ESCUELA POLITÉCNICA NACIONAL

FACULTAD DE INGENIERÍA ELÉCTRICA Y ELECTRÓNICA

IMPLEMENTACIÓN DE CÓDIGOS DE LÍNEA EN UNA TARJETA DE ENTRENAMIENTO BASADA EN UN FPGA

PROYECTO PREVIO A LA OBTENCIÓN DEL TÍTULO DE INGENIERO EN ELECTRÓNICA Y TELECOMUNICACIONES

VIVIANA ELIZABETH CHECA ROMO
vivi_chk@yahoo.com

JOSÉ DANIEL VELÁSQUEZ CÓRDOVA

jdanivelasquez@hotmail.com

DIRECTOR: DR. ROBIN ÁLVAREZ RUEDA
robin.alvarez@epn.edu.ec

Quito, Marzo 2011

DECLARACIÓN

Nosotros, Viviana Elizabeth Checa Romo y José Daniel Velásquez Córdova, declaramos bajo juramento que el trabajo aquí descrito es de nuestra autoría; que no ha sido previamente presentada para ningún grado o calificación profesional; y, que hemos consultado las referencias bibliográficas que se incluyen en este documento.

A través de la presente declaración cedemos nuestros derechos de propiedad intelectual correspondientes a este trabajo, a la Escuela Politécnica Nacional, según lo establecido por la Ley de Propiedad Intelectual, por su Reglamento y por la normatividad institucional vigente.

Viviana Elizabeth Checa Romo

José Daniel Velásquez Córdova

CERTIFICACIÓN

Certifico que el presente trabajo fue desarrollado por Viviana Elizabeth Checa Romo y José Daniel Velásquez Córdova, bajo mi supervisión.

_____

Dr. Robin Álvarez Rueda

DIRECTOR DEL PROYECTO

AGRADECIMIENTO

A DIOS, por hacer de mi, una persona que aprende todos los días a vivir, sortear las adversidades, por ponerme a las personas indicadas en el camino, y salir adelante.

A mi madre Inés Esperanza Romo Jiménez, por su gran esfuerzo y dedicación a sus hijos, para hacerlos unas personas de bien y luchar por un futuro prometedor, y a mi hermano Bolívar Xavier Checa Romo, por ser mi gran amigo y apoyo en todas las situaciones de mi vida.

A mi director de tesis, PhD. Robin Álvarez, por darme la apertura de trabajar en su grupo, por compartir su conocimiento sin restricciones ni egoísmos, y a todos los profesores de la Facultad, que me han formado académicamente para desempeñarme en mi vida profesional.

A las personas que forman parte de la Academia ACIERTE, en especial a Santiago Yépes, quien me dio la oportunidad de integrar el grupo de becarios cisco, oportunidad que me sirvió para enriquecer mis conocimientos tecnológicos y cultivar valiosas amistades.

Viviana Checa R.

AGRADECIMIENTO

A Dios Todopoderoso por brindarme una vida llena de bendiciones, salud, la motivación, el esfuerzo y la dedicación diaria. Gracias por permitirme nacer en un hogar cristiano y conocer de su palabra. El simple hecho de darme la vida es digno de todo agradecimiento.

A mis padres: María Teresa y José Gilberto, por su gran sacrificio, sin ellos esto no habría sido posible. Gracias porque aunque no me han dado conocimiento científico, nunca me han hecho faltar nada; siempre se han preocupado por mí. Gracias por el ejemplo de hogar que me han dado durante toda mi existencia.

A mi familia, en especial a mis tíos: Enmita, Fabiolita y Carlitos, quienes siempre me han apoyado y han estado pendientes de lo que haga y deje de hacer. A la familia Ebenezer, por no permitir que me desvíe del camino recto y justo.

Al Ph.D. Robin Alvarez y a todos mis profesores de la Escuela Politécnica Nacional, por brindarme su apoyo y conocimiento. Gracias por la confianza que ha depositado en mí la EPN al permitirme ser instructor de varios laboratorios. A mis profesores de escuela, colegio y en especial al Dr. Ríber Donoso por sus consejos y ser el mentor de que estudie en la EPN.

A todos mis amigos, a quienes si los nombro faltaría espacio, pero que siempre se han preocupado por mi vida, gracias por el apoyo moral. Espero no defraudarles en el mundo exterior porque sé que esperan mucho de mi vida profesional.

Daniel Velásquez C.

DEDICATORIA

Este proyecto de titulación le dedico a mi madre Inés Romo, ya que es una mujer invaluable que admiro y es mi ejemplo a seguir, y a mi hermano, el Piloto de Aviación, Xavier Checa, una persona muy intelígente y capaz, que ha sabido alcanzar sus metas gracias a su entereza por salir adelante.

Viviana Checa R.

DEDICATORIA

Este trabajo está dedicado en primer lugar a mis padres: María Teresa y José Gilberto; a mi familia; a mis amigos; y finalmente a todos los jóvenes que se sienten atraídos por las áreas de Procesamiento Digital de Señales y FPGAs, espero sea de utilidad para ustedes.

Daniel Velásquez C.

ÍNDICE DE CONTENIDOS

CONTENIDO

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

RESUMEN

Este proyecto de titulación es un referente para el procesamiento digital de señales en FPGAs. Se describe la implementación con el lenguaje de programación VHDL de la codificación y decodificación con códigos de línea, en el FPGA de la tarjeta de entrenamiento Spartan-3E Starter Kit Board.

Este trabajo se divide en seis capítulos, además de los anexos.

En el capítulo 1 se realiza una introducción a los FPGAs (Field Programmable Gate Arrays) especificando su arquitectura y funcionamiento. Se describe los componentes de la tarjeta de entrenamiento Spartan-3E Starter Kit Board, en especial los utilizados en este proyecto. Además se proporciona un panorama de los lenguajes de programación utilizados para el manejo del FPGA, dándole prioridad a VHDL (VHSIC Hardware Description Language) ya que este lenguaje se utiliza en el presente trabajo. También se desarrolla una introducción a Xilinx ISE, utilizado para la programación, síntesis y simulación.

En el capítulo 2 se detalla cada uno de los Códigos de Línea implementados, entre ellos los códigos unipolares: NRZ Unipolar, RZ Unipolar al 50%, 4B5B; los códigos polares: Diferencial tipo M, Diferencial tipo S, Manchester o Bifase L, Manchester Diferencial y CMI; y finalmente los códigos bipolares: AMI, HDB3 y MLT3. Además se visualizan las gráficas de la codificación y decodificación realizada en el software MATLAB.

En el capítulo 3 se describe la elaboración de la interfaz gráfica en MATLAB, la misma que permite la introducción de letras que llegarán al FPGA en forma de bits por el puerto RS-232 de la Tarjeta de entrenamiento. En esta interfaz se ingresan los caracteres, la velocidad de transmisión, y el tipo de código de línea a implementarse. Posterior a la decodificación en el FPGA, la interfaz gráfica presenta los caracteres inicialmente enviados, la señal enviada, los caracteres recibidos y la señal recibida.

En el capítulo 4 se presenta el banco de trabajo necesario para la implementación del proyecto, se explica la implementación de los códigos de línea y se muestra la simulación en Testbench de cada uno de ellos. Para esto se realizan varias componentes independientes y procesos en VHDL descritos a través de diagramas de flujo ya que son segmentos secuenciales. También se detalla el diseño de un circuito interpretador unipolar-bipolar y un circuito interpretador bipolar-unipolar, ya que la Spartan-3E Starter Kit Board trabaja únicamente con voltajes positivos.

En el capítulo 5 se muestran los resultados obtenidos utilizando el osciloscopio y generador de funciones DS1M12.

En el capítulo 6 se presentan las conclusiones y recomendaciones del presente proyecto.

Al final del proyecto se anexan: los scripts desarrollados en MATLAB para la simulación de los códigos de línea, los códigos VHDL desarrollados para la implementación en el FPGA, el datasheet del amplificador operacional utilizado en los circuitos externos, el circuito impreso del hardware interpretador, y el manual de usuario.

PRESENTACIÓN

La lógica programable es una familia de componentes que contienen elementos lógicos (And, Or, Not) que pueden configurarse en cualquier forma que el usuario requiera. En años recientes, la línea entre hardware y software se está borrando. Lenguajes como Verilog o VHDL permiten implementar hardware descargado en FPGAs. Es por ello que la densidad promedio de los dispositivos lógicos programables ha comenzado a subir de una manera considerable.

Dentro de los dispositivos lógicos programables se tiene: SPLD (Dispositivo Lógico Programable Simple), CPLD (Dispositivo Lógico Programable Complejo) y FPGA.
Un FPGA (Field Programable Gate Array), tiene capacidad lógica muy alta debido a que posee un arreglo bidimensional de bloques lógicos comunicados por medio de una matriz de cables e interruptores cuya interconexión y funcionalidad son programables por los usuarios.

El presente proyecto de titulación tiene como objetivo principal la implementación de los Códigos de Línea en un FPGA. El hardware utilizado es la tarjeta de entrenamiento Spartan 3E Starter Kit Board, que contiene los periféricos necesarios para los propósitos del proyecto. La comunicación con el computador se la hace mediante comunicación serial RS-232, a una velocidad máxima de 19.2 Kbps. Para la programación de los algoritmos de codificación se han requerido los siguientes paquetes de software: para la simulación e interfaz gráfica MATLAB, y para la implementación en el hardware Xilinx ISE 10.1, utilizando el lenguaje de programación VHDL. Además, se vio necesario el diseño de un circuito externo que permita la interpretación de los códigos polares y bipolares, debido a las restricciones que presenta la tarjeta de entrenamiento empleada.

CAPÍTULO I

INTRODUCCIÓN A LOS FPGAs

1.1 INTRODUCCIÓN

Este capítulo se enfoca en el estudio de los FPGAs (Field Programmable Gate Arrays), su arquitectura y sus ventajas de utilización en relación a otros PLDs. Además se examina el software utilizado para la síntesis, simulación y programación del proyecto. De la misma manera se realiza un rápido resumen de las características de la tarjeta de entrenamiento que contiene el FPGA, describiendo sus elementos utilizados.

Posteriormente se proporciona un panorama de los lenguajes de programación utilizados para el manejo del FPGA, dándole prioridad a VHDL (VHSIC Hardware Description Language) ya que este lenguaje se utiliza en el presente proyecto.

El conocimiento sobre FPGAs, el software para su programación, la tarjeta de entrenamiento que contiene el FPGA y el lenguaje VHDL permitirán entender la implementación en forma global y minuciosa.

1.2 DISPOSITIVOS LÓGICOS PROGRAMABLES

Un dispositivo lógico programable o PLD, es un circuito integrado formado por un cierto número de compuertas lógicas y/o módulos básicos cuyas conexiones pueden ser personalizadas o programadas por el usuario final para construir circuitos digitales reconfigurables.

A diferencia de una compuerta lógica que tiene una función específica, un PLD no la tiene al momento de su fabricación. Antes de su uso debe ser programado.

1.2.1 ESTADO ACTUAL DE LA LÓGICA PROGRAMABLE

La lógica programable es una familia de componentes que contienen conjuntos de elementos lógicos (And, Or, Not) que pueden configurarse en cualquier forma que el usuario requiera.

En años recientes, la línea entre hardware y software se está borrando. Lenguajes como Verilog o VHDL permiten implementar hardware descargado en FPGAs. Es por ello que la densidad promedio de los dispositivos lógicos programables ha comenzado a subir considerablemente.

1.2.2 CLASIFICACIÓN

PLD	Nombre	Compuerta	Estructura
SPLD	Dispositivo Lógico Programable Simple	Hasta 2,5 K	PAL
CPLD	Dispositivo Lógico Programable Complejo	Hasta 15 K	PAL
FPGA	Matrices de Compuertas Programables en Campo	Hasta 8 M	Matriz de Bloques Lógicos

Tabla 1.1 Clasificación de los PLDs.

SPLD

Dispositivo lógico programable simple. Son los PLDs más simples, pequeños y baratos. El término SPLDs hace referencia a una variedad de dispositivos lógicos como: ROMs (Read Only Memory), PALs (Programable Array Logic), PLAs (Programable Logic Arrays), GALs (Generic Array Logic).

CPLD

Dispositivo Lógico Programable Complejo. Un CPLD es un circuito integrado equivalente a varias PALs enlazadas por interconexiones programables. Extiende el concepto de un PLD a un mayor nivel de integración ya que respecto a los SPLDs permite implementar sistemas con un mejor desempeño y reducen costos.

FPGA

Matrices de Compuertas Programables en Campo. Consiste en arreglos de varias celdas lógicas las cuales se comunican unas con otras mediante canales de conexión verticales y horizontales. A diferencia de otros PLDs, no está limitado a la típica matriz And-Or, sino tienen una matriz interna configurable de bloques lógicos (CLBs) y un anillo de bloques de entrada/salida (IOBs).

1.2.3 COMPARACIÓN ENTRE MICROPROCESADORES Y FPGAs

En general se puede establecer la siguiente Tabla de comparación:

Dispositivo	Microprocesador	FPGA
Tipo de Funciones	No diseñado para funciones particulares	No diseñado para funciones particulares
Velocidad y Potencia	Lento, de poca potencia	Rápido y eficiente en potencia
Programación	Reprogramable	Reprogramable

Tabla 1.2 Comparación entre FPGAs, Microprocesadores.

En el FPGA todos los procesos ocurren en paralelo, no concibe un sistema operativo que ejecute algoritmos complejos en tiempo real. Mientras los microprocesadores ejecutan un conjunto de instrucciones en una manera secuencial debido a que se basan en una arquitectura de CPU.

1.2.4 COMPARACION ENTRE CPLDs Y FPGAs.

Estos dispositivos lógicos programables son elaborados por los mismos fabricantes, pero difieren en:

Lógica

En un CPLD, las funciones se implementan utilizando lógica AND-OR de dos niveles. Mientras que en un FPGA, se las realizan utilizando múltiples niveles de lógica.

Interconexión

En un CPLD la interconexión es tipo crossbar, cada salida de LBs (Bloques Lógicos) es directamente interconectable a cada entrada a través de 1 o 2 interruptores. En cambio un FPGA tiene interconexión segmentada, las conexiones entre CLBs típicamente pasan a través de varios interruptores.

Funciones embebidas de alto nivel

El FPGA tiene funciones embebidas de alto nivel (como sumadores y multiplicadores) e incluso memorias embebidas. El CPLD no lo tiene.

Cantidad de Bloques Lógicos

El FPGA contiene mayor cantidad de bloques lógicos que el CPLD.

Funciones Aritméticas

El FPGA tiene recursos especiales de ruteo para implementar funciones aritméticas eficientes. El CPLD no lo tiene.

En general, los FPGAs pueden contener grandes diseños digitales, mientras que los CPLDs pueden contener únicamente pequeños diseños.

1.3 FPGA

Figura 1.1 FPGA Spartan de Xilinx.[1]

Field Programmable Gate Array (Matrices de Compuertas Programables en Campo). Es un PLD de propósito general, que tiene capacidad lógica muy alta debido a que posee un arreglo bidimensional de bloques lógicos comunicados por

[1]Figura tomada de “http://samkerr.wordpress.com/2009/09/21/first-steps-with-an-fpga/#more-207”

medio de una matriz de cables e interruptores cuya interconexión y funcionalidad son programables por los usuarios.
El tamaño, estructura, número de bloques, cantidad de conexiones y forma de conectividad difieren de una arquitectura a otra.

1.3.1 FAMILIA SPARTAN 3E

Son FPGAs de la compañía Xilinx, basados en tecnología SRAM, está fundamentada en arreglos de bloques lógicos de dos dimensiones que pueden ser interconectados por canales horizontales y verticales.

La familia de FPGAs Spartan 3E está diseñada para necesidades de alto volumen y sensibilidad al costo. Los 5 miembros de la familia ofrece rangos de densidad desde 100.000 a 1.6 millones de compuertas, como se muestra en la Tabla 1.3.

Dispositivo	Compuertas	Celdas Lógicas Equivalentes	Arreglo CLB			Bits RAM Distribuida	Bits Bloques RAM	Multiplicadores Dedicados	DCMs	Terminales I/O (Entrada/ Salida)	Pares de Terminales I/O (Entrada/ Salida) Diferenciales
			Filas	Columnas	Total CLBs						
XC3S100E	100K	2160	22	16	240	15K	72K	4	2	108	40
XC3S250E	250K	5508	34	26	612	38K	216K	12	4	172	68
XC3S500E	500K	10476	46	34	1164	73K	360K	20	4	232	92
XC3S1200E	1200K	19512	60	46	2168	136K	504K	28	8	304	124
XS3S1600E	1600K	33192	76	58	3688	231K	648K	36	8	376	156

Tabla 1.3 Resumen de Atributos de FPGAs Spartan 3E.[2]

1.3.1.1 Arquitectura de la Familia Spartan 3E

La arquitectura de los FPGAs de la familia Spartan 3E consta de 5 elementos fundamentales programables: CLBs, IOBs, Bloque RAM, Bloques Multiplicadores y DCMs.
Los elementos del FPGA están organizados como se muestra en la Figura 1.2. Un anillo de IOBs rodea un arreglo regular de CLBs. En particular el XC3S500E, tiene dos columnas de bloques RAM, cada columna consiste de 10 bloques RAM. Cada

[2]Tabla tomada de “Spartan-3E FPGA Family” de XILINX.

bloque RAM está asociado con un multiplicador dedicado. Los DCMs están posicionados en el centro, dos arriba y dos abajo del dispositivo.

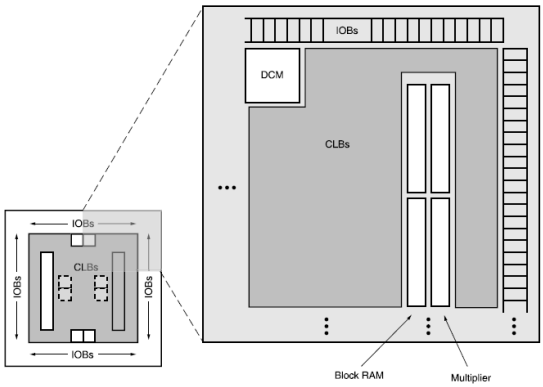


Figura 1.2 Arquitectura de la Familia Spartan-3E[3].

1.3.1.1.1 CLBs (Bloques Lógicos Configurables)

Desarrollan una gran variedad de funciones lógicas tanto como almacenaje de datos.

Las CLBs forman un arreglo de filas y columnas como se muestra en la Figura 1.3 y se enlazan vía una matriz de conmutación.

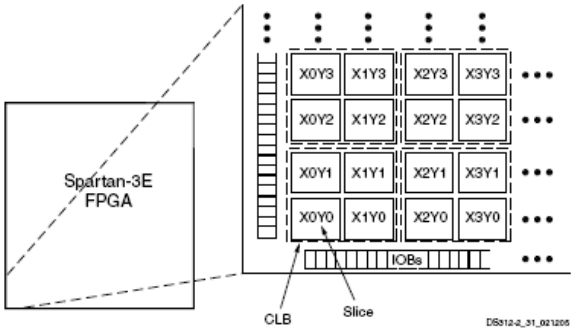


Figura 1.3 Ubicación de los CLBs.[4]

Cada CLB contiene 4 segmentos (slices) conectados entre sí. Cada segmento contiene 2 LUTs (Tablas de Consulta) flexibles, que implementan cualquier función booleana, y 2 elementos de almacenamiento utilizados como flip-flops o latches. Una LUT puede funcionar como una RAM distribuida porque logra almacenar 16 bits y puede ser usada como una memoria RAM.

[3]Figura tomada de “Spartan-3E FPGA Family” de XILINX

[4]Figura tomada de “Spartan-3E FPGA Family” de XILINX

1.3.1.1.2 IOBs (Bloques de Entrada/Salida)

Controlan el flujo de datos entre los terminales I/O y la lógica interna del dispositivo. Cada IOB soporta flujo de datos bidireccional y operación en 3 estados. Los IOBs están organizados en cuatro bancos I/O, como se muestra en la Figura 1.4.

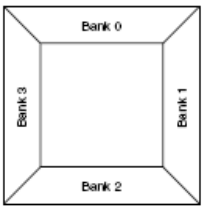


Figura 1.4 Bancos I/O de la Spartan-3E (Vista superior).[5]

1.3.1.1.3 Bloques RAM

Proveen almacenamiento de datos con una estructura de puerto dual de 18 Kbits por bloque. Los dos puertos de datos idénticos, llamados A y B, permiten acceder en forma independiente al bloque RAM. El bloque puede ser utilizado en modo de puerto simple o puerto doble.

1.3.1.1.4 Bloques Multiplicadores

Aceptan dos números binarios de 18 bits como entradas y permiten calcular el producto. Desarrollan multiplicaciones numéricas en complemento a 2 pero pueden también desarrollar almacenamiento de datos. Las entradas de 18 bits representan números desde -131072_{10} hasta $+131072_{10}$.

1.3.1.1.5 DCMs (Administradores de Reloj)

Estos bloques proveen control completo sobre: frecuencia del reloj (multiplica y/o divide la frecuencia de la señal de reloj con ciertos límites), desplazamiento en fase y delay skew. Los DCMs están rodeados de CLBs.

[5]Figura tomada de “Spartan-3E FPGA Family” de XILINX

- **Clocking Wizard**

  Para simplificar las aplicaciones utilizando los DCMs, el software de desarrollo Xilinx ISE incluye esta herramienta que proporciona instrucciones paso a paso para su configuración.

- **Sintetizador de Frecuencia (DFS)**

  Modifica la frecuencia del reloj basándose en un multiplicador y divisor, proporcionando un amplio y flexible rango de frecuencias de salida. Se puede multiplicar una señal de reloj de entrada, CLKIN, por la fracción (M/D), donde M = [2,32] $\varepsilon\, Z$ y D = [1,32] $\varepsilon\, Z$, resultando la señal de reloj CLKFX.

$$F_{CLKFX} = F_{CLKIN} \frac{Multiplicador\_CLKFX}{Divisor\_CLKFX}$$

1.3.1.1.6 Interconexión

La familia Spartan 3E tiene una gran red programable de rutas entre la entrada y salida de los elementos funcionales del FPGA, como los IOBs, CLBs, DCMs y bloques RAM, transmitiéndose señales a través de ellos.

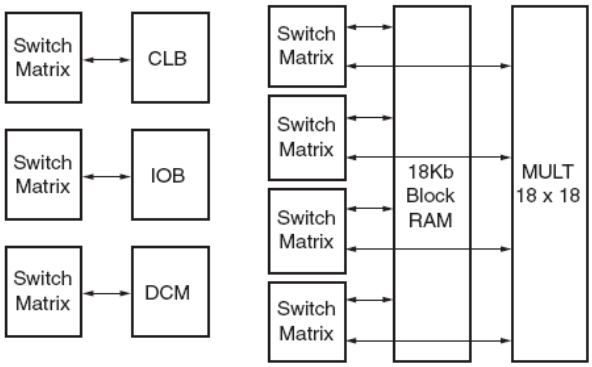


Figura 1.5 Cuatro tipos de Interconexión (CLB, IOB, DCM y Bloque RAM/Multiplicadores).[6]

[6] Figura tomada de "Spartan-3E FPGA Family" de XILINX

1.3.2 VENTAJAS DE USAR FPGAs

Entre las ventajas más destacadas se pueden mencionar:

- Los FPGAs permiten construir bloques que admiten una buena personalización del hardware, es decir incluyen la facilidad de reprogramar en campo para solucionar problemas de implementación sin necesidad de reemplazar el hardware.

- Actualmente los FPGAs se presentan en tamaños utilizables a un precio admisible. Esto los hace factores efectivos para disminuir costos cuando se realizan configuraciones individuales.

- La tecnología FPGA es admisible donde la disponibilidad a largo plazo o ambientes industriales severos están involucrados.

- Su velocidad de procesamiento y flexibilidad, por lo que el uso de FPGAs para la implementación de sistemas electrónicos cada vez es más demandante.

- Verificación efectiva del diseño mediante simuladores o en el chip.

- Densidad de integración amplia (Hasta 8 millones de compuertas).

1.3.3 APLICACIONES DE LOS FPGAs

Los FPGAs han ganado aceptación y han crecido rápidamente esta década debido a que pueden ser utilizados en un amplio rango de aplicaciones.
Los FPGAs, facilitan el desarrollo e implementación de sistemas digitales complejos. Son dispositivos que combinan muchas de las novedades en el diseño de circuitos integrados para la implementación de sistemas digitales, y brindan la posibilidad de ajustarse a necesidades individuales, definidas por el usuario.

Las aplicaciones típicas son:

- Lógica aleatoria.
- Dispositivos controladores.
- Procesamiento Digital de Señales.
- Procesamiento de imagen, audio y video.
- Soluciones militares.
- Telecomunicaciones.
- Redes.
- Sistemas con bloques SRAM.
- Supercomputadoras.
- Emulación de hardware de computadora.

Las nuevas generaciones de FPGAs pueden implementar sistemas complejos en dispositivos simples.

1.4 SPARTAN-3E STARTER KIT BOARD

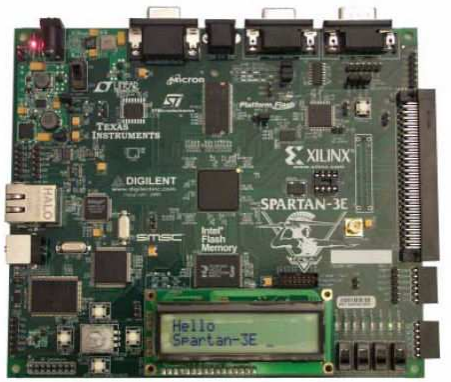


Figura 1.6 Spartan-3E Starter Kit Board.[7]

Spartan-3E Starter Kit Board es una tarjeta de entrenamiento fabricada y distribuida por Digilent Inc. En el mercado se ofertan diferentes tarjetas de evaluación y desarrollo, las cuales incluyen una diversidad de periféricos para aumentar su versatilidad.

[7] Figura tomada de "Spartan-3E Starter Kit Board User Guide"

Los principales componentes de los cuales dispone esta tarjeta son:

- FPGA Spartan-3E XC3S500E de Xilinx.
- Flash PROM de 4 Mb de Xilinx.
- CPLD CoolRunner XC2C64A de Xilinx.
- DDR SDRAM de 64 MB, interfaz x16, 100+ MHz.
- Memoria Flash de tipo NOR de 16 MB (128 Mb) para aplicaciones.
- Memoria Flash de 16 Mb, vía SPI.
- Pantalla LCD de 2 líneas de 16 caracteres.
- Puerto PS/2.
- Puerto VGA.
- Capa física Ethernet 10/100.
- 2 puertos RS-232 de 9 terminales(DTE y DCE).
- Puerto USB para descarga y depuración.
- Oscilador de 50 MHz.
- Conector de expansión Hirose FX2.
- 3 Conectores de expansión Digilent de 6 terminales.
- Conversor Digital a Analógico SPI de cuatro salidas (DAC).
- Conversor Analógico a Digital SPI de dos entradas (ADC) con pre amplificador de ganancia programable.
- Botón rotatorio.
- 8 LEDs.
- 4 Interruptores Deslizantes.
- Conector SMA para ingreso de Señal de Reloj.

A continuación se describen los periféricos que se utilizan para la implementación de este proyecto.

1.4.1 INTERRUPTORES DESLIZANTES

La Spartan-3E Starter Kit Board tiene 4 interruptores deslizantes, los cuales se muestran en la Figura 1.7. Están etiquetados de SW3 a SW0.

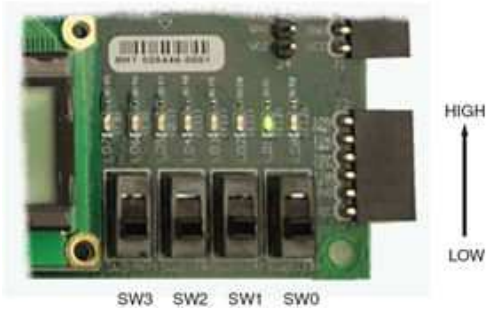


Figura 1.7. Cuatro Interruptores Deslizantes.[8]

La asignación de terminales para los 4 interruptores deslizantes es la siguiente:

SW0	L13
SW1	L14
SW2	H18
SW3	N17

1.4.2 FUENTES DE RELOJ

La Spartan-3E Starter Kit Board dispone de 3 entradas para señal de reloj, las cuales se muestran en la Figura 1.8 y se describen a continuación:

- Una señal de reloj la puede proveer el oscilador de 50 MHz incluido en la tarjeta, que tiene una exactitud de ±2500 Hz o ±50 ppm. El duty cycle de este oscilador se puede configurar en un rango entre el 40% a 60%. Electivamente, se utiliza el DCM (Digital Clock Manager) del FPGA para generar otras frecuencias con este oscilador.
- La señal de reloj también puede ser provista desde fuera de la tarjeta vía un conector SMA. También el FPGA puede generar señales de reloj o señales de alta velocidad que utilicen como puerto de salida el conector SMA.
- Además la tarjeta suministra un socket para incorporar un oscilador tipo DIP de 8 terminales. Se lo puede utilizar en aplicaciones que requieren una frecuencia diferente a 50 MHz.

[8] Figura tomada de “Spartan-3E Starter Kit Board User Guide”

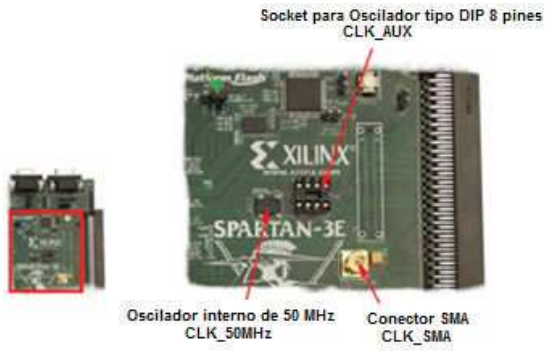


Figura 1.8 Entradas para Reloj disponibles.[9]

Asignación de terminales de las 3 fuentes de reloj:

CLK_50MHz	C9
CLK_SMA	A10
CLK_AUX	B8

1.4.3 PUERTOS SERIALES RS-232

La Spartan 3E Starter Kit Board tiene dos conectores seriales RS-232, un DB9 hembra para que la tarjeta funcione como DCE y un DB9 macho para que funcione como DTE.

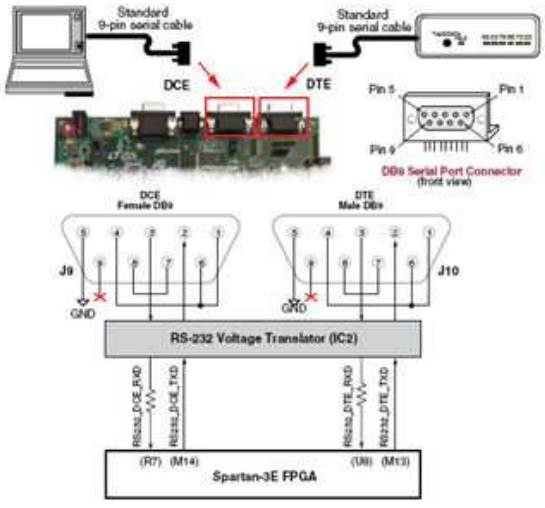


Figura 1.9 Puertos Serial RS-232.[10]

[9] Figura tomada de "Spartan-3E Starter Kit Board User Guide"

[10] Figura tomada de "Spartan-3E Starter Kit Board User Guide"

El Puerto DCE se puede conectar directamente al puerto serial de la computadora a través de un cable serial directo, no se requiere cables cruzados o Null modem. El puerto DTE se utiliza para conectar otros dispositivos periféricos RS-232.

La Figura 1.9 muestra la conexión entre el FPGA y los dos conectores DB9. El FPGA proporciona salida de datos seriales usando niveles LVTTL o LVCMOS al dispositivo Maxim embebido en la tarjeta, el cual convierte estos valores a los niveles de voltaje RS-232 apropiados. De la misma manera el Maxim convierte los datos de entrada RS-232, a niveles LVTTL para que ingresen al FPGA.

Estos puertos seriales no soportan Control de Flujo debido a que los terminales DCD (Data Carrier Detect), DTR (Data Terminal Ready) y DSR (Data Set Ready) se conectan entre ellos, como se muestra en la Figura 1.9. De manera similar los terminales RTS y CTS están interconectados.

Asignación de terminales del puerto serial RS-232 DTE:

DTE_RXD	U8
DTE_TXD	M13

Asignación de terminales del puerto serial RS-232 DCE:

DCE_RXD	R7
DCE_TXD	M14

1.4.4 CONECTORES DE EXPANSIÓN

La Spartan-3E Starter Kit Board provee una variedad de conectores de expansión para tener flexibilidad de conexión con otros componentes fuera de la tarjeta.

1.4.4.1 Conectores de Seis Terminales

La tarjeta tiene tres conectores periféricos de 6 terminales de entrada y salida. La ubicación de los conectores se observa en la Figura 1.10.

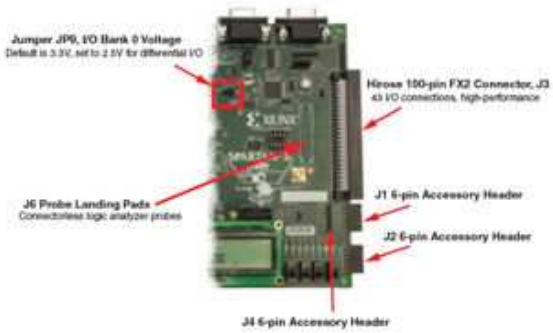


Figura 1.10 Conectores de Expansión.[11]

Los conectores J1 y J2, usan un socket de 6 terminales cada una, cuatro de los cuales conectan al FPGA, un terminal a tierra y el último terminal provee 3.3 V, tal como se muestra en la Figura 1.11.

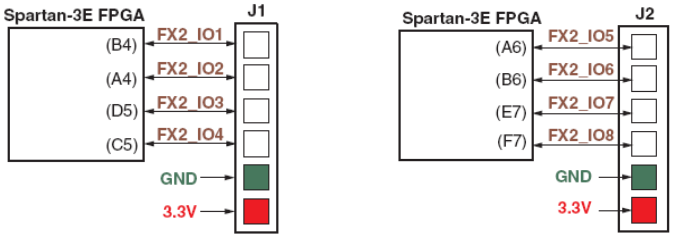


Figura 1.11 Conexiones del FPGA a las Conectores J1 y J2.[12]

Asignación de terminales de los conectores J1 y J2:

J1<0>	B4
J1<1>	A4
J1<2>	D5
J1<3>	C5

J2<0>	A6
J2<1>	B6
J2<2>	E7
J2<3>	F7

[11] Figura tomada de “Spartan-3E Starter Kit Board User Guide”

[12] Figura tomada de “Spartan-3E Starter Kit Board User Guide”

1.4.5 PROGRAMACIÓN DEL FPGA

En este proyecto, para programar el FPGA se descarga el diseño directamente al dispositivo de la Spartan-3E Starter Kit Board, vía JTAG (Joint Test Action Group), usando el interfaz USB de la tarjeta. El interfaz USB sirve solo para programación, mas no para transmisión-recepción de datos.

La Figura 1.12 indica la posición de la interfaz USB de programación en la tarjeta.

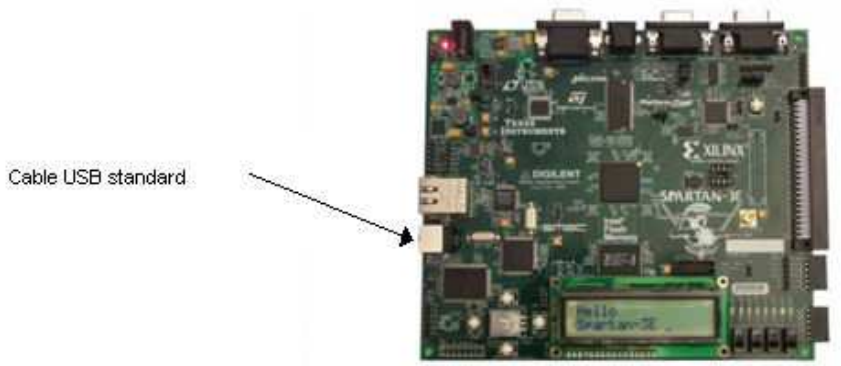


Figura 1.12 Programación de la Spartan-3E Starter Kit Board.[13]

Con el cable USB conectado al PC se puede programar el FPGA, la Flash PROM y el CPLD XC2C64A embebidos en la tarjeta, mediante el software Xilinx ISE.

Cuando se enciende la tarjeta y el cable USB está conectado, el sistema operativo de Windows debe reconocer e instalar los drivers asociados.

1.5 XILINX ISE

Xilinx ISE (Integrated Software Enviroment), es un conjunto de herramientas de desarrollo elaborado por Xilinx para facilitar la creación de diseños.

Una de las versiones de ISE es ISE WebPACK. ISE WebPACK es el software de desarrollo más fácil de obtener, se puede descargar de la página electrónica de Xilinx.

[13] Figura tomada de "Spartan-3E Starter Kit Board User Guide"

Project Navigator, una herramienta de Xilinx ISE, es el interfaz de usuario que ayuda a administrar el proceso de diseño en un FPGA, incluyendo la creación del modelo, síntesis, simulación y programación del dispositivo.

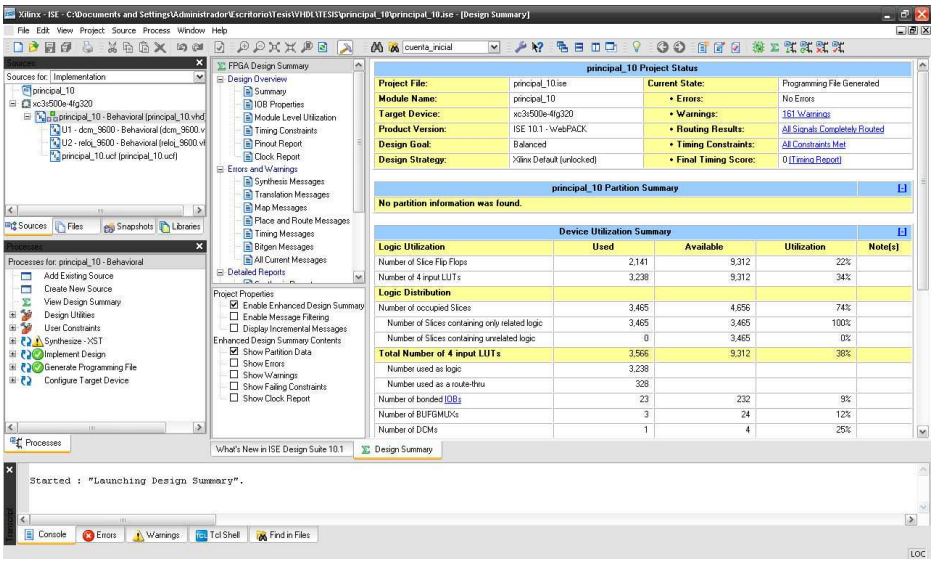


Figura 1.13 Ventana Principal del Project Navigator.

1.5.1 FLUJO DE DISEÑO SOBRE UN FPGA

El flujo de diseño de un módulo hardware sobre un FPGA comprende los siguientes pasos: creación del modelo, síntesis del diseño, implementación del diseño y programación del dispositivo Xilinx. Para el flujo de diseño que se describe se utiliza el software Xilinx ISE. En cada una de estas etapas se puede verificar la validez del diseño.

Para completar el proceso de diseño en un FPGA de la familia Xilinx 3E se necesita de los siguientes recursos:

- ✓ Computador, con Windows XP y Service Pack 3 para versiones anteriores o iguales a Xilinx 10.1. O Windows Vista para versiones superiores.
- ✓ Software Xilinx ISE
- ✓ Tarjeta de Entrenamiento con un FPGA de la familia Spartan 3E.

La Figura 1.14 muestra una versión simplificada del flujo de diseño sobre un FPGA.

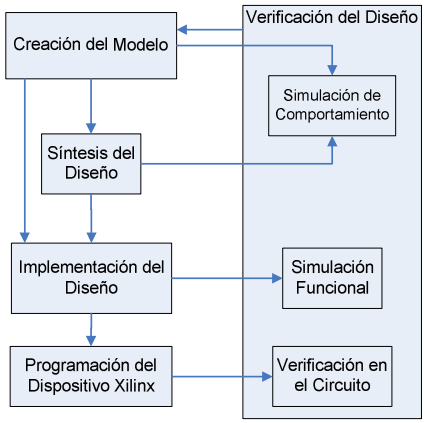


Figura 1.14 Flujo de Diseño FPGA.[14]

1.5.1.1 Creación del Modelo

Se puede Crear el Modelo basándose en: un Esquema, lenguaje de alto nivel HDL y una combinación de los dos.

Cuando el diseño es complejo o el diseñador piensa que éste puede ser resuelto con un algoritmo, entonces HDL es la mejor elección ya que puede aislar a dicho diseño de los detalles del hardware a implementar.

1.5.1.1.1 Crear un proyecto

El procedimiento en el Project Navigator de Xilinx ISE 10.1 implica escoger: Un nombre para el Proyecto, un directorio donde guardar, tipo de fuente: HDL, tal como lo muestra la Figura 1.15.

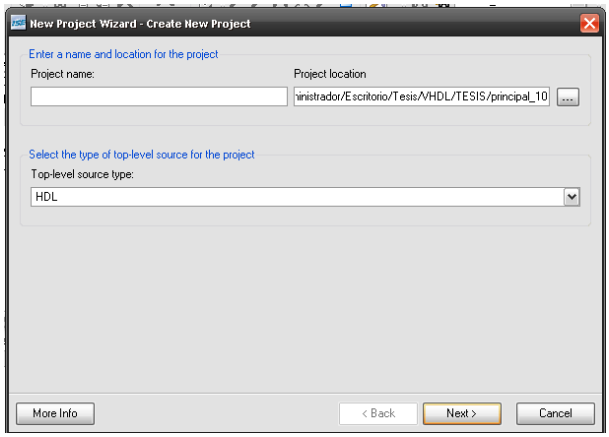


Figura 1.15 Creación de un Nuevo Proyecto en Xilinx ISE.

