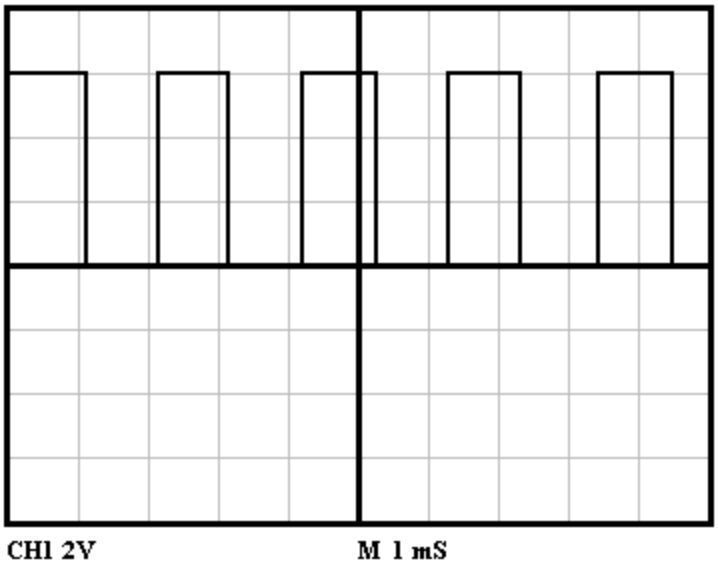

Figura 44. Osciloscopio, señal de clock del transmisor.

Luego puede verse a la salida del amplificador operacional la trama transmitida:

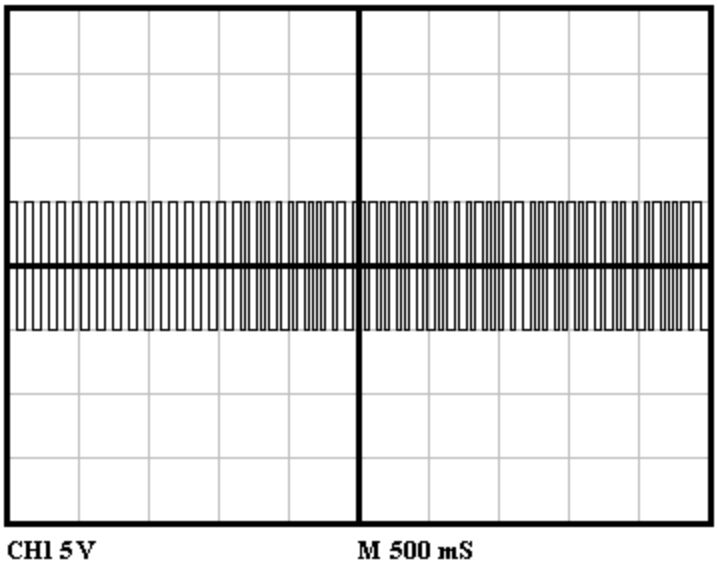

Figura 45. Osciloscopio, transmisión de trama.

Aquí se ha ejemplificado una transmisión en formato Manchester, donde se puede ver en el comienzo de la trama el patrón de 30 unos y ceros, y a continuación los restantes bits que corresponden a las palabras codificadas. También se aprecia que la amplitud es la adecuada según los cálculos efectuados en desarrollos previos.

En el siguiente gráfico se ve la salida serie del receptor en un canal, mientras que en otro se aprecia el clock que permite la lectura de los bits detectados.

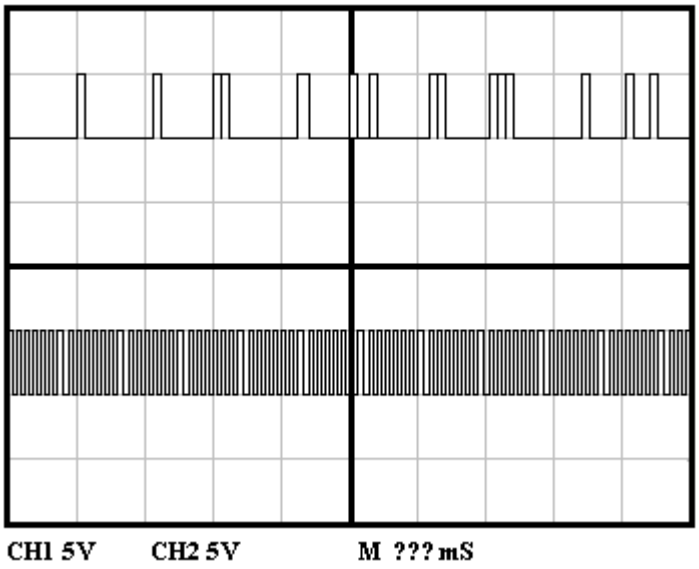

Figura 46. Osciloscopio, salida de datos y clock del receptor.

Para simplificar la comprensión aquí se ha supuesto una base de tiempo tal que por cada división temporal entran 8 ciclos de clock. En la parte inferior de la figura se graficó dicho clock, es posible ver que 1 pulso de cada 7 tiene una duración visiblemente mayor a la de los anteriores, esto se debe a que en la decodificación se extraen 7 bits de información de cada palabra de 15 bits. Como esta extracción debe estar en sincronismo con la llegada de los siguientes 15 bits, se los enfila con una duración de dos pulsos de clock cada uno, con excepción del último que ocupa tres pulsos para completar los 15 pulsos. Así la salida de los 7 bits decodificados tiene la misma duración que la palabra codificada. Esto influye en la salida de datos, ya que el símbolo que representa ese pulso de clock también tendrá una duración mayor, en el gráfico anterior se puede visualizar eso en el pulso ubicado en la 5ta división de izquierda a derecha. A continuación se altera la base de tiempo para un análisis más profundo.

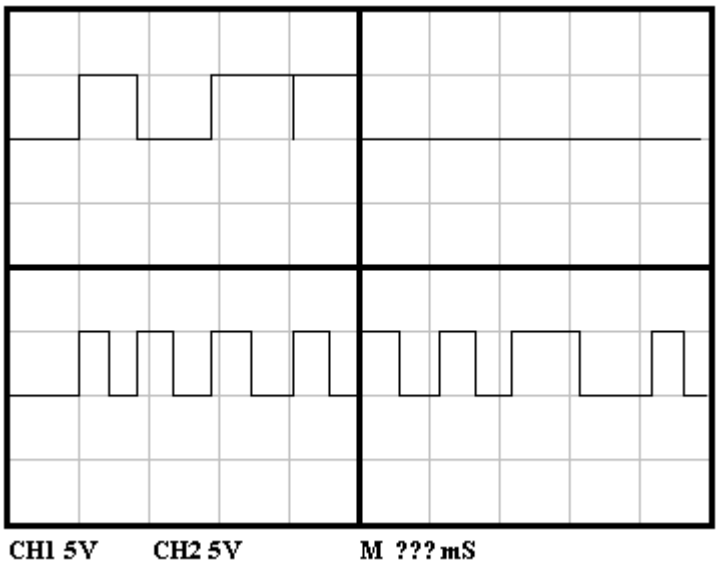

Figura 47. Osciloscopio, palabra de 7 bits recibida.

Si se observa la parte inferior, es decir el clock de salida, se ve como cada pulso de clock puede llegar a tener una duración distinta del anterior, esto se debe a que el sistema para mantener el sincronismo varía la frecuencia y la fase (variables TOT y DPOS en VHDL), para procesar cada uno de los símbolos que van ingresando. Por otra parte aquí se aprecia mejor que el séptimo pulso de clock es visiblemente mas ancho que los demás, esto se debe a lo anteriormente explicado. En cuanto a la salida de datos, si lo analizamos como una palabra de 8 bits que ingresó, es posible concluir que se trataba del número 13 en binario, 00001101, ya que en osciloscopio el primer bit de izquierda a derecha es el menos significativo. Además aquí también es posible ver como cada pulso tiene una duración distinta, esto es producto también del acomodamiento en fase y frecuencia del algoritmo de sincronismo. Por esto último resultan de mucha utilidad los flancos descendientes de cada pulso de clock, para tomarlos como un parámetro fiel para la lectura de los datos de salida.

Una vez establecida una transmisión a la frecuencia central es posible encontrar el rango de frecuencia para el cual el sistema permanece enganchado. Para hallarlo se incrementó y luego se disminuyó la frecuencia de a intervalos de 10 Hz, observando a la salida del receptor que los datos decodificados fueran acertados. El sistema al desengancharse falla en la

lectura de un bit, leyendo un mismo bit dos veces (el caso del límite de frecuencia más bajo) o perdiendo algún bit (cuando la frecuencia supera el límite superior). En cualquiera de estos dos casos el sistema pierde la capacidad de decodificar acertadamente las palabras que componen la trama. Esto se debe a que al perder o duplicarse la lectura de un bit, las palabras de 15 bits comienzan a ingresar en forma errónea al decodificador, y éste interpreta que se trata de palabras con errores y las corrige de manera incorrecta. Los límites para los cuales se verificó un correcto funcionamiento del sistema fueron una frecuencia de 520 Hz, es decir 50 Hz por encima de la frecuencia central y 420 Hz que es 50 Hz por debajo de esta última. Estos valores coinciden con los que se obtuvieron por medio de las simulaciones. Por lo tanto es posible afirmar que el sistema se comporta satisfactoriamente en un intervalo de desviación de frecuencia de aproximadamente el ±10% de la frecuencia central.

Cabe destacar que las mediciones anteriormente descriptas fueron realizadas para los dos formatos disponibles. Para configurar el modo de trabajo para cada formato en cada FPGA se asignó la entrada que controla esta característica a un switch (que trae de fábrica el dispositivo). Por lo tanto se debe cambiar manualmente la posición de este switch en ambas FPGAs para seleccionar el formato con el cual se trabajará.

CONCLUSIÓN

Para finalizar este trabajo se concluirá sobre algunos aspectos del banco de pruebas que se llevó a cabo, así como sobre algunas posibles modificaciones a realizar en el sistema en aras de mejorar su funcionamiento, y de disminuir su complejidad en ciertas funciones.

Banco de pruebas

A partir de los ensayos efectuados sobre el banco armado, se puede decir que el sistema tuvo un comportamiento satisfactorio. Se logró efectuar la transmisión para ambos formatos en un rango de frecuencia de ± 10% respecto de la frecuencia central. Esto coincide con el margen de trabajo para el cual fue diseñado el sistema, es decir que las simulaciones en Matlab del algoritmo de sincronismo, y las simulaciones en Max+Plus II del diseño en VHDL fueron coherentes a los ensayos practicados. Este margen da al sistema una gran versatilidad, dado que, si bien los cristales son muy estables, puede existir una variación considerable entre dos cristales de las mismas características. Esto implica que un mismo receptor tiene la capacidad de trabajar con varios transmisores distintos sin presentar inconvenientes a nivel de sincronismo.

Con respecto a las condiciones de ruido y jitter cabe mencionar que el banco de prueba utilizado no permitió un estudio en profundidad del sistema en condiciones de ruido considerables. Las pruebas realizadas fueron en condiciones de ruido propias del laboratorio.

La frecuencia de transmisión con la que se trabajó se vio lamentablemente, muy limitada por el conversor analógico digital, CAD 0804, el cual permite como máximo teórico una frecuencia de muestreo de 10 KHZ. Por lo tanto al tomar 50 muestras por bit transmitido la máxima frecuencia de trabajo posible con este modelo de conversor es 10000/50 Hz = 200Hz.

Con respecto al trabajo con tecnología de diseño semi-custom, como lo son las FPGA's vale decir que es una herramienta muy poderosa y de muy fácil acceso. La versatilidad, y la sencillez tanto en el diseño como en la simulación e implementación, hacen de las FPGA´s una muy buena opción para el desarrollo de dispositivos digitales.

Posibles modificaciones futuras

En una futura implementación del sistema se propone el conversor ADC0806, el cual tiene una frecuencia de muestreo máxima de 60 MHZ, con el cual se podría transmitir a 1,2 MHz. A su vez dada la capacidad de la FPGA disponible, EPF10K20RC240E, la velocidad máxima teórica de transmisión queda limitada a 3,8 KHz, esto también se soluciona utilizando un modelo de FPGA con mayor capacidad, por ejemplo el Cyclone II de Altera, la cual se promociona como la FPGA de más bajo costo que haya existido.

El conversor digital analógico CDA elegido es de 8 bits porque en principio se pensó en generar distintos formatos entre ellos el pulso de Nyquist. Para generar el formato basta con construir una tabla con los valores de las muestras como se realiza en el programa 10 del apéndice F, en el cual se construye un pulso genérico de 64 muestras. Igualmente esto no se llevó a cabo, por lo cual hubiera bastado con un conversor de dos bits, el cual permite generar los tres distintos niveles de tensión que requieren el formato Manchester y Bipolar con Retorno a Cero.

Una alternativa para incrementar la frecuencia de transmisión, con la misma tecnología es disminuir la cantidad de muestras tomadas por bit transmitido. Esto también permitiría tener un requerimiento menor en cuanto al tamaño de la FPGA, ya que se acumularía y se procesaría menos información en el receptor. Con respecto a las simulaciones en Max+Plus II, esto implicaría una gran ventaja ya que disminuirían notablemente los tiempos de simulación que han sido en algunos casos de varias horas. Pero esto haría cambiar sustancialmente el diseño, y bajo los análisis realizados para esta implementación, no se puede aseverar nada

respecto al correcto funcionamiento del sistema con estas modificaciones. Por ende esta alternativa queda solo sugerida para futuros estudios.

El tamaño de trama quedó limitado por la capacidad de la FPGA usada para el transmisor, pero este puede incrementarse usando una memoria externa. El receptor en cambio no ofrece ninguna limitación en este sentido, ya que va decodificando por palabra y se detiene una vez detectado el fin de trama. Se puede decir que es independiente al tamaño de trama.

Otra modificación que podría realizarse es aumentar la cantidad de bits con que trabajan las variables E1 y E2 y de todos los flip flops relacionados. Esto permite una mayor seguridad para evitar un desborde cuando se realizan las integraciones en el bloque "ER" (algoritmo Early Late). Esto es especialmente útil frente a ruidos cuya naturaleza no presente una media cercana a cero.

Una mejora que se podría implementar en una futura aplicación es que la entrada de formato del transmisor se maneje desde el dispositivo que proporciona los datos a transmitir, es decir que junto a los datos se brinde al transmisor una señal que controle el formato en que serán enviados. Y en el receptor se incluya un mecanismo que de acuerdo a los primeros bits se detecte el formato de la señal y se ajuste automáticamente para procesar los datos con el algoritmo que corresponda. Esta implementación se podría llevar a cabo por medio de un comparador que para cada muestra que se encuentre en el nivel de un cero, incremente la cuenta de un contador, luego de unos pocos bits se podrá decidir si el formato es Manchester o Bipolar con Retorno a Cero, ya que el único que posee el nivel de tensión cero es el último.

APENDICE A

Método de sincronismo a lazo abierto.

Programa en Matlab, formato Bipolar con retorno a cero:

```
Y=0;S=0;S3=0;pos=0;t3=0;S1=0;t1=0;    %inicialización de variables
c=0;x=0;x2=0;vec=0;nvec=0;evec=0;

f=150; %tamaño del bloque compuesto por unos y ceros alternados
g=150; %tamaño del bloque compuesto por bits aleatorios
jitt=1; %coeficiente de jitter
ruido=.3; %dispersión del ruido
p=55; %cantidad de muestras que componen un bit

for u=1:f
  n=p+round(jitt*randn(1));
  nvec(u)=n;
  S3(pos+1:pos+round(n/2))= (-1)^(u+1)*ones(1,round(n/2));
  S3(pos+round(n/2)+1:pos+n)= zeros(1,n-round(n/2));
  x2(u)=S3(pos+round(n/4));
  pos=pos+n;
end; %generación del bloque compuesto por unos y ceros alternados

for u=1:g
  n=p+round(jitt*randn(1));
  nvec(u+f)=n;
  S3(pos+1:pos+round(n/2))= sign(randn(1)*round(n/2));
  S3(pos+round(n/2)+1:pos+n)= zeros(1,n-round(n/2));
  x2(u+f)=S3(pos+round(n/4));
  pos=pos+n;
end; %generación del bloque compuesto por bits aleatorios

t3=[1:pos];
No=ruido*randn(1,pos);
Sn=S3+No; %suma de ruido a la señal
P=ones(1,50); %generación del filtro adaptado
cnv=conv(P,Sn); %convolución de la señal ruidosa con el fil. adaptado
cnvrect = abs(cnv); %rectificación de "cnv" (alinealidad)
[B,A] = cheby1(4,1,[0.03 0.05]); %generación del filtro Pasabanda entre
15K y 25K
y = filter(B,A,cnvrect); %obtención de la componente fundamental
SAL = sign(y); %onda cuadrada obtenida
plot(t3,S3,1:length(SAL),SAL)
cont=1;
for i=2:length(Sn);
  flag=SAL(i)-SAL(i-1);
    if flag==2
      x(cont)=sign(Sn(i-8)); %al utilizar un filtro dig. de orden 8 se
      cont=cont+1;           %produce un desfasaje de 8 muestras por
    end;                     %eso hacemos la lectura en i-8
end; %lectura de los bits a partir de del clock recuperado
```

```
for i=2:f+g-1
   c(i)=x(i-1)-x2(i);
end; %"c" es la diferencia entre los datos detectados y los que componen
la trama
```

A continuación se ve en azul la forma de onda bipolar con retorno y en verde la onda cuadrada recuperada, cuyos flancos de subida están cercanos al centro de la forma de onda (recuérdese el desfasaje de 8 muestras hacia la derecha que afecta a la cuadrada). Si bien se aprecia que hay variaciones de fase entre las dos señales el sistema se mantiene en sincronismo y no se producen errores en la lectura.

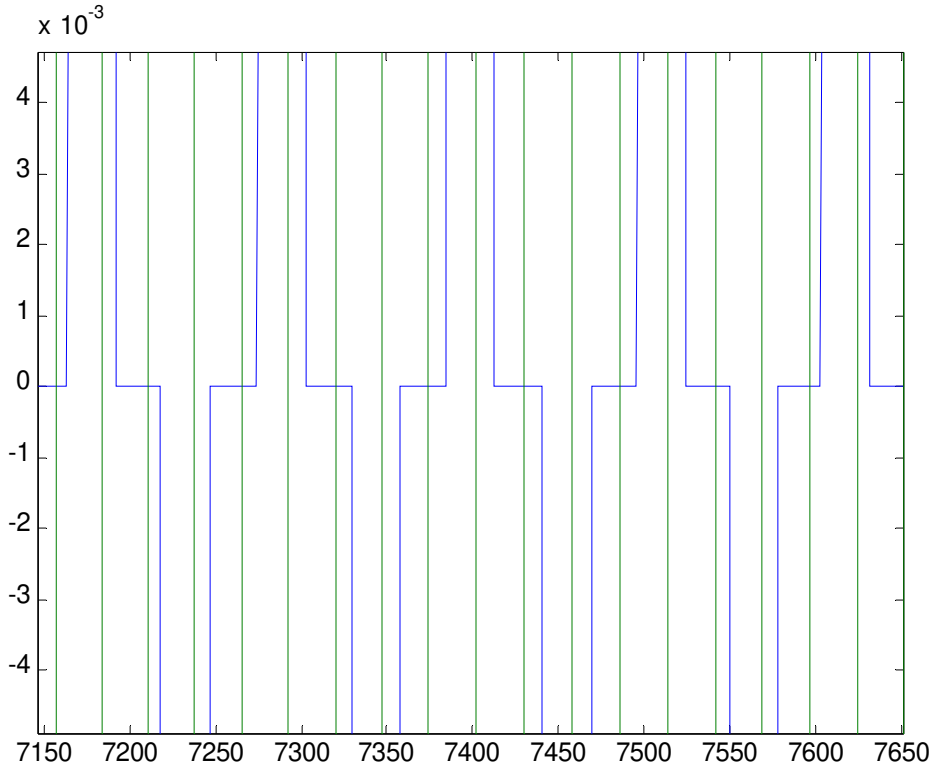

Figura 48. Señal de entrada en bipolar (azul) y clock recuperado (verde).