[14] Figura tomada de "http://www.xilinx.com/itp/xilinx8/help/iseguide/html/ise_fpga_design_flow_overview.htm"

Sobre las Propiedades del dispositivo, se debe elegir: Familia del dispositivo, Dispositivo, Paquete y Velocidad. A cerca del Diseño, escoger: Herramienta de Síntesis, Simulador y Lenguaje de Programación, tal como lo muestra la Figura 1.16.

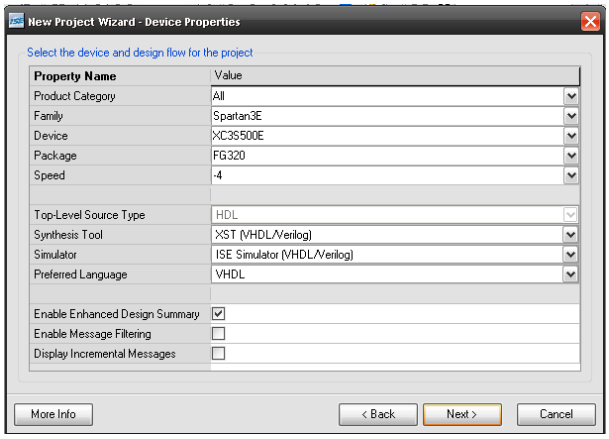


Figura 1.16 Selección de Propiedades del Dispositivo.

Posteriormente se podría crear archivos y añadirlos al proyecto. Finalmente una ventana nos presenta un resumen de las especificaciones seleccionadas.

1.5.1.1.2 Crear archivos y añadirlos al proyecto

Para crear nuevos archivos se selecciona New Source en Sources en la ventana principal del Project Navigator. Primordialmente se pueden crear archivos: UCF utilizados para Asignación de terminales, Módulo VHDL para programar en este lenguaje, y Test Bench Waveform para simulaciones de comportamiento.

1.5.1.2 Síntesis del Diseño

El sintetizador de Xilinx, XST, traduce código VHDL o Verilog en un diseño electrónico, por ejemplo en un circuito completo con elementos lógicos. Este diseño se guarda en un archivo NGC (Native Generic Circuit).

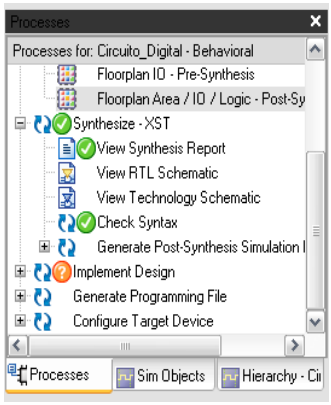


Figura 1.17 Síntesis del Diseño en la Ventana Procesos.

1.5.1.3 Implementación del Diseño

Traduce el diseño electrónico en un diseño que puede ser implementado en los elementos funcionales del FPGA.

Consiste en una secuencia de 3 pasos:

- Traducción, en esta fase el archivo NGC se convierte en un archivo NGD (Native Generic Database). El archivo NGD fusiona el diseño electrónico con las restricciones y describe el diseño lógico en términos primarios para Xilinx.

- Mapeo, todo el circuito lógico del archivo NGD es dividido en bloques que pueden ser incorporados en los elementos funcionales del FPGA, esta información se guarda en un archivo NCD (Native Circuit Description) que representa el diseño que se incorporará a los componentes del FPGA.

- Ubicación y Enrutamiento, este proceso coloca los bloques que se generaron en el anterior paso (Mapeo) en CLBs cerca a los terminales de entrada/salida que se utilizarán y los interconecta. Esta información se guarda en un nuevo NCD que contiene la información de rutas.

1.5.1.4 Programación del Dispositivo Xilinx

Una vez implementado el diseño, se debe generar el archivo de configuración (.bit) para cargarlo en el FPGA. Este archivo configura cada parte del dispositivo, para que los requerimientos del diseño sean implementados.
iMPACT es la herramienta de Xilinx ISE que permite a los usuarios cargar el archivo de configuración sobre los FPGAs de Xilinx a través del modo JTAG, también llamado Modo Boundary Scan. iMPACT detecta automáticamente el FPGA a través del cable de conexión antes de programarlo.

Procedimiento

Para programar el FPGA en Project Navigator de Xilinx ISE, en la Ventana Procesos, en el submenú Configurar Dispositivo Previsto, se hace doble click en Administrar Configuración del Proyecto (iMPACT), como lo muestra la Figura 1.18. Esto abre la herramienta iMPACT, donde seleccionamos Configurar dispositivo usando Boundary Scan, JTAG.

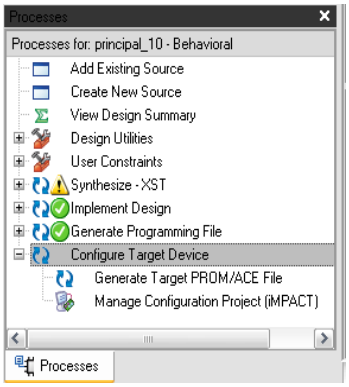


Figura 1.18 Selección de la herramienta iMPACT en la Ventana Procesos.

A continuación se abre la ventana de Asignación de nuevo archivo de configuración. Escogemos el archivo .bit que se va a programar en el FPGA, tal como lo muestra la Figura 1.19.

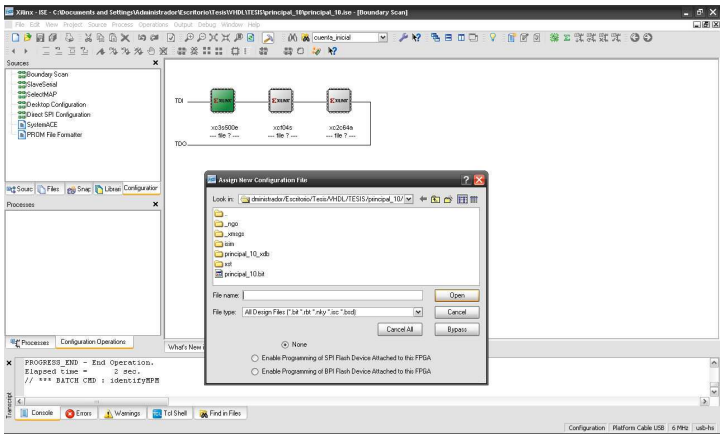


Figura 1.19 Asignación del archivo de configuración .bit

Al utilizar la Spartan-3E Starter Kit Board, en los dos cuadros de diálogo que son utilizados para programar la Flash PROM y el CPLD, seleccionamos Bypass cuando no se programa esta parte.

Finalmente con un click derecho en el dispositivo xc3s500e, correspondiente al FPGA, seleccionamos Programar, como lo muestra la Figura 1.20.

Figura 1.20 Programación del FPGA.

Xilinx ISE se tomará un momento para programar el FPGA, proceso que si es exitoso retornará con el mensaje “Programación Exitosa”.

1.5.1.5 Verificación Funcional

Se puede verificar la funcionalidad del proyecto en diferentes puntos del flujo de diseño, entre ellos:

- Antes de la síntesis mediante una Simulación de comportamiento.
- Después de la traducción a través de una Simulación funcional (conocida como simulación de nivel de compuerta lógicas).
- Después de la programación del dispositivo, verificación del diseño funcionando en el FPGA.

1.6 HDL (LENGUAJE DE DESCRIPCIÓN DE HARDWARE)

Un diseño puede ser implementado usando diagramas de bloques, que no son prácticos en diseños complejos; o haciendo uso de HDL (Hardware Description Language) que define las funciones de los CLBs y las conexiones entre ellos.

HDL, es un lenguaje de programación que se basa en expresiones de texto para descripción formal de sistemas digitales. Facilita el diseño, organización e implementación de circuitos, que se pueden verificar a través de simulaciones.

En contraste con la mayoría de lenguajes de programación, HDL incluye: una noción explícita de tiempo (formas de describir la propagación del tiempo) y dependencia de la señal. Además que es un lenguaje para síntesis y simulación.

Actualmente hay dos Lenguajes de Descripción de Hardware estandarizados y distinguidos: Verilog y VHDL.

1.6.1 VERILOG

Verilog es el HDL más comúnmente utilizado en el diseño, implementación y verificación de circuitos digitales, además que es el más fácil de aprender. Fue originalmente estandarizado en el IEEE Standard 1364-1995 y posteriormente revisado en el IEEE Standard 1364-2001.

Verilog describe un sistema digital como un conjunto de módulos. Cada uno de estos módulos tiene un interfaz a otros módulos.

Tuvo gran aceptación puesto que la sintaxis es parecida a la del lenguaje de programación C.

1.6.2 VHDL

VHDL significa Very high speed integrated circuit Hardware Description Language y es un HDL. La primera versión se llamó VHDL 87, y fue el primer HDL en ser

estandarizado por IEEE a través del estándar IEEE 1076, el mismo que se complementó posteriormente con el estándar IEEE 1164.

A pesar de que VHDL se puede simular totalmente, no todas las construcciones son sintetizables.
El código VHDL se guarda en un archivo con el mismo nombre de la entidad (Elemento de su estructura) y extensión vhd.
Este código puede ser concurrente (paralelo) o secuencial. Sólo las instrucciones ubicadas dentro de procesos, funciones o procedimientos son ejecutadas secuencialmente. Aunque la ejecución al interior de estos bloques es secuencial, el bloque en si es concurrente con las instrucciones externas del bloque.

Para realizar un código en VHDL es necesario al menos conocer: su estructura, cómo utilizar cada uno de los elementos de dicha estructura (Declaración de librerías, Entidad y Arquitectura), los operadores y atributos que se pueden utilizar.

1.6.2.1 Estructura del Código

Un código VHDL autónomo está compuesto de 3 partes fundamentales: Declaración de Librerías, Entidad y Arquitectura

1.6.2.1.1 Declaración de Librerías

Es una colección de partes de código usadas comúnmente. En la declaración se deben incluir todas las librerías que van a ser utilizadas en el diseño.

La sintaxis de la librería es de la siguiente manera:

```
LIBRARY nombre_libreria;
USE nombre_libreria.nombre_paquete.parte_paquete;
```

1.6.2.1.2 Entidad

Especifica todos los puertos de entrada y salida. Su sintaxis se muestra a continuación:

```
ENTITY nombre_entidad IS
        PORT (
                Nombre_puerto: modo_señal tipo_señal;  );
END nombre_entidad;
```

Todos los puertos deben tener: Nombre, Modo de la señal (la dirección en la que van los datos, entrada o salida) y Tipo de señal.

Modo de la señal

Puede ser: Modo in (representa señales de entrada, usado para reloj, entradas de control y datos), Modo out (representa señales de salida), Modo buffer (representa señales de realimentación interna), Modo inout (representa señales bidireccionales, de entrada y salida).

Tipo de señal

Los tipos de señal se encuentran en diferentes paquetes y librerías, los más usados son: Bit (puede tomar valores lógicos de 0L o 1L), Bit_vector (Vector de bits, representa un conjunto de bits). Integer (representa un número entero), Natural (representa enteros no negativos), Real (representa números reales, no son datos sintetizables), Std_logic (puede tomar 8 valores diferentes, 4 de ellos sintetizables: Desconocido, 0L, 1L y de alta impedancia), Std_logic_vector (representa un vector de elementos std_logic).

1.6.2.1.3 Arquitectura

Contiene el código que describe el comportamiento del circuito. Su sintaxis es la siguiente:

```
ARCHITECTURE nombre_arquitectura OF nombre_entidad IS
        [declaraciones]
BEGIN
        (código)
END nombre_arquitectura;
```

La arquitectura tiene dos partes: la declarativa (que se utiliza sólo cuando se requiera declarar señales y constantes internas a la arquitectura), y la del código. En la arquitectura se describen las operaciones que se realizan con los puertos de entrada para obtener los puertos de salida descritos en la entidad.

1.6.2.2 Operadores y Atributos

Son símbolos que indican que se deben llevar a cabo operaciones específicas, de los cuales se debe conocer antes de empezar a realizar códigos en VHDL.

1.6.2.2.1 Operadores

Los operadores predefinidos más utilizados, son:

Operadores de Asignación

Entre ellos: **<=** (asigna un valor a una señal), **:=** (asigna un valor a una variable o constante, también es usado para asignar valores iniciales); **=>** (asigna valores a elementos individuales de un vector).

Operadores Lógicos

Realizan operaciones lógicas, son: not, and, or, nand, nor, xor, xnor.

Operadores Aritméticos

Realizan operaciones aritméticas. Los sintetizables son: **+** (Suma), **-** (Resta), ***** (Multiplicación), **/** (División) y ****** (Exponenciación).

Operadores de Comparación

Realizan comparaciones. Son: **=** (Igual a), **/=** (No igual a), **<** (Menor que), **>** (Mayor que), **<=** (Menor o igual que), **>=** (Mayor o igual que).

1.6.2.2.2 Atributos

Los atributos sintetizables más utilizados son:

Atributos de Datos

Considerando un arreglo d, son: **d'low** (retorna el índice más bajo del arreglo d), **d'high** (retorna el índice más alto del arreglo d), **d'left** (retorna el índice de la izquierda del arreglo d), **d'right** (retorna el índice de la derecha del arreglo d), **d'length** (retorna el tamaño del vector).

Atributos de Señales

Considerando una señal x, son: **x'event** (retorna un valor verdadero cuando ocurre un cambio en x; por ejemplo si la señal reloj va a ser monitoreada, reloj'event retorna verdadero cada vez que sucede una transición), **x'stable** (retorna un valor verdadero cuando no cambia x).

1.6.2.3 Código Concurrente

Se puede construir código que se ejecute en forma paralela realizando asignaciones que utilicen únicamente operadores (lógicos, aritméticos, de comparación), además existen instrucciones concurrentes como: *When* y *Generate*. Una instrucción concurrente especial es *Block* que simplemente sirve para agrupar instrucciones concurrentes en bloques, para hacerlo entendible.

1.6.2.3.1 Componentes

Una componente es un simple segmento de código convencional (contiene declaración de librerías, entidad y arquitectura). Sin embargo el código que se

encuentra en una componente puede ser utilizado por otro código concurrente. Esto permite la construcción del diseño jerárquico.

A una componente se la considera una forma de segmentar el código y hacerlo reutilizable. La sintaxis para declarar una componente dentro de la parte declarativa de la arquitectura del programa principal se muestra a continuación:

```
COMPONENT nombre_componente IS
        PORT (
                nombre_puerto : modo_señal tipo_señal;
                nombre_puerto2 : modo_señal2 tipo_señal2;
                …);
END COMPONENT;
```

La sintaxis para utilizar una componente, en la parte del código de la arquitectura del programa principal se muestra a continuación:

```
Etiqueta: nombre_componente PORT MAP ( lista_de_puertos);
```

La sintaxis de la declaración de la componente es similar a la sintaxis de la entidad.

1.6.2.4 Código Secuencial

También llamado código de comportamiento. Las secciones de código que se ejecutan en forma secuencial son: procesos, funciones y procedimientos. Sin embargo, cada uno de estos bloques es concurrente con el resto del código.

Internamente en estas secciones se pueden utilizar variables. Las variables no son globales y no se pueden pasar directamente a otras secciones del código.

1.6.2.4.1 Procesos

Un proceso es una sección secuencial. Se localiza en el código principal y se ejecuta cada vez que una señal de la lista de sensibilidad cambie. Su sintaxis es la siguiente:

```
[etiqueta:] PROCESS (lista_sensibilidad_señales)
        [VARIABLE nombre tipo [rango] [:= valor_inicial;] ]
BEGIN
        (código_secuencial)
END PROCESS [etiqueta];
```

La etiqueta es opcional y se utiliza para identificar un proceso de los otros. Las variables, al igual que el rango en el que se deben encontrar y su valor inicial también es opcional, y deben ser declaradas en la parte declarativa del proceso.

En la parte principal del proceso se pueden utilizar instrucciones populares, como: *IF, WAIT, CASE, LOOP.*

1.6.2.4.2 Funciones y Procedimientos

Las funciones y procedimientos son colectivamente llamados subprogramas. Son muy similares a los procesos en su construcción ya que también emplean las instrucciones: *If, Case* y *Loop*.
Sin embargo desde el punto de vista de la aplicación, hay una diferencia fundamental. A pesar de que las funciones y procedimientos se pueden guardar en el código principal al igual que los procesos, se los almacenan en paquetes guardados en librerías siendo así que pueden ser reutilizados y compartidos por otros proyectos.

Respecto al número de señales de salida, en las funciones se puede obtener una sola variable o señal de salida, mientras en los procedimientos se obtienen varias variables y/o señales de salida, pero todas relacionadas con los puertos de la entidad del código principal.

1.7 EJEMPLOS DE IMPLEMENTACIÓN EN UN FPGA A TRAVÉS DE VHDL

Con el propósito de ilustrar el flujo de diseño sobre un FPGA y el lenguaje de programación VHDL, se realizan dos ejemplos de implementación.

1.7.1 IMPLEMENTACIÓN DE UN CIRCUITO DIGITAL EN UN FPGA USANDO ÚNICAMENTE EL CÓDIGO PRINCIPAL DE VHDL

En este ejemplo se implementa el circuito digital de la Figura 1.21 en la Spartan-3E Starter Kit Board.

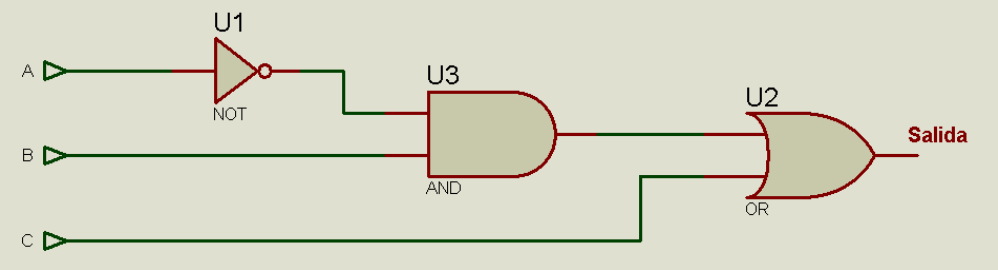


Figura 1.21 Circuito Digital a implementarse en el FPGA.

Flujo de Diseño sobre el FPGA

En la creación del modelo se seleccionan varias características del proyecto y el dispositivo a utilizarse, tal como se describe en la sección 1.5.1.1. En este ejemplo en particular, estas especificaciones se resumen en la Tabla 1.4.

Proyecto	
Nombre del proyecto:	Circuito_Digital
Tipo de fuente:	HDL
Dispositivo	
Familia del FPGA:	Spartan3E
Dispositivo:	XC3S500E
Paquete:	FG320
Velocidad:	-4
Herramienta de Síntesis:	XST
Simulador:	ISE Simulator
Lenguaje preferido:	VHDL

Tabla 1.4 Características del proyecto y dispositivo.

El módulo VHDL añadido a este proyecto, Circuito_Digital.vhd, se muestra en la Figura 1.22.

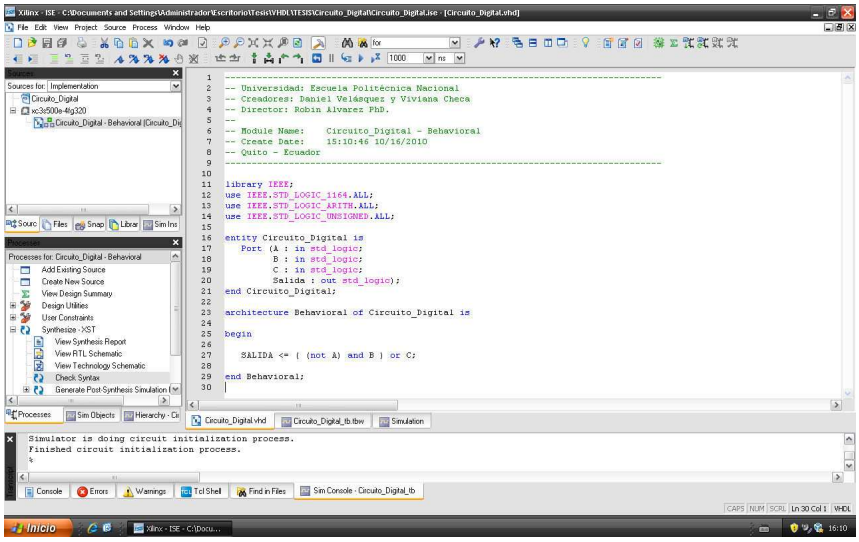


Figura 1.22 Partes fundamentales del código de Circuito_Digital.vhd

A cerca de la estructura del código VHDL, en la Figura 1.22, se puede apreciar: la declaración de librerías, la entidad y la arquitectura. Además se puede observar que sólo hay código concurrente en este ejemplo.

Una vez escrito el código en VHDL, se procede a: chequear la sintaxis, realizar la síntesis, asignar terminales tal como lo muestra la Figura 1.23, implementar el diseño (Traducción, mapeo, ubicación y enrutamiento) y programar el dispositivo.

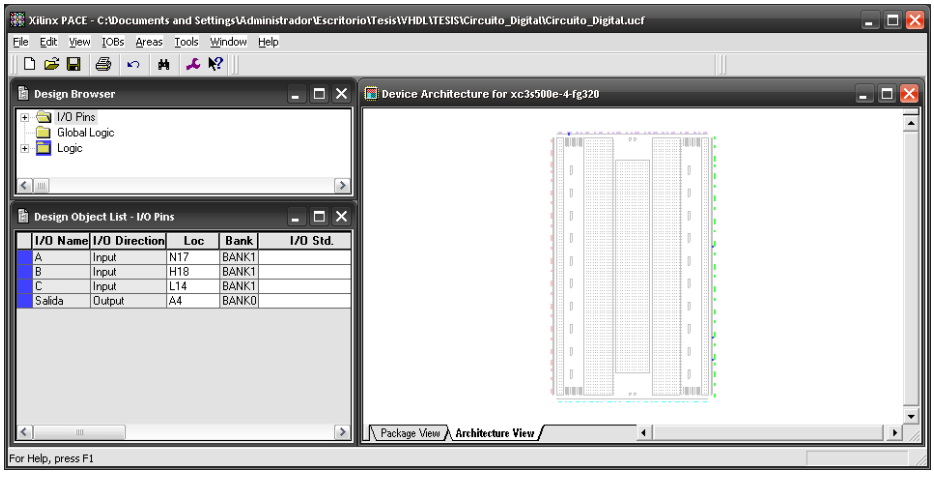
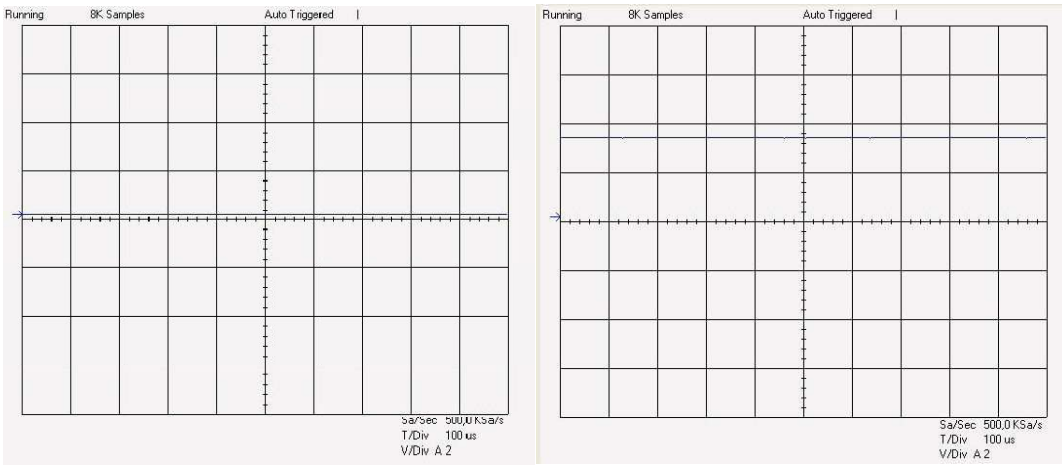


Figura 1.23 Asignación de terminalesdel proyecto Circuito_Digital.

Se verifica la funcionalidad del proyecto mediante una simulación de comportamiento, realizada mediante Testbench de ISE, tal como se muestra en la Figura 1.24.

Figura 1.24 Simulación de comportamiento del Circuito Digital.

Para comprobar la implementación del diseño en el FPGA y su correcto funcionamiento, en la Figura 1.25 se presentan los resultados observados en un osciloscopio. Se visualiza los resultados obtenidos en el terminal A4 cuando la salida es 0L en la parte (a) y 1L en la parte (b).



(a) (b)

Figura 1.25 Presentación de resultados de la implementación del Circuito Digital.

Mediante la simulación y la implementación se comprueba que se cumple la tabla de verdad correspondiente al circuito digital, que se muestra en la Tabla 1.5.

A	B	C	Salida
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Tabla 1.5 Tabla de verdad del circuito digital del primer ejemplo.

1.7.2 IMPLEMENTACIÓN DE UN CIRCUITO DIGITAL EN UN FPGA USANDO PROCESOS Y COMPONENTES DE VHDL

En este ejemplo se implementa el circuito digital de la Figura 1.26 en la Spartan-3E Starter Kit Board, empleando componentes y procesos. El código principal y cada componente son segmentos de código concurrente VHDL que se interconectan a través de señales de entrada y salida.

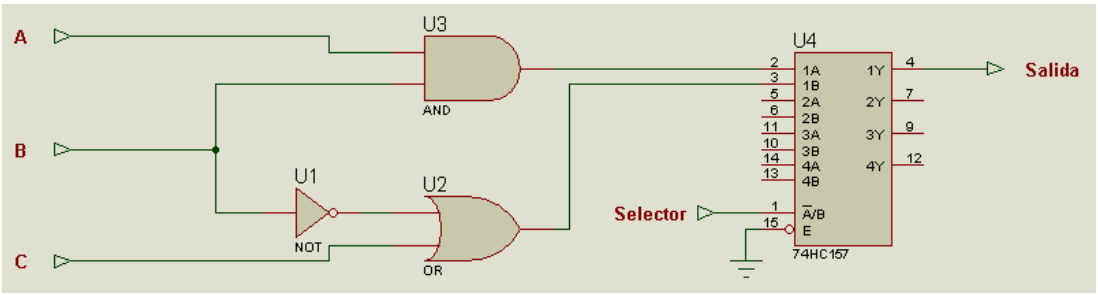
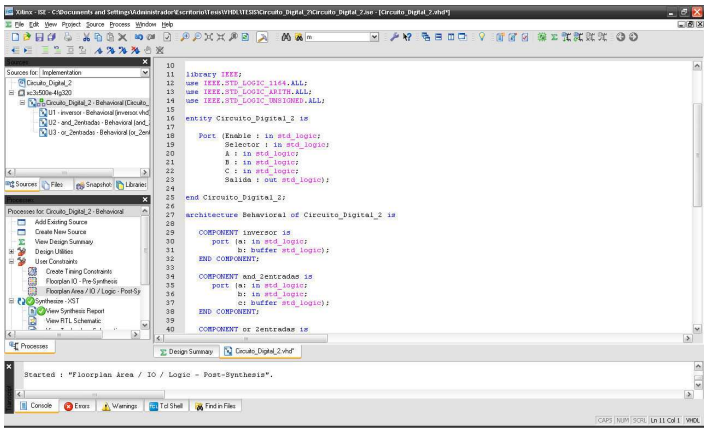


Figura 1.26 Circuito Digital a implementarse a través de procesos y componentes en el FPGA.

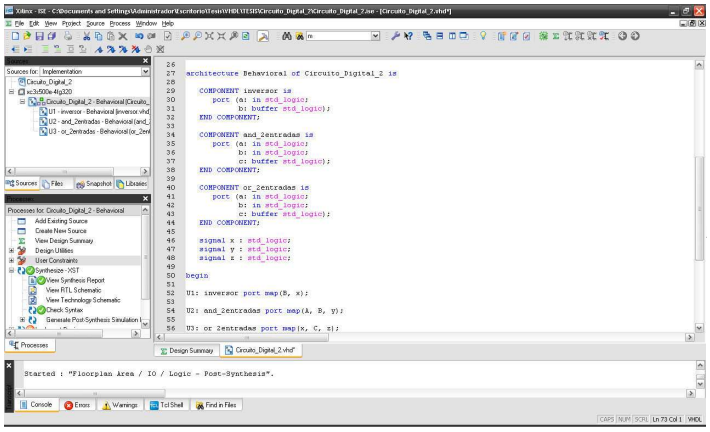
En la creación del modelo se seleccionan las características del proyecto y el dispositivo que se describe en la Tabla 1.4, excepto el nombre del proyecto; lo cual establece que se utilice VHDL para describir el comportamiento del FPGA de la Spartan 3E Starter Kit Board.

El módulo VHDL añadido a este proyecto, Circuito_Digital_2.vhd, se muestra en la Figura 1.27. En la parte (a) se observa que el código principal VHDL tiene en su estructura la declaración de librerías, la entidad y la arquitectura; al igual que en el ejemplo anterior. En la parte (b), se visualiza la declaración de componentes en la

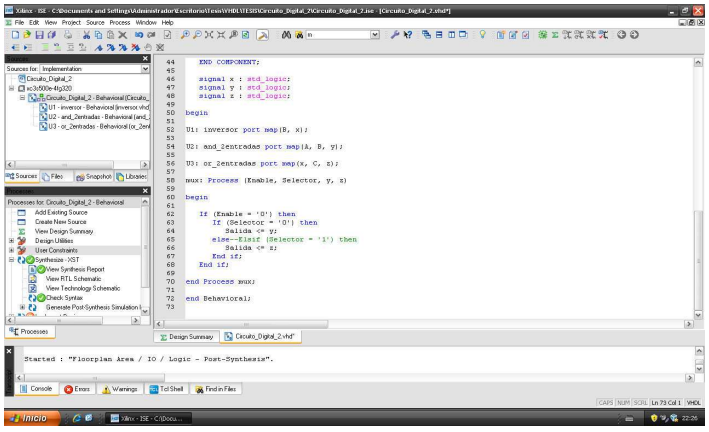
parte declarativa de la arquitectura para cada una de las compuertas lógicas (and, or y not). En la parte (c), se observa un proceso en el cuerpo de la arquitectura que permite implementar el multiplexor digital. Cada componente se lista en la ventana Fuentes (Sources) con su respectiva etiqueta.



(a)



(b)



(c)

Figura 1.27 Partes fundamentales del código principal de Circuito_Digital_2.vhd.

En la Figura 1.28 se muestra la componente denominada Inversor, con el objetivo de observar que las componentes son segmentos VHDL que constan de la estructura básica (declaración de librerías, entidad y arquitectura). Las otras dos componentes (*and_2entradas* y *or_2entradas*) que se declaran en el código principal del proyecto Circuito_Digital_2 son semejantes a ésta; excepto en el contenido de la arquitectura, el cual depende de la funcionalidad que adopte.

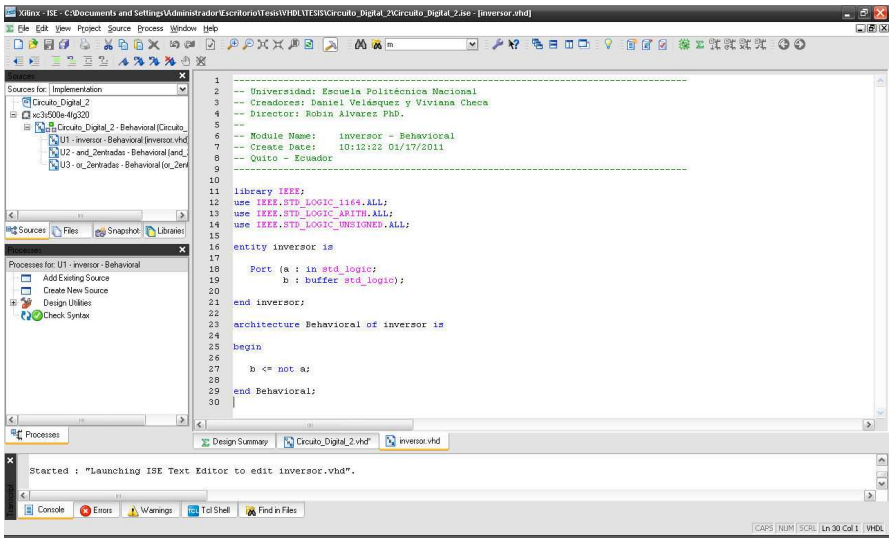


Figura 1.28 Código VHDL de la Componente denominada Inversor.

Una vez escrito en VHDL el código del programa principal y de las componentes, se continúa con el flujo de diseño incluyendo la asignación de terminales, tal como lo muestra la Figura 1.29.

Design Object List - I/O Pins

	I/O Name	I/O Direction	Loc	Bank	I/O Std.
	A	Input	H18	BANK1	
	B	Input	L14	BANK1	
	C	Input	L13	BANK1	
	Enable	Input	A4	BANK0	
	Salida	Output	B4	BANK0	
	Selector	Input	N17	BANK1	

Figura 1.29 Asignación de terminalesdel proyecto Circuito_Digital_2.

La funcionalidad del proyecto se verifica mediante la simulación de comportamiento realizada en Testbench de Xilinx ISE, la cual se muestra en la Figura 1.30.

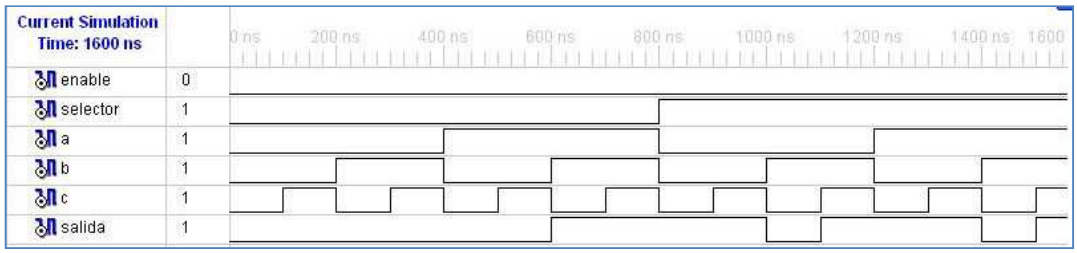


Figura 1.30 Simulación de comportamiento de Circuito_Digital_2.

Para comprobar la implementación del diseño en el FPGA y su correcto funcionamiento, se pueden observar los resultados en el terminal B4 (Salida) con un osciloscopio, que serán cero lógico o uno lógico dependiendo del estado lógico de las entradas, tal como lo muestra la Figura 1.25.

La tabla de verdad correspondiente al circuito digital de este ejemplo se la comprueba mediante la simulación en Testbench y la implementación en el FPGA, la misma que se muestra en la Tabla 1.6.

Enable	Selector	A	B	C	Salida
0	0	0	0	0	0
0	0	0	0	1	0
0	0	0	1	0	0
0	0	0	1	1	0
0	0	1	0	0	0
0	0	1	0	1	0
0	0	1	1	0	1
0	0	1	1	1	1
0	1	0	0	0	1
0	1	0	0	1	1
0	1	0	1	0	0
0	1	0	1	1	1
0	1	1	0	0	1
0	1	1	0	1	1
0	1	1	1	0	0
0	1	1	1	1	1

Tabla 1.6 Tabla de verdad del circuito digital del segundo ejemplo.

CAPÍTULO II

CÓDIGOS DE LÍNEA

2.1 INTRODUCCIÓN

Este capítulo se encarga de explicar los algoritmos que conciernen a los códigos de línea a ser analizados, los cuales son desarrollados y simulados mediante el software de MATLAB; con el propósito de que en los siguientes capítulos, sean implementados en el hardware, utilizando una tarjeta de entrenamiento que contiene un FPGA.

En primera instancia, hay que situar a los Códigos de Línea en el contexto de un Sistema de Comunicaciones. Éste tiene el propósito de transmitir información desde un emisor hasta un receptor a través de un canal. El esquema genérico de un sistema de comunicaciones se muestra en la Figura 2.1.

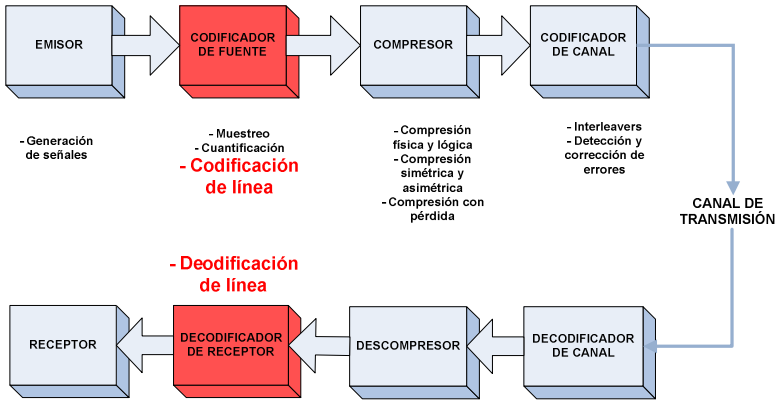


Figura 2.1 Diagrama de bloques de un Sistema de Comunicaciones.

El **Emisor** es donde se generan señales discretas, aquí se emiten los distintos símbolos del alfabeto fuente que se quiere transmitir. Los símbolos emitidos por la fuente llegan al **Codificador de Fuente** donde son transformados en símbolos de un código binario más adecuado para ser transmitido a través de un canal de comunicaciones, esta función es la que desempeñan los ***Códigos de Línea***.

Opcionalmente estos símbolos codificados pueden pasar a la etapa de **Compresión** con el objetivo de reducir su tamaño para conseguir una transmisión más rápida. Durante la transmisión de los símbolos a través del canal pueden producirse alteraciones de los mismos debido a la presencia de ruido en el canal. A estas alteraciones se las denomina errores. Por ello, antes de enviar los símbolos codificados a través del canal, se realiza una nueva codificación orientada a que el receptor pueda detectar y corregir los errores producidos en el canal, esto constituye la **Codificación de Canal**.

En el **Receptor** se realiza el proceso inverso. En primera instancia, se realiza una **Decodificación de Canal** para detectar y corregir los posibles errores que contengan los símbolos recibidos a través del canal. A continuación se procede a una posible **Descompresión** de los símbolos en el caso de haber sido comprimidos en la fuente. Por último se realiza una ***Decodificación de Línea*** en la que los símbolos codificados se transforman en los símbolos originales que fueron transmitidos por el emisor.

Una vez ubicados los Códigos de Línea dentro del esquema de un Sistema de Comunicaciones, hay que mencionar que este proyecto se desarrolla con un solo FPGA, éste constituye codificador y decodificador a la vez.

2.2 CLASIFICACIÓN DE LOS CÓDIGOS DE LÍNEA

SEGÚN LA POLARIDAD

- **Códigos unipolares:** La señal toma valores solamente positivos o solamente negativos, no los dos a la vez, además del nivel cero.

- **Códigos polares:** La señal toma valores positivos y negativos para identificar a cada uno de los niveles lógicos.

- **Códigos bipolares:** La señal varía entre tres niveles: positivo, negativo y cero.

SEGÚN EL TIEMPO DE DURACIÓN

- **Códigos RZ** (Return to Zero), en cuyo caso la forma de onda asignada para la transmisión regresa al nivel 0V por una fracción del tiempo de bit.

- **Códigos NRZ** (Non Return to Zero), en los que la forma de onda asignada para la transmisión no regresa al nivel 0.

En el presente proyecto de titulación se ha considerado desarrollar los códigos de línea presentes en la Tabla 2.1, relacionado a su polaridad.

Códigos unipolares	➢ **NRZ** ➢ **RZ al 50%** ➢ **4B5B**
Códigos Polares	➢ **Diferencial tipo M (NRZI)** ➢ **Diferencial tipo S** ➢ **Bifase L o Manchester** ➢ **Manchester Diferencial** ➢ **CMI**
Códigos Bipolares	➢ **AMI** ➢ **HDB3** ➢ **MLT-3**

Tabla 2.1 Clasificación de los códigos de línea considerados en el presente proyecto de titulación según su polaridad.

Los códigos de línea expuestos en la Tabla 2.1, van a ser presentados en base a su algoritmo de codificación, dicho algoritmo va a ser plasmado en un diagrama de flujo donde se explica la programación realizada en MATLAB (Matrix Laboratory), este software es un producto de la empresa The Mathworks Inc.,

empresa fundada en 1984[15]. Los archivos .m que corresponden a cada código se adjuntan en el **ANEXO A** de este proyecto.

La simulación en el tiempo se muestra en una figura generada por cada archivo .m donde se exhiben tres señales: **señal de reloj, señal binaria de datos y señal codificada**, con el objetivo de exponer el funcionamiento de cada código. Para todos los códigos de línea, la **señal binaria de datos** que se ha considerado codificar, corresponde a los caracteres ASCII **a** y **b,** que en binario constituye la señal 10000110 01000110, a la izquierda el LSB (Bit menos significativo)**.** Para el código de línea HDB3**,** se exponen tres ejemplos para una mejor comprensión del mismo.

Espectro de Potencia Teórico

El espectro de potencia teórico, mostrado posteriormente en cada uno de los códigos de línea se obtiene mediante un análisis de Fourier. Un dato binario se puede representar con un pulso rectangular de ancho ζ, la transformada de Fourier para el pulso se muestra en la Figura 2.2.

El pulso rectangular está definido como:

$$W(t) = \pi\left(\frac{t}{\zeta}\right) \triangleq \begin{cases} 1, & |t| \leq \zeta/2 \\ 0, & |t| \geq \zeta/2 \end{cases}$$

Luego de aplicar la Transformada de Fourier cuando $W(t) = 1$ se obtiene:

$$W(f) = \zeta S_a\left(\frac{\omega\zeta}{2}\right) = \zeta S_a(\pi\zeta f)$$

[15] Nota legal: MATLAB es un software propietario de The Mathworks Inc. y ha sido utilizado sólo con fines académicos en el presente proyecto de titulación.

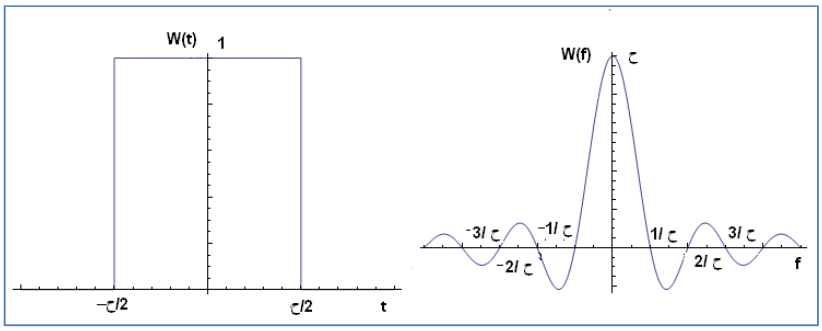


Figura 2.2 Transformada de Fourier de un pulso rectangular.

El resultado en el dominio de la frecuencia se determina mediante este espectro de potencia teórico. Las simulaciones para la obtención de los espectros se realizan a 9600 bps, por lo que la frecuencia donde se anula el espectro es de 9600 Hz, para la mayoría de códigos. En cuanto a los códigos que presentan una transición a mitad de tiempo de bit, la frecuencia de anulamiento es 19200 Hz. El código 4B5B es un caso especial, donde el ancho de banda efectivo de la señal codificada es 5/4 el ancho de banda efectivo de los datos originales, constituyendo el anulamiento en 12000 Hz.

2.2.1 CÓDIGO NRZ (NON RETURN TO ZERO)

La forma más frecuente de transmitir señales digitales es mediante la utilización de un nivel diferente de tensión para cada uno de los bits. Los códigos que siguen esta estrategia comparten la propiedad de que el nivel de tensión se mantiene constante durante la duración del bit, es decir, no hay transiciones (no hay retorno al nivel cero de tensión).

Algoritmo de codificación

Este código presenta una regla de codificación sencilla. A cada símbolo 0L ó 1L se le asigna uno de los niveles 0 o ± A, respectivamente, dependiendo de la lógica positiva o negativa. El diagrama de flujo que concierne a la programación en MATLAB de este código se muestra en la Figura 2.3.

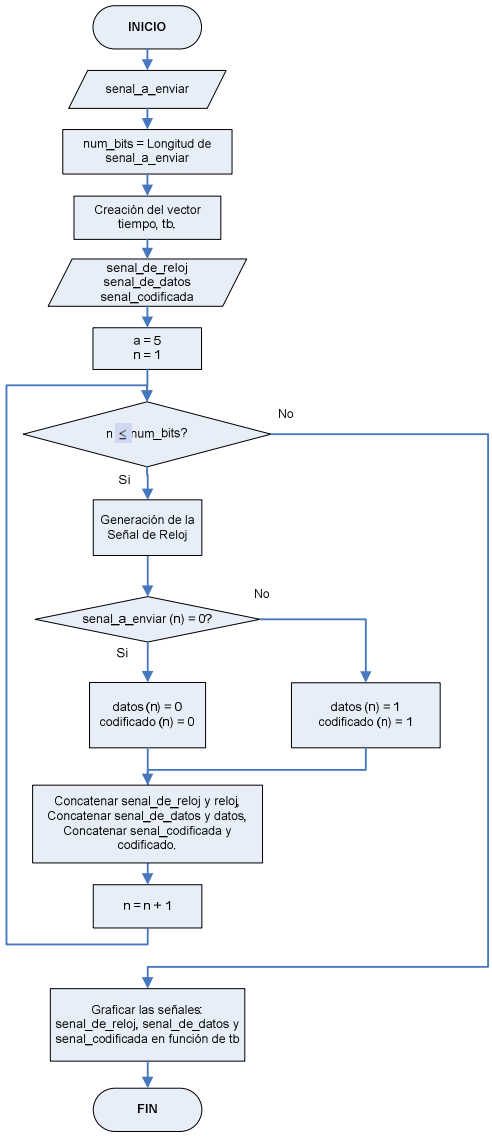


Figura 2.3 Diagrama de Flujo de la Codificación NRZ en MATLAB.

El diagrama de flujo de la figura anterior nos permite obtener la codificación NRZ. La variable de entrada *senal_a_enviar*, representa los bits que se van a codificar. Con *num_bits* se obtiene la longitud de la variable anterior.

El vector *tb* (tiempo de bit) se crea para los respectivos gráficos. Se establece la amplitud de las señales en cinco, reflejado en la variable *a*.

La variable *n,* con valor de 1, permite inicializar el lazo *while*, donde las instrucciones presentadas dentro de este lazo generan el vector *reloj*, y según sea 0L ó 1L la *senal_a_enviar* se obtienen los vectores *datos* y *codificado* de

acuerdo al algoritmo NRZ. Luego se van a concatenar las señales: *senal_de_reloj, senal_de_datos y senal_*codificada con *reloj*, *datos* y *codificado*, respectivamente, para posteriormente ser graficadas.

Las señales antes mencionadas se muestran en la Figura 2.4.

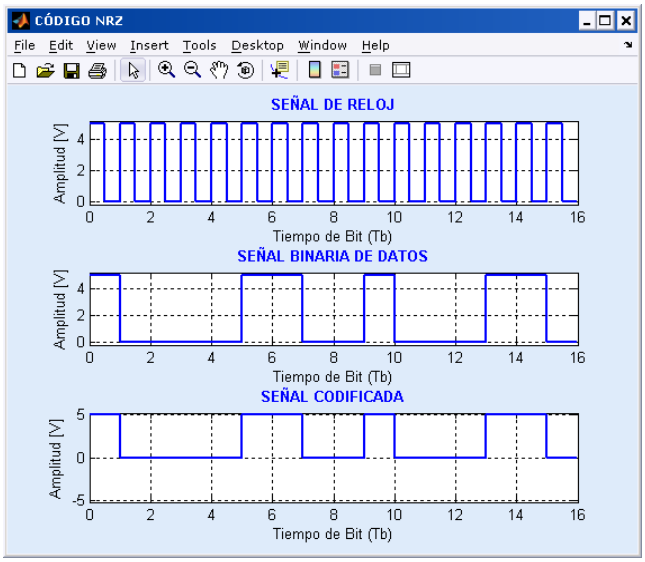


Figura 2.4 Codificación NRZ unipolar en MATLAB.

En la Figura 2.4 se observa la codificación NRZ unipolar, la señal binaria de datos tiene la misma amplitud que la señal codificada.

Este código es utilizado en lógica digital.

Espectro de Potencia Teórico

En la Figura 2.5, se presenta el espectro de potencia para el código NRZ, se observa una gran componente DC, ya que esta es una característica de dicho código.

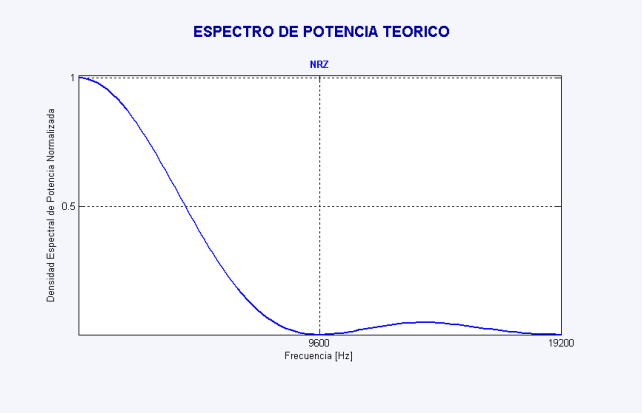


Figura 2.5 Espectro de Potencia Teórico para el código NRZ unipolar.

2.2.2 CÓDIGO RZ AL 50 %

Se dice que el código es con retorno a cero, porque cuando existe un 1L, durante un cierto porcentaje del tiempo de bit, Tb, la señal regresa nuevamente a cero.

Algoritmo de codificación

El binario 1L se representa mediante un pulso de ancho de la mitad del intervalo de tiempo de bit (50%), y un 0L es representado por la ausencia de pulso. El diagrama de flujo se exhibe en la Figura 2.6.

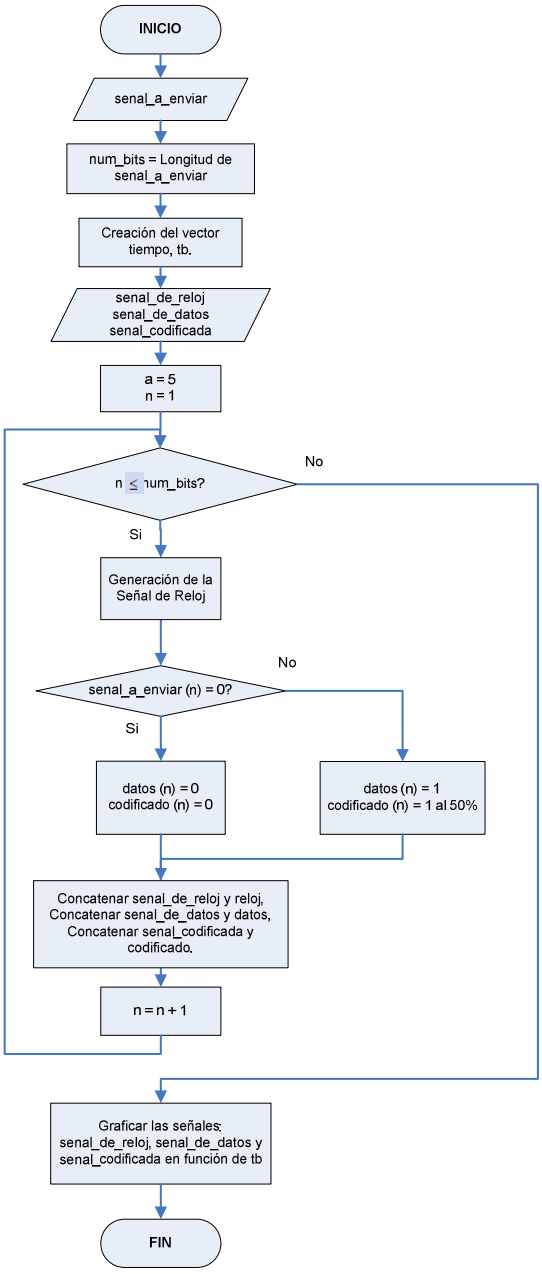


Figura 2.6 Diagrama de Flujo de la Codificación RZ al 50% en MATLAB.

La Figura 2.6 muestra la programación de la codificación RZ al 50% en MATLAB. Se muestra similar a la del código NRZ, ya que presenta las mismas señales que nos permiten obtener los resultados que se aprecian en la Figura 2.7 cumpliendo con el algoritmo de codificación.

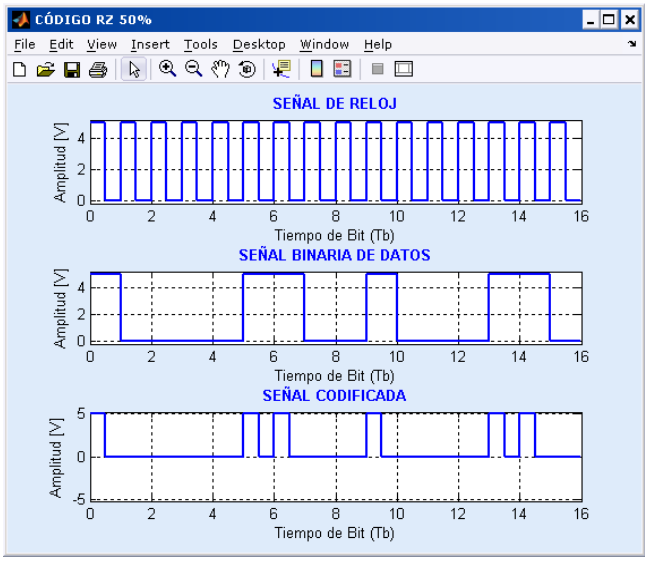


Figura 2.7 Codificación RZ 50 % unipolar en MATLAB.

Se puede observar el funcionamiento del código RZ, que presenta una transición para el caso específico de este proyecto de titulación a medio tiempo de bit, es decir al 50%, cuando se está codificando un 1L.

Como característica que concierne a este código se considera la pérdida de sincronismo en secuencias largas de 0Ls.

Una de las aplicaciones más frecuentes de la codificación RZ se encuentra en la grabación magnética de datos.

Espectro de Potencia Teórico

Al ser este código de tipo unipolar, le caracteriza una componente DC considerable. La señal codificada requiere del doble de ancho de banda efectivo de la señal de datos debido a la reducción del ancho de pulso a la mitad. En la Figura 2.8 se visualiza el espectro de potencia teórico para este código.

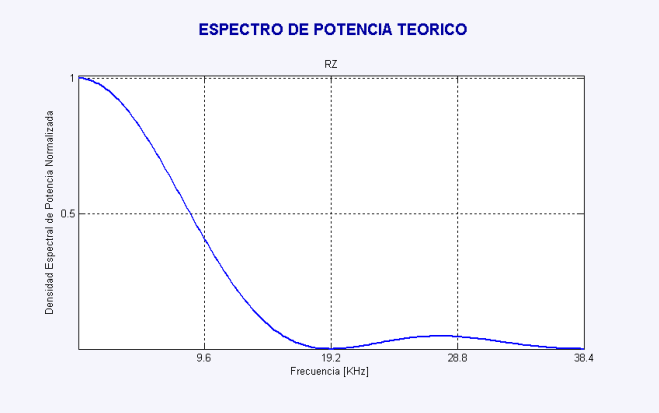


Figura 2.8 Espectro de Potencia Teórico para el código RZ al 50% unipolar.

2.2.3 CÓDIGO 4B5B (4 BINARY-5 BINARY)

Los códigos mBnB han sido desarrollados para codificar m dígitos binarios en grupos de n dígitos binarios de salida, donde m< n. Debido a que se pueden elegir 2^n valores codificados en cada grupo, existe la posibilidad de utilizar determinadas palabras código, para control de la transmisión. El código 4B5B toma 32 palabras de 5 bits, 16 para la representación de datos y las restantes para control de transmisión.

Algoritmo de codificación

El código de línea 4B5B tiene 16 símbolos para representar 16 datos binarios (0H a FH), 8 símbolos de control (Q, H, I, J, K, T, R, S) y 8 símbolos de violación (V). La codificación de los símbolos de datos está diseñada de tal manera que en condiciones normales nunca se tengan cuatro ceros consecutivos. Cabe mencionar que en este proyecto, se utilizan únicamente las palabras para la representación de datos.

En la Tabla 2.2 se ilustra la distribución de los símbolos.

Campo de Datos			Símbolos de control
	Datos	**Símbolo**	Q 00000
0	0000	11110	H 00100
1	0001	01001	I 11111
2	0010	10100	J 11000
3	0011	10101	K 10001
4	0100	01010	T 01101
5	0101	01011	R 00111
6	0110	01110	S 11001
7	0111	01111	**Nota: (algunos símbolos V pueden tomarse como H)**
8	1000	10010	**Símbolos de violación**
9	1001	10011	V o H 00001
A	1010	10110	V o H 00010
B	1011	10111	V 00011
C	1100	11010	V 00101
D	1101	11011	V 00110
E	1110	11100	V o H 01000
F	1111	11101	V 01100
			V o H 10000

Tabla 2.2 Símbolos del código de línea 4B5B.

Cabe recalcar que en la simulación que se va a presentar solo se han considerado los 16 primeros símbolos que representan los datos binarios, los símbolos de control y violación no se han tomado en cuenta.

En la Figura 2.9 se visualiza el diagrama de flujo para el código 4B5B.

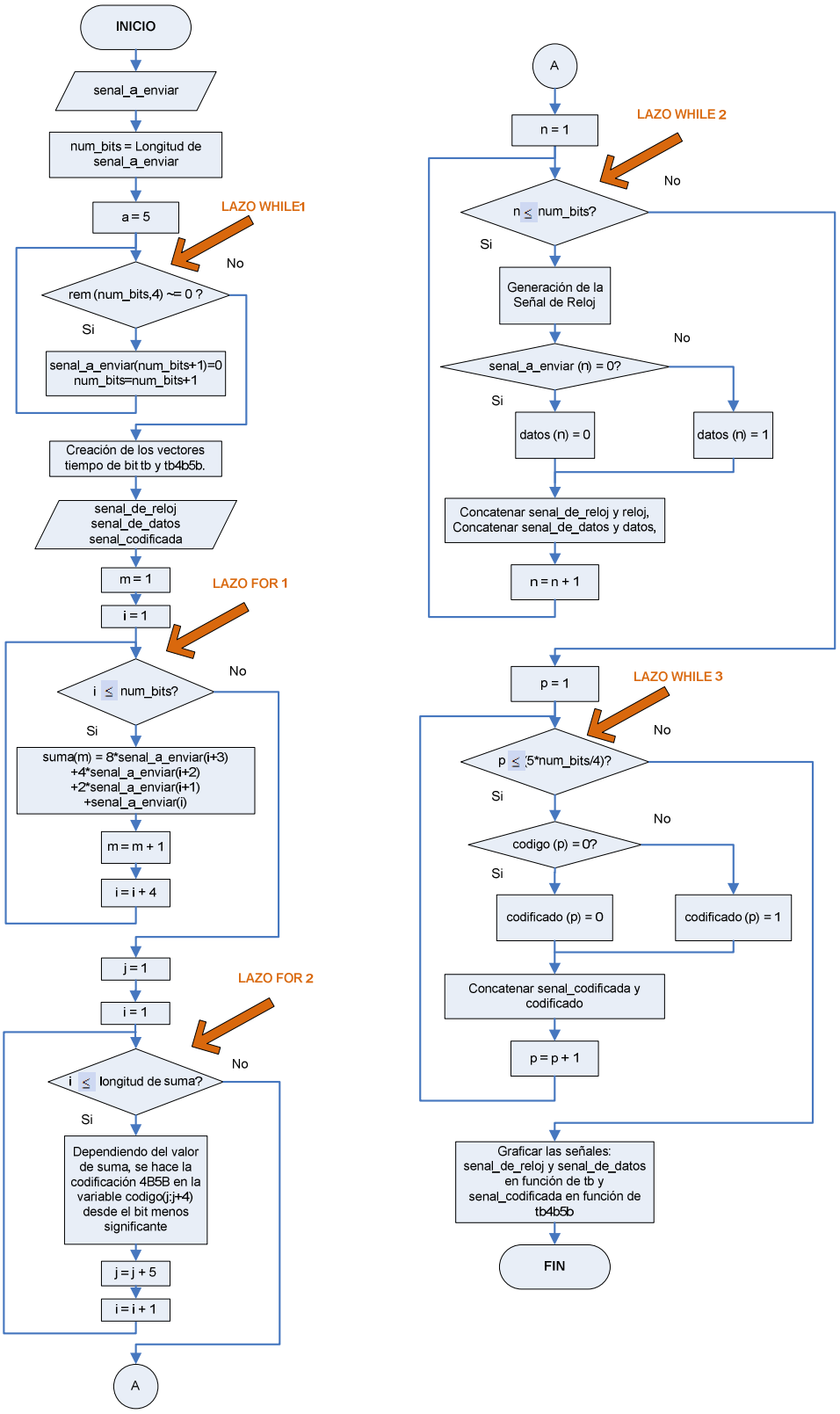


Figura 2.9 Diagrama de Flujo de la Codificación 4B5B en MATLAB.

En la programación en MATLAB del código 4B5B, tenemos la variable de entrada *num_bits* donde se almacena el número de bits que corresponden a la variable *senal_a_enviar* que ingresa el usuario.

Se crean dos vectores de tiempo de bit: *tb* y *tb4b5b*, el primero va a ser utilizado para graficar la señal de reloj y la señal binaria de datos, y el segundo va a permitir al usuario la visualización de la señal codificada. Como en todas las simulaciones, son tres señales que van a ser concatenadas *senal_de_reloj, senal_de_datos y senal_codificada.* La amplitud de las señales es igual a 5.

La señal de datos debe de ser múltiplo de cuatro bits, el primer lazo *while* cumple con este propósito; en caso que el usuario no haya ingresado datos múltiplo de cuatro bits, se rellena con 0L hasta que el número de bits completen un múltiplo de cuatro.

En el primer lazo *for*, la variable *m* tiene el valor de 1 para inicializar el vector suma, en dicho vector se almacenan los valores en decimal del 0 al 15 de cada cuatro bits que van a ser codificados.

En el lazo *for 2*, para identificar el primer elemento del vector *codigo*, el valor de la variable *j* es igual a 1, la ejecución de las instrucciones dentro del lazo admiten la obtención de los bits codificados en *codigo,* de acuerdo a la Tabla 2.2.

El lazo *while 2,* permite la generación y concatenación de la señal de reloj y la señal datos en base a los bits que presenta la variable senal_a_enviar.

En el tercer lazo *while,* mientras *p* sea menor o igual al número de símbolos codificados, se genera y concatena la señal codificada, sustentada por el vector *codigo.*

Una vez culminados todos los procesos descritos, se grafican las tres señales que fueron concatenadas, esto se aprecia en la Figura 2.10.

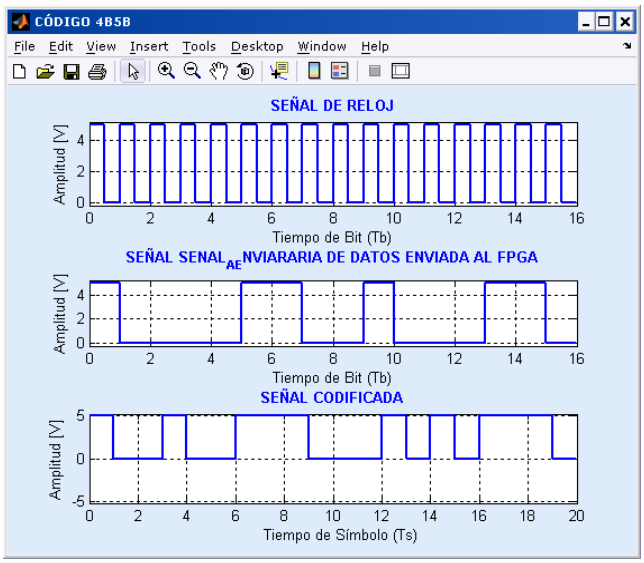


Figura 2.10 Codificación 4B5B en MATLAB.

Se puede observar en la Figura 2.10 que por cada ocho bits de datos hay diez símbolos en la señal codificada. Para la comprobación de la codificación hay que tomar en cuenta que el bit menos significante es el de la izquierda en la simulación, y el bit más significante en la Tabla 2.2 corresponde al de la izquierda. Como se observa, los primeros cuatro bits de la señal binaria de datos son 1000 (que representa 1 en decimal), lo que corresponde según la tabla mencionada a 10010.

El código 4B5B se utiliza en los siguientes estándares:

- 100BASE-TX estándar definido por la IEEE 802.3u en 1995.
- AES10-2003 MADI (Interfaz Digital de Audio de varios canales).

Además, el código en análisis se combina con el NRZI, a fin de aumentar el número de transiciones y mantener buen sincronismo.

Espectro de Potencia Teórico

El espectro de potencia para este código unipolar se muestra en la Figura 2.11. El ancho de banda efectivo para la señal codificada 4B5B, es mayor en una fracción de 5/4 al ancho de banda efectivo de la señal de datos. En este caso el ancho de banda efectivo para la codificación es de 12 KHz.

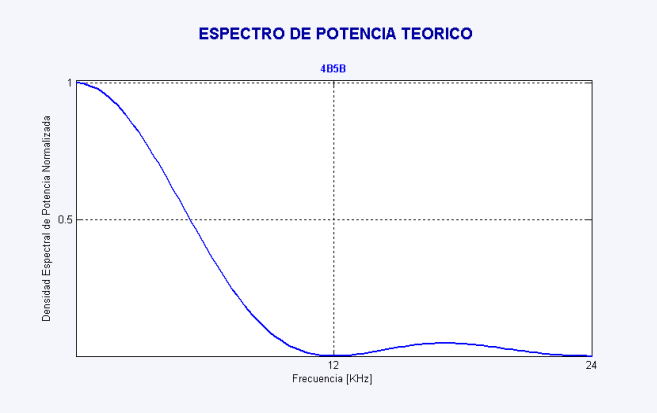


Figura 2.11 Espectro de Potencia Teórico para el código 4B5B.

2.2.4 CÓDIGOS DIFERENCIALES

Son códigos polares. El principio de codificación se basa en que uno de los dígitos binarios (1L o 0L) cambia el nivel de la señal respecto al nivel del estado precedente, y el otro no.

2.2.4.1 CÓDIGO DIFERENCIAL TIPO M (NRZI)

El primer código diferencial que se analiza es el tipo M o también llamado NRZI. Es utilizado en varios tipos de redes en combinación a otros códigos para la transmisión de datos.

Algoritmo de codificación

En el código diferencial tipo M el dígito binario 1L cambia el nivel de la señal (Si estuvo en –A cambiará a +A), en tanto que los 0L mantienen dicho nivel.

En la Figura 2.12 se visualiza el diagrama de flujo del código.

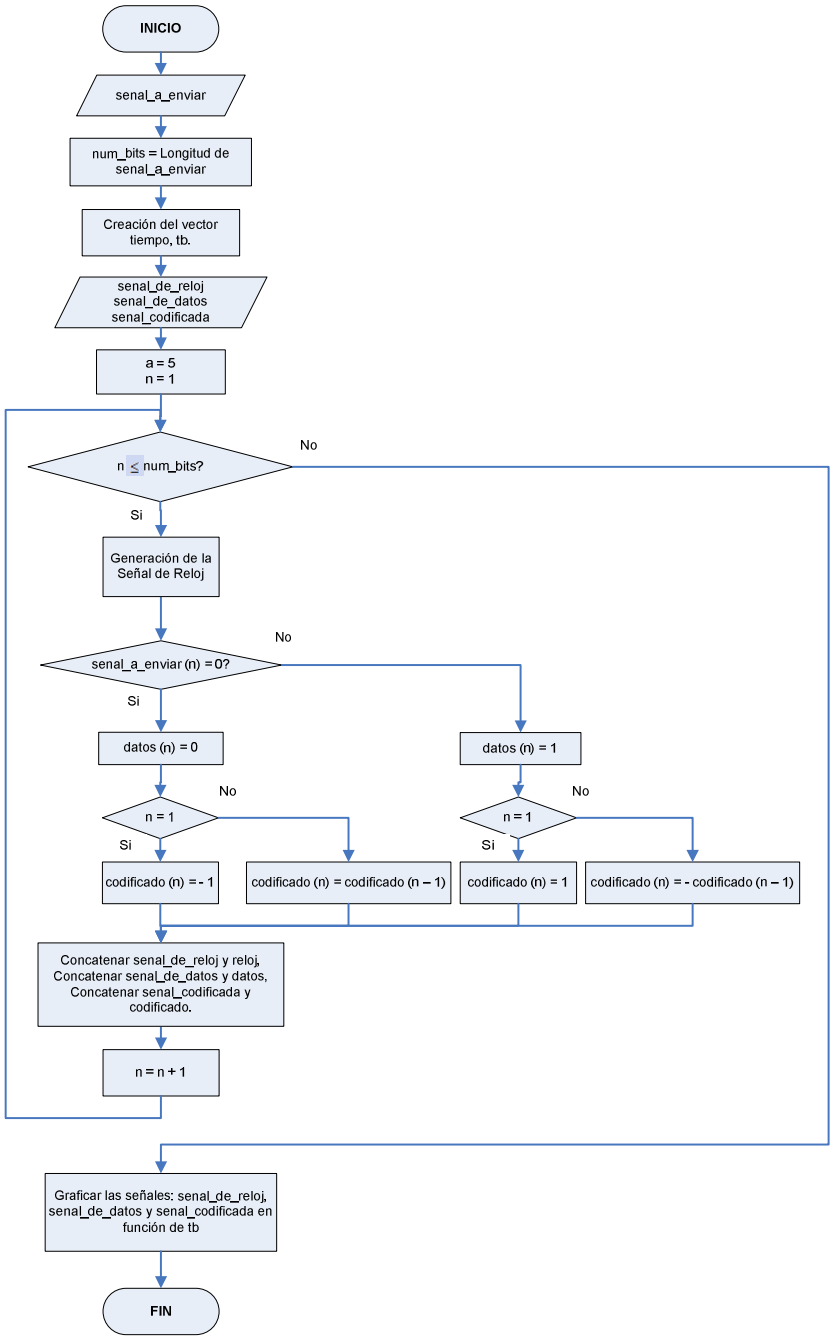


Figura 2.12 Diagrama de Flujo de la Codificación Diferencial Tipo M en MATLAB.

La programación para el código Diferencial tipo M, se aprecia de manera similar a los códigos que le anteceden, las variables de entrada *senal_a_enviar, num_bits, tb,* desempeñan las mismas funciones descritas anteriormente. Las tres señales a concatenar son las mismas que en los otros códigos con el propósito de ser graficadas, tal como se muestra en la Figura 2.13.

Dentro del lazo *while,* se ejecutan las instrucciones que permiten obtener la señal de reloj, señal binaria de datos y la señal codificada, cumpliendo con el algoritmo del código.

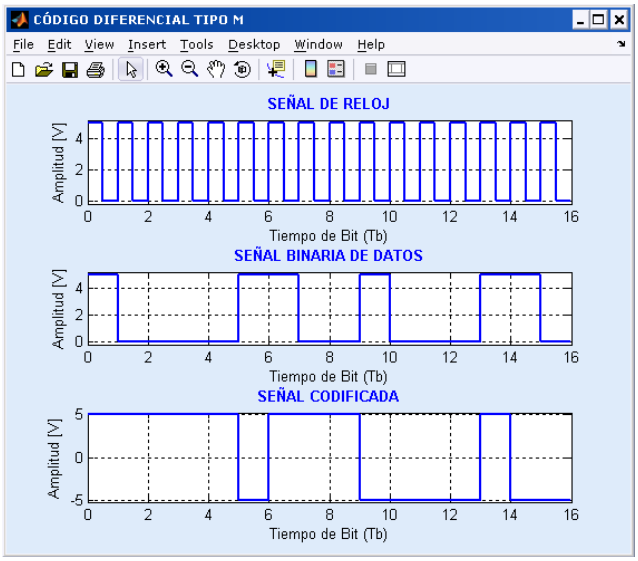


Figura 2.13 Codificación Diferencial Tipo M en MATLAB.

El código NRZI, considera un nivel inicial para poder realizar la codificación con el primer bit que presenta la señal binaria de datos. Para este caso, dicho nivel es negativo, es decir -5 [V], por lo que el primer bit 1L, cambia el nivel anterior, cumpliéndose con la regla de codificación.

Los códigos de línea diferenciales tienen buen desempeño frente al ruido. El código NRZI se usa en combinación con el código 4B5B en redes LAN de alta velocidad. En contraposición, en secuencias largas de 0Ls se generan pérdidas de sincronismo.

Espectro de Potencia Teórico

La Figura 2.14 muestra el espectro de potencia teórico para el código Diferencial Tipo M. La componente DC es significativa.

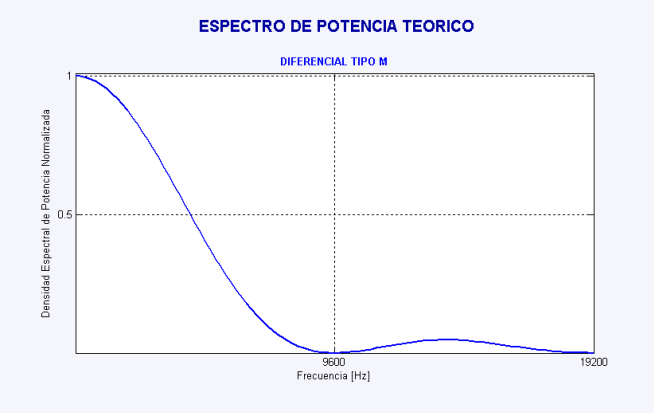


Figura 2.14 Espectro de Potencia Teórico para el código Diferencial Tipo M.

2.2.4.2 CÓDIGO DIFERENCIAL TIPO S

Este código diferencial de tipo polar, presenta una regla contraria al código descrito precedentemente.

Algoritmo de codificación

En este código los 0L cambian el nivel de la señal (si la señal se encuentra en +A, cambiará a –A), mientras que los 1L mantienen el nivel de dicha señal.

El diagrama de flujo que corresponde a este código se muestra en la Figura 2.15.

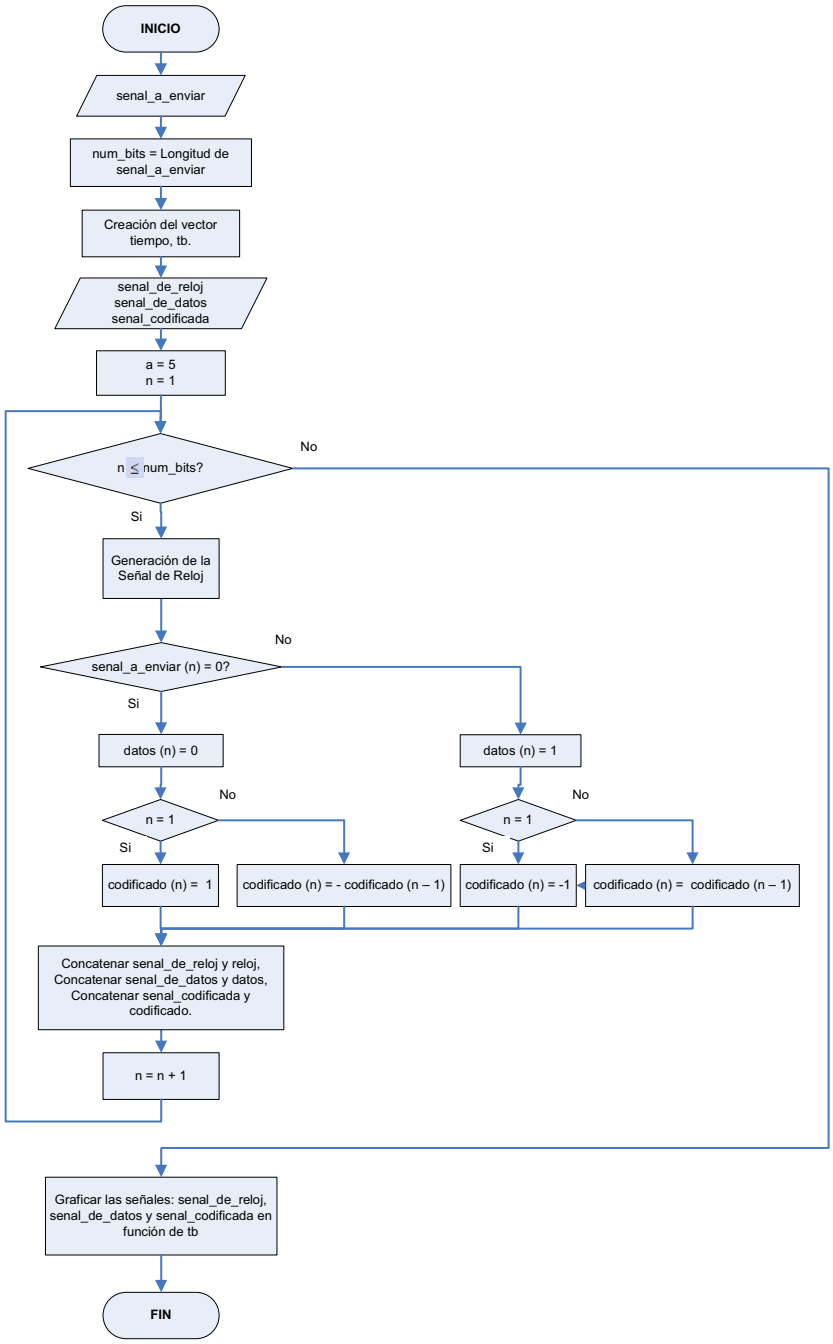


Figura 2.15 Diagrama de Flujo de la Codificación Diferencial Tipo S en MATLAB.

La programación para este código es similar a la del Diferencial Tipo M, lo que hace la diferencia, es el algoritmo que se cumple cuando se ejecutan las instrucciones dentro del lazo *while*. Un ejemplo de la simulación de la codificación Diferencial Tipo S se encuentra en la Figura 2.16.

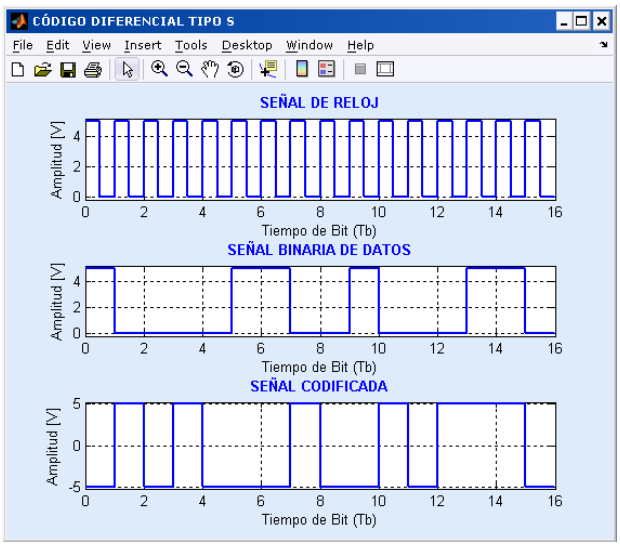


Figura 2.16 Codificación Diferencial Tipo S en MATLAB.

Para este tipo de codificación, el nivel inicial, al igual que en el Diferencial Tipo M, es negativo, por lo que el primer bit codificado mantiene el nivel anterior, correspondiente a un 1L en la señal binaria de datos.

En este caso, las pérdidas de sincronismo se dan en secuencias largas de 1Ls.

Espectro de Potencia Teórico

El código Diferencial Tipo S presenta un espectro de potencia similar al del código Diferencial Tipo M. En la Figura 2.17 se lo puede visualizar.

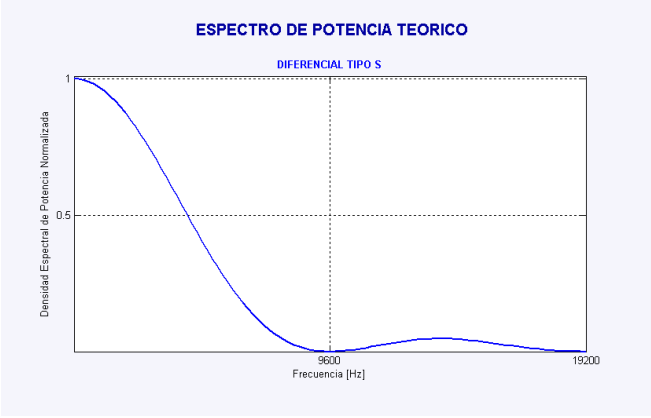


Figura 2.17 Espectro de Potencia Teórico para el código Diferencial Tipo S.

2.2.5 CÓDIGO BIFASE L O MANCHESTER

Es un código polar. En el código Manchester se garantiza siempre una transición a la mitad del período de bit esto es a Tb/2 entre los niveles +A y –A.

Algoritmo de codificación

El nivel lógico 1L en la señal de datos se codifica con una transición negativa a la mitad del tiempo de bit, en tanto que el nivel lógico 0L se codifica con una transición positiva.

En la Tabla 2.3 se resume la regla de codificación del código Manchester.

Dígito Binario	Transición
1 L	
0 L	

Tabla 2.3 Tabla de codificación Manchester.

El diagrama de flujo para la codificación Manchester se observa en la Figura 2.18.

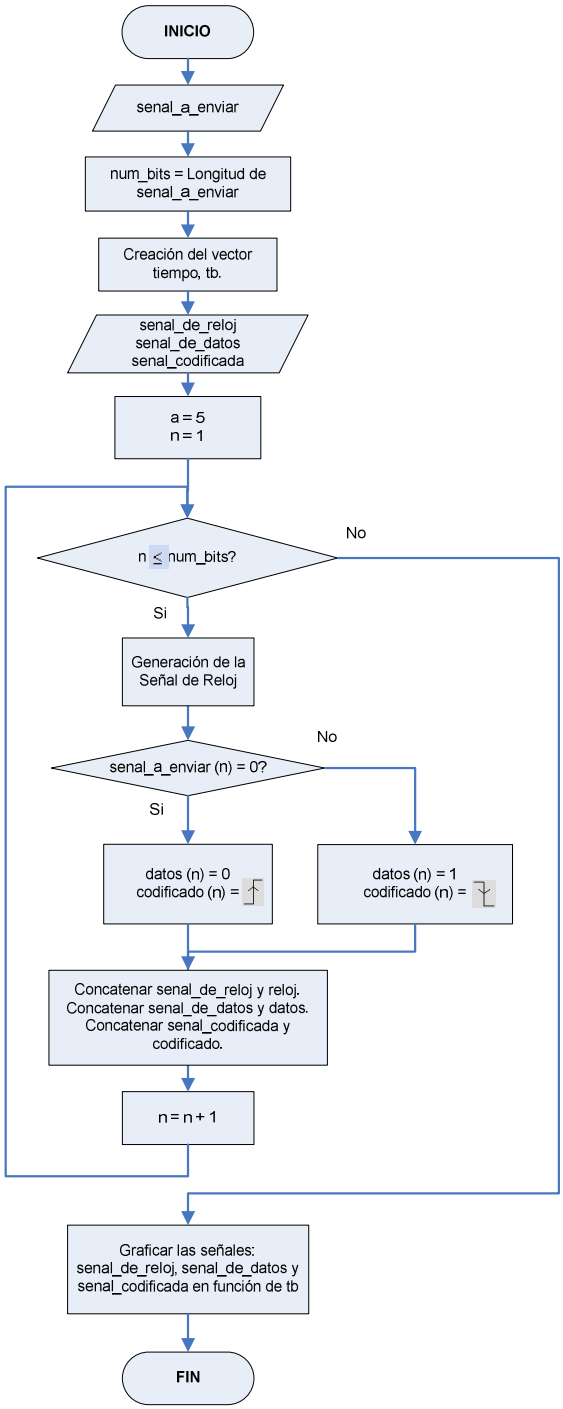


Figura 2.18 Diagrama de Flujo de la Codificación Manchester en MATLAB.