El programa que se usó para trabajar con formato Manchester es muy similar al anterior. Tiene como única diferencia la generación de la trama, que se muestra a continuación:

```
for u=1:f
  n=p+round(jitt*randn(1));
  nvec(u)=n;
  S3(pos+1:pos+round(n/2))= (-1)^(u+1)*ones(1,round(n/2));
  S3(pos+round(n/2)+1:pos+n)= (-1)^(u)*ones(1,n-round(n/2));
  x2(u)=S3(pos+round(n/4));
  pos=pos+n;
end;

for u=1:g
  n=p+round(jitt*randn(1));
  nvec(u+f)=n;
  aux=sign(randn(1));
  S3(pos+1:pos+round(n/2))= aux;
  S3(pos+round(n/2)+1:pos+n)= -1*aux;
  x2(u+f)=S3(pos+round(n/4));
  pos=pos+n;
end;
```

En la siguiente figura se aprecia como las señales se desfasan progresivamente. No pudiendo mantenerse el sincronismo. Se probaron distintos filtros y no se consiguió un buen funcionamiento en condiciones de jitter y el ruido.

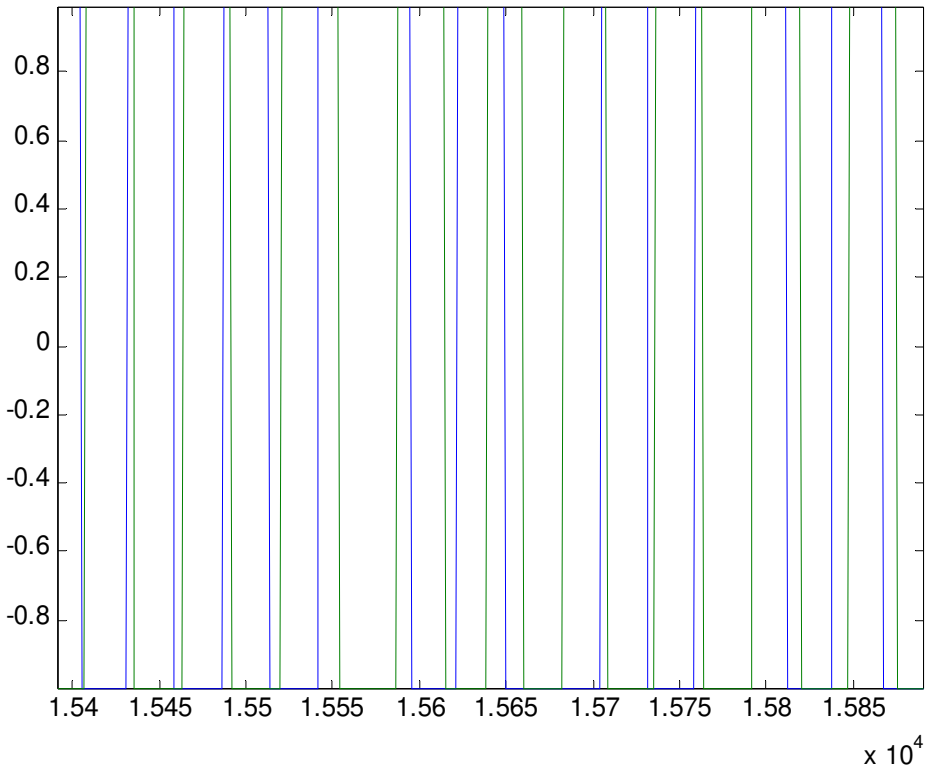

Figura 49. Señal de entrada Manchester (azul) y clock recuperado (verde).

APENDICE B

Método de sincronismo a lazo cerrado (Early/Late)

Programa en Matlab, formato Bipolar con retorno a cero:

```
Y=0;S=0;S3=0;pos=0;t3=0;S1=0;t1=0;       %inicialización de variables
c=0;x=0;x2=0;vec=0;nvec=0;evec=0;
f=50; %tamaño del bloque compuesto por unos y ceros alternados
g=50; %tamaño del bloque compuesto por bits aleatorios
jitt=.1; %coeficiente de jitter
ruido=.05; %dispersión del ruido
p=55; %cantidad de muestras que componen un bit

for u=1:f
  n=55+round(jitt*randn(1));
  nvec(u)=n;
  S3(pos+1:pos+round(n/2))= (-1)^(u+1)*ones(1,round(n/2));
  S3(pos+round(n/2)+1:pos+n)= zeros(1,n-round(n/2));
  x2(u)=S3(pos+round(n/4));
  pos=pos+n;
end;  %generación del bloque compuesto por unos y ceros alternados

for u=1:g
  n=55+round(jitt*randn(1));
  nvec(u+f)=n;
  S3(pos+1:pos+round(n/2))= sign(randn(1)*round(n/2));
  S3(pos+round(n/2)+1:pos+n)= zeros(1,n-round(n/2));
  x2(u+f)=S3(pos+round(n/4));
  pos=pos+n;
end;  %generación del bloque compuesto por bits aleatorios

t3=[1:pos];
tot=50;
pos=0;

for k=1:f+g
No=ruido*randn(1,tot+1);
Si1=S3(pos+1:tot+pos)+No(1:tot);
vecsum = cumsum(Si1);
cuenta=tot;
vec(k)=cuenta;
e1 = vecsum(cuenta)-vecsum(round(0.2*cuenta)); %compuerta temprana
e2 = vecsum(round(0.8*cuenta));                %compuerta tardía
e = 0.05*(abs(e2) - abs(e1));                  %señal de error
evec(k)=e;                              %generación del vector de errores
if abs(e)>0.001          %si error supera el umbral corrige
  if e>0                 %si error es positivo disminuye tot
   tot=tot-1;
  else                   %si error es negativo aumenta tot
   tot=tot+1;
  end;
end;
for j=1:round(tot/2)     %generación del clock de salida
```

```
   sal(j)=1;
end;
for j=round(tot/2)+1:tot
   sal(j)=-1;
end;
x(k)=sign(S3(pos+round(tot/2))); %lectura a partir de la estima de tot
Y(pos+1:pos+tot)=sal(1:tot);
pos=pos+tot;
end;

for z=1:f+g
c(z)=x2(z)-x(z);
end;
t=[1:pos];
vec
mean(vec)
c
plot(t3,S3,t,Y)
```

Dado que “e” fue calculado de la siguiente manera:

```
e = gain*(abs(e2) - abs(e1));
```

se empleó un lazo for para determinar la ganancia más conveniente:

```
gain=[0.01:.001:0.03];
for u=1:2
.
.
.

e = gain*(abs(e2) - abs(e1));
.
.
.
```

Un valor aceptable encontrado fue gain = 0.05, pero como ya se mencionó, solo se logró un buen desempeño a muy bajo ruido y jitter. De la misma forma se trabajó para determinar el valor umbral. Dependiendo de la combinación de estos dos valores se obtuvieron distintos comportamientos del sistema. En caso de períodos de símbolo de entrada diferentes del estimado inicialmente (tot=50), el sistema presentaba una respuesta oscilatoria para alcanzarlo. Tanto el tiempo de crecimiento como la frecuencia de oscilación de esta respuesta, variaba notablemente con los diferentes valores de “umbral” y “gain”. En la siguiente figura se diferencia en color azul la oscilación del error y en verde la desviación en frecuencia, otro aspecto para destacar es que utilizando este algoritmo con los

valores de coeficientes anteriormente descritos recién para el bit 61 de entrada se logra un error nulo en frecuencia. Por último se debe tener en cuenta que esta medición fue hecha con una entrada ideal, es decir sin ruido ni jitter.

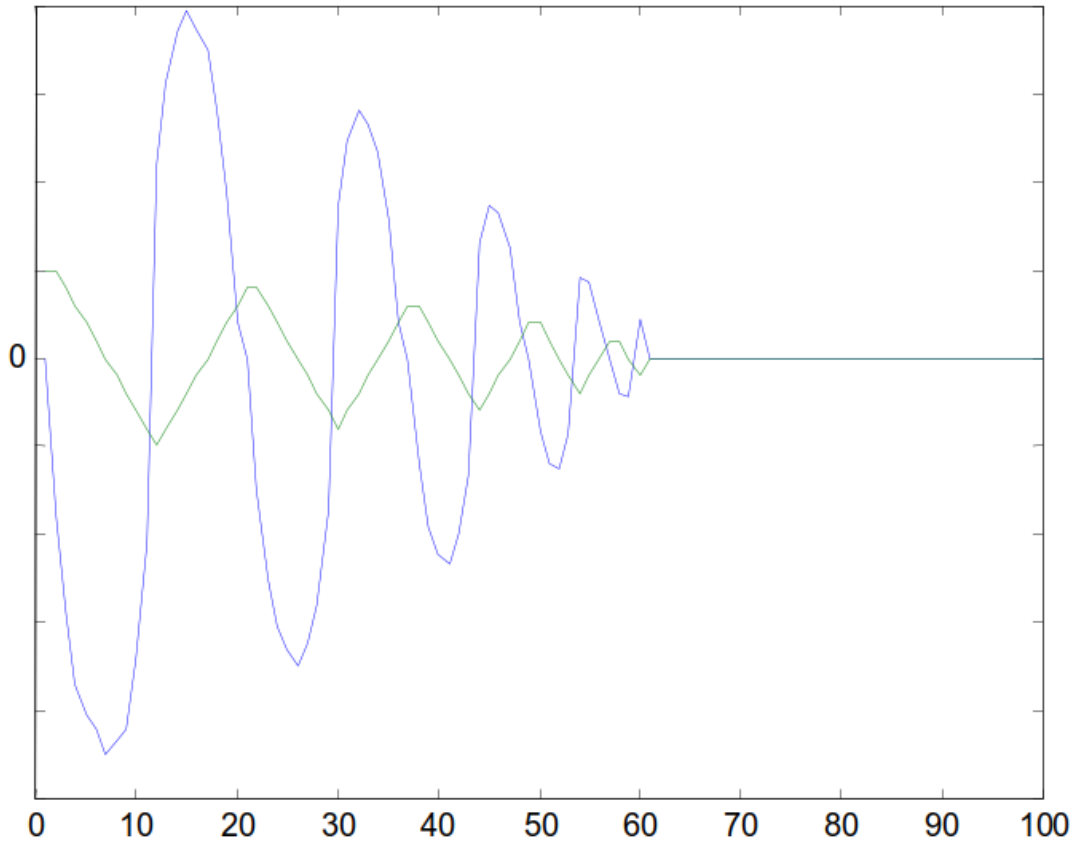

Figura 50. Error (azul) y desviación en frecuencia (verde) en relacion al número de bits.

APENDICE C

Método de sincronismo a lazo cerrado (Early/Late), versión definitiva. Programa en Matlab, formato Manchester:

```
Y=0;S3=0;pos=0;pos1=0;t3=0;lee=0;x=0;x2=0;vec=0;nvec=0;evec=0;evec2=
0;e=0;e1=0;
e2=0;dpos=0;evvec=0;evvec2=0;fase=0;frec=0;aux=0;
f=40;
g=10;
jitt=1;
ruido=0.3;
p=54;
tot=50;
for u=1:f
  n=p+round(jitt*randn(1));
  nvec(u)=n;
  S3(pos1+1:pos1+round(n/2))= 64*(-1)^(u+1)*ones(1,round(n/2));
  S3(pos1+round(n/2)+1:pos1+n)= 64*(-1)^(u)*ones(1,n-round(n/2));
  x2(u)=S3(pos1+round(n/4));
  pos1=pos1+n;
end;
for u=1:g
  n=p+round(jitt*randn(1));
  nvec(u+f)=n;
  aux=sign(randn(1));
  S3(pos1+1:pos1+round(n/2))= aux*64;
  S3(pos1+round(n/2)+1:pos1+n)= -1*aux*64;
  x2(u+f)=S3(pos1+round(n/4));
  pos1=pos1+n;
end;
t3=[1:pos1];
Sn=S3+64*ruido*randn(1,pos1);

for k=1:f+g
p1=[ones(1,round(tot/2)),(-1)*ones(1,tot-round(tot/2))];
       %Esta sería la respuesta al impulso del filtro adaptado en
formato manchester

%Para formato bipolar se usa el siguiente pulso:
% p1=[ones(1,round(tot/2)),zeros(1,tot-round(tot/2))];


if k==f+g
   tot=5;
end;

c1=conv(p1,Sn(pos+1:tot+pos));
%c1 es la convolución entre la señal de entrada y la
%respuesta al impulso del filtro adaptado.

vecsum =cumsum(c1);
vec(k)=tot;
e1 = vecsum(tot)-vecsum(round(0.2*tot));
e2 = vecsum(round(0.8*tot));

e = (abs(e1) - abs(e2));
evec(k)=e;
```

```
evec2(1)=0;
 if k==1
 else
   evec2(k)=evec(k)-evec(k-1);
      if abs(evec2(k))>16*200
            if evec(k)>0
            tot=tot-2;
            else
            tot=tot+2;
            end;
      end;
   dpos(k)=round(e/(4096));
      pos=pos-dpos(k);
 end;
if c1(tot)<0 % En formato Manchester el valor lógico del símbolo
x(k)=64;     % enviado, se considera un "1" si el valor de la de la
else         % convolución en la posición tot es menor o igual a
x(k)=-64;    % cero, y "0" en caso contrario.
end;

%  en formato bipolar se decide asi:
%  if vecsum(length(vecsum))>0
%     x(k)= 64;
%  else
%     x(k)=-64;
%  end;
%esto implica que si el área debajo de la convolución es positiva,
%entonces el valor envíado es un "1", y un "0" en caso contrario.


for j=1:round(tot/2)
   sal(j)=1;
end;
for j=round(tot/2)+1:tot
   sal(j)=-1;
end;
Y(pos+1:pos+tot)=sal(1:tot);
pos=pos+tot;
end;
t=[1:pos];
for u=1:f+g-1
   lee(u)=x2(u)-x(u);
   frec(u)=nvec(u)-vec(u);
   aux(u)=(nvec(u)-vec(u)+dpos(u));

end;
fase=cumsum(aux);
[max(lee),min(lee);max(frec),min(frec);max(fase),min(fase)]
```

APENDICE D

En este apéndice se pretende dar nociones acerca de las operaciones en los denominados campos finitos, o de Galois, iniciando tal descripción con la introducción de conceptos relacionados como lo son los grupos. El objeto es la aplicación de estas estructuras algebraicas a las operaciones de polinomios definidos sobre las mismas. El concepto más importante es que un polinomio definido sobre un campo GF(p) tiene raíces en ese campo, o en una extensión del mismo, GF(q). Igualmente cada elemento 'a' de la extensión de un campo finito GF(q) es cero de algún polinomio con coeficientes en el campo GF(p). El polinomio de grado mínimo que cumple esa condición se denomina polinomio mínimo de 'a'.

Grupos

Un grupo G_r se define como un conjunto de elementos que se relacionan por medio de operaciones. Para un conjunto G_r de elementos se define la operación binaria $*$ que es una regla de asignación tal que para dos elementos a y b del conjunto se asigna un único otro elemento de ese conjunto que es $c = a*b$. Esta operación es cerrada sobre G_r dado que resulta en otro elemento que es también integrante del grupo. Tal operación es asociativa.

Definición de Grupo:

Un conjunto G_r sobre el cual se define una operación binaria $*$ se dice que es un
grupo si se cumplen las siguientes condiciones:

1. La operación binaria $*$ es asociativa.
2. El conjunto G_r contiene un elemento llamado la identidad para la operación $*$.
3. Para todo elemento del conjunto $a \in G_r$ existe el elemento a' es denominado inverso de a.

4. Un grupo se dice conmutativo si para todo par de elementos $a,b \in G_r$, se cumple que:

$a * b = b * a$

El número de elementos de un grupo se denomina orden del grupo. Un grupo con un orden finito se denomina grupo finito.

Definición de Campo

La noción de grupo resulta útil e introductoria para la definición de los llamados campos. Un campo es un conjunto de elementos en donde la suma, la multiplicación, la resta y la división producen como resultado elementos que son siempre del campo.

Se puede definir el conjunto F de elementos bajo alguna dada operación. Un conjunto F de elementos sobre las operaciones de suma y multiplicación módulo m se realiza de acuerdo a las siguientes condiciones:

1. F es un grupo conmutativo frente a la adición. El elemento identidad para la suma se denomina cero '0'.
2. F es un grupo conmutativo frente a la multiplicación. El elemento identidad para la multiplicación se denomina uno '1'.
3. La multiplicación es distributiva frente a la adición: $a.(b + c) = a.b + a.c$

El número de elementos de un campo se denomina orden del campo. Un campo con un número finito de elementos se denomina campo finito, o campo de Galois, GF. El inverso frente a la suma para un elemento $a \in F$ de un campo se denomina -a, y el inverso para la multiplicación es llamado a^{-1}.

El conjunto $G_r = \{0,1\}$ definido bajo los operadores de suma y multiplicación módulo-2 es tal que $G_r = \{0,1\}$ es un grupo conmutativo frente a la adición, mientras que $G_1 = \{1\}$ es un grupo conmutativo bajo la multiplicación. Puede

verificarse que el grupo G_r = {0,1} es también un campo finito. Este es un campo binario denominado GF(2).

Para un dado número primo p el conjunto de enteros {0,1,2,3,...p-1} es un grupo conmutativo bajo la operación suma módulo p . El conjunto de enteros {1,2,3,...p-1} es un grupo conmutativo bajo la multiplicación módulo p . Este conjunto entonces es un campo de orden p. Se denominan campos primos GF(p). Una extensión de un campo primo GF(p) es denominado campo finito extendido GF(q) = GF(p^m), con m número positivo entero, y resulta ser también un campo de Galois. Un caso particular de estas extensiones son los campos finitos de la forma GF(2^m), donde *m* es un número positivo entero. En un campo finito GF(q), y siendo $a \in GF(q)$ resulta como consecuencia de que el campo es cerrado frente a la operación multiplicación, que las potencias del elemento $a \in GF(q)$ serán elementos del campo finito GF(q). Sin embargo el campo es finito con lo cual estas potencias deben en algún momento comenzar a repetirse. Dicho de otra forma tendrán que existir dos enteros k y m tales que m >k y $a^m = a^k$. Como a^{-k} es el inverso multiplicativo de a^k, entonces $a^{-k}a^m = a^{-k}\, a^k$ o bien $a^{m-k} = 1$. Existe entonces un número n tal que $a^n = 1$ y ese número se lo denomina el orden del elemento a. De esta manera las potencias a^1, a^2, a^3, ..., a^{n-1} son todas distintas, y forman un grupo bajo la multiplicación de GF(q). Puede demostrarse entonces que si a es un elemento no nulo del campo finito GF(q), entonces $a^{q-1} = 1$. También es cierto que si a es un elemento no nulo del campo finito GF(q), y n es el orden de ese elemento, entonces n divide a q - 1. En un campo finito GF(q) un elemento no nulo a del mismo se dice primitivo si el orden de ese elemento es q - 1. Un elemento primitivo genera a través de todas sus potencias todos los elementos no nulos del campo finito GF(q). Todo campo finito tiene algún elemento primitivo.

Aritmética de campos binarios

Los campos binarios más empleados son extensiones del campo GF(2), que se denominan campos GF(2^m). La aritmética binaria usa la multiplicación y la suma módulo-2. Un polinomio f(X) definido sobre GF(2) tiene la siguiente forma:
$f(X)=f_0+f_1X+f_2X^2+.....+f_nX^2$
Donde los f_i son 0 o 1.

El exponente más alto de la variable X se denomina grado del polinomio. Para un cierto valor de n , el grado del polinomio, existen 2^n diferentes polinomios con ese grado:

n=1 X , X+1
n=2 X^2 , 1+X , $X+X^2$, $1+X+X^2$

La suma y multiplicación se realizan usando operaciones módulo-2. Los polinomios sobre GF(2) operan multiplicando y sumando en módulo-2, y cumplen con las propiedades conmutativa, asociativa y distributiva. Una operación importante es la división entre polinomios.

Por ejemplo para dividir el polinomio X^3+X+1 por X+1, se realizan los siguientes pasos:

$$
\begin{array}{rl|l}
X^3 + \quad + X + 1 & & X + 1 \\
\cline{3-3}
X^3 + X^2 & & X^2 + X \\
\cline{1-1}
\quad X^2 + X + 1 & & \\
\quad X^2 + X & & \\
\cline{1-1}
\quad r(X) = 1 & &
\end{array}
$$

La división adopta la forma:

$r(X) + q(X).g(X) = f(X)$
donde en este ejemplo:
$r(X) = 1$
$q(X) = X + X^2$

Definición D.1: Un elemento a es cero o raíz de un polinomio f(X) si sucede que f(a)=0. Cuando un elemento a es raíz de un polinomio entonces f(a)=0, y sucede que ese polinomio es divisible por X - a.

Así por ejemplo el polinomio $f(X)=1+X^2+X^3+X^4$ tiene como raíz a a = 1, por lo cual es divisible por X+1, operación que resulta en un cociente $q(X)=1+X+X^3$ (recuérdese que el inverso aditivo de a, -a es igual a -a =a en operación módulo-2).

Definición D.2: Un polinomio p(X) sobre GF(2) de grado m se dice irreducible, si p(X) no es divisible por ningún polinomio de grado mayor que cero y menor que m.

Por ejemplo $1+X+ X^2$ es un polinomio irreducible debido a que ni X ni X+1 lo dividen con resto cero. Un polinomio de grado 2 es irreducible cuando no es divisible por ningún polinomio de grado 1.

Una propiedad de los polinomios irreducibles sobre un campo GF(2) de grado m es que dividen al polinomio $X^{2^m-1}+1$. Por ejemplo el polinomio X^3+X+1 divide a $X^{2^3-1}+1=X^7+1$.

Un polinomio irreducible p(X) de grado m es polinomio primitivo si el más pequeño entero positivo n para el que p(X) divide a X^n+1 es $n = 2^m-1$.

Por ejemplo X^4+X+1 divide $X^{2^4-1}+1=X^{15}+1$ pero no divide ningún polinomio de la forma X^n+1 siendo $1\leq n\leq 15$. Por lo tanto es un polinomio primitivo. Otra interesante propiedad de los polinomios sobre *GF(2)* es que: $(f(x))^{2^i}=f(x^{2^i})$

Construcción de un campo de Galois GF(2m)

Para construir un campo de Galois se asume que no solo los elementos '0' y '1' existen, sino que también aparece el elemento α y sus potencias.

Para este nuevo elemento:

$0.\alpha = \alpha.0 = 0$

$1.\alpha = \alpha.1 = \alpha$

$\alpha^2 = \alpha.\alpha$, $\alpha^3 = \alpha.\alpha^2$

$\alpha^i\alpha^j = \alpha^{i+j} = \alpha^j\alpha^i$

Un conjunto de estos elementos se define como:

$F = \{0,1,\alpha,\alpha^2,....,\alpha^k,....\}$

Este conjunto contendrá solo 2^m elementos. Debido a que un polinomio primitivo p(X) sobre GF(2) de grado m , divide a $X^{2^m-1}+1$, y asumiendo como cierto que $p(\alpha) = 0$, entonces:

$$p(\alpha)=0$$
$$=1$$

El conjunto F se transforma en un conjunto finito con 2^m elementos:

La condición:

$i + j < 2^m - 1$

debe cumplirse para que al multiplicar dos elementos del campo, α^i y α^j el resultado siga perteneciendo al mismo campo:

Si:

de esta manera el grupo es cerrado frente a la multiplicación. Por otro lado, para un $\alpha^j=\alpha^{(i+j)}=\alpha^{(2^m-1)+r}=\alpha^r$ número entero i tal que $0 < i < 2^m - 1$, α^{2^m-1-i} es la inversa de α^i .

Así, el conjunto $F=\{0,1,\alpha,\alpha^2,\ldots\ldots,\alpha^{2^m-2}\}$ es un grupo de orden 2^m-1 frente a la multiplicación. Para hacer que el grupo sea un grupo conmutativo frente a la adición se debe definir la operación adición sobre tal conjunto F.

Para: $0 \leq i < 2^m - 1$, X^i es dividido por p(X):

$X^i = q_i(X).p(X)+a_i(X)$

$a_i(X)$ es de grado m - 1 o menor, $a(X) = a_{i0} + a_{i1}.X + a_{i2}.X^2 +....+a_{i,m-1}.X^{m-1}$. Para $0 \leq i,j < 2^m - 1$, $a_i(X) \neq a_j(X)$ si i = 0,1,2,...,$2^m - 2$

Se pueden obtener $2^m - 1$ diferentes polinomios $a_i(X)$.

$$a_{l0}+a_{l1}.X+a_{l2}.X^2+a_{lm-1}.X^{m-1}$$

Estos polinomios representaran $2^m - 1$ elementos no nulos $\alpha^0,\alpha^1,\alpha^2,\ldots,\alpha^{2^m-2}$. Se ve que existen $2^m - 1$ polinomios diferentes de a sobre el campo GF(2) que representaran a los $2^m - 1$ diferentes elementos no nulos del campo F . Esto permitirá encontrar un equivalente en palabra binaria de cada elemento del campo.

La operación suma se define como:

$$\alpha^i=\alpha^i\oplus 0=\alpha^i$$

Luego, $\alpha^l\oplus\alpha^j=(a_{l0}\oplus a_{j0})+(a_{l1}\oplus a_{j1}).X+(a_{l2}\oplus a_{j2}).X^2+\ldots+(a_{lm-1}\oplus a_{jm-1}).X^{m-1}$

Donde la suma elemento a elemento es realizada como suma módulo 2. Esto es, la suma de dos elementos del grupo $F=\left\{0,1,\alpha,\alpha^2,\ldots\ldots,\alpha^{2^m-2}\right\}$ es la OR Exclusiva entre los bits de las representaciones binarias que corresponden a esos elementos, y que equivalen a las correspondientes expresiones polinómicas en α. Esta operación es realizada bit a bit.

Este tipo de conjuntos F de elementos definidos como se hizo previamente es un grupo conmutativo frente a la suma, y el conjunto de elementos no nulos de F es conmutativo frente a la multiplicación. Como consecuencia de esto, el conjunto:

Es un campo de Galois de 2^m elementos, GF(2^m).

Una característica interesante es poder representar la versión binaria de cada elemento del campo. La suma OR Exclusiva bit a bit entre las componentes de la representación binaria se emplea para obtener la suma de dos elementos del campo.

Ejemplo: Determine la tabla del campo de Galois GF(2^4) tomando como polinomio primitivo a $p(X) = 1 + X + X^4$.

De acuerdo al polinomio primitivo se tiene entonces que $p(\alpha) = 1 + \alpha + \alpha^4 = 0$, o bien a $\alpha^4 = 1 + \alpha$. El campo generado es GF(2^4) y se observa en la tabla:

0	0				0	0	0	0
1	1				1	0	0	0
α		α			0	1	0	0
α^2			α^2		0	0	1	0
α^3				α^3	0	0	0	1
α^4	1	$+\alpha$			1	1	0	0
α^5		α	$+\alpha^2$		0	1	1	0
α^6			$+\alpha^2$	$+\alpha^3$	0	0	1	1
α^7	1	$+\alpha$		$+\alpha^3$	1	1	0	1
α^8	1		$+\alpha^2$		1	0	1	0
α^9		α		$+\alpha^3$	0	1	0	1
α^{10}	1	$+\alpha$	$+\alpha^2$		1	1	1	0
α^{11}		α	$+\alpha^2$	$+\alpha^3$	0	1	1	1
α^{12}	1	$+\alpha$	$+\alpha^2$	$+\alpha^3$	1	1	1	1
α^{13}	1		$+\alpha^2$	$+\alpha^3$	1	0	1	1
α^{14}	1			$+\alpha^3$	1	0	0	1

Campo de Galois $GF(2^4)$ generado por $p(X) = 1 + X + X^4$

Propiedades de los campos de Galois GF(2^m)

Así como los polinomios definidos en las variables tradicionales con coeficientes en el campo de los números reales pueden tener raíces fuera de ese campo, por ejemplo en el campo de los números complejos, de la misma forma polinomios definidos sobre el campo GF(2) pueden tener sus raíces en campos de nivel extendido respecto de este, es decir por ejemplo en GF(2^m). Como un ejemplo, el polinomio $p(X) = 1 + X^3 + X^4$ es irreducible sobre GF(2) dado que no tiene raíces dentro de ese campo, pero sin embargo tiene sus cuatro raíces definidas sobre el campo GF(2^4). Realizando un reemplazo con los elementos definidos en la tabla 4 del ejemplo anterior de GF(2^4) puede verificarse que α^7, α^{11}, α^{13}, α^{14} son efectivamente raíces de este polinomio.

El siguiente teorema determina una condición sobre las raíces de un polinomio tomadas de un campo extendido. Conociendo alguna raíz β del polinomio, el teorema permite calcular todas las restantes raíces de ese polinomio. Solo lo enunciaremos a continuación:

Teorema: Sea f(X) un polinomio definido sobre el campo GF(2). Si β es un elemento del campo extendido GF(2^m) y además es raíz del polinomio f(X), entonces para cualquier entero positivo no nulo $l \geq 0, \beta^{2^l}$, que se denomina conjugado de β, es también raíz de ese polinomio.

Este teorema dice que si b es un elemento del campo extendido GF(2^m) y además es raíz del polinomio f(X), sus conjugados, que también son elementos del campo GF(2^m), son raíces de ese polinomio.

Ejemplo:

Para el caso analizado donde el polinomio sobre GF(2) $p(X) = 1 + X^3 + X^4$ tiene como raíz α^7 entonces tendrá también como raíces a $(\alpha^7)^2 = \alpha^{14}$, $(\alpha^7)^4 = \alpha^{28} = \alpha^{13}$ y a $(\alpha^7)^8 = \alpha^{56} = \alpha^{11}$. Aquí se encuentran todas las raíces dado que $(\alpha^7)^{16} = \alpha^7$ comienza a repetirse.

Polinomios mínimos

Como cada elemento β de GF(2^m) es una raíz del polinomio $X^{2^m}+X$, el mismo elemento podría ser una raíz de un polinomio definido sobre GF(2) cuyo grado sea menor que 2^m.

Definición: Se denomina polinomio mínimo de β, Φ(X), al polinomio de menor grado sobre GF(2) tal que β es raíz de ese polinomio, es decir tal que

Φ(β)=0. Así el polinomio mínimo del elemento nulo 0 es X, el polinomio del elemento 1 es 1+X.

Propiedades de los polinomios mínimos.

Los polinomios mínimos poseen algunas propiedades.

- El polinomio mínimo de un elemento β de GF(2^m) es irreducible.
- Para un polinomio f(X) definido sobre GF(2) y siendo Φ(X) el polinomio mínimo del elemento β, entonces si β es raíz de f(X) , luego f(X) es divisible por Φ(X).
- El polinomio mínimo Φ(X) de un elemento β de GF(2^m) divide a $X^{2^m}+X$.
- Si f(X) es un polinomio irreducible sobre GF(2) y siendo β un elemento de GF (2^m) cuyo polinomio mínimo es Φ(X), sucede que f (β)=0, entonces f(X)= Φ(X).
- Sea Φ(X) el polinomio mínimo de un elemento β de GF(2^m) y sea el entero más pequeño e para el cual se cumple que $\beta^{2^e}=\beta$ entonces el polinomio mínimo de β es:

APENDICE E

En este apéndice se desarrolla el primer mecanismo de codificación-decodificación que se diseño para el sistema. Fue modificado posteriormente debido a que al no contar con los síndromes de error tabulados sólo permitía la detección de errores. Si bien posee esta desventaja que se menciona, tiene como aspecto positivo una tasa de transmisión mucho más alta que el sistema que se implemento finalmente. La tasa esta dada por el cociente entre la cantidad de bits de mensaje y la cantidad de bits enviados. En consecuencia se tiene:

T_{16} = 128/144

T_{32} = 128/160

Los codificadores son muy similares al bloque “CODECICLICIRC8” que finalmente se implementó, solo difieren en la cantidad de bits que generan y en el tamaño de bloque que codifican. Para ejemplificar se mostrará el programa del codificador de 16 bits:

```
ENTITY codeciclicirc IS
          PORT
          (Datain,clockmax,habilserie,cod  : IN          STD_LOGIC;  --cod: si es 0==>16
           Dataout,habilformat             : OUT       STD_LOGIC);
END codeciclicirc;

ARCHITECTURE arc OF codeciclicirc IS

COMPONENT lpm_ff
  GENERIC (LPM_WIDTH: POSITIVE);
      PORT (data: IN STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0);
    clock: IN STD_LOGIC;
    aclr: IN STD_LOGIC := '0';
     q: OUT STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0));
END COMPONENT;

COMPONENT lpm_counter
 GENERIC (LPM_WIDTH: POSITIVE);
PORT ( clock: IN  STD_LOGIC;
       aclr: IN STD_LOGIC := '0';
       clk_en: IN STD_LOGIC := '1';
              q: OUT STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0));
END COMPONENT;

COMPONENT lpm_mux
  GENERIC (LPM_WIDTH: POSITIVE;
    LPM_WIDTHS: POSITIVE;
```

```
    LPM_SIZE: POSITIVE);
PORT (data: IN STD_LOGIC_2D(LPM_SIZE-1 DOWNTO 0, LPM_WIDTH-1 DOWNTO 0);
    sel: IN STD_LOGIC_VECTOR(LPM_WIDTHS-1 DOWNTO 0);
    result: OUT STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0));
END COMPONENT;

COMPONENT retardo
 PORT (inret,clock :IN STD_LOGIC;
    outret: OUT STD_LOGIC);
END COMPONENT;

signal and1in,mux1in: STD_LOGIC_2D(1 DOWNTO 0, 0 DOWNTO 0);
signal dff1,qff1: STD_LOGIC_VECTOR(15 DOWNTO 0);
signal sal1: STD_LOGIC_VECTOR(1 DOWNTO 0);
signal sal2: STD_LOGIC_VECTOR(4 DOWNTO 0);
signal and1out,mux1out,mux1sel,dff2,qff2,dff3,qff3: STD_LOGIC_VECTOR(0 DOWNTO 0);
signalclkff1,aclrff1,aclrff2,clk2,inret1,outret1,comp1out,g0,aclr1,aclr2,clken1,clken2,habilcod: STD_LOGIC;

BEGIN
cont1: lpm_counter GENERIC MAP (2)
PORT MAP (clockmax,aclr1,clken1,sal1);
cont2: lpm_counter GENERIC MAP (5)
PORT MAP (clockmax,aclr2,clken2,sal2);

ff1: lpm_ff GENERIC MAP (16)
PORT MAP (dff1,clockmax,aclrff1,qff1);
ff2: lpm_ff GENERIC MAP (1)
PORT MAP (dff2,clockmax,aclrff2,qff2);


mux1:lpm_mux GENERIC MAP (1,1,2)
PORT MAP (mux1in,mux1sel,mux1out);

generar : process

begin
habilcod<= habilserie and (not cod);

clken1<=habilcod and not (sal1(1));
aclr1<=sal2(4) and sal2(0);

clken2<=not (sal2(4) and sal2(0));
aclr2<=habilcod;

habilformat<=sal1(1) and (not cod);

g0<=(datain xor qff1(15))and (habilcod);

aclrff1<='0';

dff1(0) <= g0;
dff1(1) <= qff1(0);
dff1(2) <= g0 xor qff1(1);
dff1(3) <= qff1(2);
dff1(4) <= qff1(3);
dff1(5) <= qff1(4);
dff1(6) <= qff1(5);
dff1(7) <= qff1(6);
dff1(8) <= qff1(7);
dff1(9) <= qff1(8);
dff1(10) <= qff1(9);
dff1(11) <= qff1(10);
```

```
dff1(12) <= qff1(11);
dff1(13) <= qff1(12);
dff1(14) <= qff1(13);
dff1(15) <= qff1(14) xor g0;
mux1in(1,0) <= qff1(15);
mux1in(0,0) <= datain;
mux1sel(0) <= not habilcod;

dff2(0)<=mux1out(0);
aclrff2<='0';
Dataout <= qff2(0) and (not cod);

end process;
END arc;
```

La ventaja de esta primera implementación radica en la posibilidad de elección de la codificación, lo que permite variar la tasa a conveniencia. La selección de los distintos codificadores se realiza por medio de una entrada llamada “cod”. En la página siguiente se muestra el diagrama en bloques.

En la página siguiente se muestra el esquema del decodificador de 16 bits (por simplicidad se omitirá el de 32 ya que su estructura es similar, con la diferencia de que está compuesto por 32 flip flops). Los generadores que los definen son:

$g_{16}(x) = 1+ x^2+ x^{15}+ x^{16}$

y

$g_{32}(x) = 1+ x+ x^2+ x^4+ x^5+ x^7+ x^8+ x^{10}+ x^{11}+ x^{12}+ x^{16}+ x^{23}+ x^{26}+ x^{32}$

La principal diferencia que presenta frente al sistema finalmente utilizado es que este último fracciona la trama y codifica cada fragmento. Aquí se introduce la trama entera por el codificador y se agregan 16 o 32 bits al final de ésta. Si bien este sistema también puede corregir errores, se hace muy complicado la composición del síndrome, ya que se deben obtener 120 síndromes distintos para corregir sólo patrones de un error. Imagínese la complejidad de hacerlo para patrones mayores. Esto se hubiera podido realizar fácilmente si se hubieran tenido los distintos síndromes de error tabulados.

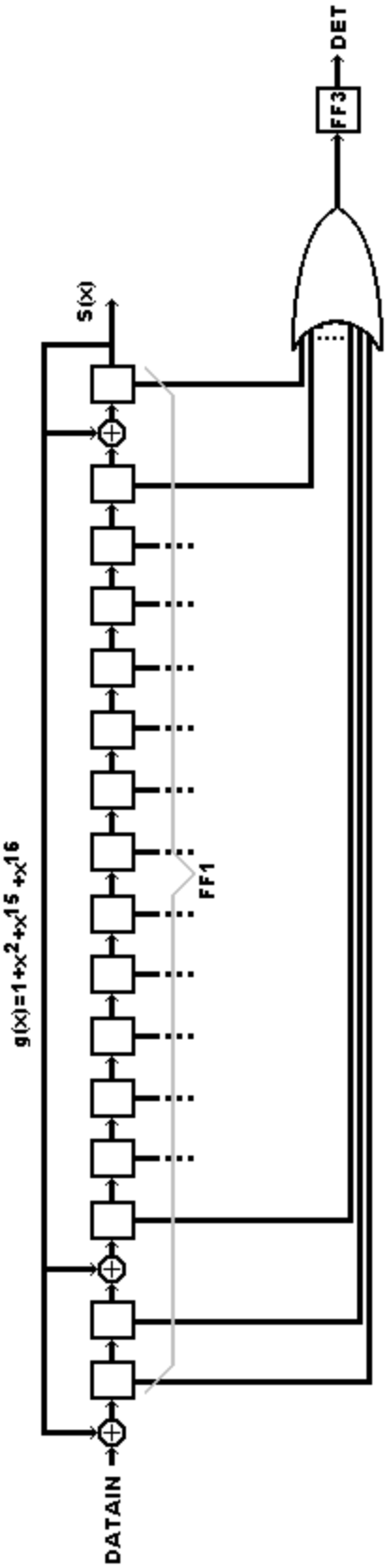

Figura 51. Diagrama del decodificador ciclico de 16 bits.