La programación de la codificación Manchester en MATLAB sigue el formato establecido en los otros códigos. Las instrucciones que se encuentran dentro del lazo *while,* permiten la ejecución del algoritmo del código, colocando una

transición positiva por cada 0L y una transición negativa por cada 1L. La Figura 2.19 permite visualizar un ejemplo de dicha programación.

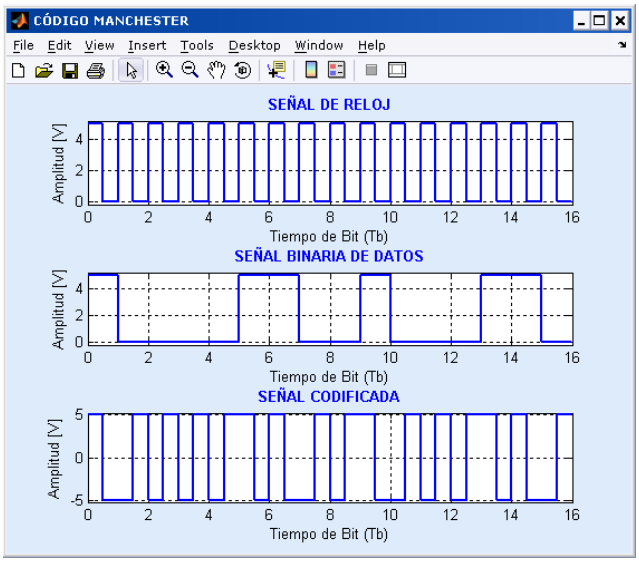


Figura 2.19 Codificación Manchester en MATLAB.

Teniendo en cuenta la Tabla 2.3, que resume la regla de codificación, se observa en la figura anterior, el cumplimiento del algoritmo como tal.

Este código presenta un buen sincronismo debido a que todos los períodos de bit tienen una transición a la mitad. Además, permite detección de errores cuando la transición esperada no ocurre.

El código Manchester es parte de la especificación de la normalización IEEE 802.3 para la transmisión en redes LAN con un bus CSMA/CD usando cable coaxial en banda base o par trenzado.

Espectro de Potencia Teórico

Este código polar presenta una componente DC nula y el ancho de banda efectivo requerido por la señal codificada es el doble de la señal de datos. El espectro de potencia teórico se presenta en la Figura 2.20.

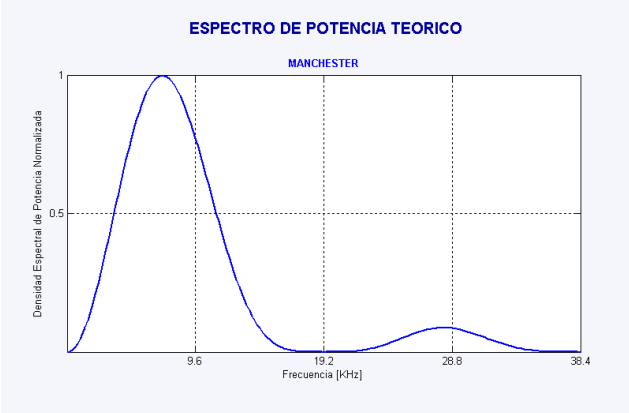


Figura 2.20 Espectro de Potencia Teórico para el código Manchester.

2.2.6 CÓDIGO MANCHESTER DIFERENCIAL

Es un código polar. Este código es una variación de la codificación Manchester, manteniendo las transiciones a la mitad del período de bit.

Algoritmo de codificación

Para la codificación del bit 1L la polaridad de la transición es opuesta a la del bit anterior, en tanto que para el bit 0L la polaridad de la transición es igual a la del bit anterior. En la Figura 2.21 se presenta el diagrama de flujo para la codificación Manchester Diferencial.

En la Tabla 2.4 se especifica el algoritmo de codificación del código Manchester Diferencial.

Dígito Binario	Transición previa	Transición siguiente
1L		
0L		

Tabla 2.4 Tabla de Codificación Manchester Diferencial.

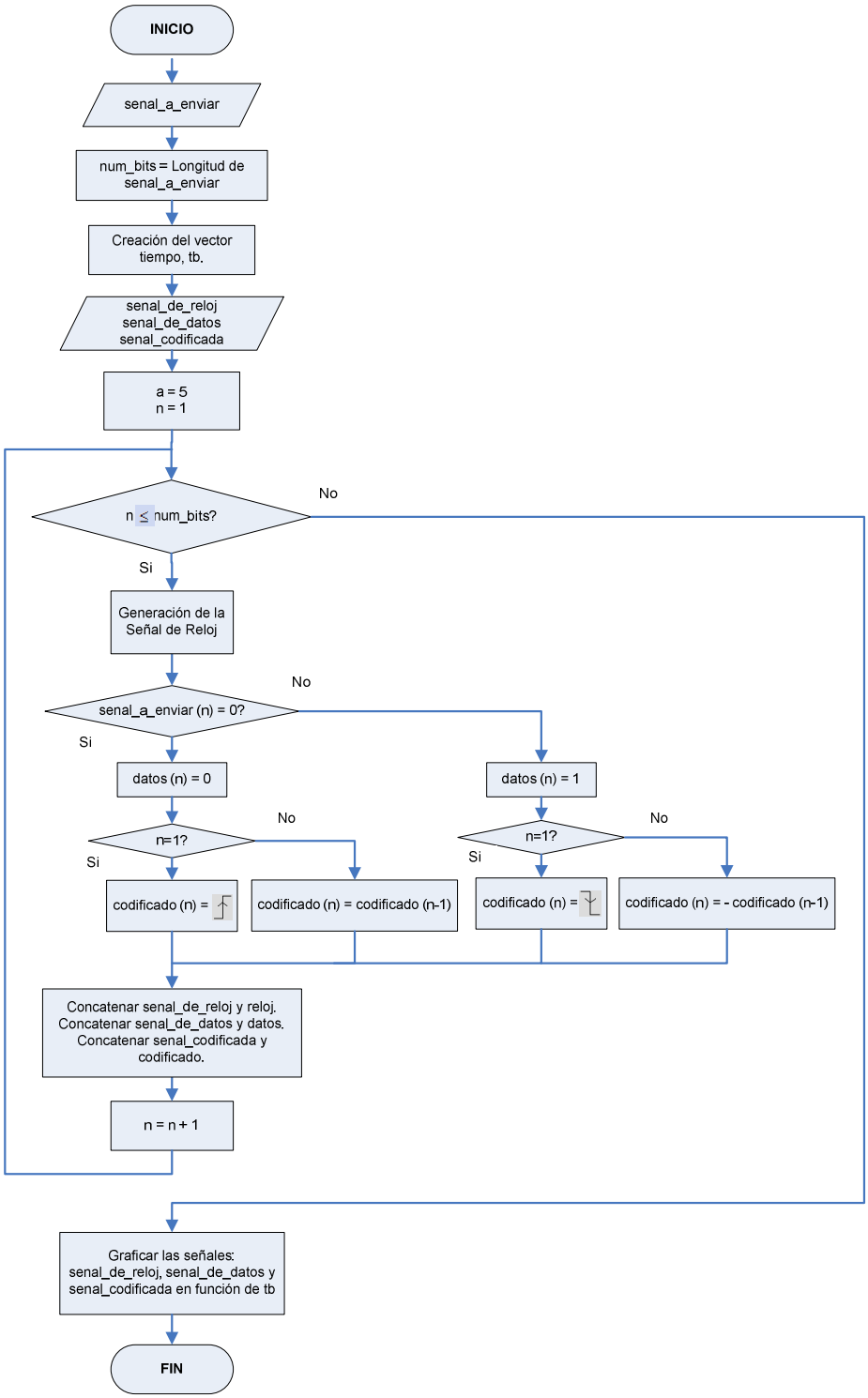


Figura 2.21 Diagrama de Flujo de la Codificación Manchester Diferencial en MATLAB.

Para el análisis del diagrama de flujo, se va a tomar en cuenta el lazo *while* presente en la programación; si el primer bit a codificar es 0L, es decir, cuando *n*

es igual a 1, la señal se codifica con una transición igual a la de referencia; en el caso de que sea 1L el primer bit, se codifica con una transición contraria a la precedente. La Figura 2.22 muestra un ejemplo de la simulación.

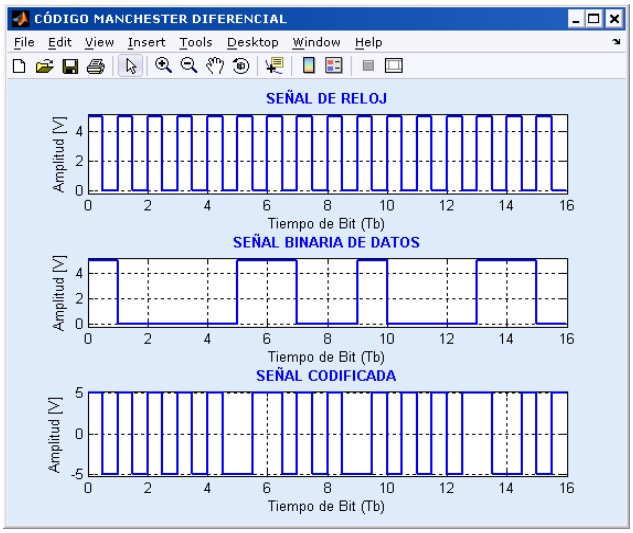


Figura 2.22 Codificación Manchester Diferencial en MATLAB.

En el código Manchester Diferencial, es necesario conocer la transición previa, ya sea esta positiva o negativa. En esta figura, la transición previa es positiva, el primer bit de la señal de datos es un 1L, por lo tanto, la transición correspondiente a este bit es negativa, como se puede verificar en la Tabla 2.4.

El código en cuestión presenta buen sincronismo por la misma razón que el código Manchester. Tiene una mayor tolerancia al ruido, dado que la codificación se da por comparación con el estado del bit precedente, lo que determina una mayor confiabilidad para detectar la transición en presencia del ruido.

El Manchester Diferencial se ha elegido en la normalización IEEE 802.5 para redes LAN en anillo con paso de testigo, en las que se usan pares trenzados apantallados.

Espectro de Potencia Teórico

Al igual que el código Manchester, éste código presenta una componente DC nula, y el ancho de banda efectivo requerido por la señal codificada es el doble de el de los datos, similar a todos los códigos que presentan una transición a la mitad

de tiempo de bit. En la Figura 2.23 se muestra el espectro de potencia teórico para este código.

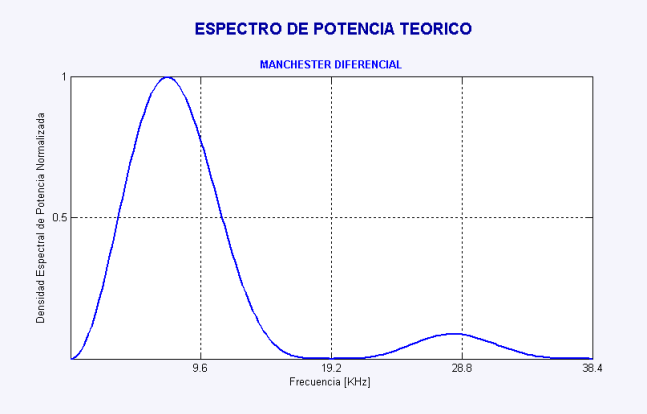


Figura 2.23 Espectro de Potencia Teórico para el código Manchester Diferencial.

2.2.7 CÓDIGO CMI (CODED MARK INVERSION)

Este código es polar y también es considerado como bifase.

Algoritmo de codificación

El 0L se codifica con un cambio de polaridad negativa a positiva, que se produce a la mitad del período de bit (Tb/2). El 1L es codificado con pulsos positivos y negativos alternados de duración Tb (sin transición a la mitad del período de bit). En la Tabla 2.5 se resume la regla de codificación CMI. El diagrama de flujo correspondiente se muestra en la Figura 2.24.

Dígito Binario	Transición
1 L	Pulsos positivos y negativos alternados de duración Tb.
0 L	

Tabla 2.5 Tabla de Codificación CMI.

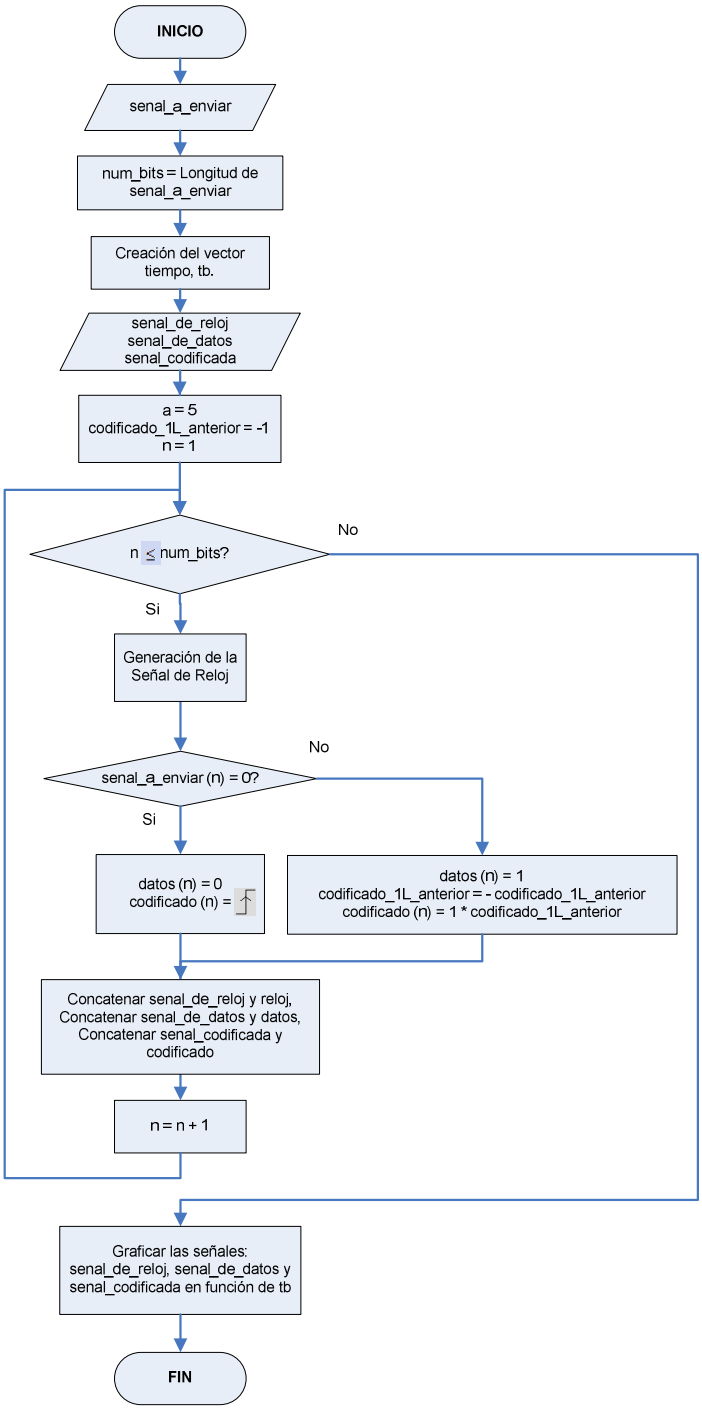


Figura 2.24 Diagrama de Flujo Codificación CMI en MATLAB.

Hay que mencionar que en este código se ha considerado crear la variable *codificado_1L_anterior*, con el propósito de que al momento de codificar los 1Ls de acuerdo con el algoritmo, se cambie el nivel, para esto se parte de una condición inicial, en este caso -1L. Se observa que si el bit a codificar es un 0L,

esto se traduce en una transición positiva; si el bit a codificar es un 1L, como ya se mencionó, el nivel va a cambiar. En la Figura 2.25 se muestra un ejemplo de codificación CMI.

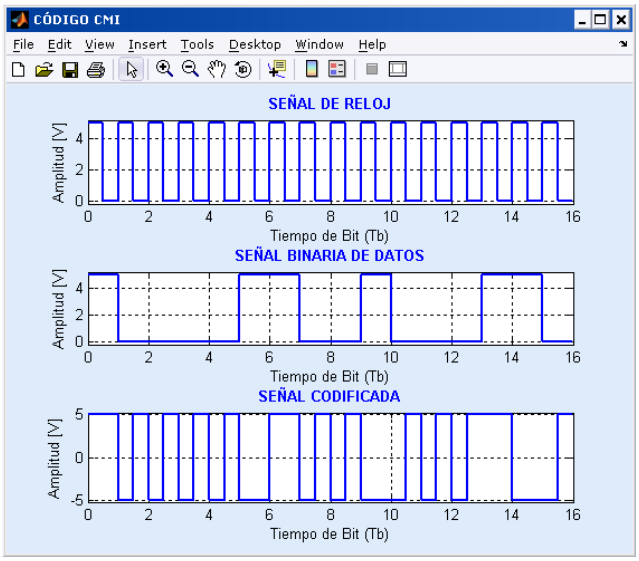


Figura 2.25 Codificación CMI en MATLAB.

En la Figura 2.25, se muestra la codificación CMI, con el primer bit de datos 1L se observa un cambio de nivel. Los 0L presentes efectivamente son codificados con un pulso positivo.

El código en cuestión presenta buen sincronismo. Es utilizado en los sistemas PDH.

Espectro de Potencia Teórico

El espectro de potencia teórico para el código CMI se presenta en la Figura 2.26. La componente DC es nula o mínima. El ancho de banda efectivo requerido por la señal codificada es el doble de la señal de datos.

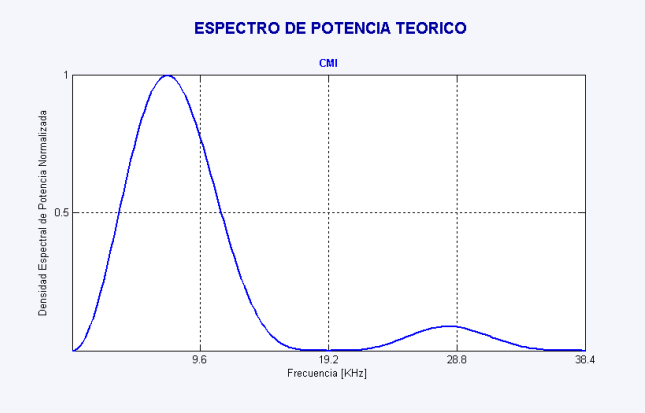


Figura 2.26 Espectro de Potencia Teórico para el código CMI.

2.2.8 CÓDIGO AMI (ALTERNATE MARK INVERSION)

Es un código bipolar. Es uno de los códigos más empleados en la transmisión digital a través de redes WAN.

Algoritmo de codificación

El proceso de codificación asigna a los 1L un estado positivo o un estado negativo en forma alternada, mientras que los 0L se transmiten como tal.

A continuación se exhibe el diagrama de flujo en la Figura 2.27.

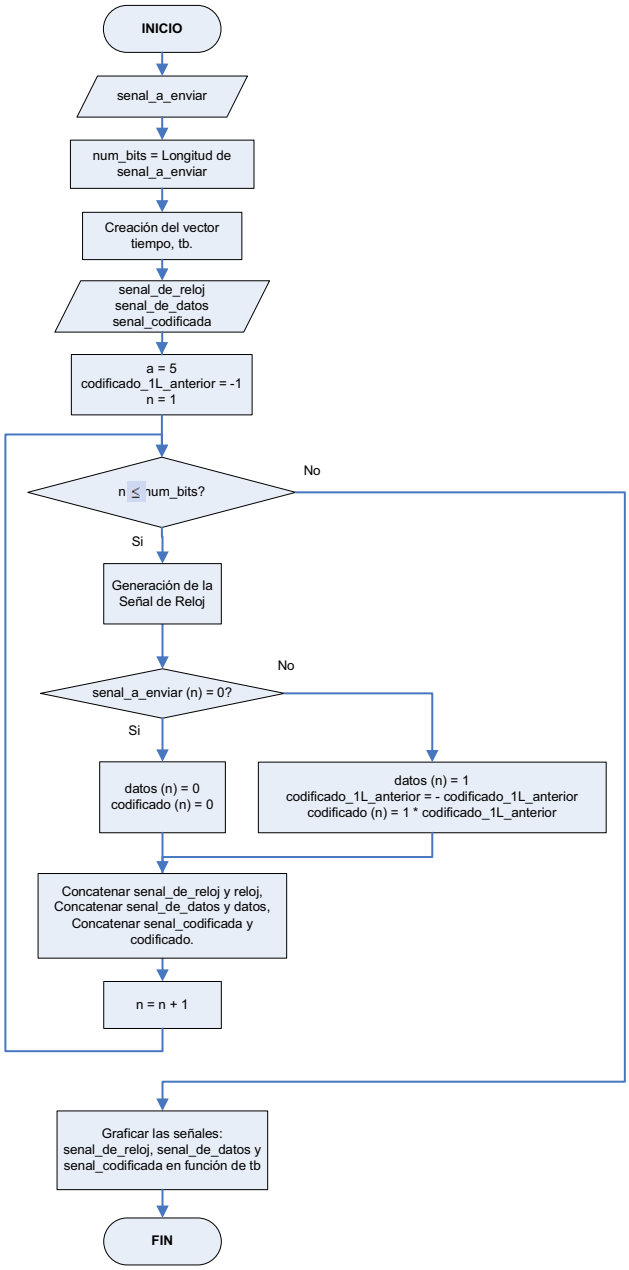


Figura 2.27 Diagrama de Flujo de la Codificación AMI en MATLAB.

Para el código AMI, también se creó una variable similar a la de CMI, de hecho, el CMI se basa en la codificación AMI en cuanto a los 1Ls se refiere. Dicha variable es *codificado_1L_anterior* y tiene el valor de referencia -1L, por lo que si el primer bit a codificar es 1L el nivel de la señal codificada va a cambiar a +1L. Como se muestra en el diagrama de flujo, los 0Ls se codifican como 0L.

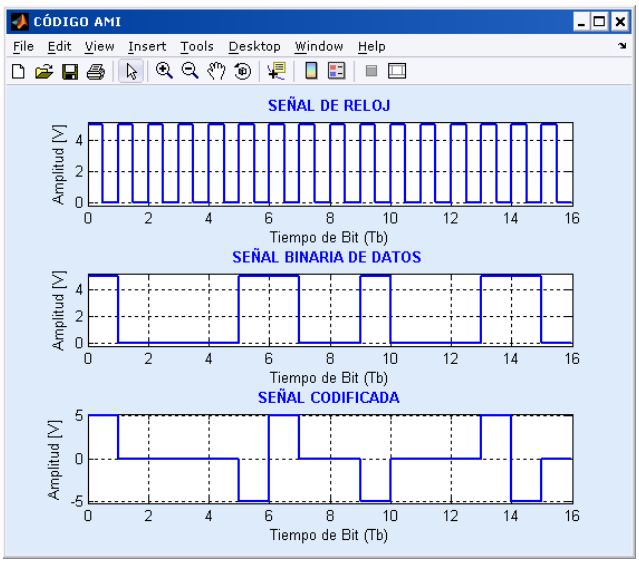


Figura 2.28 Codificación AMI en MATLAB.

De acuerdo con la regla de codificación del código bipolar en cuestión, en la Figura 2.28 se observa el cambio de nivel con el primer bit 1L a codificar.

EL código AMI presenta un buen sincronismo para los 1Ls, en tanto que con secuencias largas de 0Ls existen pérdidas de sincronismo.

Es un código normado por la UIT-T, usado en sistemas PDH y en la red ISDN.

Espectro de Potencia Teórico

Para éste código bipolar, la componente DC se aproxima a cero, el espectro de potencia se muestra en la Figura 2.29.

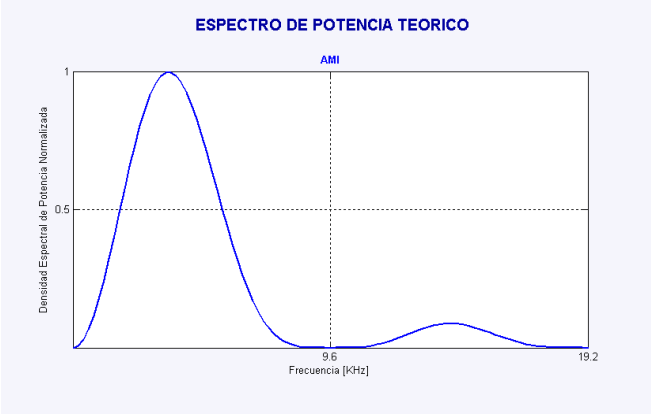


Figura 2.29 Espectro de Potencia Teórico para el código AMI.

2.2.9 CÓDIGO HDB3 (HIGH DENSITY BIPOLAR ORDER 3)

El código HDB3 pertenece a los códigos de línea llamados Técnica de Altibajos. Consisten en sustituir secuencias de bits que provocan niveles de tensión constantes por otras que garantizan la anulación de la componente continua y mejoran la sincronización del receptor. La longitud de la secuencia queda inalterada, por lo que la velocidad de transmisión de datos es la misma; además el receptor debe ser capaz de reconocer estas secuencias de datos especiales.

Algoritmo de codificación

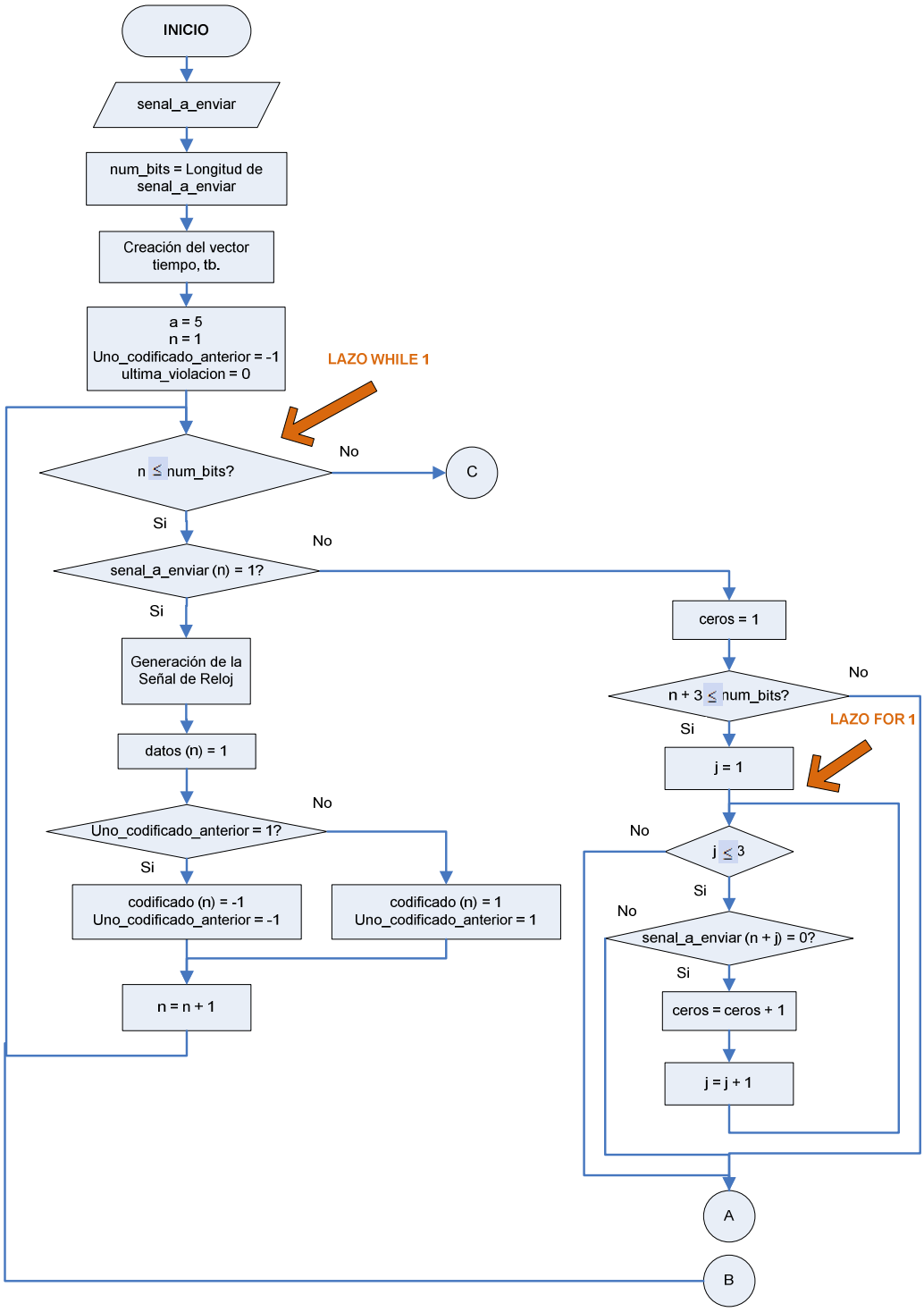
El código HDB3 no admite un número superior a 3 ceros consecutivos para una señal, y sustituye con un pulso el cuarto cero consecutivo **(pulso de violación).** Se deben tomar las precauciones necesarias para eliminar este pulso suplementario o pulso de violación en el receptor y, para ello es necesario diferenciarlo de los pulsos normales. En tal virtud, este pulso es transmitido con una polaridad idéntica a la del pulso que lo precede y se la conoce como violación de polaridad, por tanto el receptor borra la violación.

Para asegurar una componente continua nula se deben transmitir tantas violaciones positivas como negativas en forma alternada. Esta condición de alternabilidad de violaciones, para tratar de mantener una componente nula, obliga a colocar un **pulso de relleno** cuando dos violaciones de la misma polaridad ocurrirían. Esta idea se sintetiza en la regla presentada en la Tabla 2.6.

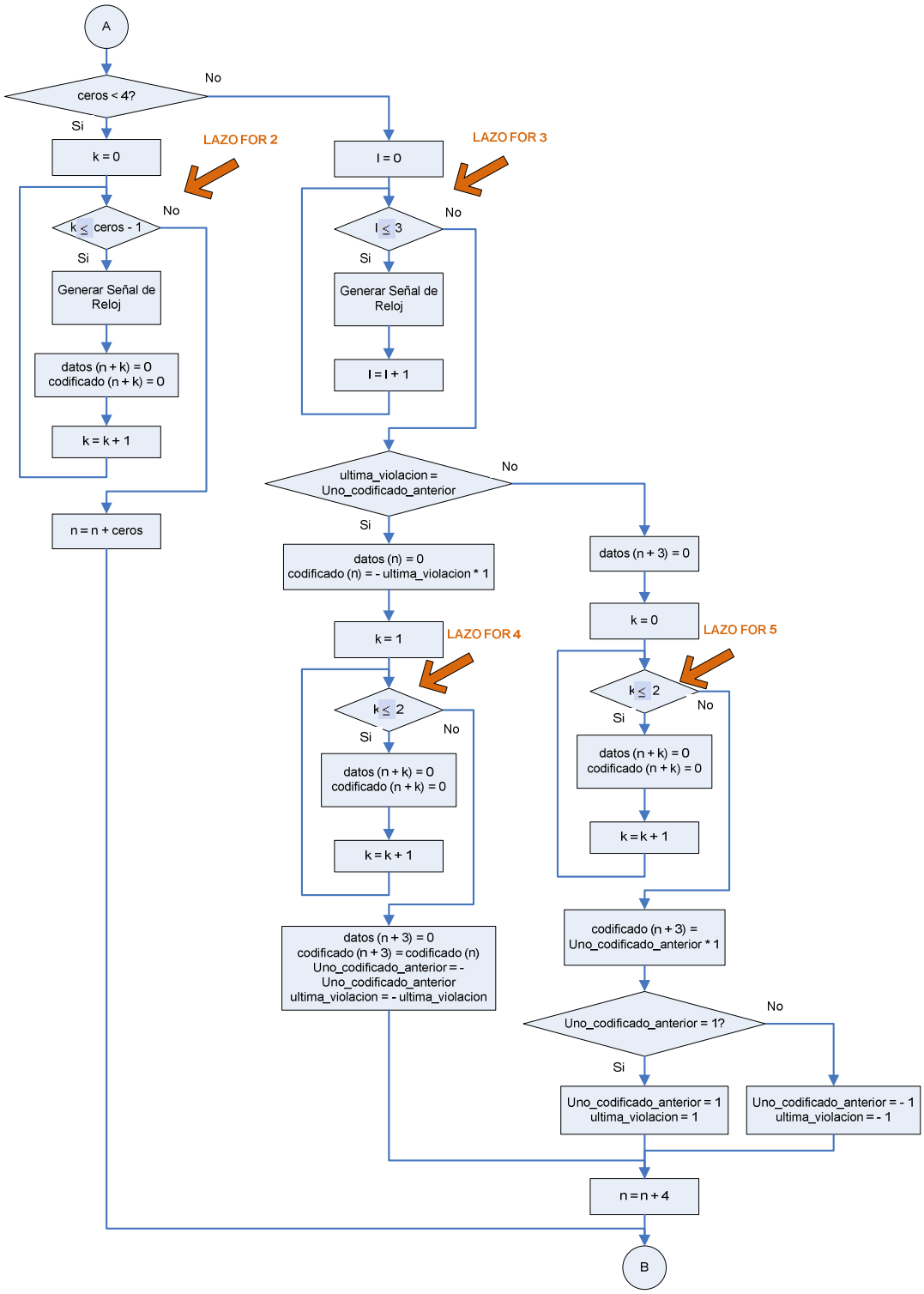
Regla de Sustitución del HDB3		
Polaridad del pulso precedente	**Número de unos lógicos desde la última sustitución.**	
	Impar	**Par**
-	000-	+00+
+	000+	-00-

Tabla 2.6 Regla de sustitución del código HDB3.

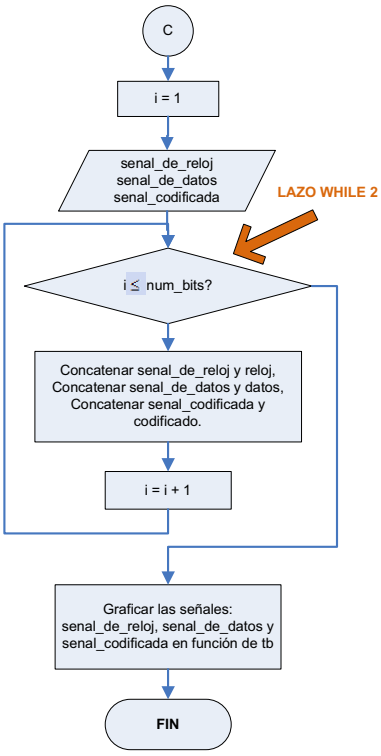
En la Figura 2.30 se exhibe el diagrama de flujo para la codificación HDB3.


INICIO
senal_a_enviar
num_bits = Longitud de senal_a_enviar
Creación del vector tiempo, tb.
a = 5
n = 1
Uno_codificado_anterior = -1
ultima_violacion = 0
LAZO WHILE 1
n ≤ num_bits?
No
C
Si
senal_a_enviar (n) = 1?
No
Si
Generación de la Señal de Reloj
datos (n) = 1
Uno_codificado_anterior = 1?
No
Si
codificado (n) = -1
Uno_codificado_anterior = -1
codificado (n) = 1
Uno_codificado_anterior = 1
n = n + 1
ceros = 1
n + 3 ≤ num_bits?
No
Si
LAZO FOR 1
j = 1
No
j ≤ 3
Si
No
senal_a_enviar (n + j) = 0?
Si
ceros = ceros + 1
j = j + 1
A
B

(a)


A
ceros < 4?
No
Si
k = 0
LAZO FOR 2
k ≤ ceros - 1
No
Si
Generar Señal de Reloj
datos (n + k) = 0
codificado (n + k) = 0
k = k + 1
n = n + ceros
l = 0
LAZO FOR 3
l ≤ 3
No
Si
Generar Señal de Reloj
l = l + 1
ultima_violacion = Uno_codificado_anterior
No
Si
datos (n) = 0
codificado (n) = - ultima_violacion * 1
k = 1
LAZO FOR 4
k ≤ 2
No
Si
datos (n + k) = 0
codificado (n + k) = 0
k = k + 1
datos (n + 3) = 0
codificado (n + 3) = codificado (n)
Uno_codificado_anterior = - Uno_codificado_anterior
ultima_violacion = - ultima_violacion
datos (n + 3) = 0
k = 0
LAZO FOR 5
k ≤ 2
No
Si
datos (n + k) = 0
codificado (n + k) = 0
k = k + 1
codificado (n + 3) = Uno_codificado_anterior * 1
Uno_codificado_anterior = 1?
No
Si
Uno_codificado_anterior = 1
ultima_violacion = 1
Uno_codificado_anterior = - 1
ultima_violacion = - 1
n = n + 4
B

(b)



(c)

Figura 2.30 Diagrama de Flujo Codificación HDB3 en MATLAB.

Para la programación del código HDB3 en MATLAB, es obligatorio establecer condiciones iniciales que permitan su ejecución, estas condiciones son *uno_codificado_anterior* y *ultima_violacion* con los valores de -1 y 0 respectivamente.

El primer lazo *while*, según el algoritmo cambia el nivel de la señal codificada cada vez que se codifica un 1L.

Si el bit a codificar es un 0L, el programa realiza un conteo del número de ceros, si este número es menor a cuatro, los ceros se codifican como tal, si es igual a cuatro, se toman en cuenta dos posibilidades: si hay violación y relleno o si sólo hay violación.

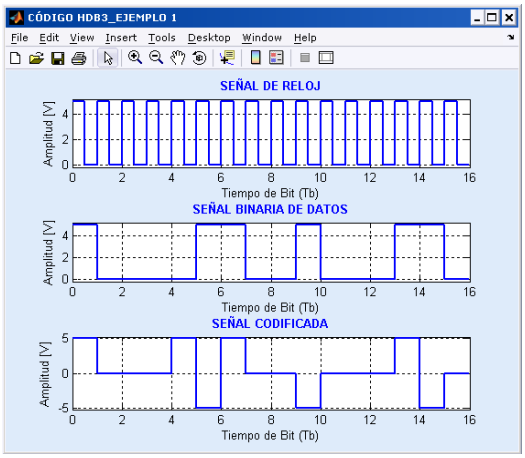
Cuando existe violación y relleno, es decir, si el signo de *ultima_violacion* es igual al signo de *uno_codificado_anterior*; el primer cero de los cuatro (bit de relleno), se codifica con un nivel inverso a la del último bit de violación, el segundo y tercer cero se codifican como 0L, y el cuarto bit (bit de violación) se codifica con un

pulso contrario al valor que se tiene en la variable *uno_codificado_anterior o ultima_violacion*. De esta manera se tiene el pulso de violación y relleno del mismo signo.

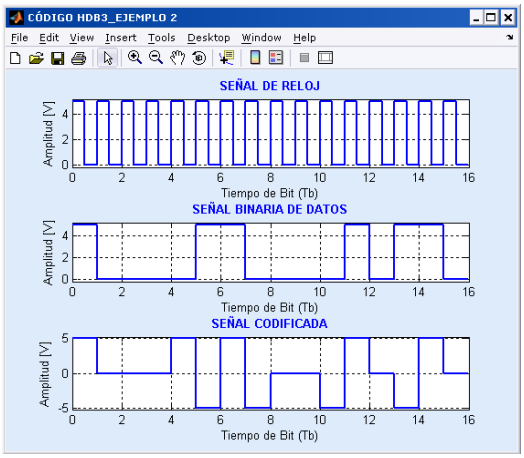
Sólo hay violación cuando las variables *ultima_violacion y uno_codificado_anterior* no son iguales, los primeros tres ceros se codifican como tal; el cuarto bit, es decir, el de violación, se codifica con el mismo signo del último bit codificado anteriormente con 1L.

El segundo lazo *while,* permite concatenar las señales de reloj, datos y codificada, para posteriormente ser graficadas.

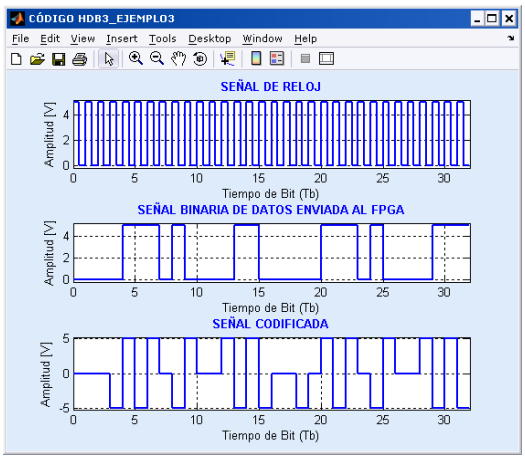
Para un mejor entendimiento del algoritmo de codificación HDB3, se ha considerado presentar tres ejemplos de simulación donde se pueda visualizar los casos de sustitución que presenta la Tabla 2.6.



(a)



(b)



(c)

Figura 2.31 Codificación HDB3 en MATLAB.

En la Figura 2.31 parte (a), la señal de datos representa los bits de los caracteres *ab*, donde se observa que hay cuatro 0Ls seguidos, en este caso solo hay violación de signo positivo, y esta constituye la primera.