Como se mencionó, para poder utilizar los distintos codificadores la primer implementación disponía de una entrada, tanto en el transmisor como en el receptor, a través de la cual se elegía el tipo de codificador y decodificador respectivamente. En la página XX de éste apéndice se muestra un diagrama en bloques del primer transmisor desarrollado. El bloque “PARMEMBLOQUESERIE” difiere en algunos aspectos del bloque que lleva el mismo nombre usado en la implementación definitiva. En este caso, su función es almacenar en memoria los bytes que le llegan desde una fuente, y mandarlos en tramas seriales, compuestas por 30 bits de un patrón de unos y ceros alternados, más 128 bits de mensaje. Más adelante se verá su implementación.

Esta es la implementación en VHDL del decodificador de 16 bits (el de 32 es muy similar):

```
ENTITY decodeciclicirc IS
          PORT
          (Datain,stposytot,habil,clock     : INSTD_LOGIC;
           Flip                             : OUT STD_LOGIC_VECTOR (15 DOWNTO 0);
           Dataout,det                      : OUT STD_LOGIC);

END decodeciclicirc;

ARCHITECTURE arc OF decodeciclicirc IS

COMPONENT lpm_counter
  GENERIC (LPM_WIDTH: POSITIVE);
 PORT ( clock   : IN  STD_LOGIC;
       clk_en  : IN STD_LOGIC := '1';
       aclr: IN STD_LOGIC := '0';
       q    : OUT STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0));
 END COMPONENT;

COMPONENT lpm_ff
   GENERIC (LPM_WIDTH: POSITIVE:=8);
     PORT (data: IN STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0);
     clock: IN STD_LOGIC;
     aclr: IN STD_LOGIC := '0';
     q: OUT STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0));
END COMPONENT;

COMPONENT lpm_mux
   GENERIC (LPM_WIDTH: POSITIVE;
     LPM_WIDTHS: POSITIVE;
     LPM_SIZE: POSITIVE);
PORT (data: IN STD_LOGIC_2D(LPM_SIZE-1 DOWNTO 0, LPM_WIDTH-1 DOWNTO 0);
     sel: IN STD_LOGIC_VECTOR(LPM_WIDTHS-1 DOWNTO 0);
     result: OUT STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0));
END COMPONENT;
```

```
COMPONENT retardo2
 PORT (inret,clock :IN STD_LOGIC;
     outret: OUT STD_LOGIC);
END COMPONENT;

signal clkff1,clken2,clkff2,aclrff1,aclrff2,clkff3,g0,aclr2,aclrff3: STD_LOGIC;
signal dff1,qff1,dff2,qff2: STD_LOGIC_VECTOR(15 DOWNTO 0);
signal sal2  :STD_LOGIC_VECTOR(1 DOWNTO 0);
signal dff3,qff3 :STD_LOGIC_VECTOR(0 DOWNTO 0);

BEGIN

cont2: lpm_counter GENERIC MAP (2)
PORT MAP (clock,clken2,aclr2,sal2);

ff1: lpm_ff GENERIC MAP (16)
PORT MAP (dff1,clkff1,aclrff1,qff1);
ff2: lpm_ff GENERIC MAP (16)
PORT MAP (dff2,clkff1,aclrff1,qff2);
ff3: lpm_ff GENERIC MAP (1)
PORT MAP (dff3,clkff3,aclrff3,qff3);

generar : process

begin

aclr2<=habil;
clken2<=not (sal2(1) and sal2(0));

clkff1<=not stposytot;
aclrff1<= sal2(1) and sal2(0);



dff3(0)<=qff1(0)or qff1(1)or qff1(2)or qff1(3)or qff1(4)or qff1(5)or qff1(6)or qff1(7)or qff1(8)or qff1(9)or
qff1(10)or qff1(11)or qff1(12)or qff1(13)or qff1(14)or qff1(15);
clkff3<=sal2(1);
aclrff3<=habil;

g0<=qff1(15);

dff1(0) <= g0 xor Datain;
dff1(1) <= qff1(0);
dff1(2) <= g0 xor qff1(1);
dff1(3) <= qff1(2);
dff1(4) <= qff1(3);
dff1(5) <= qff1(4);
dff1(6) <= qff1(5);
dff1(7) <= qff1(6);
dff1(8) <= qff1(7);
dff1(9) <= qff1(8);
dff1(10) <= qff1(9);
dff1(11) <= qff1(10);
dff1(12) <= qff1(11);
dff1(13) <= qff1(12);
dff1(14) <= qff1(13);
dff1(15) <= g0 xor qff1(14);

dff2(0) <= datain;
dff2(1) <= qff2(0);
dff2(2) <= qff2(1);
dff2(3) <= qff2(2);
```

```
dff2(4) <= qff2(3);
dff2(5) <= qff2(4);
dff2(6) <= qff2(5);
dff2(7) <= qff2(6);
dff2(8) <= qff2(7);
dff2(9) <= qff2(8);
dff2(10) <= qff2(9);
dff2(11) <= qff2(10);
dff2(12) <= qff2(11);
dff2(13) <= qff2(12);
dff2(14) <= qff2(13);
dff2(15) <= qff2(14);
aclrff2 <= not habil;

Flip<=qff1;
Dataout<=qff2(15);
det<=qff3(0);

end process;
END arc;
```

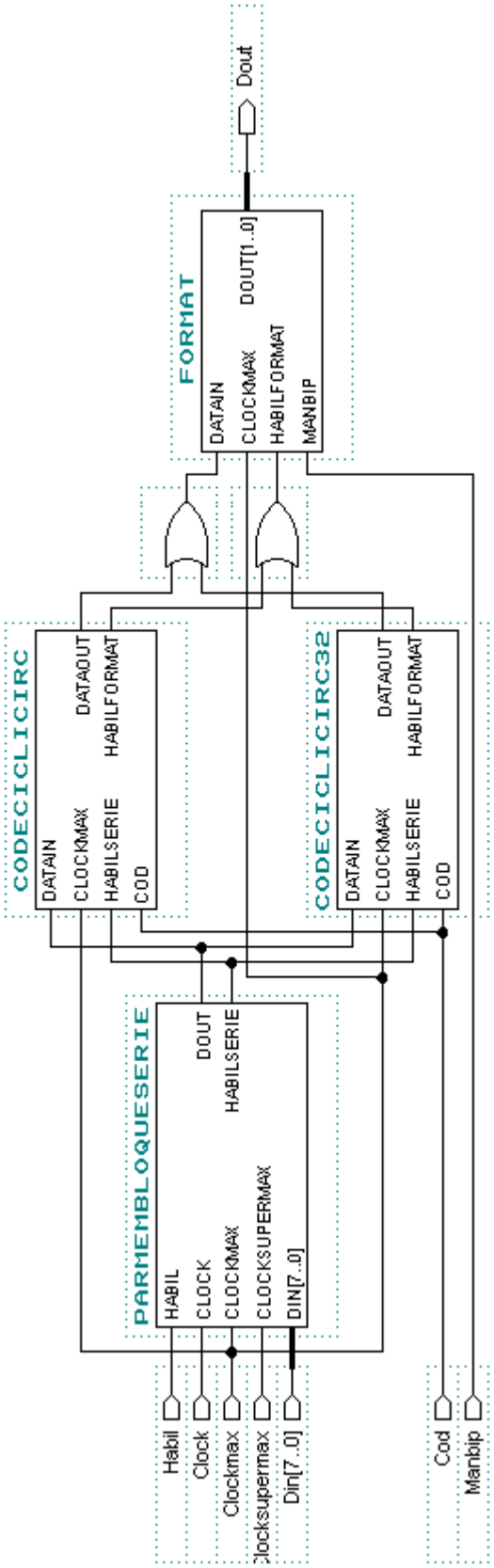

Figura 52. Transmisor con selección de codificación de 16 o 32 bits.

Implementación en VHDL de “PARMEMBLOQUESERIE”:

```
ENTITY parmembloqueserie IS
PORT(habil,clock,clockmax,clocksupermax: IN        STD_LOGIC;
Din,                                     : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
dout,habilserie                          :OUT STD_LOGIC);
END parmembloqueserie;

ARCHITECTURE arc OF parmembloqueserie IS

COMPONENT lpm_ram_dq
GENERIC (LPM_WIDTH: POSITIVE;
         LPM_WIDTHAD: POSITIVE;
      LPM_TYPE: STRING := "L_RAM_DQ";
     --LPM_NUMWORDS: POSITIVE;
       LPM_FILE: STRING := "UNUSED";
       LPM_INDATA: STRING := "REGISTERED";
       LPM_ADDRESS_CONTROL: STRING := "REGISTERED";
       LPM_OUTDATA: STRING := "UNREGISTERED";
       LPM_HINT: STRING := "UNUSED");
PORT (data: IN STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0);
     address: IN STD_LOGIC_VECTOR(LPM_WIDTHAD-1 DOWNTO 0);
     we: IN STD_LOGIC := '1';
     inclock: IN STD_LOGIC := '1';
     q: OUT STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0));
END COMPONENT;

COMPONENT lpm_ff
  GENERIC (LPM_WIDTH: POSITIVE);
  PORT (data: IN STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0);
  clock: IN STD_LOGIC;
  aclr: IN STD_LOGIC := '0';
  q: OUT STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0));
END COMPONENT;

COMPONENT lpm_mux
  GENERIC (LPM_WIDTH: POSITIVE;
     LPM_WIDTHS: POSITIVE;
     LPM_SIZE: POSITIVE);
PORT (data: IN STD_LOGIC_2D(LPM_SIZE-1 DOWNTO 0, LPM_WIDTH-1 DOWNTO 0);
     sel: IN STD_LOGIC_VECTOR(LPM_WIDTHS-1 DOWNTO 0);
     result: OUT STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0));
END COMPONENT;

COMPONENT lpm_counter
  GENERIC (LPM_WIDTH: POSITIVE);
 PORT ( clock: IN  STD_LOGIC;
 clk_en: IN STD_LOGIC := '1';
    aclr: IN STD_LOGIC := '0';
          q: OUT STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0));
END COMPONENT;

COMPONENT monoestable
PORT (Datain,clock      : IN STD_LOGIC;
Dataout                 : OUT STD_LOGIC);
END COMPONENT;

COMPONENT monoestable2
PORT (Datain,clock      : IN          STD_LOGIC;
Dataout                 : OUT STD_LOGIC);
END COMPONENT;
```

```
signal mux1in,mux2in: STD_LOGIC_2D(1 DOWNTO 0, 3 DOWNTO 0);
signal mux3in: STD_LOGIC_2D(1 DOWNTO 0, 0 DOWNTO 0);
signal mux4in: STD_LOGIC_2D(7 DOWNTO 0, 0 DOWNTO 0);
signal mux5in: STD_LOGIC_2D(1 DOWNTO 0, 7 DOWNTO 0);
signal d8,qmem1,dmem1,mux5out,qmem2,dmem2: STD_LOGIC_VECTOR(7 DOWNTO 0);
signal sal2: STD_LOGIC_VECTOR(7 DOWNTO 0);
signal sal1,sal3: STD_LOGIC_VECTOR(4 DOWNTO 0);
signal address1,address2,mux1out,mux2out,sal4: STD_LOGIC_VECTOR(3 DOWNTO 0);
signal mux4sel: STD_LOGIC_VECTOR(2 DOWNTO 0);
signal dff1,qff1,dff2,qff2,mux1sel,mux2sel,mux3out,mux3sel,mux4out,
mux5sel: STD_LOGIC_VECTOR(0 DOWNTO 0);
signal inmono1,outmono1,inmono2,outmono2,outmono3,outmono4,d1,rden1,aclr1,aclr2,aclr3,aclr4,
clk_en1,clk_en2,clk_en3,clk_en4,we1,we2,aclrff1,clock1: STD_LOGIC;

BEGIN

cont1: lpm_counter GENERIC MAP (5)
PORT MAP (clock1,clk_en1,aclr1,sal1);--direcciona escritura
cont2: lpm_counter GENERIC MAP (8)
PORT MAP (clockmax,clk_en2,aclr2,sal2);
cont3: lpm_counter GENERIC MAP (5)--cuenta patrón
PORT MAP (clockmax,clk_en3,aclr3,sal3);
cont4: lpm_counter GENERIC MAP (4)
PORT MAP (clock,clk_en4,aclr4,sal4);

ff1: lpm_ff GENERIC MAP (1)
PORT MAP (dff1,clockmax,aclrff1,qff1);
ff2: lpm_ff GENERIC MAP (1)
PORT MAP (dff2,clockmax,aclrff1,qff2);

mem1: lpm_ram_dq GENERIC MAP (8,4)
PORT MAP (dmem1,address1,we1,clocksupermax,qmem1);
mem2: lpm_ram_dq GENERIC MAP (8,4)
PORT MAP (dmem2,address2,we2,clocksupermax,qmem2);

mono1: monoestable PORT MAP(inmono1,clocksupermax,outmono1);
mono2: monoestable PORT MAP(inmono2,clocksupermax,outmono2);
mono3: monoestable2 PORT MAP(habil,clock1,outmono3);
mono4: monoestable PORT MAP(habil,clocksupermax,outmono4);


mux1:lpm_mux GENERIC MAP (4,1,2) – tamaño entrada, selección, entradas
PORT MAP (mux1in,mux1sel,mux1out);
mux2: lpm_mux GENERIC MAP (4,1,2)
PORT MAP (mux2in,mux2sel,mux2out);
mux3: lpm_mux GENERIC MAP (1,1,2)
PORT MAP (mux3in,mux3sel,mux3out);
mux4: lpm_mux GENERIC MAP (1,3,8)
PORT MAP (mux4in,mux4sel,mux4out);
mux5: lpm_mux GENERIC MAP (8,1,2)
PORT MAP (mux5in,mux5sel,mux5out);

generar : PROCESS

BEGIN

clock1<=clock;
aclr1 <= outmono4;

--cont2
```

```
clk_en2<=(not sal2(7)); -- antes or esto: sal2(0) or sal2(1) or sal2(2) or sal2(3) or sal2(4) or sal2(5) or sal2(6)
or
aclr2<=outmono4 or not (sal3(0) and sal3(1) and sal3(2) and sal3(3));

--cont3
clk_en3<=not (sal3(0) and sal3(1) and sal3(2) and sal3(3) and sal3(4));
aclr3<=outmono1 or outmono2;

--cont4 elimina el primer habilserie
clk_en4<=not (sal4(1) and sal4(3));
aclr4<=((not habil) and (sal2(7)));

--ff1
dff1(0)<=mux3out(0) and (not sal2(7)) and (not outmono3);-- antes or esto: (sal2(0) or sal2(1) or sal2(2) or
sal2(3) or sal2(4) or sal2(5) or sal2(6) or )
aclrff1<='0';

--ff2
dff2(0)<=(sal4(3)and sal4(1)) and (not sal2(7)) and (not outmono3);

--mem1
dmem1<=din;
we1 <= (not sal1(4)) and habil;

address1<=mux1out;

--mem2
dmem2<=din;
we2 <= sal1(4) and habil;
address2 <= mux2out;

--monos
inmono1<= sal1(4);
inmono2<=not sal1(4);

--mux1
mux1in(0,0) <= sal1(0);
mux1in(0,1) <= sal1(1);
mux1in(0,2) <= sal1(2);
mux1in(0,3) <= sal1(3);
mux1in(1,0) <= sal2(3);
mux1in(1,1) <= sal2(4);
mux1in(1,2) <= sal2(5);
mux1in(1,3) <= sal2(6);
mux1sel(0) <= sal1(4);

--mux2
mux2in(0,0) <= sal1(0);
mux2in(0,1) <= sal1(1);
mux2in(0,2) <= sal1(2);
mux2in(0,3) <= sal1(3);
mux2in(1,0) <= sal2(3);
mux2in(1,1) <= sal2(4);
mux2in(1,2) <= sal2(5);
mux2in(1,3) <= sal2(6);
mux2sel(0) <=  not sal1(4);

--mux3
mux3in(0,0) <= sal3(0);
mux3in(1,0) <= mux4out(0);
mux3sel(0) <= sal3(0) and sal3(1) and sal3(2) and sal3(3) and sal3(4);--todos 1 pasa del patrón a las mem
```

```
--mux4
mux4in(0,0) <= mux5out(0);
mux4in(1,0) <= mux5out(1);
mux4in(2,0) <= mux5out(2);
mux4in(3,0) <= mux5out(3);
mux4in(4,0) <= mux5out(4);
mux4in(5,0) <= mux5out(5);
mux4in(6,0) <= mux5out(6);
mux4in(7,0) <= mux5out(7);

mux4sel(0) <= sal2(0);
mux4sel(1) <= sal2(1);
mux4sel(2) <= sal2(2);

--mux5
mux5in(0,0) <= qmem1(0);
mux5in(0,1) <= qmem1(1);
mux5in(0,2) <= qmem1(2);
mux5in(0,3) <= qmem1(3);
mux5in(0,4) <= qmem1(4);
mux5in(0,5) <= qmem1(5);
mux5in(0,6) <= qmem1(6);
mux5in(0,7) <= qmem1(7);

mux5in(1,0) <= qmem2(0);
mux5in(1,1) <= qmem2(1);
mux5in(1,2) <= qmem2(2);
mux5in(1,3) <= qmem2(3);
mux5in(1,4) <= qmem2(4);
mux5in(1,5) <= qmem2(5);
mux5in(1,6) <= qmem2(6);
mux5in(1,7) <= qmem2(7);
mux5sel(0) <= not(sal1(4) and habil);

dout<=qff1(0);
habilserie<=qff2(0);
end process;
END arc;
```

APENDICE F

Programa 1

```
ENTITY parmembloqueserie3 IS
PORT(clockmax,clock                 : IN     STD_LOGIC;
        din                         : IN STD_LOGIC_VECTOR(14 DOWNTO 0);
        habil,clocksupermax         : IN     STD_LOGIC;
        dout,habilserie             : OUT STD_LOGIC);
END parmembloqueserie3;

        ARCHITECTURE arc OF parmembloqueserie3 IS

COMPONENT lpm_ram_dq
GENERIC (LPM_WIDTH: POSITIVE;
      LPM_WIDTHAD: POSITIVE;
      LPM_TYPE: STRING := "L_RAM_DQ";
      LPM_FILE: STRING := "UNUSED";
      LPM_INDATA: STRING := "REGISTERED";
      LPM_ADDRESS_CONTROL: STRING := "REGISTERED";
      LPM_OUTDATA: STRING := "UNREGISTERED";
      LPM_HINT: STRING := "UNUSED");
PORT (data: IN STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0);
    address: IN STD_LOGIC_VECTOR(LPM_WIDTHAD-1 DOWNTO 0);
    we: IN STD_LOGIC := '1';
    inclock: IN STD_LOGIC := '1';
    q: OUT STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0));
END COMPONENT;

COMPONENT lpm_ff
  GENERIC (LPM_WIDTH: POSITIVE);
  PORT (data: IN STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0);
  clock: IN STD_LOGIC;
  aclr: IN STD_LOGIC := '0';
  q: OUT STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0));
END COMPONENT;

COMPONENT lpm_mux
  GENERIC (LPM_WIDTH: POSITIVE;
    LPM_WIDTHS: POSITIVE;
    LPM_SIZE: POSITIVE);
PORT (data: IN STD_LOGIC_2D(LPM_SIZE-1 DOWNTO 0, LPM_WIDTH-1 DOWNTO 0);
    sel: IN STD_LOGIC_VECTOR(LPM_WIDTHS-1 DOWNTO 0);
    result: OUT STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0));
END COMPONENT;

COMPONENT lpm_counter
 GENERIC (LPM_WIDTH: POSITIVE);
PORT ( clock: IN  STD_LOGIC;
       aclr: IN STD_LOGIC := '0';
        clk_en: IN STD_LOGIC := '1';
          q: OUT STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0));
END COMPONENT;

COMPONENT monoestable
PORT (Datain,clock              : IN   STD_LOGIC;
        Dataout             : OUT STD_LOGIC);
END COMPONENT;

COMPONENT monoestable2
PORT (Datain,clock              : IN   STD_LOGIC;
        Dataout             : OUT STD_LOGIC);
END COMPONENT;


signal mux1in,mux2in:               STD_LOGIC_2D(1 DOWNTO 0, 3 DOWNTO 0);
```

```
signal mux3in                          :STD_LOGIC_2D(1 DOWNTO 0, 0 DOWNTO 0);
signal mux4in                          :STD_LOGIC_2D(14 DOWNTO 0, 0 DOWNTO 0);
signal mux5in                          :STD_LOGIC_2D(1 DOWNTO 0, 14 DOWNTO 0);
signal qmem1,dmem1,qmem2,dmem2,mux5out: STD_LOGIC_VECTOR(14 DOWNTO 0);
signal sal2                            :STD_LOGIC_VECTOR(7 DOWNTO 0);
signal sal1,sal3,sal6                  :STD_LOGIC_VECTOR(4 DOWNTO 0);
signal address1,address2,mux1out,mux2out,sal4,sal5,mux4sel: STD_LOGIC_VECTOR(3 DOWNTO 0);
signaldff1,qff1,dff2,qff2,mux1sel,mux2sel,mux3out,mux3sel,mux4out,mux5sel,dff3,qff3
                                       :STD_LOGIC_VECTOR (0 DOWNTO 0);
signal inmono1,outmono1,inmono2,outmono2,outmono3,outmono4,rden1,aclr1,aclr2,aclr3,aclr4,
clk_en1,clk_en2,clk_en3,clk_en4, we1, we2, aclrff1, clk6, clk_en5, clk_en6, aclr5, aclr6: STD_LOGIC;

BEGIN

cont1: lpm_counter GENERIC MAP (5)
PORT MAP (clock,aclr1,clk_en1,sal1);--direcciona escritura

cont3: lpm_counter GENERIC MAP (5)--cuenta patron
PORT MAP (clockmax,aclr3,clk_en3,sal3);
cont4: lpm_counter GENERIC MAP (4)
PORT MAP (clock,aclr4,clk_en4,sal4);
cont5: lpm_counter GENERIC MAP (4)--direcciona lectura bit a bit
PORT MAP (clockmax,aclr5,clk_en5,sal5);
cont6: lpm_counter GENERIC MAP (5)--direcciona lectura en palabras
PORT MAP (clk6,aclr6,clk_en6,sal6);

ff1: lpm_ff GENERIC MAP (1)
PORT MAP (dff1,clockmax,aclrff1,qff1);
ff2: lpm_ff GENERIC MAP (1)
PORT MAP (dff2,clockmax,aclrff1,qff2);
ff3: lpm_ff GENERIC MAP (1)
PORT MAP (dff3,clocksupermax,aclrff1,qff3);

mem1: lpm_ram_dq GENERIC MAP (15,4)
PORT MAP (dmem1,address1,we1,clocksupermax,qmem1);
mem2: lpm_ram_dq GENERIC MAP (15,4)
PORT MAP (dmem2,address2,we2,clocksupermax,qmem2);

mono1: monoestable PORT MAP(inmono1,clocksupermax,outmono1);
mono2: monoestable PORT MAP(inmono2,clocksupermax,outmono2);
mono3: monoestable2 PORT MAP(habil,clock,outmono3);
mono4: monoestable PORT MAP(habil,clocksupermax,outmono4);


mux1:lpm_mux GENERIC MAP (4,1,2) -- tamaño entrada,selecion, entradas
PORT MAP (mux1in,mux1sel,mux1out);
mux2:lpm_mux GENERIC MAP (4,1,2)
PORT MAP (mux2in,mux2sel,mux2out);
mux3:lpm_mux GENERIC MAP (1,1,2)
PORT MAP (mux3in,mux3sel,mux3out);
mux4:lpm_mux GENERIC MAP (1,4,15)
PORT MAP (mux4in,mux4sel,mux4out);
mux5:lpm_mux GENERIC MAP (15,1,2)
PORT MAP (mux5in,mux5sel,mux5out);

generar : PROCESS

BEGIN
aclr1 <= outmono4;

--cont3
clk_en3<=not (sal3(0) and sal3(1) and sal3(2) and sal3(3) and sal3(4));
aclr3<=outmono1 or outmono2;

--cont4 elimina el primer habilserie
clk_en4<=not (sal4(1) and sal4(3));
aclr4<=((not habil) and sal6(4));
```

```
--cont5
aclr5<=qff3(0) or outmono4 or not (sal3(0) and sal3(1) and sal3(2) and sal3(3));
clk_en5<=not (sal6(4));--

--cont6
clk6<=qff3(0);
aclr6<=outmono4 or not (sal3(0) and sal3(1) and sal3(2) and sal3(3));
clk_en6<='1';

--ff1
dff1(0)<=mux3out(0) and (not (sal6(4))) and (not outmono3);
aclrff1<='0';

--ff2
dff2(0)<=(sal4(3)and sal4(1)) and (not (sal6(4))) and (not outmono3);

--ff3
dff3(0)<=(sal5(0) and sal5(1) and sal5(2) and sal5(3));

--mem1
dmem1<=din;
we1 <= (not sal1(4)) and habil;
address1 <= mux1out;

--mem2
dmem2<=din;
we2 <= sal1(4) and habil;
address2 <= mux2out;

--monos
inmono1<= sal1(4);
inmono2<=not sal1(4);

--mux1
mux1in(0,0) <= sal1(0);
mux1in(0,1) <= sal1(1);
mux1in(0,2) <= sal1(2);
mux1in(0,3) <= sal1(3);

mux1in(1,0) <= sal6(0);
mux1in(1,1) <= sal6(1);
mux1in(1,2) <= sal6(2);
mux1in(1,3) <= sal6(3);

mux1sel(0) <= sal1(4);

--mux2
mux2in(0,0) <= sal1(0);
mux2in(0,1) <= sal1(1);
mux2in(0,2) <= sal1(2);
mux2in(0,3) <= sal1(3);

mux2in(1,0) <= sal6(0);
mux2in(1,1) <= sal6(1);
mux2in(1,2) <= sal6(2);
mux2in(1,3) <= sal6(3);

mux2sel(0) <=  not sal1(4);

--mux3
mux3in(0,0) <= sal3(0);
mux3in(1,0) <= mux4out(0);
mux3sel(0) <= sal3(0) and sal3(1) and sal3(2) and sal3(3) and sal3(4);--todos 1 pasa del patrón a las mem

--mux4
mux4in(0,0) <= mux5out(0);
```

```
mux4in(1,0) <= mux5out(1);
mux4in(2,0) <= mux5out(2);
mux4in(3,0) <= mux5out(3);
mux4in(4,0) <= mux5out(4);
mux4in(5,0) <= mux5out(5);
mux4in(6,0) <= mux5out(6);
mux4in(7,0) <= mux5out(7);
mux4in(8,0) <= mux5out(8);
mux4in(9,0) <= mux5out(9);
mux4in(10,0) <= mux5out(10);
mux4in(11,0) <= mux5out(11);
mux4in(12,0) <= mux5out(12);
mux4in(13,0) <= mux5out(13);
mux4in(14,0) <= mux5out(14);

mux4sel(0) <= sal5(0);
mux4sel(1) <= sal5(1);
mux4sel(2) <= sal5(2);
mux4sel(3) <= sal5(3);

--mux5
mux5in(0,0) <= qmem1(0);
mux5in(0,1) <= qmem1(1);
mux5in(0,2) <= qmem1(2);
mux5in(0,3) <= qmem1(3);
mux5in(0,4) <= qmem1(4);
mux5in(0,5) <= qmem1(5);
mux5in(0,6) <= qmem1(6);
mux5in(0,7) <= qmem1(7);
mux5in(0,8) <= qmem1(8);
mux5in(0,9) <= qmem1(9);
mux5in(0,10) <= qmem1(10);
mux5in(0,11) <= qmem1(11);
mux5in(0,12) <= qmem1(12);
mux5in(0,13) <= qmem1(13);
mux5in(0,14) <= qmem1(14);

mux5in(1,0) <= qmem2(0);
mux5in(1,1) <= qmem2(1);
mux5in(1,2) <= qmem2(2);
mux5in(1,3) <= qmem2(3);
mux5in(1,4) <= qmem2(4);
mux5in(1,5) <= qmem2(5);
mux5in(1,6) <= qmem2(6);
mux5in(1,7) <= qmem2(7);
mux5in(1,8) <= qmem2(8);
mux5in(1,9) <= qmem2(9);
mux5in(1,10) <= qmem2(10);
mux5in(1,11) <= qmem2(11);
mux5in(1,12) <= qmem2(12);
mux5in(1,13) <= qmem2(13);
mux5in(1,14) <= qmem2(14);
mux5sel(0) <= not(sal1(4) and habil);


dout<=qff1(0);
habilserie<=qff2(0);
end process;
END arc;
```

Programa 2

```
ENTITY format IS
        PORT
        (clockmax,Datain,habilserie,manbip   : IN     STD_LOGIC;
         Dataout                             : OUT STD_LOGIC_VECTOR(1 DOWNTO 0));
END format;

ARCHITECTURE arc OF format IS

COMPONENT lpm_mux
 GENERIC (LPM_WIDTH: POSITIVE;
    LPM_WIDTHS: POSITIVE;
    LPM_SIZE: POSITIVE);
PORT (data: IN STD_LOGIC_2D(LPM_SIZE-1 DOWNTO 0, LPM_WIDTH-1 DOWNTO 0);
    sel: IN STD_LOGIC_VECTOR(LPM_WIDTHS-1 DOWNTO 0);
    result: OUT STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0));

signal mux1in: STD_LOGIC_2D(1 DOWNTO 0, 0 DOWNTO 0);
signal mux2in: STD_LOGIC_2D(1 DOWNTO 0, 1 DOWNTO 0);
signal mux2out: STD_LOGIC_VECTOR(1 DOWNTO 0);
signal mux1out,mux1sel,mux2sel: STD_LOGIC_VECTOR(0 DOWNTO 0);

BEGIN

mux1:lpm_mux GENERIC MAP (1,1,2)
PORT MAP (mux1in,mux1sel,mux1out);
mux2:lpm_mux GENERIC MAP (2,1,2)
PORT MAP (mux2in,mux2sel,mux2out);

generar : process

begin

--mux1
mux1in(1,0) <= clockmax;
mux1in(0,0) <= not clockmax;
mux1sel(0) <= datain;

--mux2
mux2in(0,0) <=clockmax;
mux2in(0,1) <=clockmax and (not datain);
mux2in(1,0) <= '1';
mux2in(1,1) <=not mux1out(0);
mux2sel(0) <= manbip;

--Salidas
dataout(0)<=mux2out(0) and habilserie;
dataout(1)<=mux2out(1) or not habilserie;
end process;
END arc;
```

Programa 3

```
ENTITY compa2 IS
PORT(din                                : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
        dout                            : OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
END compa2;

ARCHITECTURE arc OF compa2 IS

COMPONENT lpm_add_sub
  GENERIC (LPM_WIDTH: POSITIVE);
    PORT (dataa, datab: IN STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0);
    add_sub: IN STD_LOGIC := '0';
    result: OUT STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0));
END COMPONENT;

BEGIN
add1: lpm_add_sub GENERIC MAP (8)
PORT MAP (din,dataddb1,add_sub1,add1out);

signal dataddb1,add1out:STD_LOGIC_VECTOR(7 DOWNTO 0);
signal add_sub1:STD_LOGIC;

generar : PROCESS
BEGIN

--add1
dataddb1<="10000000";
add_sub1<='0';  --el valor ‘0’ hace funcionar al componente como un restador

--Salida
dout<=add1out;

end process;
END arc;
```

Programa 4

```
ENTITY habiltrama IS
PORT(din                    : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
        clockr,clocksupermaxr,reset:IN STD_LOGIC;
        habil          : OUT    STD_LOGIC);
END habiltrama;

ARCHITECTURE arc OF habiltrama IS

COMPONENT lpm_counter
  GENERIC (LPM_WIDTH: POSITIVE);
 PORT ( clock: IN  STD_LOGIC;
        aclr: IN STD_LOGIC := '0';
          clk_en: IN STD_LOGIC := '1';
            q: OUT STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0));
END COMPONENT;

COMPONENT lpm_ff
   GENERIC (LPM_WIDTH: POSITIVE);
   PORT (data: IN STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0);
      clock: IN STD_LOGIC;
     q: OUT STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0));
END COMPONENT;

signal sal2,sal3                        :STD_LOGIC_VECTOR(4 DOWNTO 0);
signal dff3,qff3                        :STD_LOGIC_VECTOR(2 DOWNTO 0);
signal sal1                             :STD_LOGIC_VECTOR(1 DOWNTO 0);
signal dff1,qff1,dff2,qff2              :STD_LOGIC_VECTOR(0 DOWNTO 0);
signal clock1,clock2,aclr1,aclr2,aclr3,clk_en1,clk_en2,clk_en3,aclrff1    :STD_LOGIC;

BEGIN

ff1: lpm_ff GENERIC MAP (1)
PORT MAP (dff1,clocksupermaxr,qff1);
ff2: lpm_ff GENERIC MAP (1)
PORT MAP (dff2,clocksupermaxr,qff2);
ff3: lpm_ff GENERIC MAP (3)
PORT MAP (dff3,clocksupermaxr,qff3);

cont1: lpm_counter GENERIC MAP (2)
PORT MAP (clockr,aclr1,clk_en1,sal1);
cont2: lpm_counter GENERIC MAP (5)
PORT MAP (clockr,aclr2,clk_en2,sal2);
cont3: lpm_counter GENERIC MAP (5)
PORT MAP (clockr,aclr3,clk_en3,sal3);

generar : PROCESS

BEGIN

--cont1
aclr1<=qff3(0) or reset;
clk_en1<=(not(sal1(0)and sal1(1))) and ((not din(7))and(din(6)or din(5)));

--cont2
aclr2<=qff3(1);
clk_en2<= (sal1(0)and sal1(1)) and  not(sal2(4)and sal3(3)and sal3(4)) and ((not din(7))and(din(6)or din(5)));

--cont3
aclr3<=qff3(2);
clk_en3<= (sal1(0)and sal1(1)) and  not(sal2(4)and sal3(3)and sal3(4));      -- si sal3 llega a 26, resetea todo

--ff1
dff1(0)<= sal1(0)or sal1(1);
```

```
--ff2
dff2(0)<=(din(7) or(not (din(6) or din(5))));

--ff3
dff3(0) <= ( qff2(0) and (not(sal1(0)and sal1(1))) ) or (sal3(1) and sal3(3) and sal3(4));
dff3(1)<=(sal3(1) and sal3(3) and sal3(4)) or reset;
dff3(2)<=(sal3(1) and sal3(3) and sal3(4)) or reset;

--Salida
habil<=qff1(0) ;

end process;
END arc;
```