En la parte (b), se tiene como señal de datos los bits de los caracteres *ah,* hay dos bloques de cuatro 0Ls consecutivos; el primer bloque solo presenta una violación de signo positivo, el segundo tiene bit de relleno y violación, ambos de signo negativo, ya que la violación anterior fue positiva, y el último 1L codificado es positivo.

En la parte (c) se codifica la palabra "*papá"*, donde se tienen cuatro bloques de cuatro 0Ls consecutivos. El primer bloque presenta sólo una violación de signo negativo por ser la primera; en el segundo, la violación sería negativa de acuerdo al último 1L anterior codificado, por lo tanto hay violación y relleno de signo positivo; en el tercero, al igual que en el caso anterior, hay relleno y violación de signo negativo; y por último, el cuarto grupo de 0Ls muestra pulsos de violación y relleno de signo positivo.

Este código es adecuado para la transmisión a altas velocidades, además posee una buena sincronización.
Ofrece detección de errores ocasionales observando la alternabilidad de las violaciones, ya que la introducción de un simple error causa una discontinuidad en la secuencia de alternabilidad de las violaciones. Se usa en los sistemas PDH.

Espectro de Potencia Teórico

El espectro de potencia para el código HDB3 se ostenta en la Figura 2.32. Se observa una componente DC mínima.

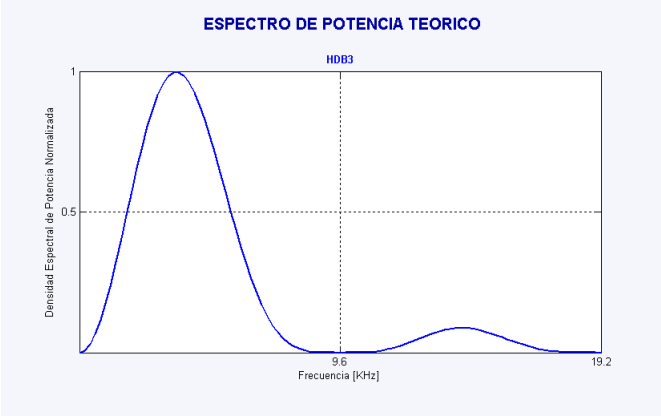


Figura 2.32 Espectro de Potencia Teórico para el código HDB3.

2.2.10 CÓDIGO MLT-3 (MULTI LEVEL TRANSMIT)

El código MLT-3 utiliza tres niveles de voltaje, requiere menos ancho de banda efectivo que la mayoría de los otros códigos binarios o ternarios.

Algoritmo de codificación

La codificación MLT-3 produce una transición para cada bit de datos 1L, usa tres niveles de voltaje +V, -V, y 0. Los niveles de voltaje para los 1L son seleccionados de manera secuencial (+V, 0, -V, 0, +V). El bit 0L mantiene el nivel de voltaje del estado anterior. La Figura 2.33 presenta el diagrama de flujo de la programación en MATLAB de este código.

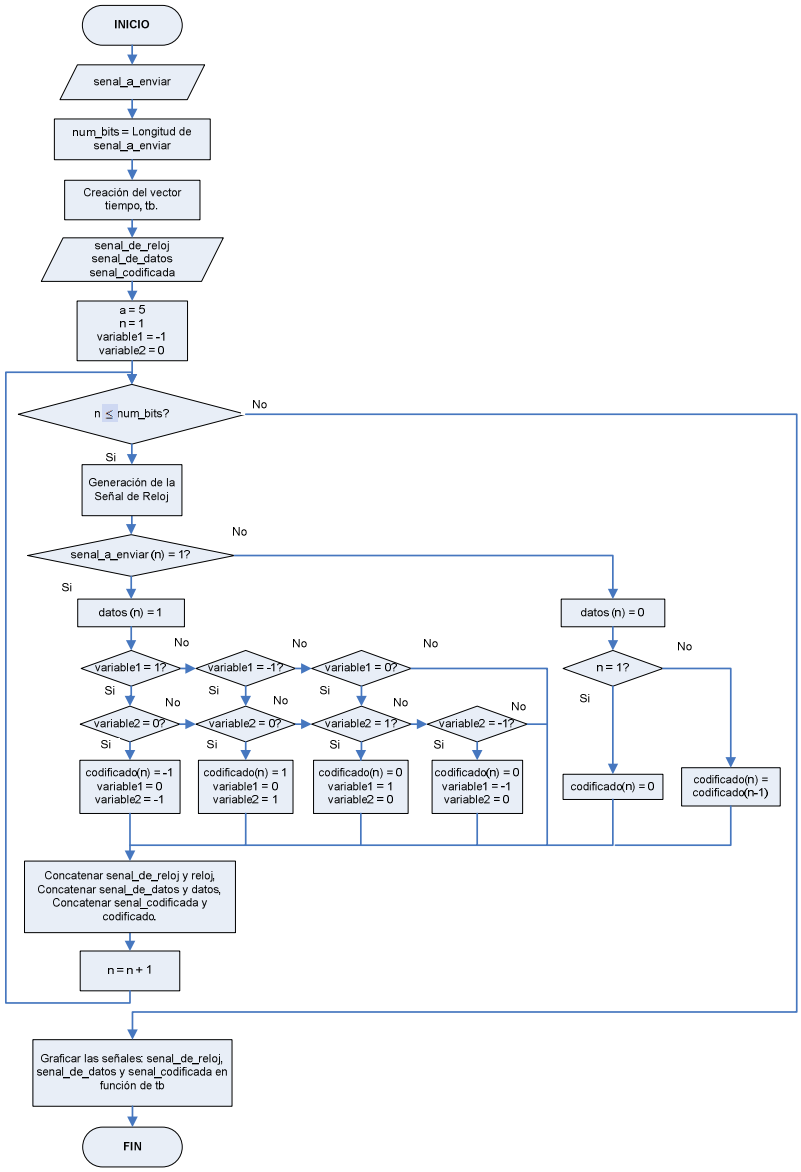
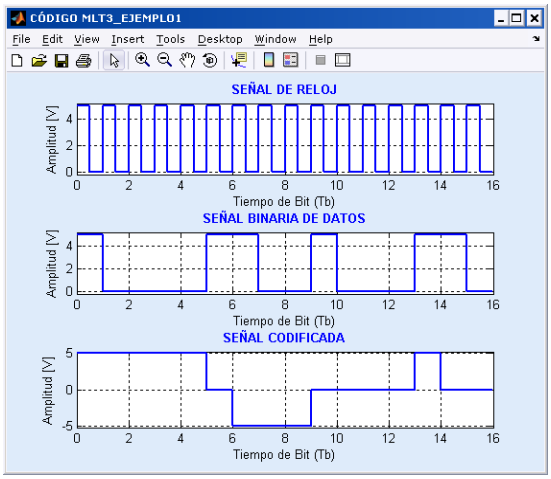


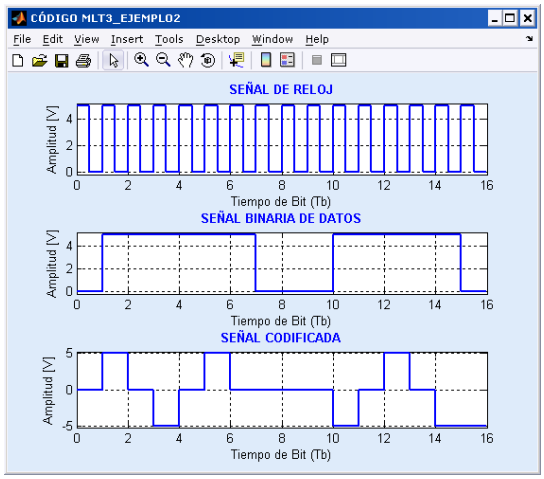
Figura 2.33 Diagrama de Flujo de la Codificación MLT3 en MATLAB.

Para este código, se hace uso de dos variables para poder dar inicio a la codificación MLT3, estas son *variable1* y *variable2* con valores de -1 y 0, que representan el penúltimo y último estado anterior respectivamente. Si el primer bit de la señal de datos es 1L, el valor codificado que le corresponde sería de + 5 [V], ya que la codificación se hace de manera secuencial, es decir, -5, 0, +5. En la programación se consideran todas las combinaciones de los valores que pueden tomar estas variables dentro del lazo *while*, y a la vez se queda almacenada la combinación de valores para el siguiente 1L a ser codificado. En cuanto a los bits

de datos 0L, la señal codificada mantendrá el nivel. En la Figura 2.34 se muestran dos ejemplos de la simulación de la codificación MLT3.



(a)



(b)

Figura 2.34 Codificación MLT-3 en MATLAB.

En la Figura 2.34 (a) se puede observar que se cumple satisfactoriamente con el algoritmo de codificación MLT3, considerando los valores iniciales que se usaron en la programación, se codifica el primer 1L de datos con +5 [V] y el 0L mantiene el nivel. En la parte (b) se puede visualizar de mejor manera la codificación de los 1Ls, ya que la señal binaria de datos está compuesta por dos secuencias largas de 1Ls.

Para este código, se produce pérdida de sincronismo en secuencias largas de 0Ls.

Es utilizado en algunas redes de área local de alta velocidad como en el caso de las redes 100Base-TX.

Espectro de Potencia Teórico

En la Figura 2.35 se exhibe el espectro de potencia teórico para el código MLT3. Este código presenta una componente DC considerable.

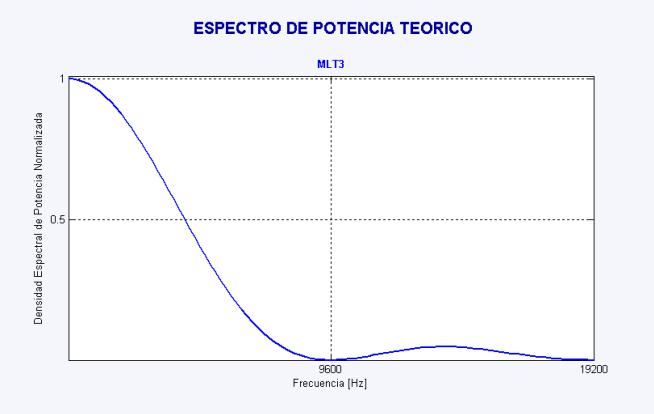


Figura 2.35 Espectro de Potencia Teórico para el código MLT3.

CAPÍTULO III

DESARROLLO DE LA INTERFAZ GRÁFICA

3.1 DESCRIPCIÓN DE LA INTERFAZ GRÁFICA

En este capítulo se detalla la realización de la interfaz gráfica, la misma que tiene como objetivo que el usuario disfrute de un entorno amigable ante el ingreso de la secuencia de caracteres a ser codificada y el tipo de código de línea a implementarse en el FPGA. Estos caracteres se transmiten del PC al FPGA (Dispositivo que se encarga de la codificación y decodificación) en forma de bits a través del puerto RS-232 (Puerto de Comunicación de la Spartan-3E Starter Kit Board).
Posterior a la decodificación, el PC recibe los caracteres decodificados desde el FPGA, entonces la interfaz gráfica presenta: la secuencia de caracteres antes de ser codificados enviada al FPGA, la señal en forma de bits enviada, la secuencia de caracteres decodificados recibida desde el FPGA y la señal en forma de bits recibida.

3.2 ELECCIÓN DEL PROGRAMA IDÓNEO PARA EL DESARROLLO DE LA INTERFAZ GRÁFICA

La interfaz gráfica se desarrolló en el software MATLAB versión 7.1, porque presenta facilidades para el desarrollo de Interfaces Gráficas.

GUIDE (Ambiente de desarrollo de Interfaces Gráficas de Usuario) de MATLAB combina un ambiente gráfico con la programación en código.
El ambiente gráfico es un entorno que permite diseñar la Interfaz simplemente seleccionando y arrastrando al área de diseño los componentes, como: botones, barras de desplazamiento, campos de edición de texto, textos estáticos, menús, campos de ingreso de gráficos.

La programación en código se realiza en los archivos .m que GUIDE genera automáticamente y que controlan cómo opera la Interfaz Gráfica. Utilizando el editor de archivos .m se puede añadir código para que cada componente desarrolle las funciones requeridas.

Además, se seleccionó MATLAB porque permite: transmisión y recepción de señales a través de la interfaz RS-232 (interfaz de comunicación que posee la Tarjeta de entrenamiento), análisis de datos, extracción de información y visualización de resultados a través de la interfaz gráfica.

3.3 DIAGRAMA DE BLOQUES DE LA INTERFAZ GRÁFICA

Toda la interfaz consta de varias GUIs (Interfaces Gráficas de Usuario) de MATLAB, las mismas que se interconectan como se observa en la Figura 3.1.

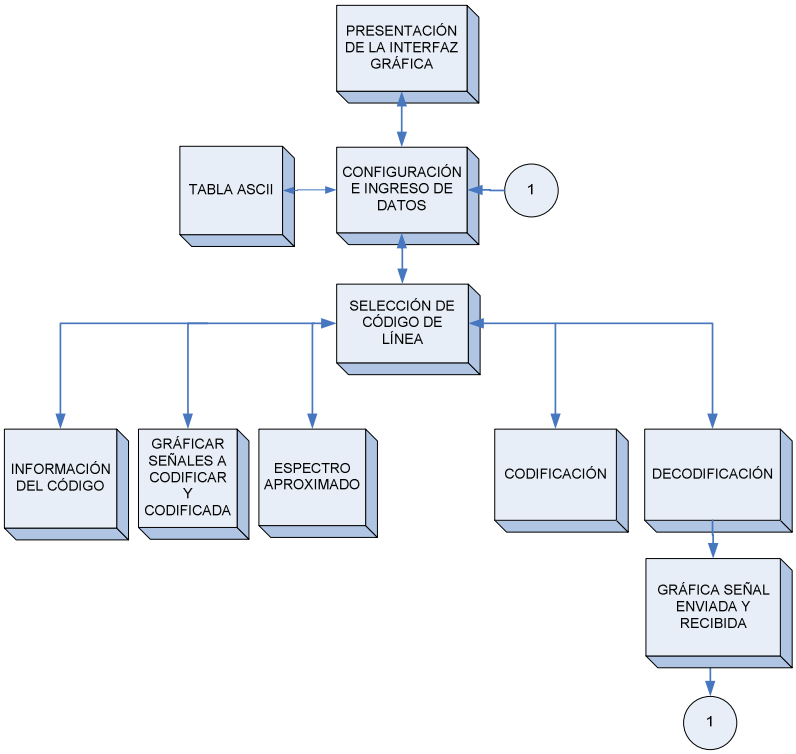


Figura 3.1 Diagrama de Bloques de la Interfaz Gráfica.

3.4 DESARROLLO DE LAS GUIs (INTERFACES GRÁFICAS DE USUARIO)

A continuación se detalla el propósito, la visualización, y los diagramas de flujo de la programación en código, de cada una de las Interfaces Gráficas de Usuario.

3.4.1 PRESENTACIÓN DE LA INTERFAZ GRÁFICA

Al inicializar la Interfaz Gráfica, la primera ventana que se visualiza es la de Presentación, tal como se muestra en la Figura 3.2.

Esta ventana posee dos botones: Continuar (Abre la GUI de Configuración e Ingreso de Datos) y Salir del software.

Figura 3.2 Presentación de la Interfaz Gráfica.

3.4.2 CONFIGURACIÓN E INGRESO DE DATOS

Esta GUI permite al usuario seleccionar las opciones de configuración e ingresar los datos (caracteres) a ser codificados y decodificados en el FPGA, tal como lo muestra la Figura 3.3.

Figura 3.3 Configuración e Ingreso de Datos.

Las opciones de configuración permiten escoger: el puerto de comunicación (COM) que el PC va a utilizar para la transmisión y recepción de datos (debe ser uno de los que se observa en la ventana de Administración de Dispositivos y al mismo tiempo el que está conectado a la Tarjeta de Entrenamiento), y la velocidad de transmisión entre las que se debe elegir: 2400, 4800, 9600 o 19200 bps para que el reloj de la Tarjeta de Entrenamiento funcione a la frecuencia adecuada.

Al aceptar las dos opciones de configuración se establece el puerto de comunicación seleccionado, por el mismo que se envía una señal binaria identificando la velocidad a la que se transmitirán los datos.
En el campo Introducción de Datos, el usuario debe ingresar una secuencia de caracteres, cada uno de los cuales se transforma a su equivalente binario de 8 bits de acuerdo a la tabla ASCII para su posterior transmisión a la Spartan-3E Starter Kit Board.

El botón Tabla ASCII abre la GUI correspondiente, con el objetivo de verificar que los bits que aparecen en esta interfaz gráfica de usuario pertenecen a los caracteres ingresados.

El diagrama de flujo de la programación en código de esta GUI lo podemos visualizar en la Figura 3.4.

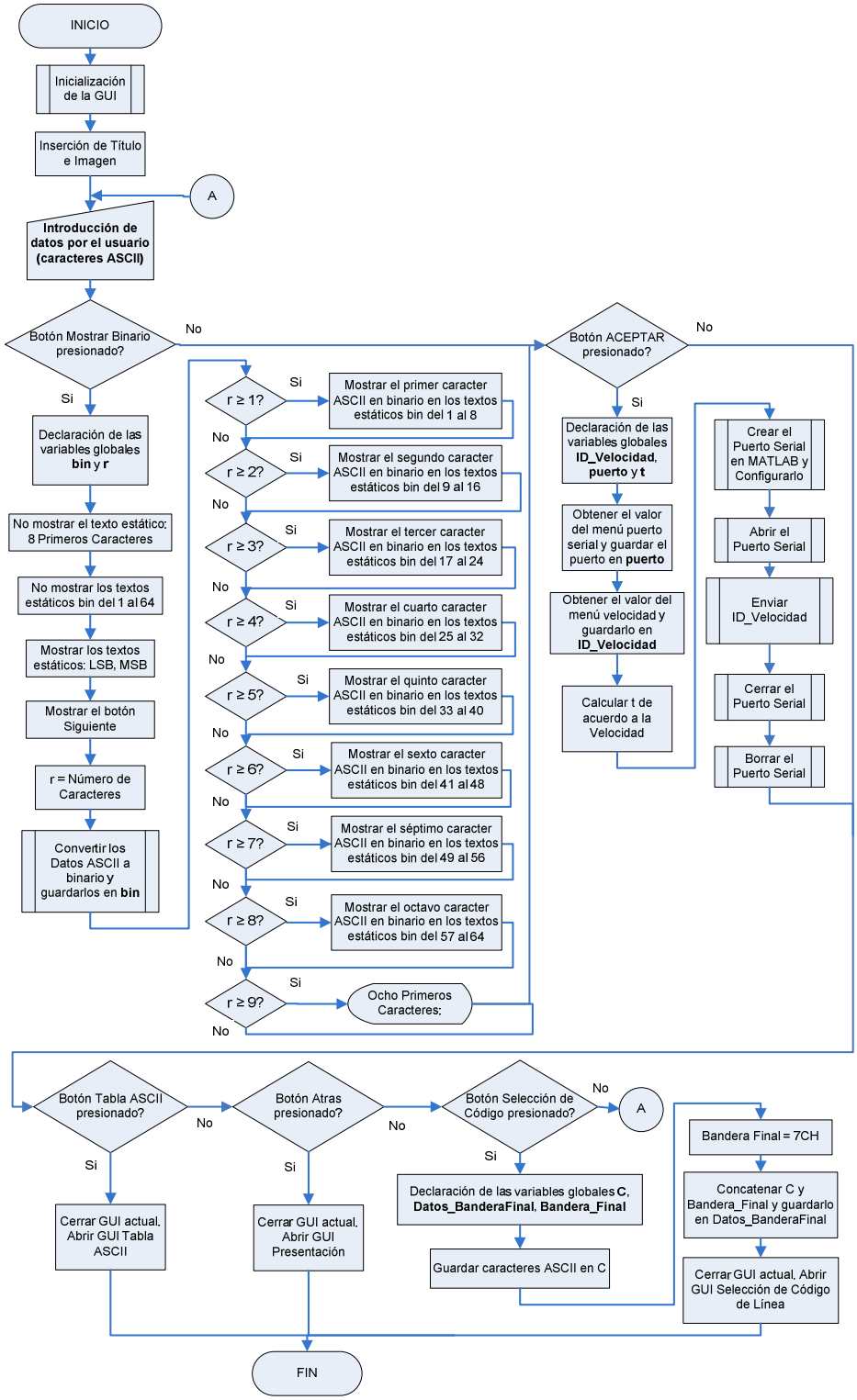


Figura 3.4 Diagrama de flujo de la programación de Configuración e Ingreso de Datos.

El botón Mostrar Binario, permite observar los 8 bits correspondientes a cada carácter que se ingresó, de izquierda a derecha desde el LSB (Bit menos significativo) al MSB (Bit más significativo). Si se ingresan más de 8 caracteres se presentarán sólo los 64 primeros bits por razones de visualización y espacio. Solamente al presionar Mostrar Binario se visualiza el botón Selección de Código que nos lleva a la GUI Selección de Código de Línea.

3.4.2.1 Comunicación del PC con la Spartan-3E Starter Kit Board

En el Capítulo 1 se puede observar que la Spartan-3E Starter Kit Board tiene un puerto USB orientado únicamente a depuración y programación del FPGA, mas no a transmisión y recepción de datos. Además posee dos puertos seriales RS-232, uno para que funcione como DTE y otro como DCE. Razón principal por la que la comunicación del computador a través de la interfaz gráfica con la Spartan-3E Starter Kit Board se la realiza mediante el interfaz RS-232.

En Windows, para conocer los puertos de comunicación (COM) que se encuentran habilitados en el PC para comunicación serial, se selecciona Administrar después de hacer un clic derecho en Mi PC, tal como lo muestra la Figura 3.5

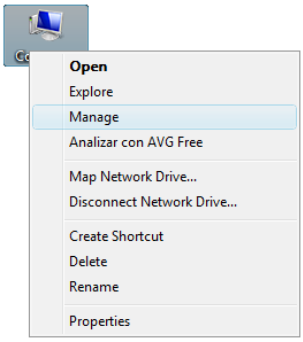


Figura 3.5 Ingreso a la ventana Administración de Dispositivos desde Mi PC.

Se visualiza la pantalla de Administración de Dispositivos, tal como se muestra en la Figura 3.6.

Con respecto al número del Puerto de comunicación, este varía dependiendo de la cantidad de puertos habilitados en el computador. El puerto de comunicación

habilitado que se conecta a la Spartan-3E Starter Kit Board debe coincidir con el que el usuario selecciona en la Interfaz Gráfica, caso contrario no existirá comunicación con la Tarjeta de Entrenamiento.

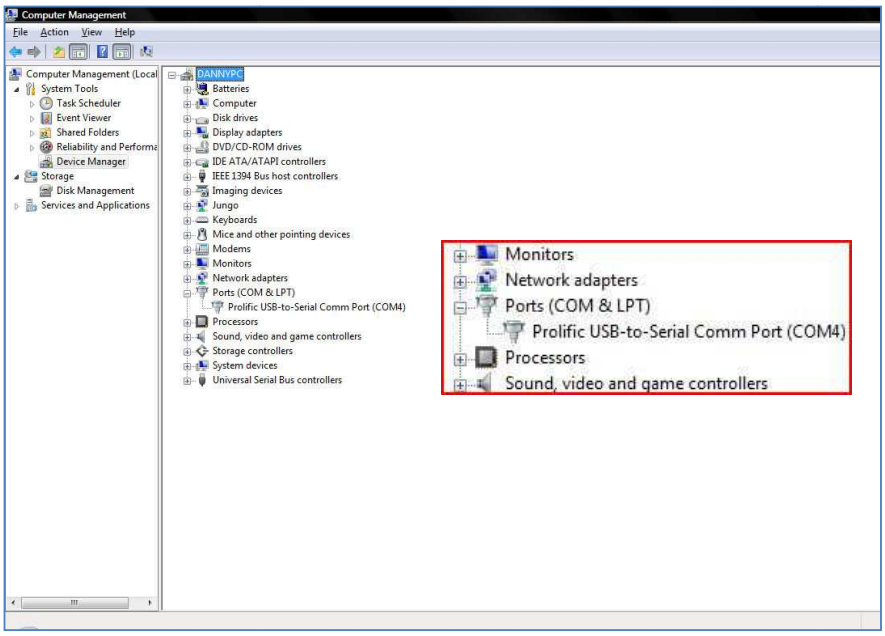


Figura 3.6 Ventana de Administración de Dispositivos de Windows Vista.

Actualmente los PCs no se fabrican con puertos seriales, caso en el cual se puede conectar a un puerto USB del computador un cable USB a RS232, dispositivo que también se lo observa como puerto de comunicación en la ventana de Administración de Dispositivos en el caso de que se lo utilice.

En la Figura 3.7 se muestra el esquema de comunicación entre el Computador y la Spartan-3E Starter Kit Board. No obstante, en el Capítulo 4 se detalla el modo de transmisión, el formato de los caracteres, y los códigos de los procesos VHDL para que la tarjeta de entrenamiento se pueda comunicar en forma serial con el computador.

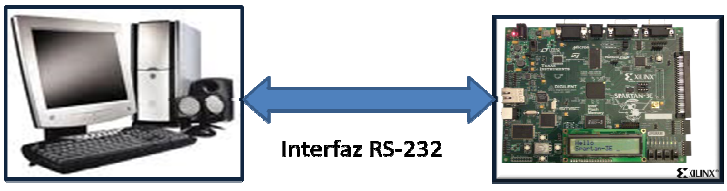


Figura 3.7 Esquema de Comunicación entre el PC y la Spartan-3E Starter Kit Board.

3.4.3 TABLA ASCII

Esta Interfaz Gráfica de Usuario tiene como propósito que el usuario pueda comprobar que los bits que se envían y reciben corresponden a los caracteres ingresados.

La GUI correspondiente se muestra en la Figura 3.8

Bits				5	0	1	0	1	0	1	0	1
				6	0	0	1	1	0	0	1	1
				7	0	0	0	0	1	1	1	1
1	2	3	4									
0	0	0	0		NUL	DEL	SP	0	@	P	‘	p
1	0	0	0		SOH	DC1	¡	1	A	Q	a	q
0	1	0	0		STX	DC2	“	2	B	R	b	r
1	1	0	0		ETX	DC3	#	3	C	S	c	s
0	0	1	0		EOT	DC4	%	4	D	T	d	t
1	0	1	0		ENQ	NAK	$	5	E	U	e	u
0	1	1	0		ACK	SYN	&	6	F	V	f	v
1	1	1	0		BEL	ETB	‘	7	G	W	g	w
0	0	0	1		BS	CAN	(	8	H	X	h	x
1	0	0	1		HT	EM	)	9	I	Y	i	y
0	1	0	1		LF	SUB	*	:	J	Z	j	z
1	1	0	1		VT	ESC	+	;	K	[	k	{
0	0	1	1		FF	FS	,	<	L	\	l	|
1	0	1	1		CR	GS	-	=	M	]	m	}
0	1	1	1		SO	RS	.	>	N	^	n	~
1	1	1	1		SI	US	/	?	O	-	o	DEL

Figura 3.8 Tabla ASCII.

3.4.4 SELECCIÓN DE CÓDIGO DE LÍNEA

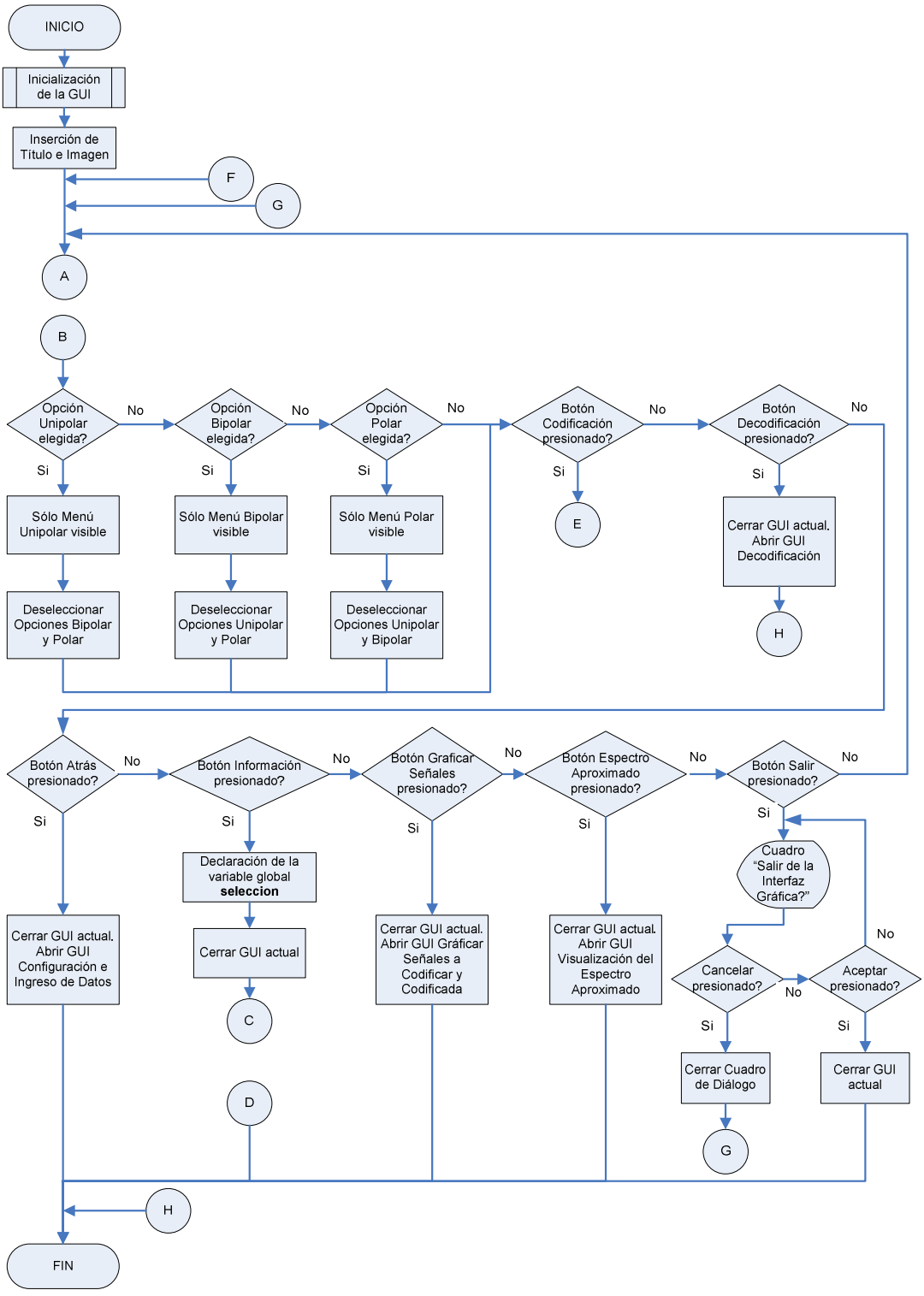
En esta ventana se debe escoger el código de línea a implementarse en el FPGA, para esto se han separado los códigos unipolares, polares y bipolares en menús diferentes.

Figura 3.9 Selección del Código de Línea.

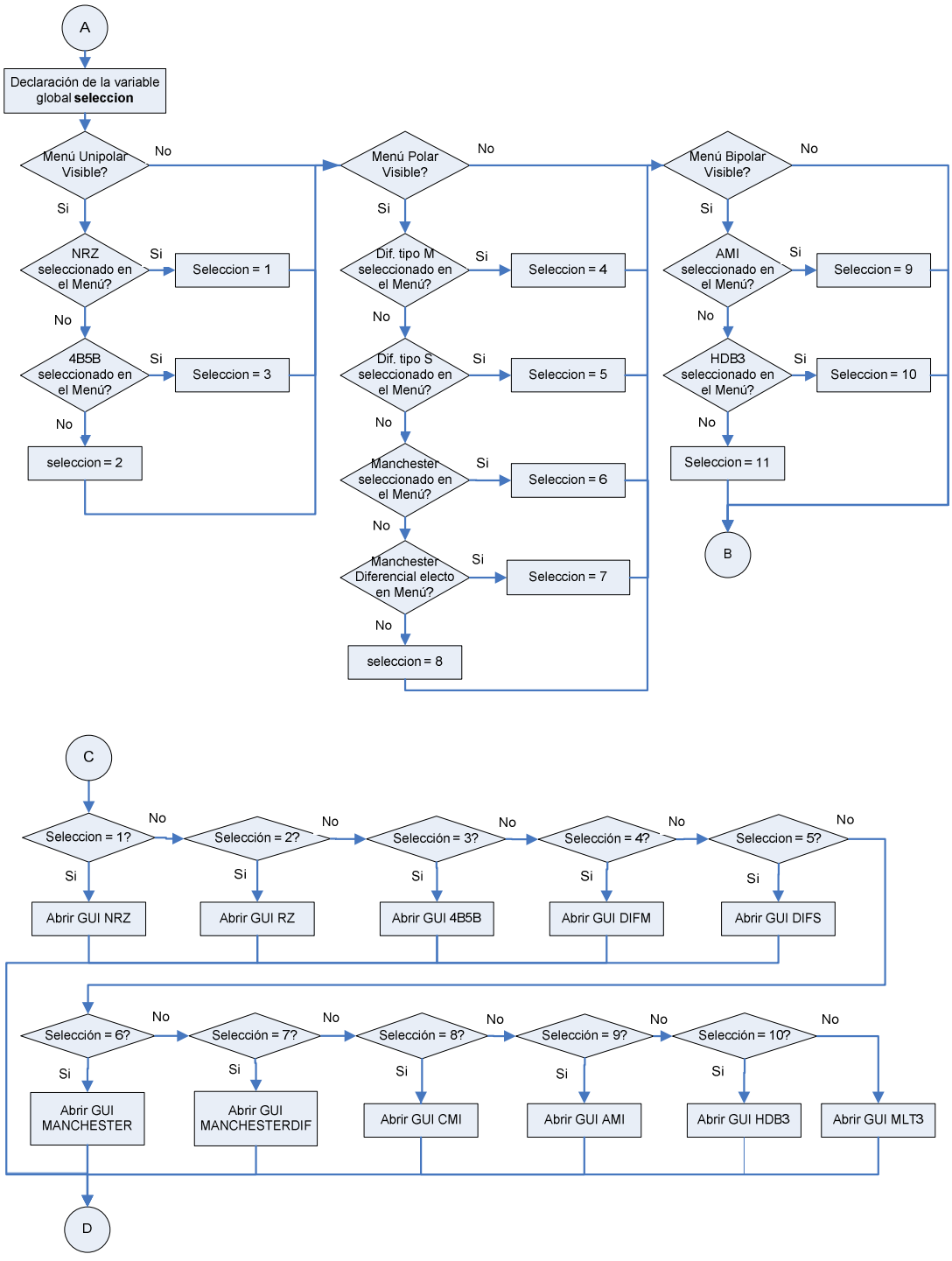
Esta Interfaz Gráfica de Usuario se muestra en la Figura 3.9 y el diagrama de flujo de la programación en código en la Figura 3.10.

Existen tres marcos específicos de botones: Generales, Simulación y Hardware.

Una vez seleccionado el Código de Línea; en **Generales** se pueden abrir las GUIs: Configuración e Ingreso de Datos con el botón Atrás, Información del Código seleccionado, y Visualización del Espectro Teórico; en **Simulación** se puede abrir la GUI Graficar Señales a Codificar y Codificada. Los dos botones de **Hardware** se describen a continuación.


INICIO
Inicialización de la GUI
Inserción de Título e Imagen
F
G
A
B
Opción Unipolar elegida?
No
Opción Bipolar elegida?
No
Opción Polar elegida?
No
Botón Codificación presionado?
No
Botón Decodificación presionado?
No
Si
Sólo Menú Unipolar visible
Sólo Menú Bipolar visible
Sólo Menú Polar visible
E
Cerrar GUI actual. Abrir GUI Decodificación
H
Deseleccionar Opciones Bipolar y Polar
Deseleccionar Opciones Unipolar y Polar
Deseleccionar Opciones Unipolar y Bipolar
Botón Atrás presionado?
No
Botón Información presionado?
No
Botón Graficar Señales presionado?
No
Botón Espectro Aproximado presionado?
No
Botón Salir presionado?
No
Si
Declaración de la variable global seleccion
Cerrar GUI actual
C
Cuadro "Salir de la Interfaz Gráfica?"
Cancelar presionado?
No
Aceptar presionado?
No
Cerrar GUI actual. Abrir GUI Configuración e Ingreso de Datos
Cerrar GUI actual. Abrir GUI Gráficar Señales a Codificar y Codificada
Cerrar GUI actual. Abrir GUI Visualización del Espectro Aproximado
Cerrar Cuadro de Diálogo
Cerrar GUI actual
G
D
H
FIN

(a)



(b)

Figura 3.10 Diagrama de Flujo de la programación de la GUI Selección de Código de Línea.

3.4.4.1 Botón Enviar Datos al FPGA para la Codificación

Al presionar este botón, el PC envía una trama binaria (su objetivo y formato se detalla en el Capítulo 4) que contiene un número binario que identifica el código de línea seleccionado que se desea implementar en el FPGA, seguido de los datos a ser codificados. Esta trama se envía por el puerto RS-232 del computador.

El diagrama de flujo de la programación en código de este botón se muestra en la Figura 3.11. Cabe recalcar que este botón es parte de la Interfaz Selección de Código de Línea, por ende es un segmento del diagrama de flujo de la Figura 3.10.

3.4.4.2 Botón Ver datos recibidos Decodificados

Al presionar este botón MATLAB toma los datos binarios que llegan por el pin de recepción del puerto RS-232 y los muestra en las GUIs: Decodificación, y Gráfica de Señal Enviada y Recibida.

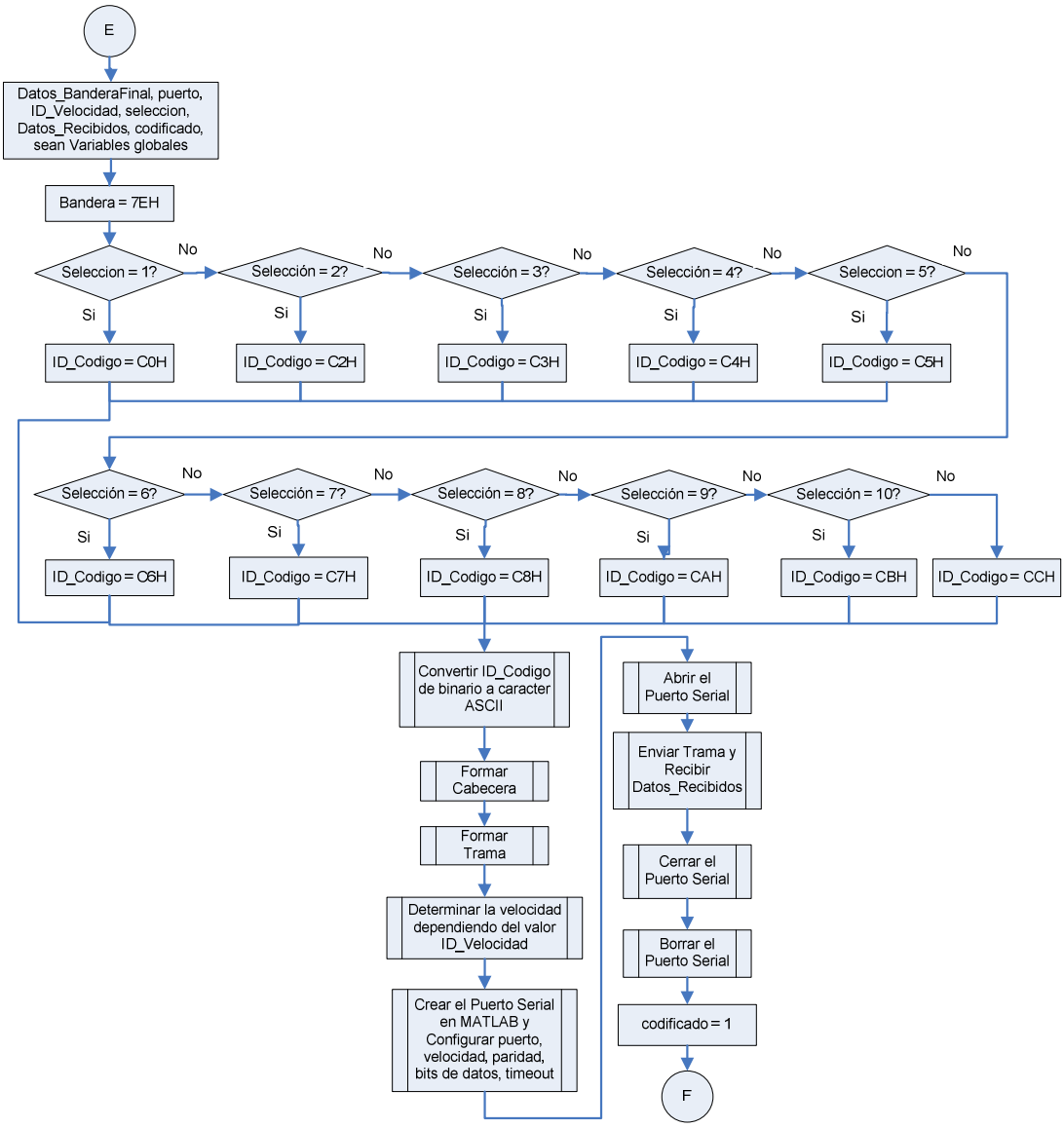


Figura 3.11 Diagrama de Flujo de la programación del Botón Enviar Datos al FPGA para la Codificación.

3.4.5 INFORMACIÓN DE CÓDIGO DE LÍNEA

Esta Interfaz Gráfica de Usuario, presenta información de cada uno de los códigos de línea que se implementan en este proyecto, en esta reseña se detallan reglas de codificación, características y aplicaciones.

La GUI correspondiente a la Información del código de línea 4B5B, se visualiza en la Figura 3.12. Mientras que el diagrama de flujo de la programación se presenta en la Figura 3.13.

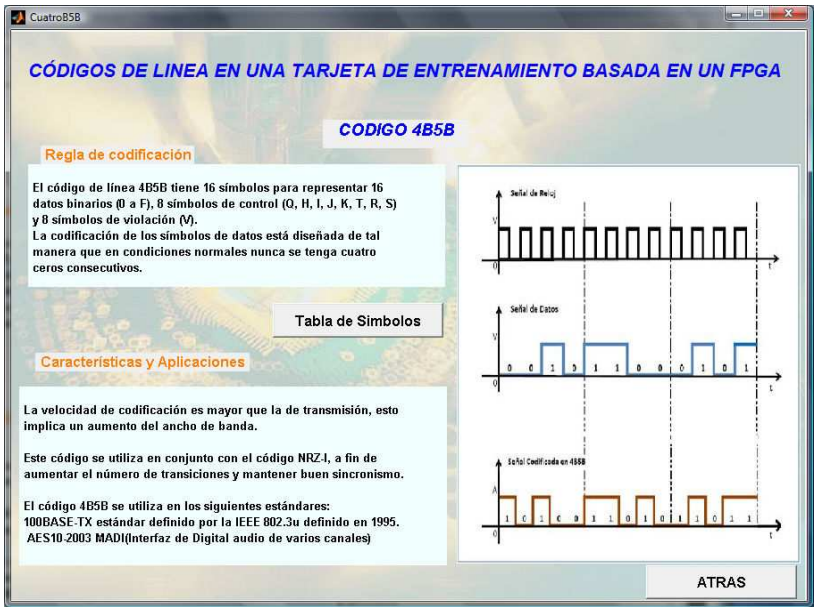


Figura 3.12 Información del Código 4B5B.

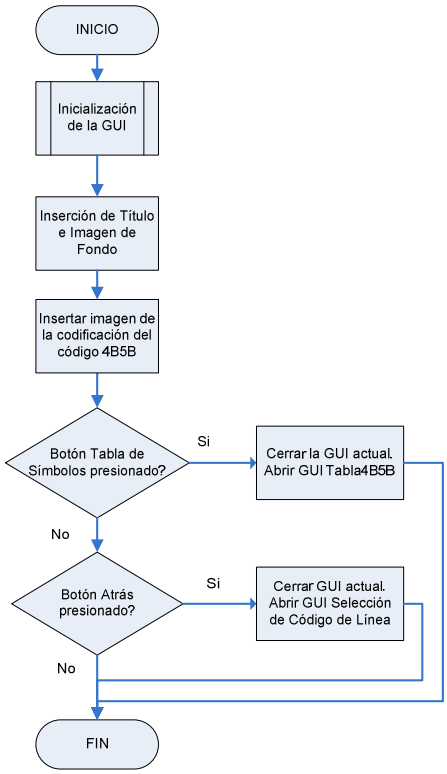


Figura 3.13 Diagrama de flujo de Información del Código 4B5B.

3.4.6 GRAFICAR SEÑALES A CODIFICAR Y CODIFICADA

Esta ventana presenta tres gráficas: la señal de reloj, la señal a codificar (originada como resultado de ensamblar los identificadores de inicio y fin de trama y todos los bits de datos producidos por la secuencia de caracteres introducida), y la señal codificada dependiendo del código de línea seleccionado en la GUI Selección de Código de Línea. Estos gráficos se los realiza a través de programación en MATLAB, cuyo diagrama de flujo se muestra en la Figura 3.15. La GUI correspondiente se muestra en la Figura 3.14.

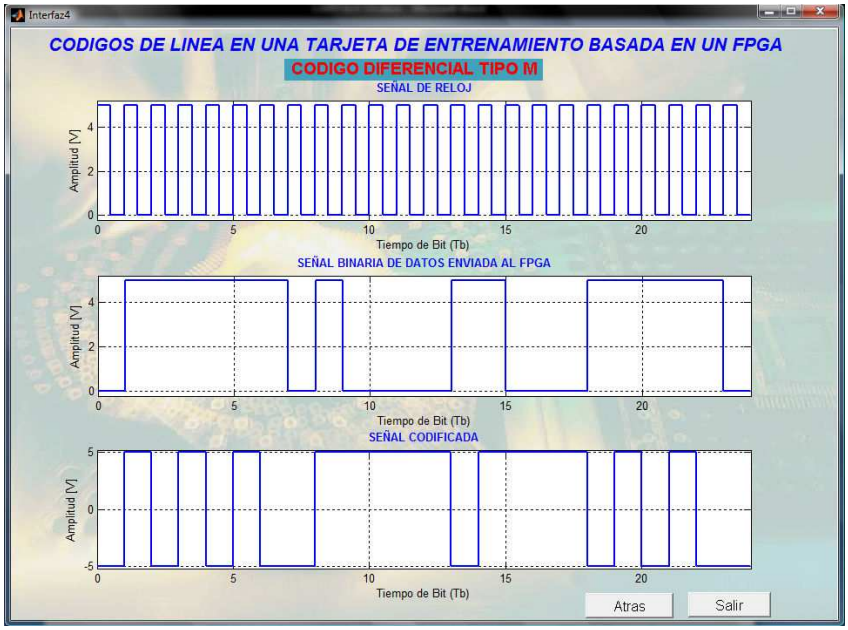
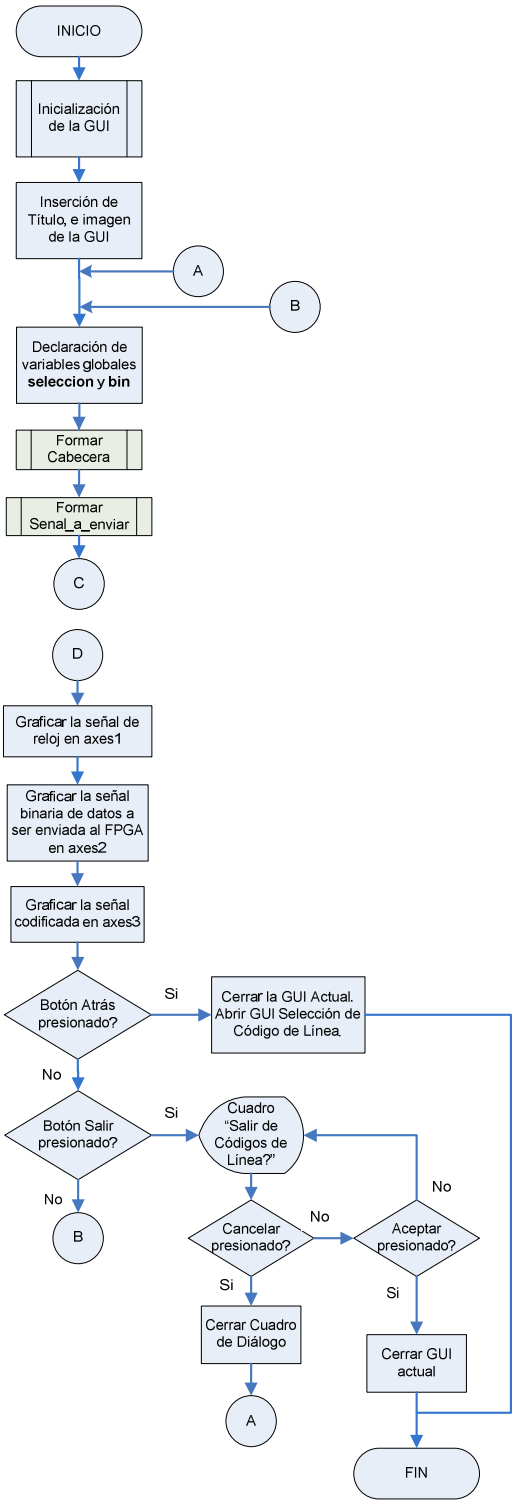
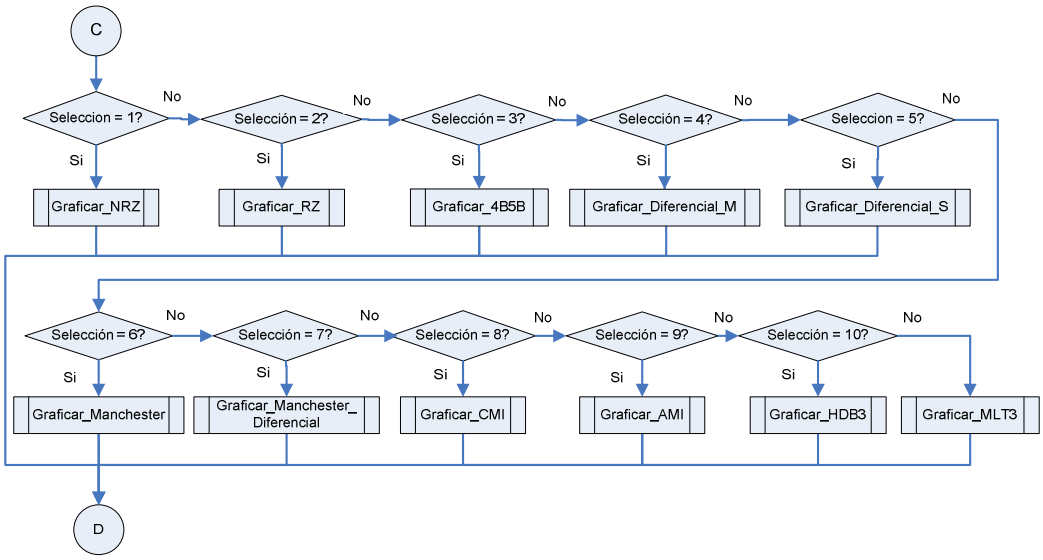


Figura 3.14 Gráfica de la Señal de Reloj, Señal a Codificar y Señal Codificada.

La Figura 3.14 muestra el caso específico cuando el carácter introducido por el usuario es “a” y el código de línea seleccionado es Diferencial Tipo M. Se puede comprobar que la señal binaria de datos a ser codificada en el FPGA es: el identificador de inicio (7EH), los datos (61H), y el identificador de fin de trama (7CH). Se verifica también la correcta codificación de acuerdo a las reglas de este código de línea.


INICIO
Inicialización de la GUI
Inserción de Título, e imagen de la GUI
A
B
Declaración de variables globales seleccion y bin
Formar Cabecera
Formar Senal_a_enviar
C
D
Graficar la señal de reloj en axes1
Graficar la señal binaria de datos a ser enviada al FPGA en axes2
Graficar la señal codificada en axes3
Botón Atrás presionado?
Si
Cerrar la GUI Actual. Abrir GUI Selección de Código de Línea.
No
Botón Salir presionado?
Si
Cuadro "Salir de Códigos de Línea?"
No
B
Cancelar presionado?
No
Aceptar presionado?
No
Si
Si
Cerrar Cuadro de Diálogo
Cerrar GUI actual
A
FIN

(a)



(b)

Figura 3.15 Diagrama de Flujo de Graficar Señales a Codificar en el FPGA y Codificada.

3.4.7 VISUALIZACIÓN DEL ESPECTRO TEÓRICO

En esta Interfaz Gráfica se programó una figura que permite visualizar la Densidad Espectral de Potencia teórica de la Señal Codificada dependiendo del Código de Línea que el usuario escoja en la GUI Selección de Código de Línea. Esta ventana se muestra en la Figura 3.16.

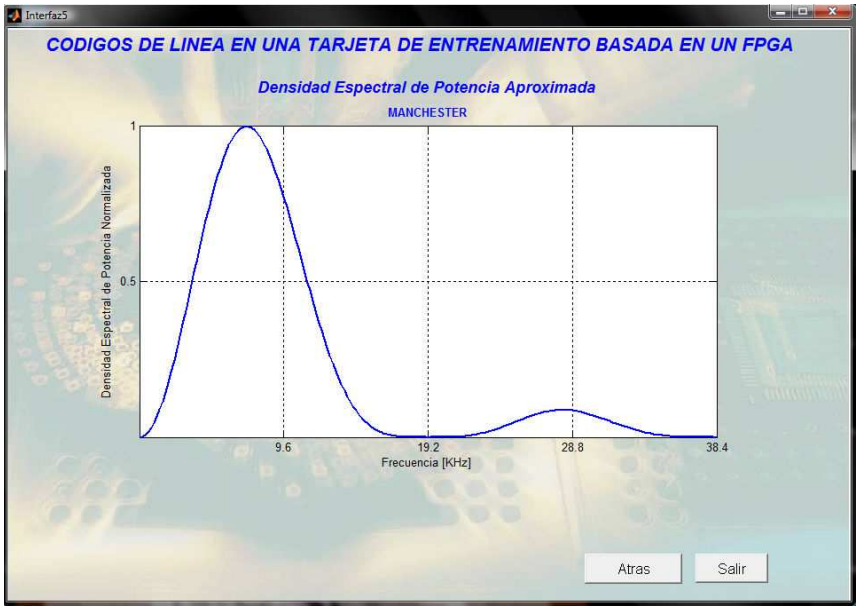


Figura 3.16 Densidad Espectral de Potencia Teórica del Código de Línea seleccionado.

El gráfico presentado en la Figura 3.16 corresponde a la Densidad Espectral de Potencia Teórica del código de línea Manchester.

El diagrama de flujo correspondiente a esta GUI se presenta en la Figura 3.17.

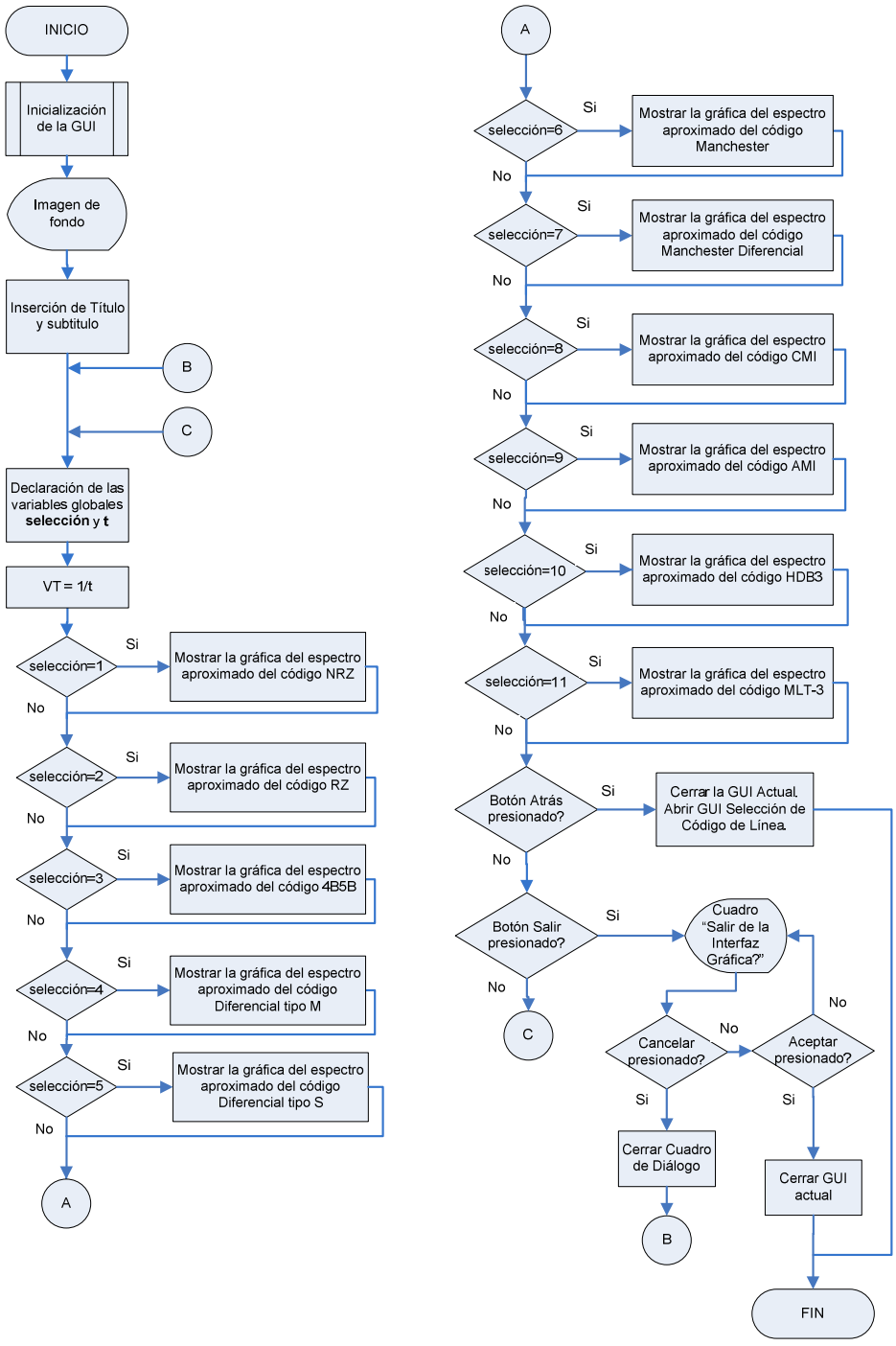


Figura 3.17 Diagrama de Flujo de Visualización del Espectro de Potencia Teórico del Código de Línea seleccionado.

3.4.8 DECODIFICACIÓN

La Interfaz Gráfica de Usuario Decodificación presenta la Secuencia de Caracteres Enviada que el usuario introdujo, la Secuencia de Caracteres Recibida que se formó después de haber acogido la señal decodificada desde el FPGA, la secuencia de bits enviada, y la secuencia de bits recibida. Con lo cual podemos hacer una comparación entre la secuencia a ser codificada enviada y la secuencia decodificada recibida, que por supuesto debe ser la misma. Cabe mencionar que esta interfaz puede visualizar todo lo manifestado, únicamente después de realizarse la decodificación en el FPGA y la transmisión de los datos decodificados desde este dispositivo programable a la interfaz gráfica, lo cual se detalla en el Capítulo 4.

Además esta GUI permite visualizar la Velocidad de Transmisión y Velocidad de Señal, tal como se muestra en la Figura 3.18.

Figura 3.18 Visualización de Caracteres Enviados y Recibidos del FPGA.

El diagrama de flujo de la programación en código de esta Interfaz Gráfica de Usuario se muestra en la Figura 3.19.

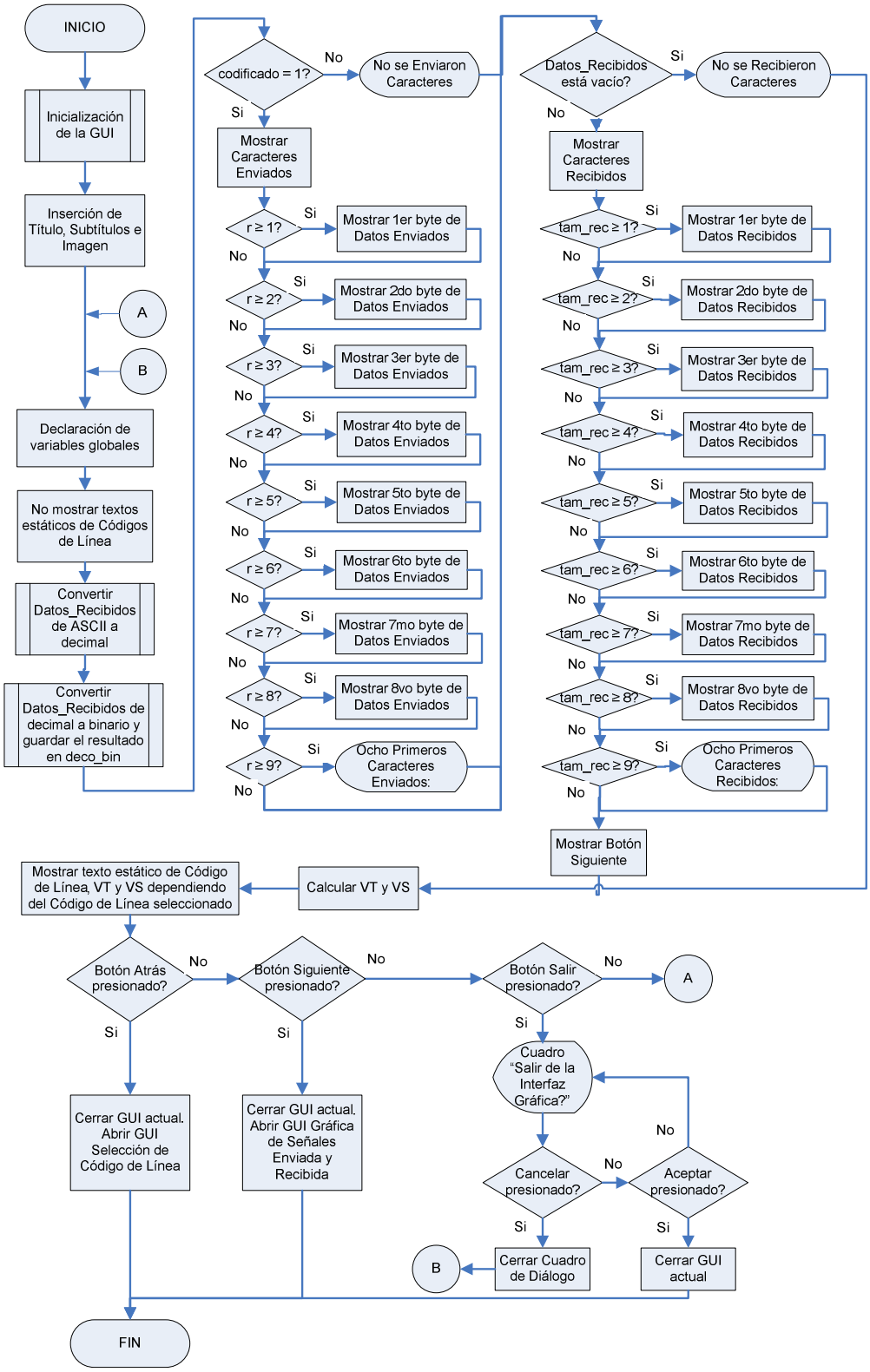


Figura 3.19 Diagrama de Flujo de la programación de la GUI Decodificación.

3.4.9 GRÁFICA DE SEÑALES ENVIADA Y RECIBIDA

En esta ventana se presentan tres Gráficas: la Señal de Reloj, la Señal que se envió para que sea codificada y la Señal decodificada recibida desde el FPGA. Con estas gráficas se puede comparar fácilmente las señales enviada y recibida, que deben ser iguales en el caso de un exitoso proceso de transmisión, codificación, decodificación y recepción.

Esta GUI se muestra en la Figura 3.20, mientras que el diagrama de flujo de la programación en código de esta Interfaz Gráfica se muestra en la Figura 3.21.

Figura 3.20 Gráfica de la Señal de Reloj, Señal Enviada y Señal Recibida del FPGA.

La Figura 3.20 es un ejemplo específico en el caso que el usuario ingrese como dato el carácter “a” que en ASCII tiene el valor 61H o [1 0 0 0 0 1 1 0] en binario, y escoja una velocidad de transmisión de 9600 bps. Se puede observar que el tiempo de bit es aproximadamente 104.17 us en la señal enviada y recibida.

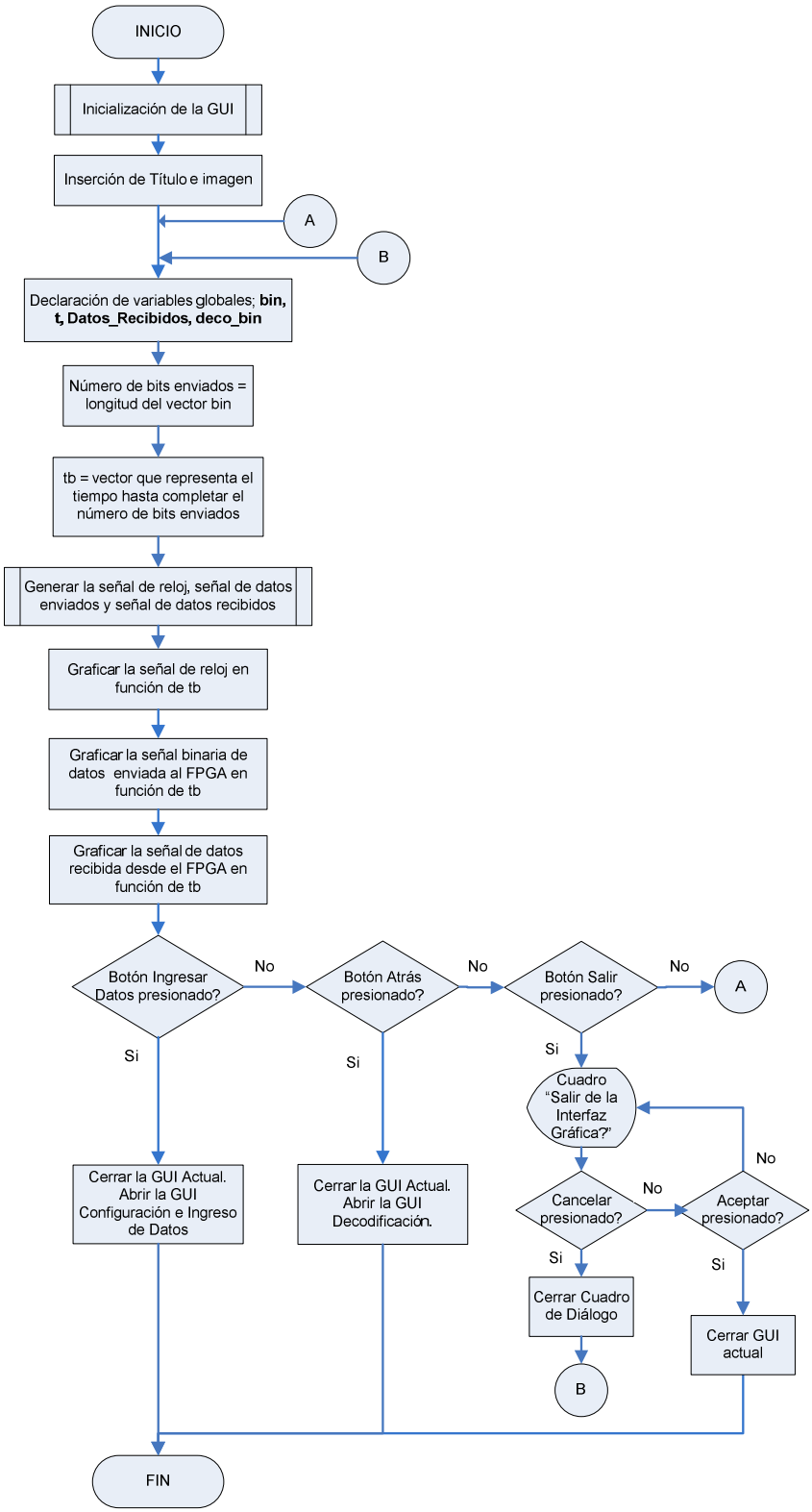


Figura 3.21 Diagrama de Flujo de Gráfica de la Señal de Reloj, Señal Enviada y Señal Recibida del FPGA.

Se puede apreciar en el diagrama de flujo que para obtener esta GUI se toman las variables bin, t, Datos_Recibidos y deco_bin de otras Interfaces Gráficas, debido a que en este segmento no se envía o recibe datos sino sólo se grafica los resultados ya obtenidos en otras GUIs. Se incluye tres botones: Ingresar Datos, que permite abrir la GUI Configuración e Ingreso de Datos; Atrás, que permite regresar a Decodificación; y Salir, que cierra la Interfaz Gráfica.

CAPÍTULO IV

IMPLEMENTACIÓN Y SIMULACIÓN DE LOS CÓDIGOS DE LÍNEA

4.1 DESCRIPCIÓN Y REQUERIMIENTOS

En este capítulo se detallan las funciones que cumple la Spartan-3E Starter Kit Board y por consiguiente también el FPGA, el mismo que en este proyecto tiene como objetivo principal: la codificación y decodificación de datos.

Para cumplir este propósito, además es necesario:

- ✓ Configurar la velocidad en el FPGA a la que se enviarán y se recibirán los datos, es decir generar un reloj interno de frecuencia determinada que permita llevar a cabo procesos en VHDL a la velocidad seleccionada por el usuario.
- ✓ Transmitir y recibir los datos entre el FPGA y la Interfaz Gráfica elaborada en MATLAB.
- ✓ Almacenar los datos después de su recepción en el FPGA para su posterior codificación.
- ✓ Identificar el tipo de código de línea a implementarse.
- ✓ Después del almacenamiento de datos serializarlos para que sean codificados.
- ✓ Implementar un Interpretador Unipolar – Bipolar y un Interpretador Bipolar – Unipolar debido a las características de la Spartan – 3E Starter Kit Board (No trabaja con niveles de voltaje negativos).
- ✓ Posterior a la decodificación, ensamblar los datos para que sean enviados a la Interfaz Gráfica.
- ✓ Enviar los datos decodificados desde el FPGA a MATLAB para visualizarlos en la Interfaz Gráfica.

4.2 BANCO DE TRABAJO

En el esquema de trabajo que se ha implementado, constan los siguientes elementos: la tarjeta de entrenamiento que contiene el FPGA que va a efectuar las tareas de codificación y decodificación, un circuito externo que nos permite la visualización de los códigos polares y bipolares debido a las restricciones que presenta la tarjeta, dos fuentes DC para la alimentación del circuito externo, un osciloscopio, una computadora con el software MATLAB donde se va a desplegar la interfaz gráfica y el programa Xilinx ISE 10.1 para la programación y ejecución del software realizado en VHDL.

Hardware

- Tarjeta de entrenamiento SPARTAN 3E STARTER KIT BOARD.
- Circuito externo interpretador Unipolar-Bipolar y Bipolar-Unipolar.
- 2 Fuentes DC.
- Osciloscopio Tektronix TDS 1002B.
- Computador, con las siguientes características:
  - ✓ Sistema operativo Windows XP con Service Pack 3.
  - ✓ Dos puertos USB.
  - ✓ Memoria RAM, mínimo de 512 Mbytes.
  - ✓ Procesador Pentium 4 o superior.
- Cable USB-Serial.

Software

- MATLAB 7.1
- Xilinx ISE 10.1

En la Figura 4.1 se presenta el Diagrama de Bloques del Banco de Trabajo

Figura 4.1 Diagrama de Bloques del Banco de Trabajo.

En la Figura 4.2 se ha capturado una imagen donde se presentan los equipos utilizados en el proyecto.

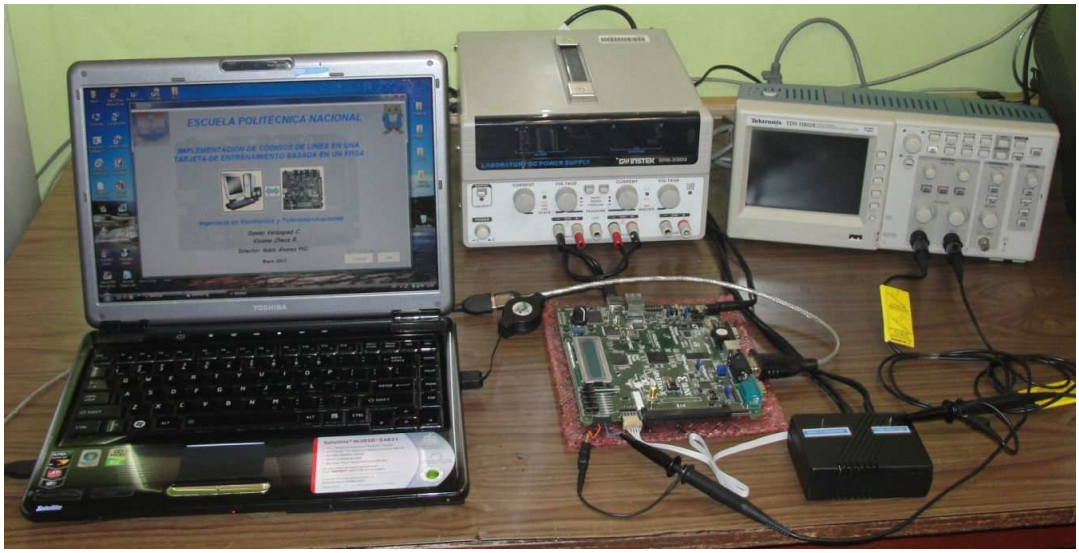


Figura 4.2 Banco de Trabajo.

4.3 CARACTERÍSTICAS DE LA TARJETA DE ENTRENAMIENTO

La tarjeta de entrenamiento que se utiliza en el proyecto se denomina SPARTAN 3E STARTER KIT BOARD. Esta tarjeta es conveniente para el desarrollo del proyecto debido a que posee los periféricos necesarios, como: dos puertos seriales RS-232, un conector hembra DCE y un conector macho DTE DB9. Para la transmisión y recepción de datos se emplea el conector hembra de la tarjeta que actúa como DCE (Data Communications Equipment), en tanto que el computador de donde se envían los datos a través de la interfaz gráfica desempeña el papel de DTE (Data Terminal Equipment), tal como se aprecia en la Figura 4.3. Además contiene los pines de entrada y salida adecuadas para la operación del esquema propuesto.

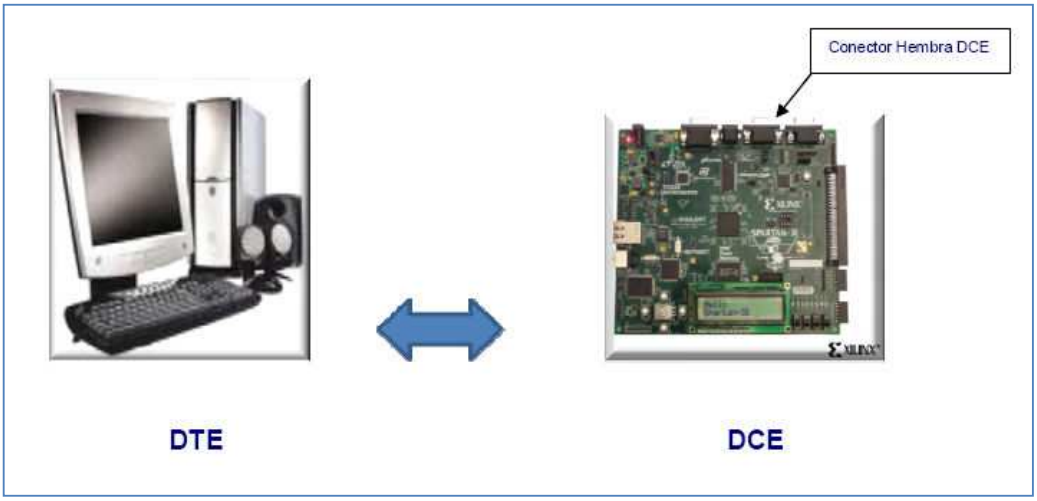


Figura 4.3 Esquema de comunicación Computador – Tarjeta de Entrenamiento.

La tarjeta de entrenamiento proporciona solo voltajes LVTTL, es decir 0 [V] y +3.3 [V]. Para el análisis de los códigos polares y bipolares que se implementan es ineludible el diseño de dos circuitos externos que nos permitan interpretar dichos códigos. El primero de ellos se lo denomina Circuito Interpretador Unipolar-Bipolar, este admite dos entradas binarias que provienen de la tarjeta de entrenamiento y como salida entrega una señal con dos o tres niveles de voltaje según sea el código. El segundo es el Circuito Interpretador Bipolar-Unipolar, la entrada de este circuito es la señal codificada (puede ser bipolar) que entrega el

primero; y las salidas son de tipo binarias, que a su vez son entradas a la tarjeta de entrenamiento.

4.4 TRANSMISIÓN Y RECEPCIÓN DE SEÑALES MATLAB - FPGA

En la comunicación MATLAB – FPGA se utiliza el modo de transmisión asincrónica (no hay ninguna relación de tiempo entre MATLAB y el FPGA), debido a que este modo de transmisión típicamente se usa en transmisión de códigos ASCII a través del puerto RS-232, tal como lo requiere el proyecto. Además se utiliza este modo de transmisión debido a que primero se transmite un carácter de configuración de velocidad y luego de un tiempo indeterminado se transmite los datos, es decir el FPGA no conoce exactamente el momento en que recibirá los mismos. Al utilizar el protocolo asincrónico de MATLAB cada dispositivo utiliza su propia señal de reloj interna pudiendo transmitirse bytes a tiempos arbitrarios.

El formato y conjunto de caracteres transmitidos se especifican a continuación.

4.4.1 FORMATO DEL CARÁCTER EN TRANSMISIÓN ASINCRÓNICA

El formato que se utiliza en el proyecto para la transmisión serial de un carácter incluye 1 bit de inicio, 8 bits de datos, 1 bit de paridad y 1 bit de parada.

La Figura 4.4 ilustra este formato.

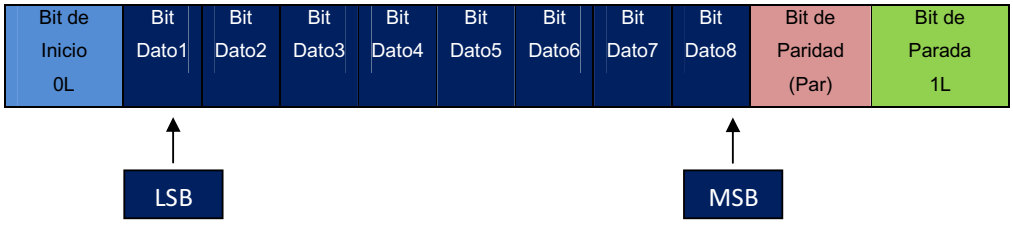


Figura 4.4 Formato de un carácter en transmisión asincrónica.

El número de bits transmitidos por segundo está dado por la velocidad en bits por segundo, bps. En esta consideración se incluyen los bits de inicio, datos, paridad

y parada. Como se puede apreciar en la Figura 4.4, los bits son transmitidos uno cada tiempo de bit, de la siguiente manera:

- Un bit de inicio con valor 0L
- Ocho bits de datos, el primer bit corresponde al bit menos significativo (LSB), mientras que el octavo bit corresponde al bit más significativo (MSB).
- Un bit de paridad par
- Un bit de parada de valor 1L.

En el FPGA, los caracteres transmitidos son identificados por los bits de inicio y parada. El bit de inicio indica cuando el carácter empieza y el bit de parada indica la finalización del carácter. Cuando el pin de transmisión está en reposo (no transmite), está en estado lógico 1L. El bit de paridad permite realizar un chequeo de error simple en el FPGA; al utilizar paridad par, los bits de datos más el bit de paridad deben dar como resultado un número par de estados lógicos en alto.

Los bits de datos transmitidos pueden representar: comandos al FPGA (Identificador de Velocidad, Identificador de Inicio, Identificador de Código, Identificador de Fin) o Datos (uno de los caracteres ASCII introducidos por el usuario). Los datos se basan en el formato de caracteres ASCII extendido (8 bits conforman el carácter) permitiendo utilizar 256 caracteres (excepto los caracteres restringidos).

4.4.2 IDENTIFICADOR DE VELOCIDAD

Al configurar el puerto de comunicación y la velocidad de transmisión en la Interfaz Gráfica se envía un carácter al FPGA. Este carácter identifica la velocidad a la que se transmitirán los datos. Para el efecto, se utilizan señales de reloj independientes en MATLAB y en el FPGA, tal como se especifica en el proceso configuración de velocidad. Este identificador tiene el formato de un carácter en transmisión asincrónica, es decir 8 bits de datos de acuerdo a la Tabla 4.2.

4.4.3 FORMATO DEL STREAM DE DATOS

Al enviar en modo asincrónico los datos (caracteres ASCII) al FPGA, se lo realiza en forma continua y se incluyen los siguientes comandos: Identificador de Inicio de datos, un Identificador de Código y un Identificador de Fin de datos, tal como lo muestra la Figura 4.5. Todas estas componentes del stream de datos son caracteres que tienen el formato para transmisión asincrónica (bit de inicio, bit de paridad, bit de parada).

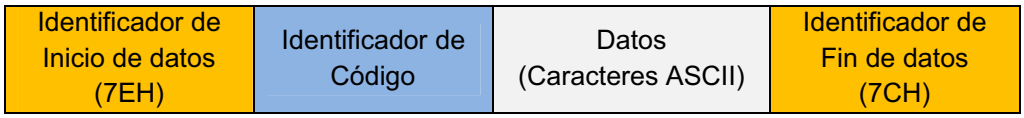


Figura 4.5 Formato del stream de datos.

Los Identificadores no se los utiliza en el modo de transmisión sino con los siguientes objetivos:

- Identificador de Inicio de datos, permite conocer cuál es el primer bit (bit menos significativo) del primer carácter después de haber sido decodificado en el FPGA. En la codificación no es necesario identificar el primer bit, pero posterior a la decodificación si se requiere conocer el primer bit para su posterior transmisión a la Interfaz Gráfica.
- Identificador de Código, permite conocer el código seleccionado por el usuario con el cual se realizará la codificación y decodificación en el FPGA, es establecido de acuerdo a la Tabla 4.1.
- Identificador de fin de datos, posterior a la decodificación permite conocer cuando se terminan los bits decodificados, para enviarlos a la Interfaz Gráfica de MATLAB.

En la Figura 4.6 se muestra el esquema de transmisión de las señales digitales de MATLAB al FPGA. Primero se envía el Identificador de Velocidad y después de un tiempo indeterminado el stream de datos.

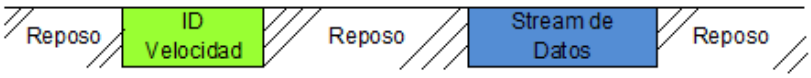


Figura 4.6 Transmisión de señales de MATLAB al FPGA.