Programa 5

```
ENTITY mem IS
        PORT(din                                : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
      add3in,tot                                : IN STD_LOGIC_VECTOR (5 DOWNTO 0);
      dpos                                      : IN STD_LOGIC_VECTOR (4 DOWNTO 0);
      habil,clockr,clocksupermaxr,stposytot     : IN      STD_LOGIC;
      stconv,reset                              : OUT STD_LOGIC;
      dout                                      : OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
END mem;

ARCHITECTURE arc OF mem IS

COMPONENT lpm_counter
  GENERIC (LPM_WIDTH: POSITIVE;
                    LPM_AVALUE: STRING := "unused");
 PORT ( clock: IN  STD_LOGIC;
      aclr: IN STD_LOGIC := '0';
      clk_en: IN STD_LOGIC := '1';
      aset: IN STD_LOGIC := '0';
      q: OUT STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0));
END COMPONENT;

COMPONENT lpm_ram_dq
  GENERIC (LPM_WIDTH: POSITIVE;
        LPM_WIDTHAD: POSITIVE;
        LPM_NUMWORDS: POSITIVE;
                        LPM_TYPE: STRING := "L_RAM_DQ";
        LPM_FILE: STRING := "UNUSED";
        LPM_INDATA: STRING := "REGISTERED";
        LPM_ADDRESS_CONTROL: STRING := "REGISTERED";
        LPM_OUTDATA: STRING := "UNREGISTERED";
        LPM_HINT: STRING := "UNUSED");
  PORT (data: IN STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0);
                    address: IN STD_LOGIC_VECTOR(LPM_WIDTHAD-1 DOWNTO 0);
                    we: IN STD_LOGIC := '1';
                    inclock: IN STD_LOGIC := '1';
                    q: OUT STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0));
END COMPONENT;

COMPONENT lpm_compare
  GENERIC (LPM_WIDTH: POSITIVE);
  PORT (dataa, datab: IN STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0);
           ageb: OUT STD_LOGIC);
END COMPONENT;

COMPONENT lpm_ff
  GENERIC (LPM_WIDTH: POSITIVE);
    PORT (data: IN STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0);
    clock,enable,aclr: IN STD_LOGIC;
    q: OUT STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0));
END COMPONENT;

COMPONENT lpm_add_sub
  GENERIC (LPM_WIDTH: POSITIVE);
    PORT (dataa, datab: IN STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0);
    result: OUT STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0));
END COMPONENT;

COMPONENT lpm_mux
  GENERIC (LPM_WIDTH: POSITIVE;
    LPM_WIDTHS: POSITIVE;
    LPM_SIZE: POSITIVE);
PORT (data: IN STD_LOGIC_2D(LPM_SIZE-1 DOWNTO 0, LPM_WIDTH-1 DOWNTO 0);
    sel: IN STD_LOGIC_VECTOR(LPM_WIDTHS-1 DOWNTO 0);
    result: OUT STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0));
```

```
END COMPONENT;

COMPONENT monoestable
PORT (Datain,clock              : IN  STD_LOGIC;
      Dataout             : OUT STD_LOGIC);
END COMPONENT;

COMPONENT retardo
        PORT (inret,clock       : IN  STD_LOGIC;
          outret          : OUT       STD_LOGIC );
END COMPONENT;

signal mux1in                           :STD_LOGIC_2D(1 DOWNTO 0, 7 DOWNTO 0);
signal mux2in,mux3in                    :STD_LOGIC_2D(1 DOWNTO 0, 6 DOWNTO 0);
signal mux4in                           :STD_LOGIC_2D(1 DOWNTO 0, 5 DOWNTO 0);
signal mux5in                           :STD_LOGIC_2D(1 DOWNTO 0, 0 DOWNTO 0);
signal qmem1,dmem1,qmem2,dmem2,qmem3,dmem3,mux1out,dff3
                                        :STD_LOGIC_VECTOR(7 DOWNTO 0);
signal address1,address2,sal1,sal2,datab1,datab2,tot7,dpos7,datadda2,add2out,dataddb2,
mux2out,mux3out,add3out             :STD_LOGIC_VECTOR(6 DOWNTO 0);
signal address3,mux4out,sal5,sal7   :STD_LOGIC_VECTOR(5 DOWNTO 0);
signal sal3,sal4,dataddb1,add1out   :STD_LOGIC_VECTOR(4 DOWNTO 0);
signal sal8                         :STD_LOGIC_VECTOR(3 DOWNTO 0);
signal dff2,qff2                    :STD_LOGIC_VECTOR(2 DOWNTO 0);
signal sal6                         :STD_LOGIC_VECTOR(1 DOWNTO 0);
signal dff1,qff1,qff4,dff5,qff5,dff6,qff6,dff7,qff7,mux1sel,mux2sel,mux3sel,mux4sel,mux5out,
mux5sel                             :STD_LOGIC_VECTOR(0 DOWNTO 0);
signal clock1,aclr1,aclr2,aclr3,aclr4,aclr5,aclr6,aclr7,aclr8,we1,we2,we3,clken1,clken2,
clken3,comp2out,comp3out,clk_en1,clk_en2,clk_en3,clk_en4,clk_en5,clk_en6,clk_en7,
clk_en8,aset1,monout,monin,comp1out,ena1,ena4,clr1,clr7:STD_LOGIC;

BEGIN
add1: lpm_add_sub GENERIC MAP (5)
PORT MAP (dpos,dataddb1,add1out);
add2: lpm_add_sub GENERIC MAP (7)
PORT MAP (datadda2,dataddb2,add2out);
add3: lpm_add_sub GENERIC MAP (7)
PORT MAP (dpos7,tot7,add3out);

comp1: lpm_compare GENERIC MAP (7)
PORT MAP (sal1,datab1,comp1out);
comp2: lpm_compare GENERIC MAP (7)
PORT MAP (sal2,datab2,comp2out);
comp3: lpm_compare GENERIC MAP (6)
PORT MAP (sal5,tot,comp3out);

cont1: lpm_counter GENERIC MAP (7,"10")
PORT MAP (clockr,aclr1,clk_en1,monout,sal1);
cont2: lpm_counter GENERIC MAP (7)
PORT MAP (clockr,aclr2,clk_en2,aset1,sal2);
cont3: lpm_counter GENERIC MAP (5)
PORT MAP (clockr,aclr3,clk_en3,aset1,sal3);
cont4: lpm_counter GENERIC MAP (5,"20")
PORT MAP (clockr,aclr4,clk_en4,monout,sal4);
cont5: lpm_counter GENERIC MAP (6)
PORT MAP (clocksupermaxr,aclr5,clk_en5,aset1,sal5);
cont6: lpm_counter GENERIC MAP (2)
PORT MAP (qff1(0),aclr6,clk_en6,aset1,sal6);
cont7: lpm_counter GENERIC MAP (6)
PORT MAP (clocksupermaxr,aclr7,clk_en7,aset1,sal7);
cont8: lpm_counter GENERIC MAP (4)
PORT MAP (clockr,aclr8,clk_en8,aset1,sal8);


ff1: lpm_ff GENERIC MAP (1)
PORT MAP (dff1,clockr,ena1,clr1,qff1);
ff2: lpm_ff GENERIC MAP (3)
```

```
PORT MAP (dff2,clocksupermaxr,ena1,clr1,qff2);
ff3: lpm_ff GENERIC MAP (8)
PORT MAP (dff3,clocksupermaxr,ena1,clr1,dout);
ff4: lpm_ff GENERIC MAP (1)
PORT MAP (qff1,clocksupermaxr,ena4,clr1,qff4);
ff5: lpm_ff GENERIC MAP (1)
PORT MAP (dff5,clocksupermaxr,ena1,clr1,qff5);
ff6: lpm_ff GENERIC MAP (1)
PORT MAP (dff6,clockr,ena1,clr1,qff6);
ff7: lpm_ff GENERIC MAP (1)
PORT MAP (dff7,stposytot,ena1,clr7,qff7);

mem1: lpm_ram_dq GENERIC MAP (8,7,85)
PORT MAP (dmem1,address1,we1,clocksupermaxr,qmem1);
mem2: lpm_ram_dq GENERIC MAP (8,7,85)
PORT MAP (dmem2,address2,we2,clocksupermaxr,qmem2);
mem3: lpm_ram_dq GENERIC MAP (8,6,64)
PORT MAP (dmem3,address3,we3,clocksupermaxr,qmem3);

mono1: monoestable PORT MAP(monin,clocksupermaxr,monout);

mux1:lpm_mux GENERIC MAP (8,1,2) -- tamaño entrada,selecion, entradas
PORT MAP (mux1in,mux1sel,mux1out);
mux2: lpm_mux GENERIC MAP (7,1,2)
PORT MAP (mux2in,mux2sel,mux2out);
mux3: lpm_mux GENERIC MAP (7,1,2)
PORT MAP (mux3in,mux3sel,mux3out);
mux4: lpm_mux GENERIC MAP (6,1,2)
PORT MAP (mux4in,mux4sel,mux4out);

ret1:retardo PORT MAP(habil,clocksupermaxr,monin);

generar : PROCESS

BEGIN
--add1
dataddb1<="01010";

--add2
datadda2(0)<=add1out(0);
datadda2(1)<=add1out(1);
datadda2(2)<=add1out(2);
datadda2(3)<=add1out(3);
datadda2(4)<=add1out(4);
datadda2(5)<='0';
datadda2(6)<='0';

dataddb2(0)<=sal5(0);
dataddb2(1)<=sal5(1);
dataddb2(2)<=sal5(2);
dataddb2(3)<=sal5(3);
dataddb2(4)<=sal5(4);
dataddb2(5)<=sal5(5);
dataddb2(6)<='0';

--add3
tot7(0)<=tot(0);
tot7(1)<=tot(1);
tot7(2)<=tot(2);
tot7(3)<=tot(3);
tot7(4)<=tot(4);
tot7(5)<=tot(5);
tot7(6)<='0';

dpos7(0)<=dpos(0);
dpos7(1)<=dpos(1);
dpos7(2)<=dpos(2);
```

```
dpos7(3)<=dpos(3);
dpos7(4)<=dpos(4);
dpos7(5)<=dpos(4);
dpos7(6)<=dpos(4);

-- comp1
datab1<=add3out;

-- comp2
datab2<=add3out;

--          (A)
aclr1 <= (sal3(2)and sal3(4)) or (not habil);
clk_en1<='1';
aclr3 <= qff2(1)or qff7(0);
clk_en3<=qff2(0)and not(sal3(2)and sal3(4));
aset1<='0';

--          (B)
aclr2 <= (sal4(2)and sal4(4))or (not habil);
clk_en2<='1';
aclr4 <= qff2(0) or qff7(0);
clk_en4<=qff2(1)and not(sal4(2)and sal4(4));

--cont5
aclr5 <=not((sal3(2)and sal3(4))or(sal4(2)and sal4(4)))or (not habil);
clk_en5<=((sal3(2)and sal3(4))or(sal4(2)and sal4(4)))and not(sal5(0)and sal5(1)and sal5(2)and sal5(3)and
sal5(4)and sal5(5));

--cont6
clk_en6<=not sal6(1);
aclr6<=qff6(0);
dff6(0)<=(not habil);
--aclr6s<=aclr6;

--cont7
aclr7<=qff5(0);
clk_en7<=(not sal7(5)) and ((sal3(2)and sal3(4))or(sal4(2)and sal4(4)))and not(sal5(0)and sal5(1)and
sal5(2)and sal5(3)and sal5(4)and sal5(5)) and ((dmem3(7)and dmem3(6) and dmem3(5)) or((not
dmem3(7))and(not (dmem3(6)or dmem3(5))))) and qff2(2);
dff5(0)<=((not((sal3(2)and sal3(4))or(sal4(2)and sal4(4)))) and (not sal7(5))) or qff7(0) or(not sal6(1));

--cont8
aclr8<=not qff7(0);
clk_en8<='1';
clr7<=sal8(3);

--ff1
dff1(0)<=sal5(0)and sal5(1)and sal5(2)and sal5(3)and sal5(4)and sal5(5);
ena1<='1';
clr1<=sal7(5) and stposytot;

--ff2
dff2(0)<=comp1out;
dff2(1)<=comp2out;
dff2(2)<=not comp3out;

--ff3
dff3(0)<=qmem3(0) and (not sal7(5));
dff3(1)<=qmem3(1) and (not sal7(5));
dff3(2)<=qmem3(2) and (not sal7(5));
dff3(3)<=qmem3(3) and (not sal7(5));
dff3(4)<=qmem3(4) and (not sal7(5));
dff3(5)<=qmem3(5) and (not sal7(5));
dff3(6)<=qmem3(6) and (not sal7(5));
dff3(7)<=qmem3(7) and (not sal7(5));
```

```
--ff4
ena4<=sal6(1);


--mux1
mux1in(0,0)<=qmem1(0);
mux1in(0,1)<=qmem1(1);
mux1in(0,2)<=qmem1(2);
mux1in(0,3)<=qmem1(3);
mux1in(0,4)<=qmem1(4);
mux1in(0,5)<=qmem1(5);
mux1in(0,6)<=qmem1(6);
mux1in(0,7)<=qmem1(7);

mux1in(1,0)<=qmem2(0);
mux1in(1,1)<=qmem2(1);
mux1in(1,2)<=qmem2(2);
mux1in(1,3)<=qmem2(3);
mux1in(1,4)<=qmem2(4);
mux1in(1,5)<=qmem2(5);
mux1in(1,6)<=qmem2(6);
mux1in(1,7)<=qmem2(7);

mux1sel(0)<=we1;

--mux2
mux2in(0,0)<=sal1(0);
mux2in(0,1)<=sal1(1);
mux2in(0,2)<=sal1(2);
mux2in(0,3)<=sal1(3);
mux2in(0,4)<=sal1(4);
mux2in(0,5)<=sal1(5);
mux2in(0,6)<=sal1(6);

mux2in(1,0)<=add2out(0);
mux2in(1,1)<=add2out(1);
mux2in(1,2)<=add2out(2);
mux2in(1,3)<=add2out(3);
mux2in(1,4)<=add2out(4);
mux2in(1,5)<=add2out(5);
mux2in(1,6)<=add2out(6);

mux2sel(0)<= not we1;

--mux3
mux3in(0,0)<=sal2(0);
mux3in(0,1)<=sal2(1);
mux3in(0,2)<=sal2(2);
mux3in(0,3)<=sal2(3);
mux3in(0,4)<=sal2(4);
mux3in(0,5)<=sal2(5);
mux3in(0,6)<=sal2(6);

mux3in(1,0)<=add2out(0);
mux3in(1,1)<=add2out(1);
mux3in(1,2)<=add2out(2);
mux3in(1,3)<=add2out(3);
mux3in(1,4)<=add2out(4);
mux3in(1,5)<=add2out(5);
mux3in(1,6)<=add2out(6);

mux3sel(0)<= not we2;

--mux4
mux4in(0,0)<=sal5(0);
mux4in(0,1)<=sal5(1);
mux4in(0,2)<=sal5(2);
```

```
mux4in(0,3)<=sal5(3);
mux4in(0,4)<=sal5(4);
mux4in(0,5)<=sal5(5);

mux4in(1,0)<=add3in(0);
mux4in(1,1)<=add3in(1);
mux4in(1,2)<=add3in(2);
mux4in(1,3)<=add3in(3);
mux4in(1,4)<=add3in(4);
mux4in(1,5)<=add3in(5);

mux4sel(0)<=qff4(0);

--mem1
dmem1<=din;
address1<=mux2out;
we1<= not(sal3(2)and sal3(4));

--mem2
dmem2<=din;
address2<=mux3out;
we2<= not(sal4(2)and sal4(4));

--mem3
dmem3(0)<=mux1out(0) and qff2(2);
dmem3(1)<=mux1out(1) and qff2(2);
dmem3(2)<=mux1out(2) and qff2(2);
dmem3(3)<=mux1out(3) and qff2(2);
dmem3(4)<=mux1out(4) and qff2(2);
dmem3(5)<=mux1out(5) and qff2(2);
dmem3(6)<=mux1out(6) and qff2(2);
dmem3(7)<=mux1out(7) and qff2(2);

address3(0)<=mux4out(0);
address3(1)<=mux4out(1);
address3(2)<=mux4out(2);
address3(3)<=mux4out(3);
address3(4)<=mux4out(4);
address3(5)<=mux4out(5);

--Salidas
we3<=not qff4(0);
dff7(0)<=sal7(5);
reset<=qff7(0);
stconv<=qff4(0);

END process;
END arc;
```

Programa 6

```
ENTITY er IS
PORT(din                                        : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
     stconv,clocksupermaxr,clockffr,manbip,habil,reset    : IN STD_LOGIC;
     tot,tota,totb,totbb,totc                   : IN STD_LOGIC_VECTOR (5 DOWNTO 0);
     e1,e2                                      : OUT STD_LOGIC_VECTOR (16 DOWNTO 0);
     add3out                                    : OUT STD_LOGIC_VECTOR (5 DOWNTO 0);
     stposytot,salida,habilout                  : OUT STD_LOGIC);
  END er;

ARCHITECTURE arc OF er IS

COMPONENT lpm_counter
GENERIC (LPM_WIDTH: POSITIVE;
            LPM_AVALUE: STRING := "unused");
 PORT (clock: IN  STD_LOGIC;
          aclr: IN STD_LOGIC := '0';
          clk_en: IN STD_LOGIC := '1';
          q: OUT STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0));
END COMPONENT;

COMPONENT lpm_compare
GENERIC (LPM_WIDTH: POSITIVE);
PORT (dataa, datab: IN STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0);
         agb: OUT STD_LOGIC);
END COMPONENT;

COMPONENT lpm_ff
GENERIC (LPM_WIDTH: POSITIVE);
PORT (data: IN STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0);
         clock,enable,aclr: IN STD_LOGIC;
         q: OUT STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0));
END COMPONENT;

COMPONENT lpm_add_sub
GENERIC (LPM_WIDTH: POSITIVE;
            LPM_PIPELINE: INTEGER := 1);
PORT (dataa, datab: IN STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0);
         add_sub: IN STD_LOGIC := '1';
         clock,aclr : IN STD_LOGIC := '0';
         result: OUT STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0));
END COMPONENT;

COMPONENT lpm_mux
GENERIC (LPM_WIDTH: POSITIVE;
            LPM_WIDTHS: POSITIVE;
            LPM_SIZE: POSITIVE);
PORT (data: IN STD_LOGIC_2D(LPM_SIZE-1 DOWNTO 0, LPM_WIDTH-1 DOWNTO 0);
         sel: IN STD_LOGIC_VECTOR(LPM_WIDTHS-1 DOWNTO 0);
        result: OUT STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0));
END COMPONENT;

signal mux2in                                   :STD_LOGIC_2D(1 DOWNTO 0, 5 DOWNTO 0);
signal mux3in                                   :STD_LOGIC_2D(1 DOWNTO 0, 8 DOWNTO 0);
signal mux4in                                   :STD_LOGIC_2D(1 DOWNTO 0, 0 DOWNTO 0);
signal aclr1,comp1out,comp2out,comp3out,comp4out,comp5out,clk_en1,clk_en2,clk_en3,
clk_en4,ena1,ena2,ena3,ena4,ena5,ena6,ena7,add_sub1,add_sub5,clken,clken4,clken1,clken5,
addclr1,addclr4,clockff2,clockff3,aclr7,ena8,ena9,aclrff11       :STD_LOGIC;
signal mux2sel,mux3sel,qff5,dff5,dff7,qff7,dff8,qff8,
dff9,qff9,dff11,qff11,mux4out                   :STD_LOGIC_VECTOR(0 DOWNTO 0);
signal sal2,sal3,sal4                           :STD_LOGIC_VECTOR(1 DOWNTO 0);
signal mux2out,sal1,add6out                     :STD_LOGIC_VECTOR(5 DOWNTO 0);
signal mux3out                                  :STD_LOGIC_VECTOR(8 DOWNTO 0);
signal dff1,qff1,dff4,qff4,datad,qff6,add5out,add1out,datadand
                                                :STD_LOGIC_VECTOR(15 DOWNTO 0);
signal dff2,qff2,dff3,qff3,dataaddb2,qff10      :STD_LOGIC_VECTOR(16 DOWNTO 0);
```

```
BEGIN

add1: lpm_add_sub GENERIC MAP (16)
PORT MAP (qff1,datad,add_sub1,clocksupermaxr,addclr1,dff1);
add2: lpm_add_sub GENERIC MAP (17)--e2 0-0.8
PORT MAP (qff10,dataaddb2,add_sub1,clocksupermaxr,addclr1,dff2);
add3: lpm_add_sub GENERIC MAP (17)--e1 0.2-tot
PORT MAP (qff3,dataaddb2,add_sub1,clocksupermaxr,addclr1,dff3);
add4: lpm_add_sub GENERIC MAP (16)
PORT MAP (qff4,datadand,add_sub1,clocksupermaxr,addclr4,dff4);
add5: lpm_add_sub GENERIC MAP (16)  --restador
PORT MAP (qff1,qff4,add_sub5,clocksupermaxr,addclr1,add5out);
add6: lpm_add_sub GENERIC MAP (16)  --restador
PORT MAP (sal1,totb,add_sub5,clocksupermaxr,addclr1,add6out);

comp1: lpm_compare GENERIC MAP (6)
PORT MAP (tota,sal1,comp1out);
comp2: lpm_compare GENERIC MAP (6)
PORT MAP (sal1,totb,comp2out);
comp3: lpm_compare GENERIC MAP (6)
PORT MAP (sal1,totc,comp3out);
comp4: lpm_compare GENERIC MAP (6)
PORT MAP (sal1,tot,comp4out);
comp5: lpm_compare GENERIC MAP (6)
PORT MAP (sal1,totbb,comp5out);

cont1: lpm_counter GENERIC MAP (6)
PORT MAP (qff5(0),aclr1,clk_en1,sal1);
cont2: lpm_counter GENERIC MAP (2)
PORT MAP (qff5(0),aclr1,clk_en2,sal2);
cont3: lpm_counter GENERIC MAP (2)
PORT MAP (qff5(0),aclr1,clk_en3,sal3);
cont4: lpm_counter GENERIC MAP (2)
PORT MAP (qff5(0),aclr1,clk_en4,sal4);

ff1: lpm_ff GENERIC MAP (16)
PORT MAP (dff1,clocksupermaxr,ena1,aclr1,qff1);
ff2: lpm_ff GENERIC MAP (17)
PORT MAP (dff2,clocksupermaxr,ena2,aclr1,qff2);
ff3: lpm_ff GENERIC MAP (17)
PORT MAP (dff3,not qff5(0),ena3,aclr1,qff3);
ff4: lpm_ff GENERIC MAP (16)
PORT MAP (dff4,clockffr,ena4,aclr1,qff4);
ff5: lpm_ff GENERIC MAP (1)
PORT MAP (dff5,clocksupermaxr,ena5,aclr1,qff5);
ff6: lpm_ff GENERIC MAP (16)
PORT MAP (add5out,qff5(0),ena6,aclr1,qff6);
ff7: lpm_ff GENERIC MAP (1)
PORT MAP (dff7,clocksupermaxr,ena7,aclr7,qff7);
ff8: lpm_ff GENERIC MAP (1)
PORT MAP (dff8,clocksupermaxr,ena5,aclr7,qff8);
ff9: lpm_ff GENERIC MAP (1)
PORT MAP (mux4out,qff7(0),ena9,aclr7,qff9);
ff10: lpm_ff GENERIC MAP (17)
PORT MAP (dff2,clocksupermaxr,ena6,aclr1,qff10);
ff11: lpm_ff GENERIC MAP (1)
PORT MAP (dff11,stconv,ena5,aclrff11,qff11);

mux2:lpm_mux GENERIC MAP (6,1,2) -- tamaño entrada,seleccion, entradas
PORT MAP (mux2in,mux2sel,mux2out);
mux3: lpm_mux GENERIC MAP (9,1,2)
PORT MAP (mux3in,mux3sel,mux3out);
mux4: lpm_mux GENERIC MAP (1,1,2)
PORT MAP (mux4in,mux3sel,mux4out);

generar : PROCESS
```

```
BEGIN

--add1
add_sub1<='1';
datad(0)<=din(0);
datad(1)<=din(1);
datad(2)<=din(2);
datad(3)<=din(3);
datad(4)<=din(4);
datad(5)<=din(5);
datad(6)<=din(6);
datad(7)<=din(7);
datad(8)<=din(7);
datad(9)<=din(7);
datad(10)<=din(7);
datad(11)<=din(7);
datad(12)<=din(7);
datad(13)<=din(7);
datad(14)<=din(7);
datad(15)<=din(7);
addclr1<='0';

--add2
dataaddb2(0)<=qff6(0);
dataaddb2(1)<=qff6(1);
dataaddb2(2)<=qff6(2);
dataaddb2(3)<=qff6(3);
dataaddb2(4)<=qff6(4);
dataaddb2(5)<=qff6(5);
dataaddb2(6)<=qff6(6);
dataaddb2(7)<=qff6(7);
dataaddb2(8)<=qff6(8);
dataaddb2(9)<=qff6(9);
dataaddb2(10)<=qff6(10);
dataaddb2(11)<=qff6(11);
dataaddb2(12)<=qff6(12);
dataaddb2(13)<=qff6(13);
dataaddb2(14)<=qff6(14);
dataaddb2(15)<=qff6(15);
dataaddb2(16)<=qff6(15);

--add4
datadand(0)<=mux3out(0) and (not qff5(0));
datadand(1)<=mux3out(1) and (not qff5(0));
datadand(2)<=mux3out(2) and (not qff5(0));
datadand(3)<=mux3out(3) and (not qff5(0));
datadand(4)<=mux3out(4) and (not qff5(0));
datadand(5)<=mux3out(5) and (not qff5(0));
datadand(6)<=mux3out(6) and (not qff5(0));
datadand(7)<=mux3out(7) and (not qff5(0));
datadand(8)<=mux3out(8) and (not qff5(0));
datadand(9)<=din(7) and (not qff5(0));
datadand(10)<=din(7) and (not qff5(0));
datadand(11)<=din(7) and (not qff5(0));
datadand(12)<=din(7) and (not qff5(0));
datadand(13)<=din(7) and (not qff5(0));
datadand(14)<=din(7) and (not qff5(0));
datadand(15)<=din(7) and (not qff5(0));
addclr4<=not comp5out;

--add5
add_sub5<='0';

--cont1
aclr1<=(not stconv) or reset;
clk_en1<=qff8(0);
```

```
--cont2
clk_en2<=(not comp1out)and not(sal2(0)and sal2(1));

--cont3
clk_en3<=(comp3out)and not(sal3(0)and sal3(1));

--cont4
clk_en4<=(comp4out)and not(sal4(0)and sal4(1));

--ff1
ena1<=(not (sal4(0)and sal4(1)))and (not qff5(0));

--ff2
ena2<=not (sal3(0)and sal3(1));

--ff3
ena3<=(sal2(0)and sal2(1)) and (not (sal4(0)and sal4(1)));

--ff4
ena4<= comp2out and qff5(0);


--ff5
dff5(0)<=not qff5(0);
ena5<='1';

--ff6
ena6<=(not sal4(1));

--ff7
dff7(0)<=(sal4(0)and sal4(1));
ena7<='1';
aclr7<=(not habil);

--ff8
dff8(0)<=not (sal4(0)and sal4(1));
ena8<=qff7(0);

--ff9
ena9<='1';

--ff11
aclrff11<='0';
dff11(0)<=habil;

--mux2
mux2in(0,0)<=sal1(0);
mux2in(0,1)<=sal1(1);
mux2in(0,2)<=sal1(2);
mux2in(0,3)<=sal1(3);
mux2in(0,4)<=sal1(4);
mux2in(0,5)<=sal1(5);

mux2in(1,0)<=add6out(0);
mux2in(1,1)<=add6out(1);
mux2in(1,2)<=add6out(2);
mux2in(1,3)<=add6out(3);
mux2in(1,4)<=add6out(4);
mux2in(1,5)<=add6out(5);
mux2sel(0)<=comp2out and (not qff5(0));

--mux3
mux3in(0,0)<=din(0);
mux3in(0,1)<=din(1);
mux3in(0,2)<=din(2);
mux3in(0,3)<=din(3);
```

```
        mux3in(0,4)<=din(4);
        mux3in(0,5)<=din(5);
        mux3in(0,6)<=din(6);
        mux3in(0,7)<=din(7);
        mux3in(0,8)<=din(7);

        mux3in(1,0)<='0';
        mux3in(1,1)<=din(0);
        mux3in(1,2)<=din(1);
        mux3in(1,3)<=din(2);
        mux3in(1,4)<=din(3);
        mux3in(1,5)<=din(4);
        mux3in(1,6)<=din(5);
        mux3in(1,7)<=din(6);
        mux3in(1,8)<=din(7);
        mux3sel(0)<=manbip; --con 1 es manchester

        --mux4
        mux4in(0,0)<=not qff10(16);
        mux4in(1,0)<=qff6(15);

        salida<=qff9(0) and (not reset);
        add3out<=mux2out;
        e2<=qff2;
        e1<=qff3;
        stposytot<=qff7(0);
        habilout<=qff11(0);

        END process;
END arc;
```