4.5 DIAGRAMA DE BLOQUES DE LA IMPLEMENTACIÓN EN VHDL

VHDL es inherentemente concurrente (contrario a otros lenguajes de programación, que son secuenciales); para implementar circuitos que dependen de una señal de reloj, se utiliza procesos en VHDL para que la programación sea secuencial.

El código principal VHDL tiene ocho procesos, cada uno de los cuales se describe posteriormente en este capítulo. Estos procesos son concurrentes entre sí, pero secuenciales en su estructura interna.

Las señales de reloj y los códigos de línea se desarrollan en componentes diferentes e independientes, las cuales son segmentos de código VHDL convencional ejecutadas por el programa principal de manera concurrente, permitiendo la construcción del diseño en forma jerárquica. Estos bloques de código están declarados en el código principal para que sean reconocidos e intercambien señales con el mismo, tal como lo muestra la Figura 4.7.

Las componentes de VHDL que forman parte de la implementación se encuentran en el **ANEXO B**.

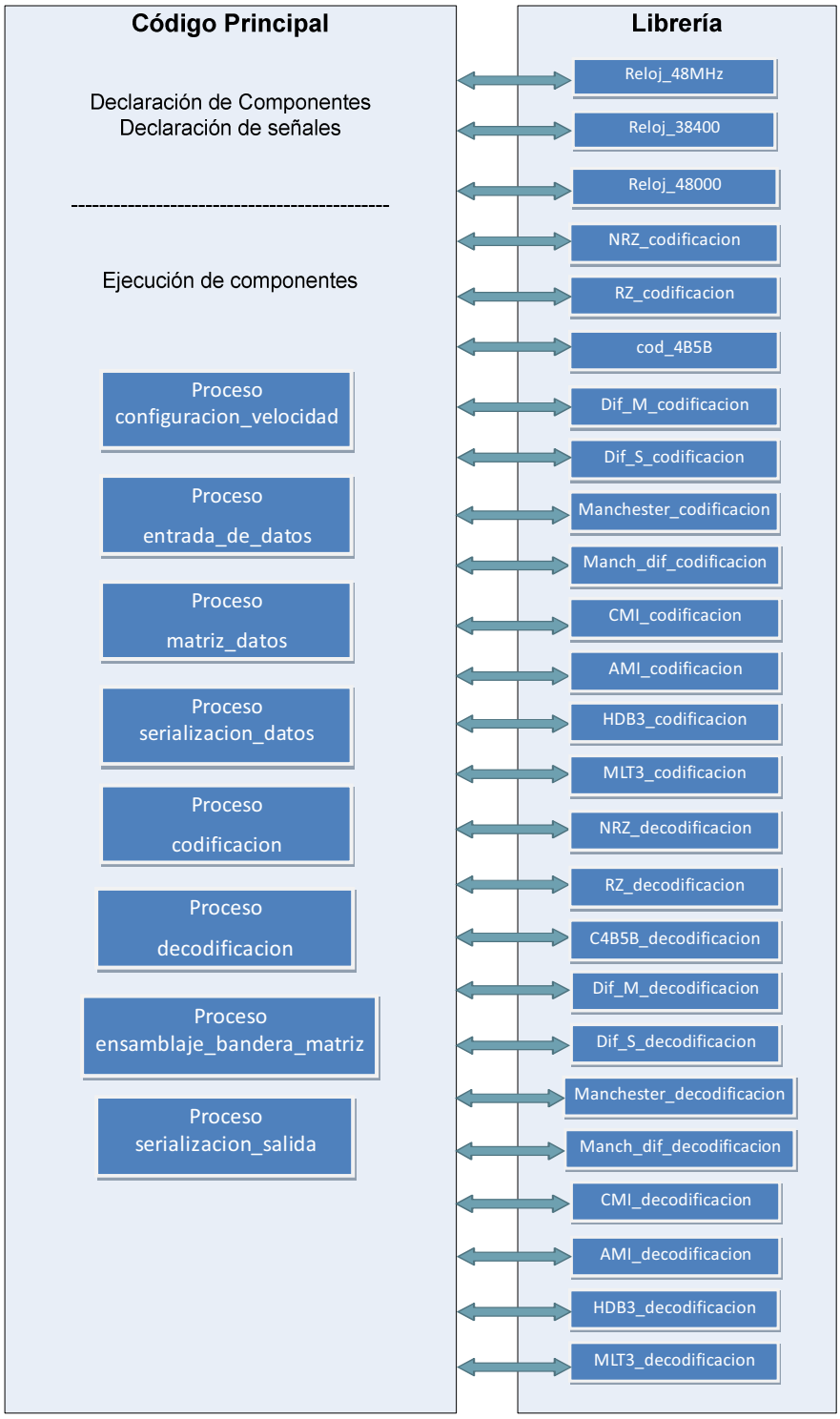


Figura 4.7 Diagrama de Bloques del programa principal en VHDL.

4.6 COMPONENTES DE RELOJ

Para cumplir con los propósitos del proyecto, específicamente con los procesos de configuración de velocidad, transmisión y recepción de datos, codificación y decodificación, se requieren de señales de reloj de diferente frecuencia. Como la tarjeta de entrenamiento dispone de un único reloj interno de 50 MHz, a partir de este se realizan las derivaciones requeridas que se explican en la Figura 4.8.

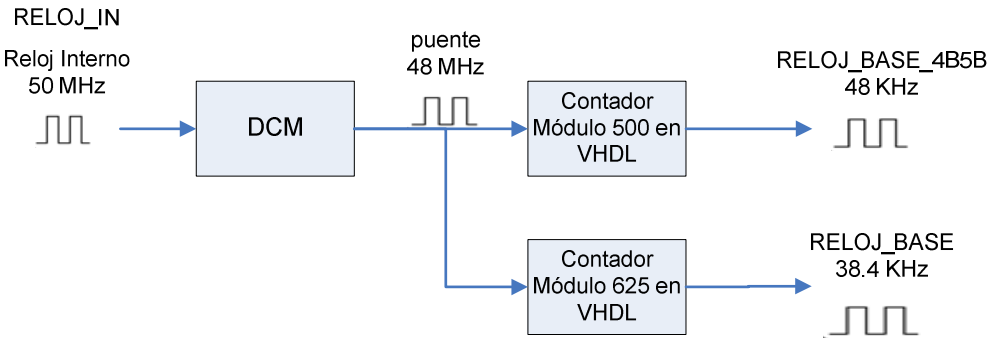


Figura 4.8 Señales de Reloj a partir del Reloj interno de la Tarjeta de Entrenamiento.

Configuración de Velocidad

Para configurar la velocidad a la que se transmitirán los datos se realiza una transmisión asincrónica desde MATLAB al FPGA a 38400 bps. Para identificar lo que se recibe se necesita una señal de reloj de entrada al FPGA de 38400 Hz, a la que se le ha denominado *RELOJ_BASE*, que permite almacenar los 8 bits del Identificador de Velocidad, uno cada transición positiva de la señal de reloj mencionada.

En la configuración del puerto de comunicación, el computador a través de la Interfaz Gráfica envía a esta velocidad determinada al FPGA un carácter especificando la velocidad escogida por el usuario a la que se enviará posteriormente la trama de datos. Para que el FPGA conozca la velocidad a la cual se enviarán los datos, el reloj interno de la tarjeta de entrenamiento de 50 MHz se lo configura para que funcione a la frecuencia específica de 38400 Hz utilizando el DCM y un contador implementado en VHDL módulo 625.

Transmisión y recepción de datos

En la Interfaz Gráfica se pueden elegir las siguientes velocidades a las que se enviarán y recibirán datos desde y hacia la Tarjeta de entrenamiento: 2400, 4800, 9600 y 19200 bps. Entonces para la transmisión y recepción de datos en el FPGA se configura la velocidad con la señal de reloj *RELOJ_BASE* de 38400 Hz en el proceso configuración de velocidad, obteniéndose una señal de reloj, *RELOJ_OUT*, a la frecuencia específica dependiendo de la velocidad seleccionada, pudiendo esta última ser de 2400, 4800, 9600 o 19200 Hz.

Codificación y decodificación de datos

La codificación y decodificación de datos debe realizarse en base a una señal de reloj. La señal de reloj que debería utilizarse es *RELOJ_OUT*, ya que tiene la frecuencia adecuada, pero debido a que en VHDL no se pueden considerar ambas transiciones positiva y negativa de una misma señal dentro de un único proceso (ya que se produciría el error Xst: 827, este tipo de error es de síntesis no de sintaxis), no se puede codificar y decodificar únicamente en base a esta señal, esto se da en caso de los códigos que presentan transición a mitad de tiempo de bit.

Una solución eficiente para este caso es crear una señal del doble de frecuencia que la señal *RELOJ_OUT, RELOJ_OUT_Doble*, que a través de un contador módulo 2 represente la transición positiva y negativa de la señal RELOJ_OUT, tal como se muestra en la Figura 4.9, y por consiguiente utilizando condicionales cada transición positiva de esta última, sea equivalente a ejecutar un conjunto de instrucciones cada transición positiva y negativa de *RELOJ_OUT*.

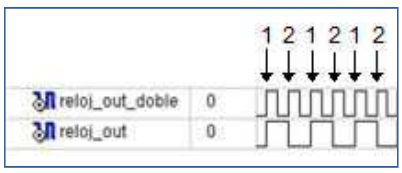


Figura 4.9 Señales de reloj, *RELOJ_OUT y RELOJ_OUT_Doble*.

Los códigos que requieren de la señal *RELOJ_OUT_Doble*, son aquellos que tienen transición a mitad de tiempo de bit. De esta manera los códigos que

codifican en base a la señal de reloj, *RELOJ_OUT* son: NRZ, Diferencial tipo M, Diferencial tipo S, AMI, HDB3 y MLT3. Los códigos que se basan en la señal de reloj *RELOJ_OUT_Doble* son: RZ al 50%, Manchester, Manchester Diferencial y CMI. 4B5B es un código diferente en cuanto a que la señal codificada se debe basar en una señal de reloj de frecuencia 5/4 de la frecuencia de la señal de reloj con la que llegan los datos, para este código en particular se utiliza la señal *RELOJ_OUT_4B5B*.

4.6.1 SEÑAL DE RELOJ DE 48 MHz

La señal de 48 MHz es intermedia entre la señal de reloj generada por la tarjeta de entrenamiento y las señales requeridas. La frecuencia de esta señal es específicamente de 48 MHz ya que a partir de esta se pueden generar las dos señales de reloj básicas (48 KHz y 38.4 KHz), a través de contadores en VHDL, requeridas para la transmisión y recepción de datos, y la codificación y decodificación.

Para este propósito se utiliza el DFS (Sintetizador de Frecuencia), que constituye una de las herramientas de software del DCM. El DFS permite multiplicar y/o dividir la frecuencia de reloj propia de la tarjeta para sintetizarla en una nueva.

Al utilizar el DCM se debe especificar el tipo de archivo de salida (VHDL para que sea compatible con el resto del proyecto) y las características específicas del FPGA (Familia, dispositivo, paquete, velocidad).

La Figura 4.10 muestra la configuración con el DCM; el único ítem que se habilita es el de salida *CLKFX.*

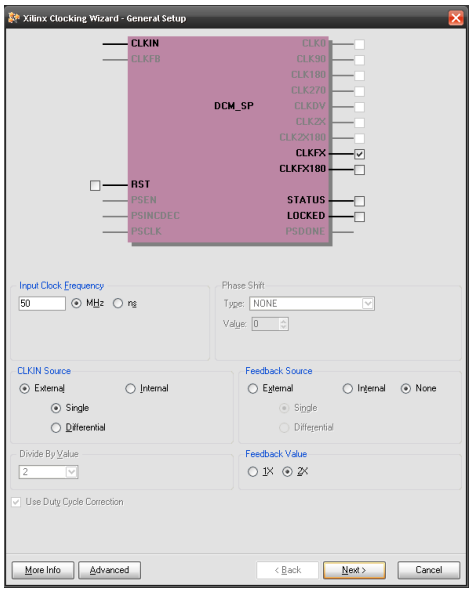


Figura 4.10 Configuración general de la señal de reloj de 48 MHz con DCM.

La frecuencia de la señal de salida está en función de la frecuencia de la señal de reloj de entrada, de M (Multiplicador) y D (Divisor). M y D se especifican en el cuadro de diálogo Clock Frecuency Synthesizer, tal como se muestra en la Figura 4.11. Este cuadro de diálogo aparece cuando CLKFX o CLKFX180 están habilitados.

Para reducir la frecuencia de la señal de reloj de la tarjeta de 50 MHz a 48 MHz se realiza la siguiente asignación:

$$f_{reloj\ interno} * \left(\frac{M}{D}\right) = 48\ MHz$$

$$50\ MHz * \left(\frac{M}{D}\right) = 48\ MHz$$

$$\left(\frac{M}{D}\right) = \frac{48\ MHz}{50\ MHz}$$

$$\frac{M}{D} = \frac{24}{25}$$

$$\Rightarrow M = 24\ y\ D = 25$$

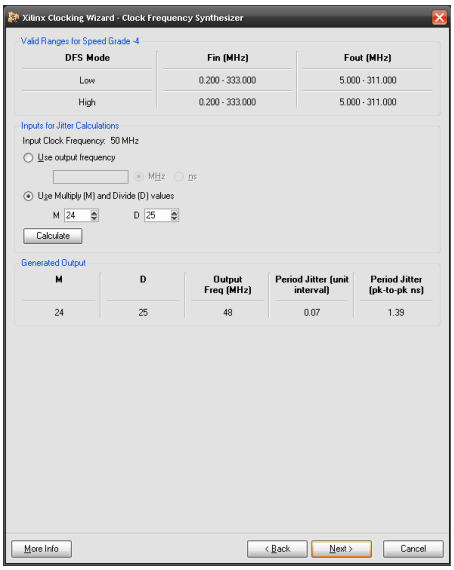


Figura 4.11 Asignación de los factores de multiplicación y división para la obtención de la señal de reloj de 48MHz.

Cabe mencionar que no se sintetiza directamente de la señal de 50 MHz a las señales de reloj de 48 KHz y 38.4 KHz debido a los límites del DCM, siendo necesario crear esta señal de reloj intermedia. Los factores de multiplicación M y de división D son enteros entre los límites, de 2 a 32 y de 1 a 32, respectivamente. Al finalizar la utilización del DCM, se crea un archivo .vhd que corresponde al código VHDL que obtiene la señal de reloj de 48 MHz.

Para utilizar este archivo en el proyecto, se le asigna la componente *Reloj_48MHz*, en la cual sólo se ocupa la señal de entrada: *CLKIN_IN* y la señal de salida: *CLKFX_OUT*. La entrada y salida de esta componente se relacionan con el código principal a través de las señales *RELOJ_IN* y puente, respectivamente. *RELOJ_IN* es la señal proveniente del reloj de la tarjeta de entrenamiento de 50 MHz, mientras que puente es la señal de reloj de 48 MHz. Con la señal puente se generan las señales de reloj de 48 KHz y de 38.4 KHz.

4.6.2 SEÑAL DE RELOJ DE 48 KHz

A partir de la señal de reloj puente se genera la señal de reloj *RELOJ_BASE_4B5B* de 48 KHz. Para este propósito se implementa un contador ascendente módulo 500 (De 0 a 499) en VHDL. Cada vez que el contador llega a

499, la señal de reloj *RELOJ_BASE_4B5B* se invierte y el contador toma nuevamente el valor 0, tal como lo muestra la Figura 4.12.

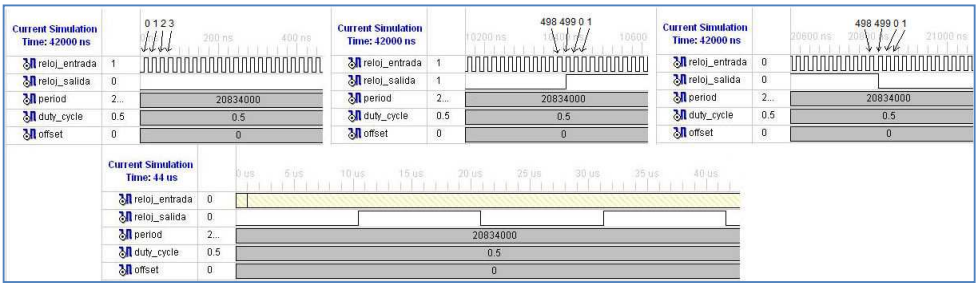


Figura 4.12 Obtención de la señal de reloj de 48 KHz a partir de 48 MHz.

El código VHDL que obtiene esta señal de reloj, está en la componente reloj_48000. El diagrama de flujo de esta componente se muestra en la parte (b) de la Figura 4.14.

4.6.3 SEÑAL DE RELOJ DE 38.4 KHz

La señal de reloj *puente* también es de la frecuencia adecuada, 48 MHz, para generar la señal de reloj *RELOJ_BASE* de 38.4 KHz. Esta es la señal de salida de la componente *reloj_38400* donde se implementa un contador ascendente módulo 625 (De 0 a 624) en VHDL. Cada vez que el contador toma el valor 624, la señal de reloj *RELOJ_BASE* se invierte y el contador retorna al valor 0, tal como lo muestra la Figura 4.13.

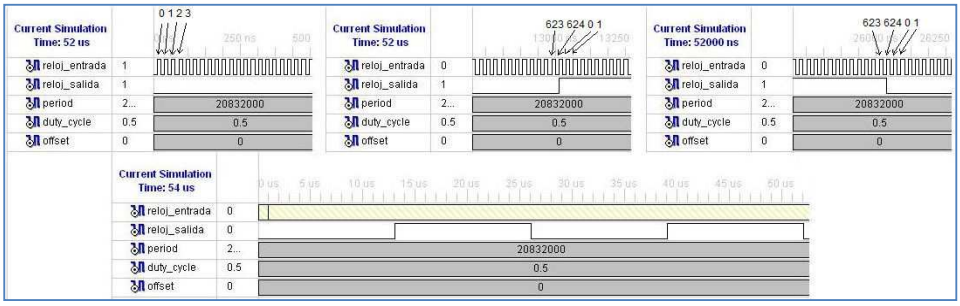


Figura 4.13 Obtención de la señal de reloj de 38.4 KHz a partir de 48 MHz.

El diagrama de flujo de la componente *reloj_38400* se muestra en la parte a de la Figura 4.14.

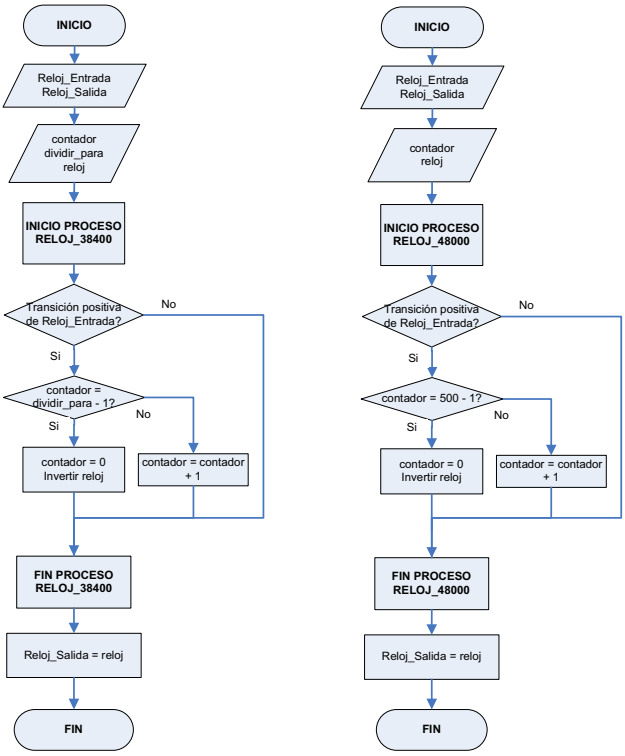


Figura 4.14 Diagrama de Flujo de las Componentes de Reloj de 38400 Hz y 48000 Hz en VHDL.

Los diagramas de flujo mostrados en la Figura 4.14 son semejantes, debido a que los dos corresponden a contadores ascendentes en VHDL. Se diferencian en el módulo del contador; el de la parte (a) módulo 625, valor que se almacena en la señal *dividir_para*; y el de la parte (b) módulo 500. Los procesos de las componentes se ejecutan en cada transición positiva de *Reloj_Entrada* ya que esta señal se encuentra en la lista de sensibilidad de los procesos. *Reloj_Entrada* es el nombre que toma la señal puente que ingresa del código principal.

4.7 DESARROLLO DE PROCESOS

Hay que recordar que VHDL es inherentemente concurrente, sin embargo un proceso es una sección de código que es ejecutada secuencialmente. Estos bloques, procesos, son concurrentes con el resto de instrucciones localizadas fuera del mismo.

Entonces las instrucciones que se encuentran adentro de cada proceso son específicas para código secuencial VHDL, y por tanto se lo puede representar en diagramas de flujo que se ejecutan cuando cambia el estado de al menos una de las señales de la lista de sensibilidad.

También hay que recordar que de acuerdo a VHDL en los procesos se declaran únicamente variables y no señales, las cuales se utilizan explícitamente en código secuencial.

En el proyecto se utilizan variables en cada proceso en lugar de señales en todo el código principal porque se requiere la creación de datos internos válidos únicamente para cada proceso específico; hay que recordar que las variables no son globales, es decir, no tienen validez afuera del proceso. Además se utilizan variables y no señales debido a que la actualización de las variables es inmediata, es decir toma el nuevo valor en la siguiente instrucción del código, contrario a las señales (cuando un proceso las utiliza) que se actualizan al concluir la actual ejecución del proceso.

En el código principal cada proceso no se utiliza simplemente para separar líneas de código VHDL y hacerlo más entendible, sino que además se le asigna una función, tal como se describe en cada uno de ellos.

4.7.1 CONFIGURACIÓN DE VELOCIDAD

Uno de los procesos que contiene el programa principal es el de configuración de velocidad, proceso que tiene como finalidad obtener en el FPGA señales de reloj de frecuencia adecuada de acuerdo a la velocidad que el usuario ha seleccionado en la Interfaz Gráfica, para que posteriormente se pueda realizar la comunicación (Envío y recepción de datos MATLAB – FPGA) a esa velocidad, la codificación y decodificación. La frecuencia de las señales de reloj a obtener, *RELOJ_OUT, RELOJ_OUT_Doble* y *RELOJ_OUT_4B5B*, depende de la velocidad seleccionada en la Interfaz Gráfica, tal como se especifica en la Tabla 4.1. Las señales de reloj *RELOJ_OUT* y *RELOJ_OUT_Doble* se obtienen a partir de la señal de reloj *RELOJ_BASE*, mientras que *RELOJ_OUT_4B5B* se obtiene a partir de *RELOJ_BASE_4B5B*. Estas señales se consiguen a través de un contador

ascendente en VHDL (excepto *RELOJ_OUT_Doble* para 19200 bps) tal como se muestra en el diagrama de flujo de la Figura 4.15.

Velocidad [bps]	RELOJ_OUT [Hz]	RELOJ_OUT_Doble [Hz]	RELOJ_OUT_4B5B [Hz]
2400	2400	4800	3000
4800	4800	9600	6000
9600	9600	19200	12000
19200	19200	38400	24000

Tabla 4.1 Señales de Reloj obtenidas de acuerdo a la velocidad de transmisión seleccionada.

La frecuencia de la señal de reloj, *RELOJ_OUT*, debe ser numéricamente igual a la velocidad para que en el proceso *entrada_de_datos* el código VHDL en cada transición positiva de *RELOJ_OUT* reconozca un bit recibido en el FPGA. *RELOJ_OUT_Doble* es del doble de frecuencia que *RELOJ_OUT* debido a que se le utiliza en la codificación y decodificación con los códigos de línea que tienen transición a mitad de tiempo de bit. La frecuencia de *RELOJ_OUT_4B5B* es 5/4 la frecuencia de *RELOJ_OUT* debido a que la codificación y decodificación en VHDL con el código de línea 4B5B requieren de una señal de reloj de esta característica.

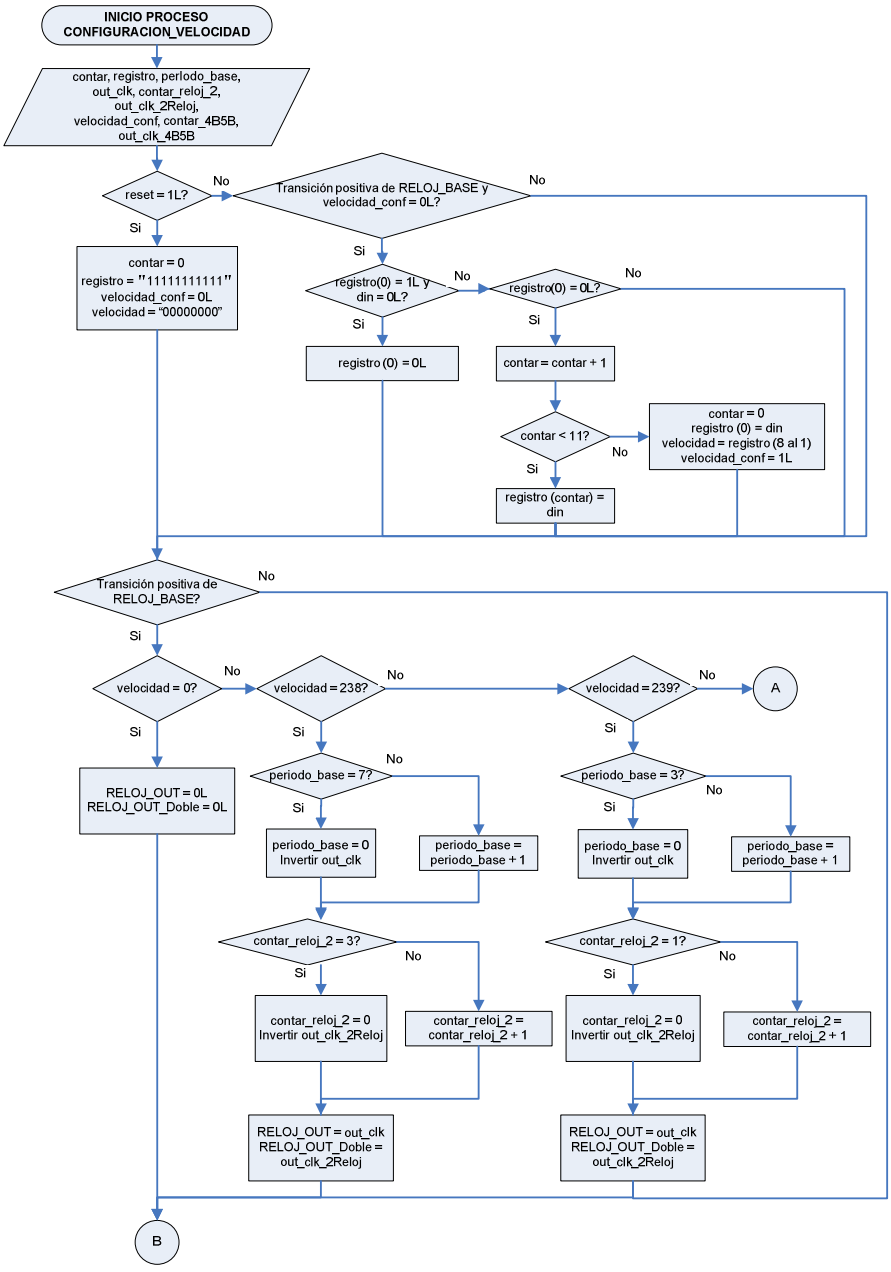
Para realizar la configuración de velocidad, MATLAB envía un Identificador de Código (Carácter que identifica la velocidad a la que se llevará a cabo la transmisión y recepción de datos MATLAB - FPGA) al FPGA. Este identificador de velocidad es seleccionado en MATLAB de acuerdo a la Tabla 4.2.

Velocidad [bps]	ID Velocidad Formato binario y decimal	
2400	1110 1110	238
4800	1110 1111	239
9600	1111 0100	244
19200	1111 0101	245

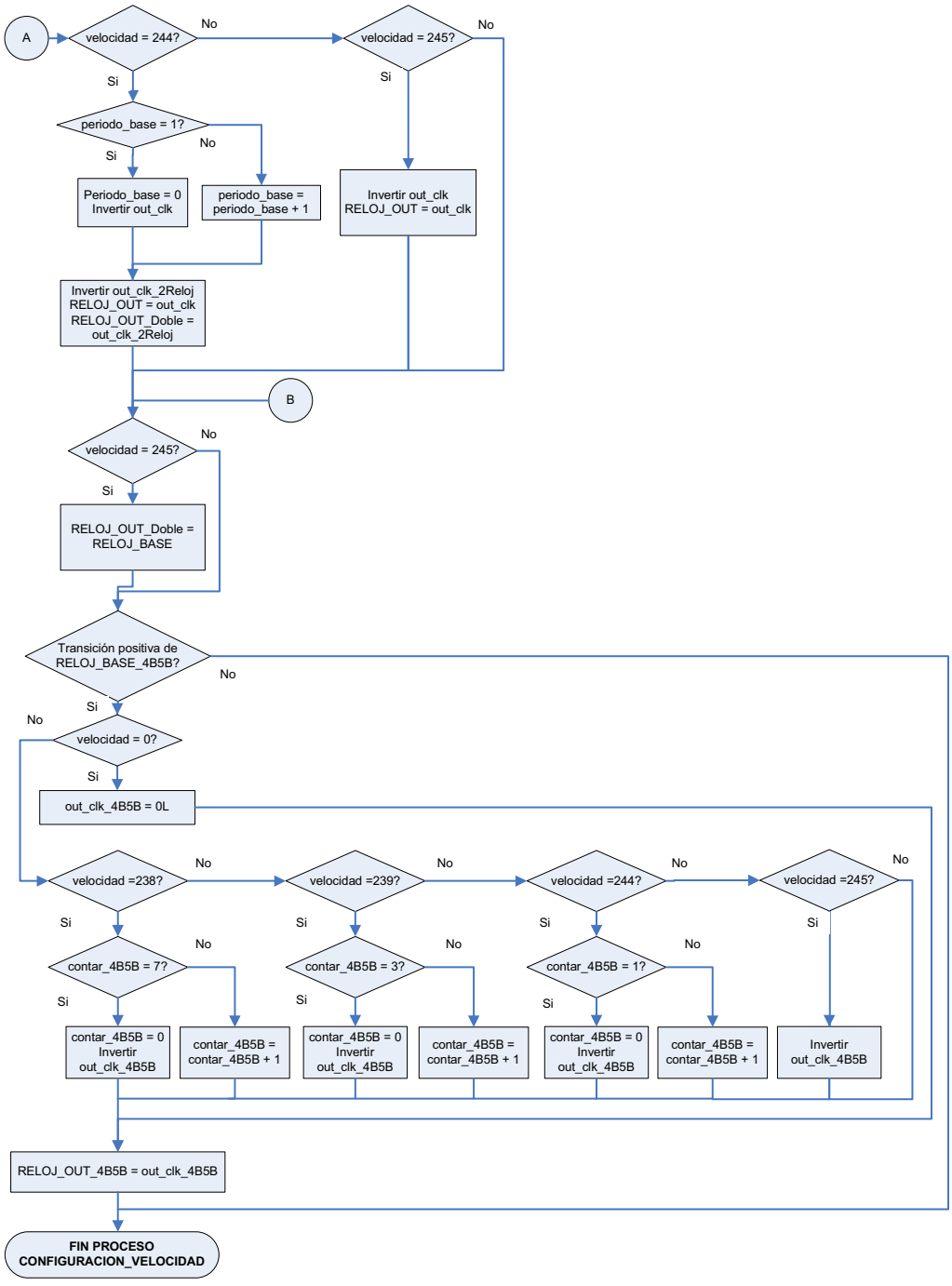
Tabla 4.2 Identificador de Velocidad enviado al FPGA de acuerdo a la velocidad seleccionada.

MATLAB envía el Identificador de velocidad a 38400 bps implícitamente utilizando un reloj de 38400 Hz, por lo que este proceso es sensible a la señal de reloj *RELOJ_BASE* (De 38400 Hz) debido a que en transmisión asincrónica, transmisor y receptor deben actuar a la misma frecuencia.

El diagrama de flujo del código en VHDL del proceso *configuracion_velocidad* se muestra en la Figura 4.15.



(a)



(b)

Figura 4.15 Diagrama de Flujo del proceso de Configuración de Velocidad en VHDL.

4.7.2 ENTRADA DE DATOS

La interfaz gráfica desarrollada en el presente proyecto permite enviar el stream binario de datos desde el puerto serial del computador al puerto serial RS-232 que posee la tarjeta de entrenamiento, estos datos son procesados en el FPGA mediante el proceso *entrada_de _datos*.

Este proceso realiza una recepción de datos serial, contiene una señal de entrada de datos: din, y una salida de datos en forma de vector: datos (7 : 0). Además la señal de reloj *RELOJ_OUT* es necesaria en la lista de sensibilidad para que el proceso se ejecute. Este proceso genera una señal de supervisión, *err* (error), si ésta es 1L significa que hubo un error en la recepción de datos.

El flujo de datos consiste en 11 bits, tal como se muestra en la Figura 4.16, el primero es el bit de inicio, el cual, si está en alto, indica el inicio de la recepción. Los siguientes ocho bits son los de datos (en este caso los datos son caracteres ASCII ingresados por el usuario en la interfaz gráfica), el décimo bit es el de paridad, si éste es cero lógico significa que el número de unos en los datos es par, si es uno lógico sucederá el otro caso. Finalmente el undécimo bit es el de parada, el mismo que debe ser uno lógico al concluir la transmisión del carácter. Cuando la recepción ha concluido y ningún error ha sido detectado, los datos se almacenan en registros internos (*reg*) para ser transferidos al vector *datos* (7:0).

El código en VHDL presenta unas variables internas de este proceso como son: *count,* esta variable es usada para determinar el número de bits recibidos; *reg*, en esta variable se almacenan los datos y *temp*, con la que se calcula el error.

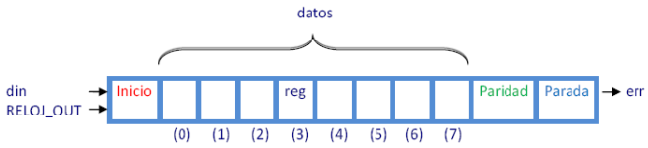


Figura 4.16 Recepción de datos seriales.

En la Figura 4.17 se presenta el diagrama de flujo de la programación en VHDL del proceso de Entrada de Datos.

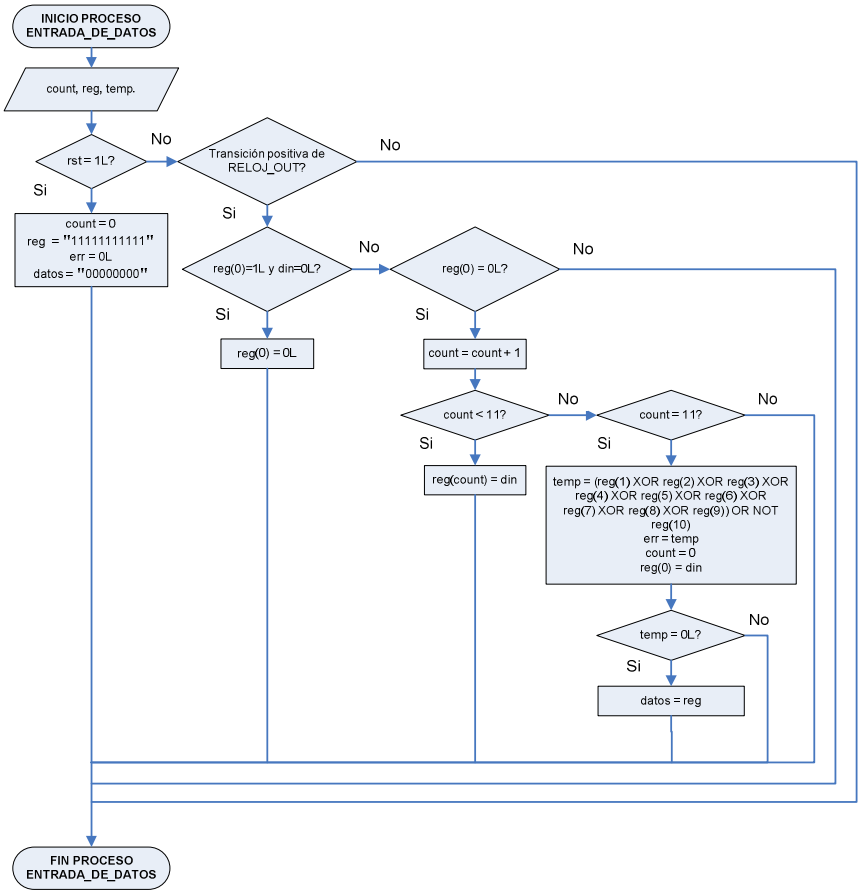


Figura 4.17 Diagrama de Flujo del proceso Entrada de Datos en VHDL.

4.7.3 MATRIZ DE DATOS

En el proceso precedente se puede extraer la señal datos, que es un vector de ocho elementos. El propósito de este proceso es almacenar en una matriz 100x8, los valores de los datos, que representan el valor binario de los caracteres ASCII ingresados por el usuario. El proceso *matriz_datos* tiene en su lista de sensibilidad la señal *RELOJ_OUT*, es decir, que con cada transición positiva de esta señal se van guardando cada uno de los bits de la señal datos en la matriz.

En la transmisión serial desde MATLAB se recibe el stream de datos y un carácter especial llamado LF (en hexadecimal es 0AH, en notación decimal tiene un valor de 10), cuando el proceso en cuestión recibe este carácter significa que la transmisión ha finalizado y consecuentemente la señal *datos_validos* toma el valor de 1L para ser utilizado en el proceso serialización de datos.

Es importante señalar que en la matriz no se van a recopilar valores que tome la señal datos que sean cero o el carácter de terminación de datos LF. En la Figura 4.18 se muestra el formato de la matriz de datos que se genera de este proceso.

$a_{(1,0)}$	$a_{(1,1)}$	$a_{(1,2)}$	$a_{(1,3)}$	$a_{(1,4)}$	$a_{(1,5)}$	$a_{(1,6)}$	$a_{(1,7)}$
$a_{(2,0)}$	$a_{(2,1)}$	$a_{(2,2)}$	$a_{(2,3)}$	$a_{(2,4)}$	$a_{(2,5)}$	$a_{(2,6)}$	$a_{(2,7)}$
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
$a_{(100,0)}$	$a_{(100,1)}$	$a_{(100,2)}$	$a_{(100,3)}$	$a_{(100,4)}$	$a_{(100,5)}$	$a_{(100,6)}$	$a_{(100,7)}$

Figura 4.18 Formato de la Matriz de Datos.

El diagrama de flujo que representa la programación en VHDL de este proceso se encuentra en la Figura 4.19.

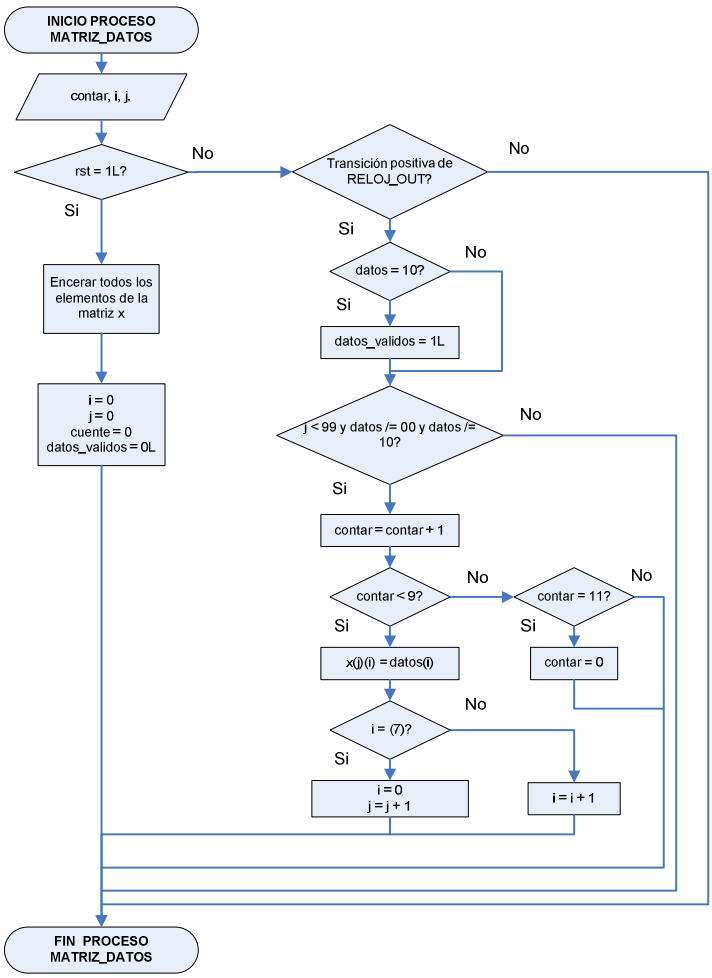


Figura 4.19 Diagrama de Flujo del proceso Matriz de Datos en VHDL.

4.7.4 SERIALIZACIÓN DE DATOS

Una vez almacenada la señal de datos en la matriz, el código VHDL del proceso *serializacion_datos* permite tomar bit a bit todos los caracteres a ser codificados (desde el bit menos significativo del primer carácter hasta el bit más significativo del último carácter ingresado), obteniéndose la señal *senal_a_codificar*.

En la transmisión de datos, MATLAB incorpora en el flujo de bits enviado, antes de los datos, un Identificador de Código, carácter que permite identificar el código de línea seleccionado por el usuario con el cual se realiza la codificación y decodificación. Este identificador no es parte de la señal a ser codificada por lo que en este proceso se lo excluye de la señal a codificar, pero se lo utiliza para asignar un valor a la señal ID_Codigo para identificar en el programa principal el código de línea escogido, tal como se muestra en la Tabla 4.3.