Programa 7

```
ENTITY dpostot IS
PORT(e1,e2                                          : IN STD_LOGIC_VECTOR (16 DOWNTO 0);
       stconv,stposytot,clocksupermaxr,manbip,habil  : IN STD_LOGIC;
       tot                                          : OUT STD_LOGIC_VECTOR (5 DOWNTO 0);
       dpos                                         : OUT STD_LOGIC_VECTOR (4 DOWNTO 0);
       es,evec2s                                    : OUT STD_LOGIC_VECTOR (16 DOWNTO 0));
END dpostot;

ARCHITECTURE arc OF dpostot IS

COMPONENT abso
  PORT ( din: IN STD_LOGIC_VECTOR(16 DOWNTO 0);
         dout: OUT STD_LOGIC_VECTOR(16 DOWNTO 0));
END COMPONENT;

COMPONENT lpm_add_sub
  GENERIC (LPM_WIDTH: POSITIVE;
           LPM_PIPELINE: INTEGER := 1);
    PORT (dataa, datab: IN STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0);
    add_sub: IN STD_LOGIC := '1';
    clock: IN STD_LOGIC := '0';
             aclr: IN STD_LOGIC := '0';
             result: OUT STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0));
END COMPONENT;

COMPONENT lpm_compare
  GENERIC (LPM_WIDTH: POSITIVE);
  PORT (dataa, datab: IN STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0);
              agb: OUT STD_LOGIC);
END COMPONENT;

COMPONENT lpm_ff
  GENERIC (LPM_WIDTH: POSITIVE;
           LPM_AVALUE: STRING := "UNUSED");
  PORT (data: IN STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0);
     clock,enable: IN STD_LOGIC;
    aset,aclr: IN STD_LOGIC := '0';
    q: OUT STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0));
END COMPONENT;

COMPONENT lpm_mux
  GENERIC (LPM_WIDTH: POSITIVE;
    LPM_WIDTHS: POSITIVE;
    LPM_SIZE: POSITIVE);
PORT (data: IN STD_LOGIC_2D(LPM_SIZE-1 DOWNTO 0, LPM_WIDTH-1 DOWNTO 0);
    sel: IN STD_LOGIC_VECTOR(LPM_WIDTHS-1 DOWNTO 0);
    result: OUT STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0));
END COMPONENT;

COMPONENT lpm_counter
 GENERIC (LPM_WIDTH: POSITIVE);
 PORT ( clock: IN  STD_LOGIC;
      aclr: IN STD_LOGIC := '0';
      clk_en: IN STD_LOGIC := '1';
      q: OUT STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0));
END COMPONENT;

COMPONENT monoestable
PORT (Datain,clock              : IN  STD_LOGIC;
          Dataout             : OUT STD_LOGIC);
END COMPONENT;

signal mux1in,mux2in                        : STD_LOGIC_2D(1 DOWNTO 0, 16 DOWNTO 0);
signal mux3in                               : STD_LOGIC_2D(3 DOWNTO 0, 5 DOWNTO 0);
```

```
signal mux4in                                   : STD_LOGIC_2D(3 DOWNTO 0, 4 DOWNTO 0);
signal mux6in                                   : STD_LOGIC_2D(1 DOWNTO 0, 4 DOWNTO 0);
signal mux5in                                   : STD_LOGIC_2D(1 DOWNTO 0, 11 DOWNTO 0);
signal mux7in                                   : STD_LOGIC_2D(1 DOWNTO 0, 0 DOWNTO 0);
signal aclr1,aclr2,add_sub3,comp1out,comp2out,comp3out,comp4out,comp5out,clk_en1,
clk_en2,clk_en3,clk_en4,ena1,ena3,ena4,ena5,ena6,add_sub1,add_sub5,clken,clken4,
clken1,clken5,aset1,aset2,monout,addclr1,addclr4,clkff2,clkff3,addclr2,aclrff      :STD_LOGIC;
signal dflag,flag,mux1sel,mux2sel,qff5,dff5,mux7out:STD_LOGIC_VECTOR(0 DOWNTO 0);
signal mux3sel,mux4sel,sal1,sal3,sal4           :STD_LOGIC_VECTOR(1 DOWNTO 0);
signal qff3,dff3,add4ina,add4inb,add4out,add5inb,
add5out,mux4out,mux6out                         :STD_LOGIC_VECTOR(4 DOWNTO 0);
signal qff2,add3inb,add3out,aload2,mux3out:STD_LOGIC_VECTOR(5 DOWNTO 0);
signal mux5out                                  :STD_LOGIC_VECTOR(11 DOWNTO 0);
signal qff1,add1out,mode1,mode2,mux1out,mux2out,e,
evec2,modevec2,aload1,compinb                   :STD_LOGIC_VECTOR(16 DOWNTO 0);

BEGIN
abso1: abso
PORT MAP (e1,mode1);
abso2: abso
PORT MAP (e2,mode2);
abso3: abso
PORT MAP (evec2,modevec2);

add1: lpm_add_sub GENERIC MAP (17)
PORT MAP (mux1out,mux2out,add_sub1,clocksupermaxr,addclr1,e);
add2: lpm_add_sub GENERIC MAP (17)
PORT MAP (e,qff1,add_sub1,clocksupermaxr,addclr2,evec2);
add3: lpm_add_sub GENERIC MAP (6)
PORT MAP (qff2,mux3out,add_sub3,clocksupermaxr,addclr1,add3out);
add4: lpm_add_sub GENERIC MAP (5)
PORT MAP (add4ina,add4inb,add_sub3,clocksupermaxr,addclr4,add4out);
add5: lpm_add_sub GENERIC MAP (5)
PORT MAP (mux4out,add5inb,add_sub3,clocksupermaxr,addclr4,add5out);

comp1: lpm_compare GENERIC MAP (17)
PORT MAP (modevec2,compinb,comp1out);

cont1: lpm_counter GENERIC MAP (2)
PORT MAP (stconv,aclr1,clk_en1,sal1);

ff1: lpm_ff GENERIC MAP (17)
PORT MAP (e,clkff2,ena1,aset1,aclrff,qff1);
ff2: lpm_ff GENERIC MAP (6,"50")
PORT MAP (add3out,clkff2,ena1,aset2,aclrff,qff2);
ff3: lpm_ff GENERIC MAP (5)
PORT MAP (dff3,clkff3,ena3,aset1,aclrff,qff3);

mono1: monoestable PORT MAP(habil,clocksupermaxr,monout);

mux1:lpm_mux GENERIC MAP (17,1,2)
PORT MAP (mux1in,mux1sel,mux1out);
mux2: lpm_mux GENERIC MAP (17,1,2)
PORT MAP (mux1in,mux2sel,mux2out);
mux3: lpm_mux GENERIC MAP (6,2,4)
PORT MAP (mux3in,mux3sel,mux3out);
mux4: lpm_mux GENERIC MAP (5,2,4)
PORT MAP (mux4in,mux4sel,mux4out);
mux5: lpm_mux GENERIC MAP (12,1,2)
PORT MAP (mux5in,mux1sel,mux5out);
mux6: lpm_mux GENERIC MAP (5,1,2)
PORT MAP (mux6in,mux2sel,mux6out);
mux7: lpm_mux GENERIC MAP (1,1,2)
PORT MAP (mux7in,mux2sel,mux7out);

generar : PROCESS
```

```
BEGIN
--add1
add_sub1<='0';
addclr1<=not stposytot;

--add2
addclr2<=(not stposytot)or (not(sal1(0)and sal1(1)));

--add3
add_sub3<='1';

--add4
add4ina(0)<=mux6out(0);
add4ina(1)<=mux6out(1);
add4ina(2)<=mux6out(2);
add4ina(3)<=mux6out(3);
add4ina(4)<=mux6out(4);

add4inb(0)<=mux7out(0);
add4inb(1)<='0';
add4inb(2)<='0';
add4inb(3)<='0';
add4inb(4)<='0';

addclr4<=not habil;

--add5
add5inb<= not add4out;

--comp1
compinb(0)<=mux5out(0);
compinb(1)<=mux5out(1);
compinb(2)<=mux5out(2);
compinb(3)<=mux5out(3);
compinb(4)<=mux5out(4);
compinb(5)<=mux5out(5);
compinb(6)<=mux5out(6);
compinb(7)<=mux5out(7);
compinb(8)<=mux5out(8);
compinb(9)<=mux5out(9);
compinb(10)<=mux5out(10);
compinb(11)<=mux5out(11);
compinb(12)<='0';
compinb(13)<='0';
compinb(14)<='0';
compinb(15)<='0';
compinb(16)<='0';

--cont1
aclr1<=not habil;
clk_en1<= not(sal1(0)and sal1(1));

--ff1
aset1<='0';
aclrff<=not habil;
ena1<='1';

--ff2
clkff2<=not stconv;
aset2<=monout or (not habil);
aload2<="110010";

--ff3
dff3<=add5out;
clkff3<=clocksupermaxr;
ena3<=stconv and(sal1(0)and sal1(1));
```

```
--mux1
mux1in(0,0)<=mode1(0);
mux1in(0,1)<=mode1(1);
mux1in(0,2)<=mode1(2);
mux1in(0,3)<=mode1(3);
mux1in(0,4)<=mode1(4);
mux1in(0,5)<=mode1(5);
mux1in(0,6)<=mode1(6);
mux1in(0,7)<=mode1(7);
mux1in(0,8)<=mode1(8);
mux1in(0,9)<=mode1(9);
mux1in(0,10)<=mode1(10);
mux1in(0,11)<=mode1(11);
mux1in(0,12)<=mode1(12);
mux1in(0,13)<=mode1(13);
mux1in(0,14)<=mode1(14);
mux1in(0,15)<=mode1(15);
mux1in(0,16)<=mode1(16);

mux1in(1,0)<=mode2(0);
mux1in(1,1)<=mode2(1);
mux1in(1,2)<=mode2(2);
mux1in(1,3)<=mode2(3);
mux1in(1,4)<=mode2(4);
mux1in(1,5)<=mode2(5);
mux1in(1,6)<=mode2(6);
mux1in(1,7)<=mode2(7);
mux1in(1,8)<=mode2(8);
mux1in(1,9)<=mode2(9);
mux1in(1,10)<=mode2(10);
mux1in(1,11)<=mode2(11);
mux1in(1,12)<=mode2(12);
mux1in(1,13)<=mode2(13);
mux1in(1,14)<=mode2(14);
mux1in(1,15)<=mode2(15);
mux1in(1,16)<=mode2(16);

mux1sel(0)<=not manbip;

--mux2
mux2sel(0)<=manbip;

--mux3
mux3in(0,0)<='0';
mux3in(0,1)<='0';
mux3in(0,2)<='0';
mux3in(0,3)<='0';
mux3in(0,4)<='0';
mux3in(0,5)<='0';

mux3in(1,0)<='0';
mux3in(1,1)<='0';
mux3in(1,2)<='0';
mux3in(1,3)<='0';
mux3in(1,4)<='0';
mux3in(1,5)<='0';

mux3in(2,0)<='0';
mux3in(2,1)<='1';
mux3in(2,2)<='1';
mux3in(2,3)<='1';
mux3in(2,4)<='1';
mux3in(2,5)<='1';

mux3in(3,0)<='0';
mux3in(3,1)<='1';
mux3in(3,2)<='0';
```

```
mux3in(3,3)<='0';
mux3in(3,4)<='0';
mux3in(3,5)<='0';

mux3sel(0)<=e(16);
mux3sel(1)<=comp1out;

--mux4
mux4in(0,0)<='1';
mux4in(0,1)<='0';
mux4in(0,2)<='0';
mux4in(0,3)<='0';
mux4in(0,4)<='0';

mux4in(1,0)<='1';
mux4in(1,1)<='0';
mux4in(1,2)<='0';
mux4in(1,3)<='0';
mux4in(1,4)<='0';

mux4in(2,0)<='1';
mux4in(2,1)<='1';
mux4in(2,2)<='1';
mux4in(2,3)<='1';
mux4in(2,4)<='1';

mux4in(3,0)<='1';
mux4in(3,1)<='1';
mux4in(3,2)<='0';
mux4in(3,3)<='0';
mux4in(3,4)<='0';

mux4sel(0)<=add4out(4);
mux4sel(1)<=comp1out;

--mux5  "000001100100000000";
mux5in(0,0)<='0';
mux5in(0,1)<='0';
mux5in(0,2)<='0';
mux5in(0,3)<='0';
mux5in(0,4)<='0';
mux5in(0,5)<='0';
mux5in(0,6)<='0';
mux5in(0,7)<='1';
mux5in(0,8)<='0';
mux5in(0,9)<='0';
mux5in(0,10)<='1';
mux5in(0,11)<='1';

mux5in(1,0)<='0';
mux5in(1,1)<='0';
mux5in(1,2)<='0';
mux5in(1,3)<='0';
mux5in(1,4)<='0';
mux5in(1,5)<='0';
mux5in(1,6)<='0';
mux5in(1,7)<='0';
mux5in(1,8)<='0';
mux5in(1,9)<='0';
mux5in(1,10)<='1';
mux5in(1,11)<='0';

--mux6
mux6in(0,0)<=e(10);
mux6in(0,1)<=e(11);
mux6in(0,2)<=e(12);
mux6in(0,3)<=e(13);
```

```
mux6in(0,4)<=e(14);

mux6in(1,0)<=e(12);
mux6in(1,1)<=e(13);
mux6in(1,2)<=e(14);
mux6in(1,3)<=e(15);
mux6in(1,4)<=e(16);

--mux7
mux7in(0,0)<=e(9);
mux7in(1,0)<=e(11);

es<=e;
evec2s<=evec2;
dpos<=qff3;
tot<=qff2;
END process;
END arc;
```

Programa 8

```
ENTITY totdiv IS
        PORT(tot                        : IN STD_LOGIC_VECTOR (5 DOWNTO 0);
             tota,totb,totbb,totc       : OUT STD_LOGIC_VECTOR(5 DOWNTO 0));
  END totdiv ;

ARCHITECTURE arc OF totdiv IS

BEGIN

generar : PROCESS

BEGIN
case tot is     --a partir del valor de tot, se presentan los distintos valores de tota, totb, totbb y totc
when "100101" => tota<="001000";
                 totb<="010011";
                 totbb<="010100";
                 totc<="011110";
when "100110" => tota<="001001"; --38
                 totb<="010011";
                 totbb<="010100";
                 totc<="011110";
when "100111" => tota<="001001"; --39
                 totb<="010100";
                 totbb<="010101";
                 totc<="011111";
when "101000" => tota<="001001";--40
                 totb<="010100";
                 totbb<="010101";
                 totc<="100000";
when "101001" => tota<="001001"; --41
                 totb<="010101";
                 totbb<="010110";
                 totc<="100001";
when "101010" => tota<="001001"; --42
                 totb<="010101";
                 totbb<="010110";
                 totc<="100010";
when "101011" => tota<="001010"; --43
                 totb<="010110";
                 totbb<="010111";
                 totc<="100010";
when "101100" => tota<="001010"; --44
                 totb<="010110";
                 totbb<="010111";
                 totc<="100011";
when "101101" => tota<="001010"; --45
                 totb<="010111";
                 totbb<="011000";
                 totc<="100100";
when "101110" => tota<="001010"; --46
                 totb<="010111";
                 totbb<="011000";
                 totc<="100101";
when "101111" => tota<="001010"; --47
                 totb<="011000";
                 totbb<="011001";
                 totc<="100110";
when "110000" => tota<="001011"; --48
                 totb<="011000";
                 totbb<="011001";
                 totc<="100110";
when "110001" => tota<="001011"; --49
                 totb<="011001";
                 totbb<="011010";
```

```
                totc<="100111";
when "110010" => tota<="001011"; --50
                totb<="011001";
                totbb<="011010";
                totc<="101000";
when "110011" => tota<="001011"; --51
                totb<="011010";
                totbb<="011011";
                totc<="101001";
when "110100" => tota<="001011"; --52
                totb<="011010";
                totbb<="011011";
                totc<="101010";
when "110101" => tota<="001100"; --53
                totb<="011011";
                totbb<="011100";
                totc<="101010";
when "110110" => tota<="001100"; --54
                totb<="011011";
                totbb<="011100";
                totc<="101011";
when "110111" => tota<="001100"; --55
                totb<="011100";
                totbb<="011101";
                totc<="101100";
when "111000" => tota<="001100"; --56
                totb<="011100";
                totbb<="011101";
                totc<="101101";
when "111001" => tota<="001100"; --57
                totb<="011101";
                totbb<="011110";
                totc<="101110";
when "111010" => tota<="001101"; --58
                totb<="011101";
                totbb<="011110";
                totc<="101110";
when "111011" => tota<="001101"; --59
                totb<="011110";
                totbb<="011111";
                totc<="101111";
when "111100" => tota<="001101"; --60
                totb<="011110";
                totbb<="011111";
                totc<="110000";
when "111101" => tota<="001101"; --61
                totb<="011111";
                totbb<="100000";
                totc<="110001";
when "111110" => tota<="001101"; --62
                totb<="011111";
                totbb<="100000";
                totc<="110010";
when "111111" => tota<="001110"; --63
                totb<="100000";
                totbb<="100001";
                totc<="110010";

when others =>

end case;
end process;
END arc;
```

Programa 9

```
ENTITY coefs IS
PORT(habil,stposytot,clockmaxr,clocksupermaxr,din,reset: IN STD_LOGIC;
dout,clockout,correc0,correc1,correc2,correc3,correc4,
      correc5,correc6,correcor,dinread                    : OUT STD_LOGIC);
END coefs;

ARCHITECTURE arc OF coefs IS

COMPONENT lpm_ram_dq
GENERIC (LPM_WIDTH: POSITIVE;
         LPM_WIDTHAD: POSITIVE;
     LPM_TYPE: STRING := "L_RAM_DQ";
    --LPM_NUMWORDS: POSITIVE;
      LPM_FILE: STRING := "UNUSED";
      LPM_INDATA: STRING := "REGISTERED";
      LPM_ADDRESS_CONTROL: STRING := "REGISTERED";
      LPM_OUTDATA: STRING := "UNREGISTERED";
      LPM_HINT: STRING := "UNUSED");
PORT (data: IN STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0);
    address: IN STD_LOGIC_VECTOR(LPM_WIDTHAD-1 DOWNTO 0);
    we: IN STD_LOGIC := '1';
    inclock: IN STD_LOGIC := '1';
    q: OUT STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0));
END COMPONENT;

COMPONENT lpm_ff
  GENERIC (LPM_WIDTH: POSITIVE);
  PORT (data: IN STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0);
  clock: IN STD_LOGIC;
  enable: IN STD_LOGIC := '1';
  aclr: IN STD_LOGIC := '0';
  q: OUT STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0));
END COMPONENT;

COMPONENT lpm_mux
  GENERIC (LPM_WIDTH: POSITIVE;
    LPM_WIDTHS: POSITIVE;
    LPM_SIZE: POSITIVE);
PORT (data: IN STD_LOGIC_2D(LPM_SIZE-1 DOWNTO 0, LPM_WIDTH-1 DOWNTO 0);
    sel: IN STD_LOGIC_VECTOR(LPM_WIDTHS-1 DOWNTO 0);
    result: OUT STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0));
END COMPONENT;

COMPONENT lpm_counter
 GENERIC (LPM_WIDTH: POSITIVE);
PORT ( clock: IN  STD_LOGIC;
       aclr: IN STD_LOGIC := '0';
        clk_en: IN STD_LOGIC := '1';
         q: OUT STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0));
END COMPONENT;

COMPONENT monoestable
PORT (Datain,clock              : IN  STD_LOGIC;
          Dataout           : OUT STD_LOGIC);
END COMPONENT;

COMPONENT monoestable2
PORT (Datain,clock              : IN  STD_LOGIC;
          Dataout           : OUT STD_LOGIC);
END COMPONENT;

signal mux1in                           : STD_LOGIC_2D(15 DOWNTO 0, 3 DOWNTO 0);
signal mux2in                           : STD_LOGIC_2D(15 DOWNTO 0, 6 DOWNTO 0);
signal mux3in                           : STD_LOGIC_2D(7 DOWNTO 0, 0 DOWNTO 0);
signal mux2out,dff4,qff4,dff5,qff5      : STD_LOGIC_VECTOR(6 DOWNTO 0);
```

```
signal sal1                                          : STD_LOGIC_VECTOR(4 DOWNTO 0);
signal dff1,qff1,dff2,qff2,sal2,sal3,mux1out         : STD_LOGIC_VECTOR(3 DOWNTO 0);
signal mux1sel,mux2sel                               :STD_LOGIC_VECTOR(3 DOWNTO 0);
signal mux3sel                                       :STD_LOGIC_VECTOR(2 DOWNTO 0);
signal dff3,qff3,dff6,qff6,dff7,qff7,dff8,qff8,dff9,qff9,dff10,qff10,mux3out
                                                     : STD_LOGIC_VECTOR(0 DOWNTO 0);
signal inmono1,outmono1,clockff1,clockff2,clockff4,clockff5,clockff7,rden1,aclr1,aclr2,aclr3,
clk_en1,clk_en2,clk_en3,aclrff1,aclrff2,st,aclrff4,aclrff5,enable1,enable2,
monin1,monout1,aclrff6                               : STD_LOGIC;

BEGIN

cont1: lpm_counter GENERIC MAP (5)
PORT MAP (stposytot,aclr1,clk_en1,sal1);--cuenta patron
cont2: lpm_counter GENERIC MAP (4)
PORT MAP (stposytot,aclr2,clk_en2,sal2);
cont3: lpm_counter GENERIC MAP (4)
PORT MAP (stposytot,aclr2,clk_en3,sal3);--cuenta ceros del final

ff1: lpm_ff GENERIC MAP (4)
PORT MAP (dff1,clockff1,enable1,aclrff1,qff1);
ff2: lpm_ff GENERIC MAP (4)
PORT MAP (dff2,clockff2,enable1,aclrff2,qff2);
ff3: lpm_ff GENERIC MAP (1)
PORT MAP (dff3,clockmaxr,enable1,aclrff4,qff3);
ff4: lpm_ff GENERIC MAP (7)
PORT MAP (dff4,clockff4,enable2,aclrff4,qff4);
ff5: lpm_ff GENERIC MAP (7)
PORT MAP (dff5,clockff5,enable1,aclrff5,qff5);
--ff5: lpm_ff GENERIC MAP (7)
--PORT MAP (dff5,clockff5,enable2,aclrff1,qff5);

ff6: lpm_ff GENERIC MAP (1)--clock de salida
PORT MAP (dff6,clocksupermaxr,enable1,aclrff6,qff6);
--ff6: lpm_ff GENERIC MAP (1)--clock de salida
--PORT MAP (dff6,clockmaxr,enable1,aclrff6,qff6);
ff7: lpm_ff GENERIC MAP (1)--habilita clock de salida
PORT MAP (dff7,clockff7,enable1,aclrff2,qff7);
ff8: lpm_ff GENERIC MAP (1)--habilita clock de salida(evita glitch)
PORT MAP (dff8,clocksupermaxr,enable1,aclrff2,qff8);
ff9: lpm_ff GENERIC MAP (1)--salida(evita glitch)
PORT MAP (dff9,clocksupermaxr,enable1,aclrff2,qff9);
ff10: lpm_ff GENERIC MAP (1)--para cont3(evita glitch)
PORT MAP (dff10,clocksupermaxr,enable1,aclrff2,qff10);


mux1: lpm_mux GENERIC MAP (4,4,16)  -- tamaño entrada,selecion, entradas
PORT MAP (mux1in,mux1sel,mux1out);
mux2: lpm_mux GENERIC MAP (7,4,16)
PORT MAP (mux2in,mux2sel,mux2out);
mux3: lpm_mux GENERIC MAP (1,3,8)   -- saca salida en serie
PORT MAP (mux3in,mux3sel,mux3out);

generar : PROCESS

BEGIN
--cont1
clk_en1<= habil and not(sal1(4)and sal1(3)and sal1(2) and sal1(1)and sal1(0));

st<=sal1(4)and sal1(3)and sal1(2)and sal1(1)and sal1(0);

aclr1<=reset;

--cont2
clk_en2<=st;
aclr2<=qff3(0) or reset;
dff3(0)<=(sal2(0) and sal2(1) and sal2(2) and sal2(3));
```

```
--cont3
clk_en3<=st and (not din);
aclr3<=qff3(0) or reset;
dff10(0)<=(sal3(0) and sal3(1) and sal3(2) and sal3(3));

--mux
mux1in(0,0)<='1';mux1in(0,1)<='0';mux1in(0,2)<='0';mux1in(0,3)<='0';--'1'
mux1in(1,0)<='0';mux1in(1,1)<='1';mux1in(1,2)<='0';mux1in(1,3)<='0';--alfa
mux1in(2,0)<='0';mux1in(2,1)<='0';mux1in(2,2)<='1';mux1in(2,3)<='0';--alfa2
mux1in(3,0)<='0';mux1in(3,1)<='0';mux1in(3,2)<='0';mux1in(3,3)<='1';--alfa3
mux1in(4,0)<='1';mux1in(4,1)<='1';mux1in(4,2)<='0';mux1in(4,3)<='0';--alfa4
mux1in(5,0)<='0';mux1in(5,1)<='1';mux1in(5,2)<='1';mux1in(5,3)<='0';--alfa5
mux1in(6,0)<='0';mux1in(6,1)<='0';mux1in(6,2)<='1';mux1in(6,3)<='1';--alfa6

mux1in(7,0)<='1';mux1in(7,1)<='1';mux1in(7,2)<='0';mux1in(7,3)<='1';--alfa7
mux1in(8,0)<='1';mux1in(8,1)<='0';mux1in(8,2)<='1';mux1in(8,3)<='0';--alfa8
mux1in(9,0)<='0';mux1in(9,1)<='1';mux1in(9,2)<='0';mux1in(9,3)<='1';--alfa9
mux1in(10,0)<='1';mux1in(10,1)<='1';mux1in(10,2)<='1';mux1in(10,3)<='0';--alfa10
mux1in(11,0)<='0';mux1in(11,1)<='1';mux1in(11,2)<='1';mux1in(11,3)<='1';--alfa11
mux1in(12,0)<='1';mux1in(12,1)<='1';mux1in(12,2)<='1';mux1in(12,3)<='1';--alfa12
mux1in(13,0)<='1';mux1in(13,1)<='0';mux1in(13,2)<='1';mux1in(13,3)<='1';--alfa13
mux1in(14,0)<='1';mux1in(14,1)<='0';mux1in(14,2)<='0';mux1in(14,3)<='1';--alfa14

mux1in(15,0)<='0';mux1in(15,1)<='0';mux1in(15,2)<='0';mux1in(15,3)<='0';

mux1sel<=sal2;

mux2in(0,0)<='0';mux2in(0,1)<='0';mux2in(0,2)<='0';mux2in(0,3)<='0';mux2in(0,4)<='0';mux2in(0,
5)<='0';mux2in(0,6)<='0';
mux2in(1,0)<='1';mux2in(1,1)<='0';mux2in(1,2)<='0';mux2in(1,3)<='0';mux2in(1,4)<='0';mux2in(1,5
)<='0';mux2in(1,6)<='0';
mux2in(2,0)<='0';mux2in(2,1)<='1';mux2in(2,2)<='0';mux2in(2,3)<='0';mux2in(2,4)<='0';mux2in(2,5
)<='0';mux2in(2,6)<='0';
mux2in(3,0)<='0';mux2in(3,1)<='0';mux2in(3,2)<='0';mux2in(3,3)<='0';mux2in(3,4)<='1';mux2in(3,5
)<='0';mux2in(3,6)<='0';
mux2in(4,0)<='0';mux2in(4,1)<='0';mux2in(4,2)<='1';mux2in(4,3)<='0';mux2in(4,4)<='0';mux2in(4,5
)<='0';mux2in(4,6)<='0';
mux2in(5,0)<='0';mux2in(5,1)<='0';mux2in(5,2)<='0';mux2in(5,3)<='0';mux2in(5,4)<='0';mux2in(5,5
)<='0';mux2in(5,6)<='0';
mux2in(6,0)<='0';mux2in(6,1)<='0';mux2in(6,2)<='0';mux2in(6,3)<='0';mux2in(6,4)<='0';mux2in(6,5
)<='1';mux2in(6,6)<='0';
mux2in(7,0)<='0';mux2in(7,1)<='0';mux2in(7,2)<='0';mux2in(7,3)<='0';mux2in(7,4)<='0';mux2in(7,5
)<='0';mux2in(7,6)<='0';
mux2in(8,0)<='0';mux2in(8,1)<='0';mux2in(8,2)<='0';mux2in(8,3)<='1';mux2in(8,4)<='0';mux2in(8,5
)<='0';mux2in(8,6)<='0';
mux2in(9,0)<='0';mux2in(9,1)<='0';mux2in(9,2)<='0';mux2in(9,3)<='0';mux2in(9,4)<='0';mux2in(9,5
)<='0';mux2in(9,6)<='0';
mux2in(10,0)<='0';mux2in(10,1)<='0';mux2in(10,2)<='0';mux2in(10,3)<='0';mux2in(10,4)<='0';mux
2in(10,5)<='0';mux2in(10,6)<='0';
mux2in(11,0)<='0';mux2in(11,1)<='0';mux2in(11,2)<='0';mux2in(11,3)<='0';mux2in(11,4)<='0';mux
2in(11,5)<='0';mux2in(11,6)<='0';
mux2in(12,0)<='0';mux2in(12,1)<='0';mux2in(12,2)<='0';mux2in(12,3)<='0';mux2in(12,4)<='0';mux
2in(12,5)<='0';mux2in(12,6)<='1';
mux2in(13,0)<='0';mux2in(13,1)<='0';mux2in(13,2)<='0';mux2in(13,3)<='0';mux2in(13,4)<='0';mux
2in(13,5)<='0';mux2in(13,6)<='0';
mux2in(14,0)<='0';mux2in(14,1)<='0';mux2in(14,2)<='0';mux2in(14,3)<='0';mux2in(14,4)<='0';mux
2in(14,5)<='0';mux2in(14,6)<='0';
mux2in(15,0)<='0';mux2in(15,1)<='0';mux2in(15,2)<='0';mux2in(15,3)<='0';mux2in(15,4)<='0';mux
2in(15,5)<='0';mux2in(15,6)<='0';
mux2sel<=qff2;

mux3in(0,0)<=qff5(0);
mux3in(1,0)<=qff5(1);
mux3in(2,0)<=qff5(2);
mux3in(3,0)<=qff5(3);
mux3in(4,0)<=qff5(4);
```

```
mux3in(5,0)<=qff5(5);
mux3in(6,0)<=qff5(6);
mux3in(7,0)<=qff5(6);
mux3sel(0)<=sal2(1);
mux3sel(1)<=sal2(2);
mux3sel(2)<=sal2(3);

--ffs
dff1(0)<=qff1(0) xor (din and mux1out(0));
dff1(1)<=qff1(1) xor (din and mux1out(1));
dff1(2)<=qff1(2) xor (din and mux1out(2));
dff1(3)<=qff1(3) xor (din and mux1out(3));

enable1<='1';

dff2<=qff1;

clockff2<=not sal2(3);
clockff1<=not stposytot;
aclrff1<=qff3(0) or (not st);
aclrff2<=not st;
aclrff5<=(not st) or qff10(0);

dff4(0)<=qff4(1);
dff4(1)<=qff4(2);
dff4(2)<=qff4(3);
dff4(3)<=qff4(4);
dff4(4)<=qff4(5);
dff4(5)<=qff4(6);
dff4(6)<=din;
enable2<=not((sal2(0)and sal2(1)and sal2(2))or sal2(3));
aclrff4<='0';
clockff4<=not stposytot;

dff5(0)<=qff4(0)xor mux2out(0);
dff5(1)<=qff4(1)xor mux2out(1);
dff5(2)<=qff4(2)xor mux2out(2);
dff5(3)<=qff4(3)xor mux2out(3);
dff5(4)<=qff4(4)xor mux2out(4);
dff5(5)<=qff4(5)xor mux2out(5);
dff5(6)<=qff4(6)xor mux2out(6);
clockff5<=not sal2(3);

aclrff6<=sal2(1)and sal2(2)and sal2(3);
dff6(0)<=not sal2(0);

dff7(0)<='1';
clockff7<=qff8(0);

dff8(0)<=sal2(0)and sal2(1)and sal2(2)and sal2(3);

dff9(0)<=mux3out(0);
dout<=qff9(0);
clockout<=qff6(0) and qff7(0);
correc0 <=mux2out(0);
correc1 <=mux2out(1);
correc2 <=mux2out(2);
correc3 <=mux2out(3);
correc4 <=mux2out(4);
correc5 <=mux2out(5);
correc6 <=mux2out(6);
correcor<=mux2out(0)or mux2out(1)or mux2out(2)or mux2out(3)or mux2out(4)or mux2out(5)or
mux2out(6);
dinread<=qff4(6);
end process;

END arc;
```

Programa 10

```
ENTITY generador IS
          PORT
          (habil, clock         : IN STD_LOGIC;
          dout                  : OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
END generador;

ARCHITECTURE arc OF generador IS

COMPONENT lpm_counter
  GENERIC (LPM_WIDTH: POSITIVE);
 PORT ( clock: IN  STD_LOGIC;
       aclr: IN STD_LOGIC := '0';
       q: OUT STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0));
END COMPONENT;

signal sal1,qout: STD_LOGIC_VECTOR(7 DOWNTO 0);
signal aclr1: STD_LOGIC;

BEGIN

cont1: lpm_counter GENERIC MAP (7)
PORT MAP (clock,aclr1,sal1);
generar : process
BEGIN
aclr1<=not habil;
dout<=qout;
if habil= '1' then

case sal1 is
   when "0000001"=> qout <= M1;
   when "0000010"=> qout <= M2;
   when "0000011"=> qout <= M3;
   when "0000100"=> qout <= M4;
.
.
.
.
when "1000000"=> qout <= M64;
when others  =>  qout <= "00000000" ;
end case;

else
qout <= "0000000" ;

end if;
end process;

END arc;
```

Bibliografía consultada.

COMMUNICATION SYSTEMS- AN INTRODUCTION TO SIGNALS AND NOISE IN ELECTRICAL COMMUNICATION. Tercera Edicion (1986). A. Bruce Carlson. Mc. Graw Hill.

DIGITAL COMMUNICATIONS - FUNDAMENTALS AND APPLICATIONS. Bernard Sklar. Prentice Hall.

DIGITAL AND ANALOG COMMUNICATIONS SYSTEMS. Leon W. Couch. Mr. Millan Publishing Co. Inc.

COMMUNICATION SYSTEMS ENGINEERING. John G. Proakis, Masoud Salehi. Prentice Hall.

ERROR CONTROL CODING - FUNDAMENTALS AND APPLICATIONS. Shu Lin, Daniel J. Costello. Prentice Hall.