Código de Línea	Identificador de Código		ID_Código
	Decimal	Binario	
NRZ	192	1100 0000	1
RZ	194	1100 0010	2
4B5B	195	1100 0011	3
Diferencial Tipo M	196	1100 0100	4
Diferencial Tipo S	197	1100 0101	5
Manchester	198	1100 0110	6
Manchester Diferencial	199	1100 0111	7
CMI	200	1100 1000	8
AMI	202	1100 1010	9
HDB3	203	1100 1011	10
MLT3	204	1100 1100	11

Tabla 4.3 Identificadores asignados a cada uno de los códigos de línea.

El diagrama de flujo de este proceso se lo muestra en la Figura 4.20.

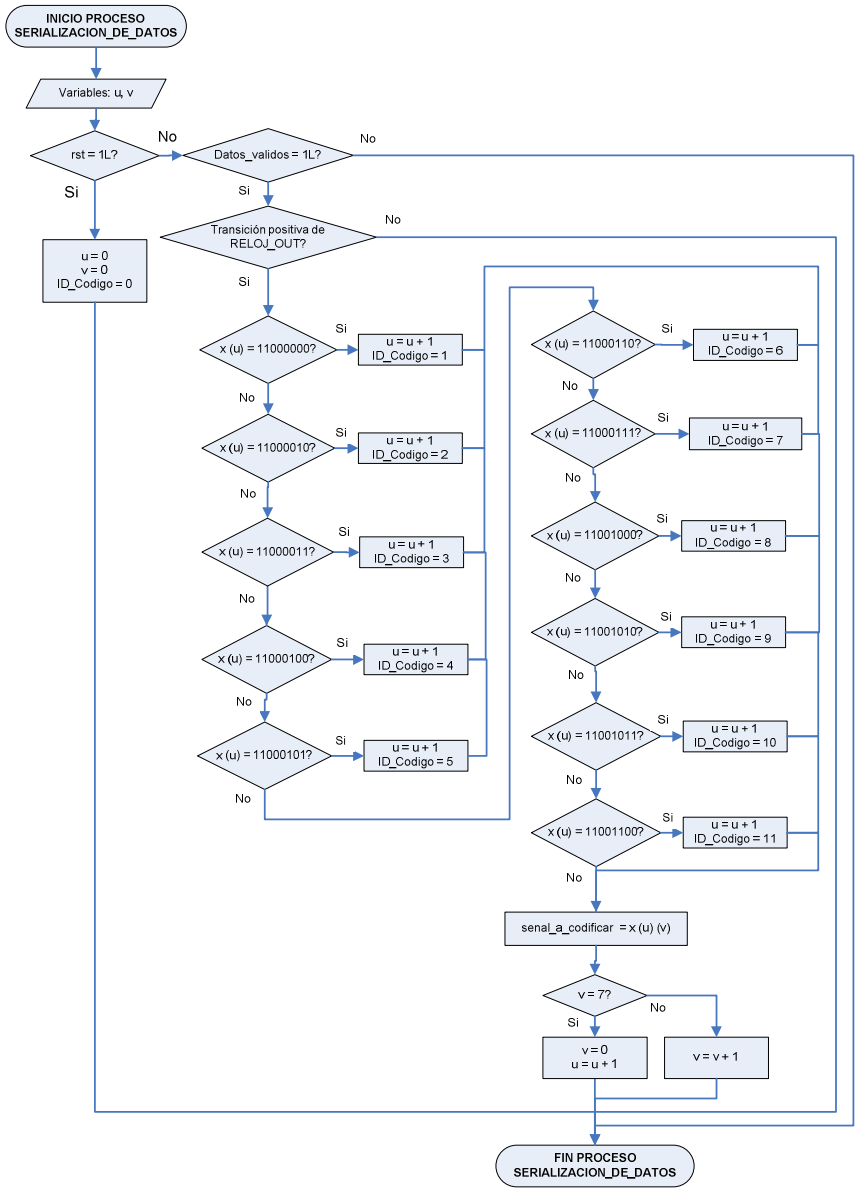


Figura 4.20 Diagrama de Flujo del proceso Serialización de Datos en VHDL.

4.7.5 CODIFICACIÓN

La programación en VHDL para la codificación con cada código de línea se realiza en componentes diferentes, las cuales son segmentos de código convencional utilizadas por el programa principal.

La elección del Código de Línea que se va a implementar se establece de acuerdo al Identificador de Código, ID_Codigo, que forma parte del flujo de bits

que se envía de MATLAB a la Spartan – 3E Starter Kit Board. En relación a las señales de entrada, siempre ingresan del código principal a todas las componentes, realizándose la codificación con todos los códigos de línea de manera simultánea. A pesar de esto, los valores que se muestran en la Tabla 4.4 identifican al código de línea respectivo, lo que permite la salida de las señales, desde la componente respectiva (sólo una componente dependiendo del código seleccionado) al código principal, obteniéndose la codificación con uno sólo de los Códigos de Línea.

Código de Línea	Identificador de Codigo		Componente
	Decimal	Binario	
NRZ	192	1100 0000	NRZ_codificacion
RZ al 50%	194	1100 0010	RZ_codificacion
4B5B	195	1100 0011	C4B5B_codificacion
Diferencial Tipo M	196	1100 0100	DIF_M_codificacion
Diferencial Tipo S	197	1100 0101	DIF_S_codificacion
Manchester	198	1100 0110	Manchester_codificacion
Manchester Diferencial	199	1100 0111	Manch_dif_codificacion
CMI	200	1100 1000	CMI_codificacion
AMI	202	1100 1010	AMI_codificacion
HDB3	203	1100 1011	HDB3_codificacion
MLT3	204	1100 1100	MLT3_codificacion

Tabla 4.4 Elección de un Código de Línea a ser Implementado de acuerdo al ID_Codigo.

Todos los códigos en VHDL de cada componente de codificación contienen dos entradas digitales: *RELOJ_OUT* o *RELOJ_OUT_Doble*, señales de reloj con la frecuencia adecuada (dependiendo de la velocidad escogida por el usuario en la interfaz gráfica: 2400, 4800, 9600 y 19200 bps) con una de las cuales se procederá a codificar; y *senal_a_codificar* que es la señal de datos.

Para la visualización de la codificación de cada uno de los códigos ya sean unipolares, polares o bipolares, se crean dos señales *salida1* y *salida2*, las cuales toman valores 0L o 1L cumpliendo con cada uno de los procesos de la

componente seleccionada. Los valores que toman dichas señales constituyen las entradas al circuito externo interpretador unipolar a bipolar, dando como resultado tres niveles de voltaje, - 5, 0 y 5 [V], tal como se muestra en la Tabla 4.5.

Niveles de Voltaje	salida1	salida2
0 [V]	0L	0L
- 5 [V]	0L	1L
+ 5 [V]	1L	0L
Restringido	1L	1L

Tabla 4.5 Niveles de voltaje de acuerdo a las señales de salida.

El diagrama de flujo del proceso codificación se muestra en la Figura 4.21.

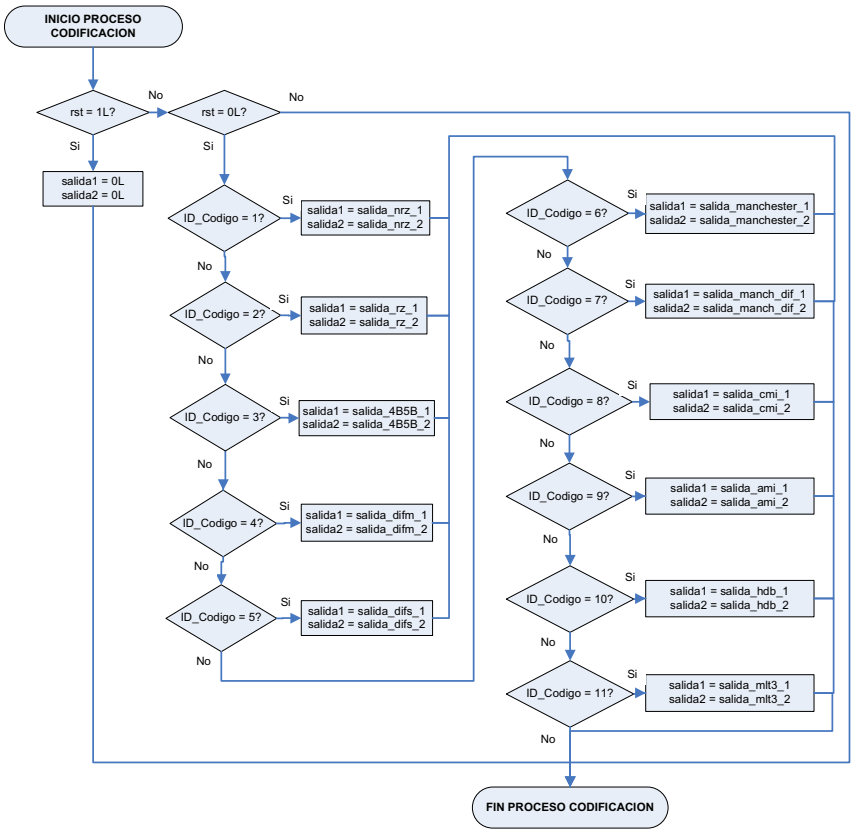


Figura 4.21 Diagrama de Flujo del proceso Codificación en VHDL.

La simulación de la codificación de cada componente se la realiza con la herramienta Testbench del Software Xilinx ISE 10, dicha herramienta permite

verificar el funcionamiento de la programación en VHDL de cada uno de los códigos de línea.

Para esto, se crea una entidad vacía como testbench, incluyendo en la arquitectura una señal interna por cada entrada o salida de la entidad de control, y por último se ingresan patrones en las señales de entrada.

Para todos los casos de simulación, para la entrada *senal_a_codificar* se ha escogido como patrón los caracteres ASCII de 8 bits *a* y *b*, que en binario equivale a 1000110 y 01000110 respectivamente. La señal de reloj, por motivos de simulación y visualización, es de 50 MHz, con un período de 20 ns.

4.7.5.1 Codificación NRZ

La componente *NRZ_codificacion* implementa el algoritmo de dicho código.

Esta componente consta de las señales descritas en el proceso de codificación. Dos señales de entrada: *RELOJ_OUT* y *senal_a_codificar*, de tipo std_logic; y dos señales de salida: *salida1* y *salida2*, también de tipo std_logic.

Además contiene el proceso llamado *NRZ_cod*. Este proceso es sensible a la señal de reloj *RELOJ_OUT*; y contiene dos variables internas: *qin_1* y *qin_2*, con valores iniciales iguales a 0L. En estas variables se almacenan los valores que deben tomar las salidas: *salida1* y *salida2* respectivamente.

En la Figura 4.22 se muestra el diagrama de flujo de la componente de codificación NRZ, *NRZ_codificacion*.

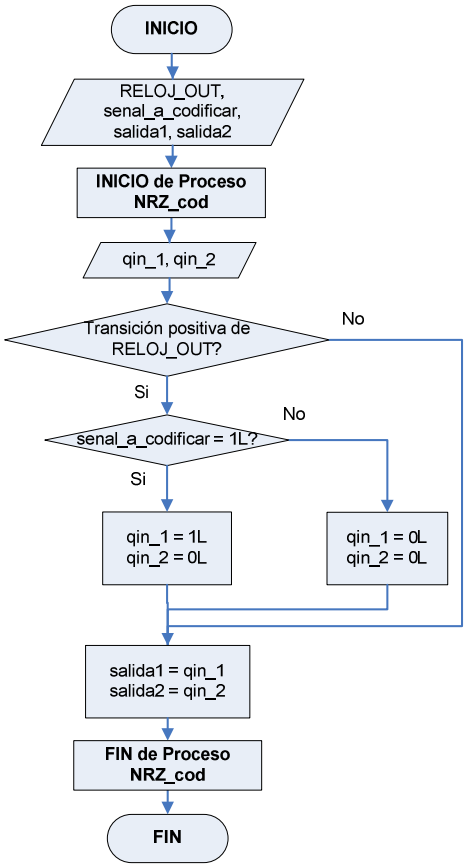


Figura 4.22 Diagrama de Flujo Codificación NRZ en VHDL.

Tal como se puede observar en el diagrama de flujo de la Figura 4.22, las señales de salida sólo cambiarán cada transición positiva de la señal de reloj, *RELOJ_OUT*.

En la Figura 4.23 se presentan los resultados de la simulación realizada en Testbench de Xilinx ISE, confirmando la correcta operación del código sintetizado. La primera columna muestra los nombres de las señales que intervienen en esta componente.

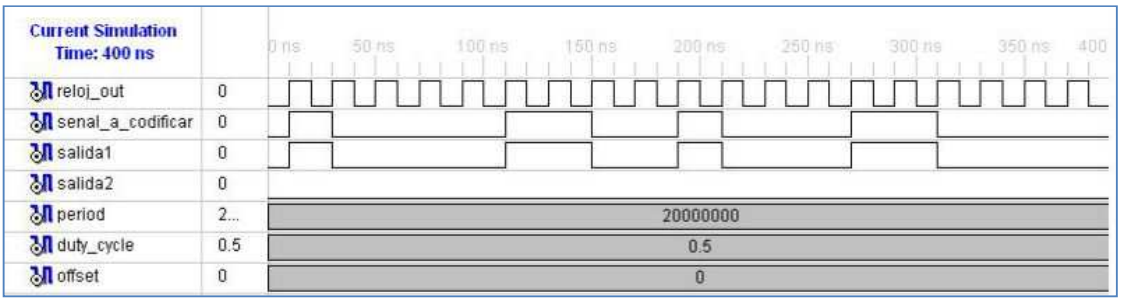


Figura 4.23 Simulación en Testbench de la codificación NRZ en VHDL.

Debe tomarse en cuenta que *salida2* siempre será igual a 0L debido a que es un código Unipolar.

4.7.5.2 Codificación RZ al 50%

Para la codificación RZ al 50% se hace uso de la componente *RZ_codificacion*, cumpliendo con el retorno a 0L de la señal transcurrido la mitad del tiempo de bit.

Esta componente consta de las señales de entrada: *RELOJ_OUT_Doble* y *senal_a_codificar*, ambas de tipo std_logic; y las señales de salida: *salida1* y *salida2*, también de tipo std_logic.

La componente incluye el proceso *RZ_cod*, sensible a la señal de reloj *RELOJ_OUT_Doble*. Este proceso tiene tres variables: *qin_1* de tipo std_logic y valor inicial 0L, *qin_2* de tipo std_logic y valor inicial igual a 0L, contar de tipo natural en el rango de 0 a 2, con valor inicial 1.

En la Figura 4.24 se muestra el diagrama de flujo de la componente de codificación RZ.

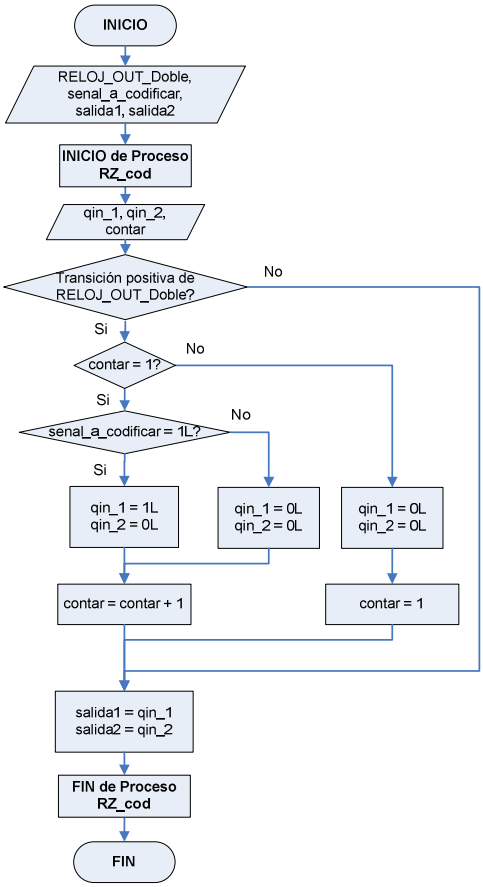


Figura 4.24 Diagrama de Flujo Codificación RZ al 50% en VHDL.

Como se puede apreciar en el diagrama de flujo, se utiliza la señal de reloj *RELOJ_OUT_Doble* (señal del doble de frecuencia de la señal *RELOJ_OUT*), debido a que la señal codificada necesita transiciones a la mitad del tiempo de bit. En cada transición positiva de esta señal de reloj se actualizan los valores de las salidas: *salida1* y *salida2*, valores que previamente son guardados en las variables *qin_1* y *qin_2.*

La variable *contar* es igual a 1 cuando ocurre una transición positiva de *RELOJ_OUT*, mientras que cuando ocurre una transición negativa *contar* es igual a 2, valores que son establecidos mediante la señal *RELOJ_OUT_Doble*. De esta manera cuando *contar* es 2, las señales de salida representarán 0 [V].

En la Figura 4.25 se presentan los resultados de la simulación en Testbench, donde se puede apreciar que la señal *RELOJ_OUT_DOBLE* es del doble de frecuencia que la señal *RELOJ_OUT*, además que la señal *salida1* es 0L cada vez que sucede una transición negativa de *RELOJ_OUT*.

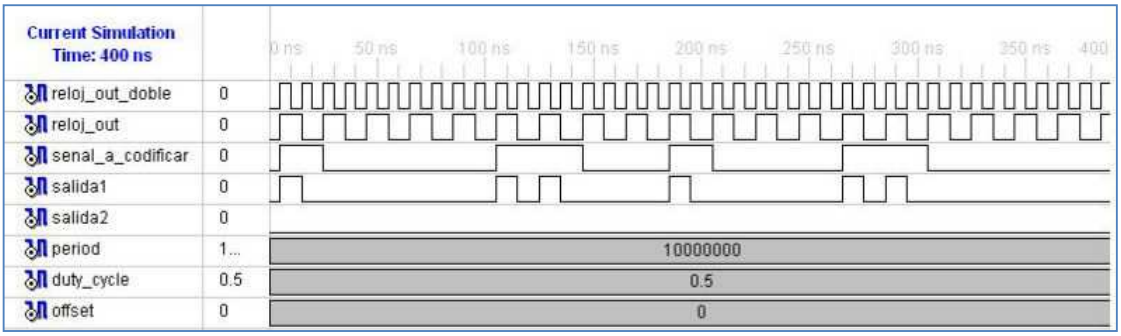


Figura 4.25 Simulación en Testbench de la codificación RZ al 50% en VHDL.

La señal *salida2* siempre es igual a 0L debido a que es un código Unipolar.

4.7.5.3 Codificación 4B5B

La componente que contiene el código unipolar 4B5B es *C4B5B_codificacion*, tiene las señales de entrada: *RELOJ_OUT, RELOJ_OUT_4B5B*, *rst* y *senal_a_codificar*, todas de tipo std_logic; en cuanto a las señales de salida están: *salida1* y *salida2*. La señal *codift* está declarada dentro de la arquitectura, ya que dicha señal va a ser utilizada en dos procesos distintos que constituyen la componente.

En la Figura 4.26 se muestra el diagrama de flujo de la componente de codificación 4B5B.

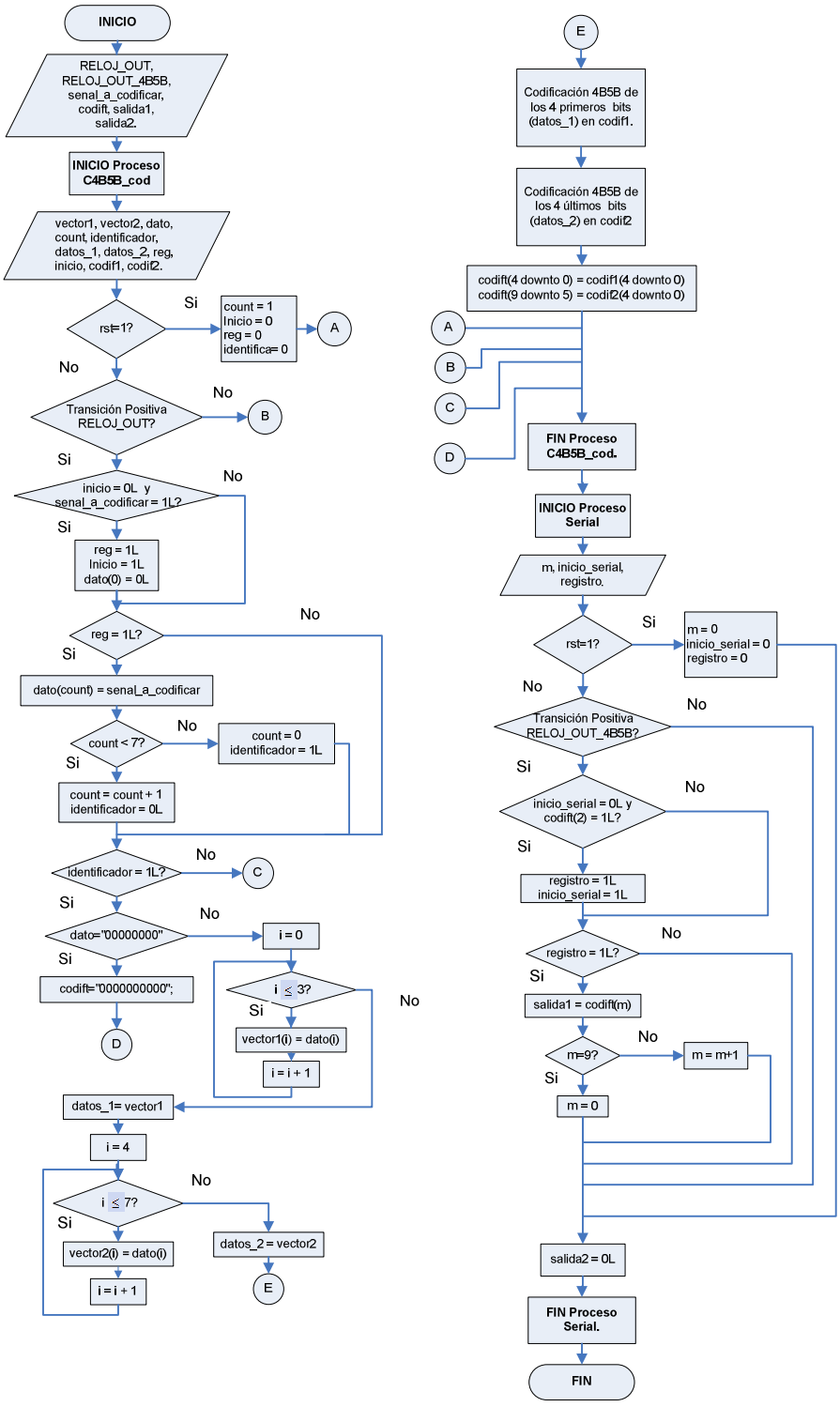


Figura 4.26 Diagrama de Flujo Codificación 4B5B en VHDL.

El primer proceso, *C4B5B_cod,* es sensible a la señal de entrada *RELOJ_OUT*, contiene variables internas que nos permiten almacenar los valores necesarios

para que se ejecute la codificación 4B5B. En este proceso se toma la señal de entrada *senal_a_codificar*, se extraen los bits en vectores de cuatro elementos (almacenado en las variables *vector1* y *vector2*) para su respectiva equivalencia en vectores de cinco bits (almacenado en *codif1* y *codif2*). La señal *codift* antes mencionada, es un vector de 10 elementos que va a estar constituida por el resultado de los diferentes valores que tomen las variables *codif1* y *codif2*.
El segundo proceso, *serial,* está regido por la señal de reloj *RELOJ_OUT_4B5B*; cabe mencionar que *RELOJ_OUT_4B5B* es 5/4 de *RELOJ_OUT*, según lo establece el código 4B5B. Este proceso dispone la presentación de los datos codificados *(codift)* en forma serial. Con este propósito, se consideró crear variables que nos permitan identificar el inicio de datos a codificarse, para una correcta visualización de los resultados en la señales de salida.

Este es el único caso que se codifica a partir del identificador inicio de datos.

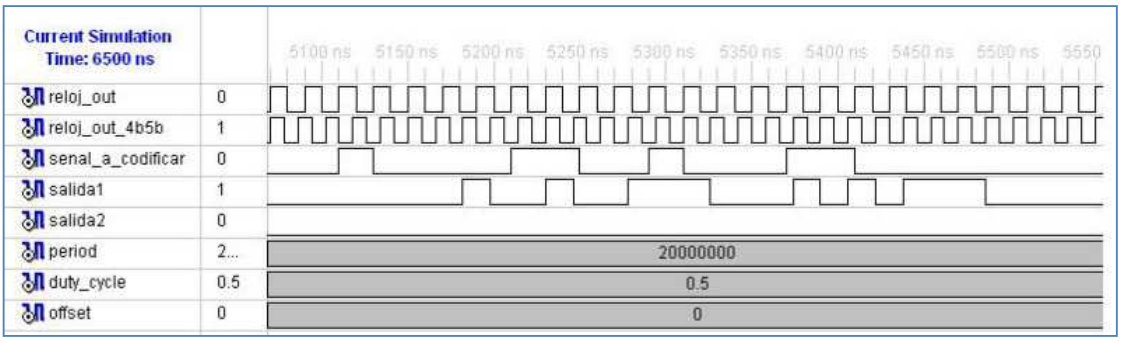


Figura 4.27 Simulación en Testbench de la codificación 4B5B en VHDL.

La Figura 4.27 muestra la simulación de la codificación 4B5B en Testbench, se puede observar que la señal de datos, *senal_a_codificar*, está dispuesta de acuerdo a *RELOJ_OUT*; el tiempo de bit de *salida1*, que representa la señal codificada, corresponde a un ciclo de *RELOJ_OUT_4B5B*.

4.7.5.4 Codificación Diferencial tipo M

La componente *DIF_M_codificacion* contiene las instrucciones en VHDL que permite ejecutar el algoritmo del código de línea Diferencial tipo M.

La componente consta de las señales de entrada: *RELOJ_OUT* y *senal_a_codificar*, y las señales de salida: *salida1* y *salida2*, todas de tipo std_logic.

Esta componente contiene el proceso *Diferencial_M_cod*, sensible a la señal *RELOJ_OUT*, utilizada debido a que la señal codificada puede cambiar de estado cada tiempo de bit. Las variables que intervienen en este proceso: *qin_1* y valor inicial 0L y *qin_2* de valor inicial 1L, almacenan los valores que deben tomar las señales de salida cada tiempo de bit. Los valores iniciales de las variables determinan que el voltaje de referencia es – 5 [V]. El diagrama de flujo de la componente que realiza la codificación Diferencial tipo M o también llamada NRZI se muestra en la Figura 4.28.

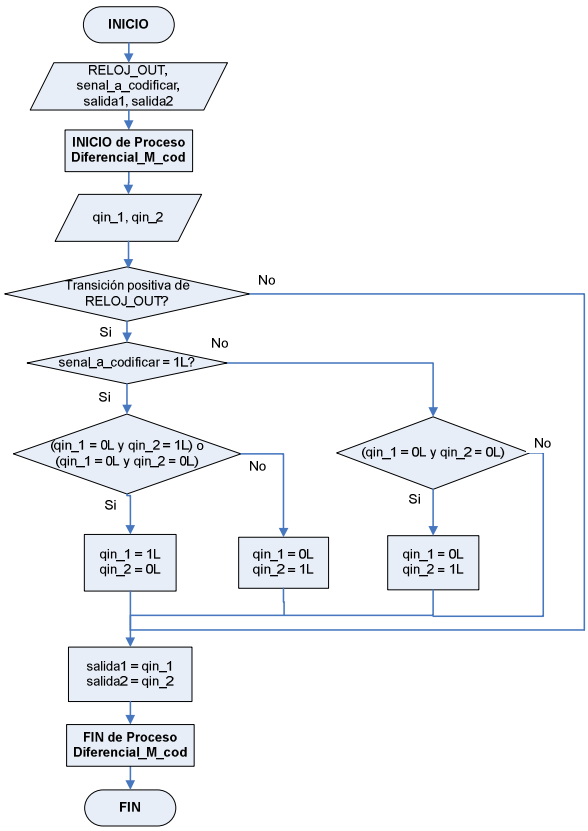


Figura 4.28 Diagrama de Flujo Codificación Diferencial Tipo M en VHDL.

Como se puede apreciar en el diagrama de flujo, se utiliza la señal de reloj *RELOJ_OUT* y cada transición positiva de esta señal se actualiza los valores de

las variables *qin_1* y *qin_2*. Estos valores se envían a las salidas: *salida1* y *salida2*, cada vez que se ejecuta el proceso *Diferencial_M_cod*.

La simulación en Testbench se presenta en la Figura 4.29. Se verifica que cada transición positiva de la señal *RELOJ_OUT* dependiendo de los niveles de voltaje de *senal_a_codificar*, pueden cambiar *salida1* y *salida2*; si es 1L las salidas cambian el nivel de voltaje y si es 0L mantienen el nivel de voltaje.

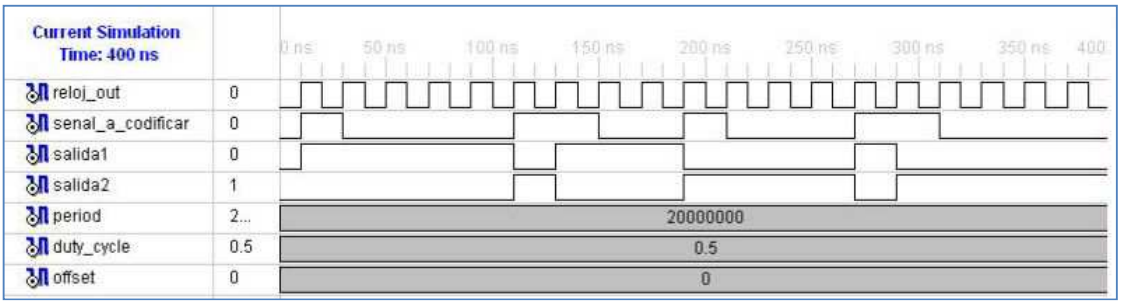


Figura 4.29 Simulación en Testbench de la codificación Diferencial Tipo M en VHDL.

Las salidas únicamente pueden tomar los valores de: *salida1* = 1L y *salida2* = 0L, o *salida1* = 0L y *salida2* = 1L, representando los niveles de voltaje +5 [V] y – 5 [V], debido a que es un código polar.

4.7.5.5 Codificación Diferencial tipo S

El contenido VHDL de la componente *Diferencial_Tipo_S* cumple con el algoritmo del código en cuestión.

Al igual que las anteriores componentes, ésta contiene una señal *RELOJ_OUT* y *senal_a_codificar*, y las señales de salida: *salida1* y *salida2*, todas de tipo std_logic.

El diagrama de flujo que refleja la programación de la componente *Diferencial_Tipo_S* se expone en la Figura 4.30.

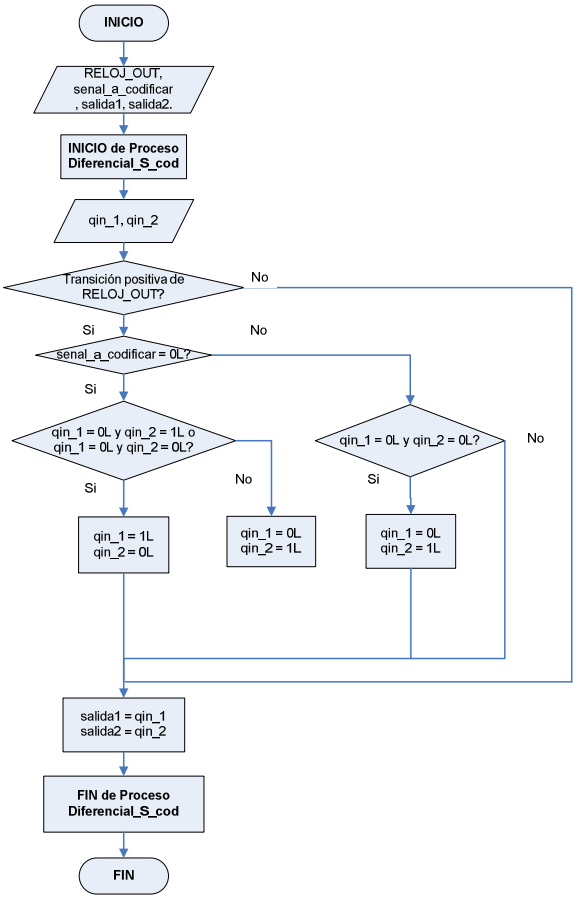


Figura 4.30 Diagrama de Flujo Codificación Diferencial Tipo S en VHDL.

El proceso involucrado en la componente es *Diferencial_S_cod,* el cual está regido con la señal de reloj *RELOJ_OUT*, tiene dos variables internas *qin_1* y *qin_2*, las mismas que almacenan los valores que van a tomar las salidas cada transición positiva del *RELOJ_OUT*, en relación a si la entrada *senal_a_codificar* es un 1L o 0L. Los valores iniciales de las variables *qin_1* y *qin_2* son 0L y 1L respectivamente. Estos valores representan el nivel de voltaje -5V. Es el mismo valor considerado en el código Diferencial Tipo M.

La simulación en Testbench de la componente se denota en la Figura 4.31.

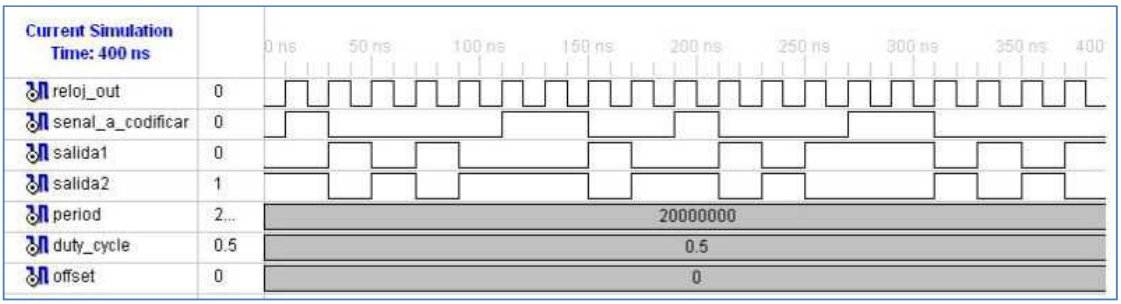


Figura 4.31 Simulación en Testbench de la codificación Diferencial Tipo S en VHDL.

Se puede apreciar en la figura anterior que se cumple con la regla de codificación, ya que el valor de referencia es -5V y como el primer bit a codificar es 1L, se mantiene el nivel, es decir *salida1* toma el valor 0L y *salida2* es 1L.

4.7.5.6 Codificación Manchester

La implementación en VHDL de la codificación Manchester se realiza en la componente *Manchester_codificacion.* En esta componente intervienen las señales de entrada: *RELOJ_OUT_Doble* y *senal_a_codificar*, y las señales de salida: *salida1* y *salida2*.

Se incluye el proceso *Manchester_cod*, sensible a la señal de reloj *RELOJ_OUT_Doble* debido a que la señal codificada tiene transiciones a medio tiempo de bit. En este proceso intervienen las variables: *qin_1* de tipo std_logic y valor inicial 0L, *qin_2* de tipo std_logic y valor inicial 0L, *senal_a_codificar_actual* también de tipo std_logic y valor inicial 0L, *contar* de tipo natural en el rango de 0 a 2 con valor inicial 1. Los valores iniciales de las variables *qin_1* y *qin_2* son igual a 0L debido a que este código no requiere de un voltaje de referencia.

En la Figura 4.32 se aprecia el diagrama de flujo de la componente *Manchester_codificacion.*

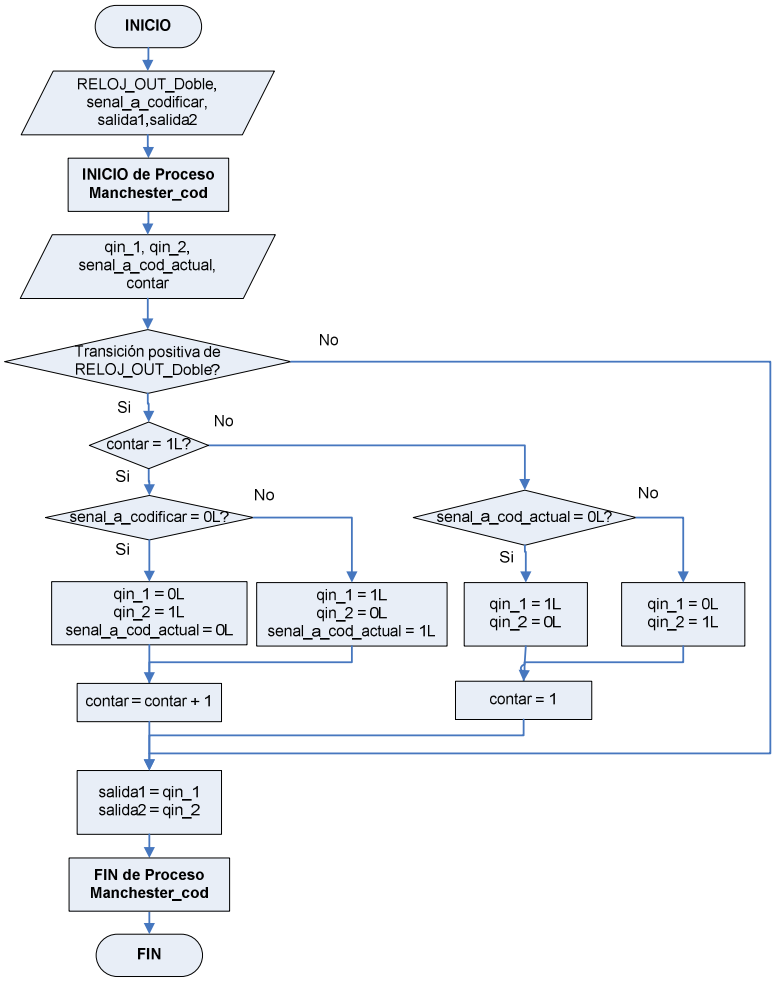


Figura 4.32 Diagrama de Flujo Codificación Manchester en VHDL.

Como se puede apreciar en el diagrama de flujo, en cada transición positiva de la señal de reloj *RELOJ_OUT_Doble* se actualizan los valores de las variables *contar*, *qin_1* y *qin_2*. Si *contar* es igual a 1 quiere decir que existe una transición positiva de la señal *RELOJ_OUT*, mientras que si *contar* es igual a 2 la transición de la señal de reloj es negativa. La variable *senal_a_codificar_actual* permite almacenar el nivel de voltaje de la señal a ser codificada cuando la señal *RELOJ_OUT* tiene un nivel alto de voltaje, para utilizarlo cuando esta señal de reloj tiene un nivel bajo. En cada transición (positiva y negativa) de *RELOJ_OUT* se actualizan los valores de *qin_1 y qin_2*, para posteriormente enviarlos a las señales de salida: *salida1* y *salida2*.

En la Figura 4.33 se muestra la simulación en Testbench del código VHDL de esta componente. Se puede observar que en las salidas codificadas existen las transiciones deseadas a la mitad del tiempo de bit, esto se logra con la señal *RELOJ_OUT_DOBLE*.

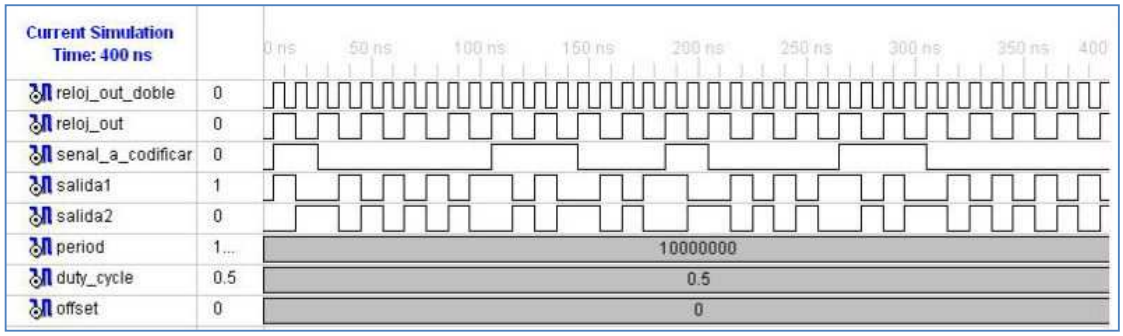


Figura 4.33 Simulación en Testbench de la codificación Manchester en VHDL.

Por ejemplo cuando *senal_a_codificar* es 1L las salidas: *salida1* y *salida2* toman los valores 1L y 0L (es decir +5 [V]) respectivamente durante el estado en alto de la señal *RELOJ_OUT*, mientras que toman los valores 0L y 1L (es decir -5 [V]) en el estado en bajo de la misma señal.

4.7.5.7 Codificación Manchester Diferencial

La componente designada para realizar la codificación Manchester Diferencial es *Manch_dif_codificacion*. Está constituida, de manera similar a la componente del código Manchester, por las señales de entrada: *RELOJ_OUT_Doble*, y *senal_a_codificar*, y las señales de salida: *salida1* y *salida2.*

En la Figura 4.34 se visualiza el diagrama de flujo de la codificación Manchester Diferencial en VHDL.

El proceso involucrado para la codificación es *Manchester_dif_cod,* el mismo que es sensible a la señal de *RELOJ_OUT_Doble*, contiene variables tales como: *qin1_anterior, qin2_anterior*, estas variables fueron creadas debido a que éste código depende del estado anterior para codificar el bit en cuestión, sus valores iniciales son 0L y 1L respectivamente, es decir, para este código, los valores asignados representan un pulso preliminar negativo. Estas variables conservan

los valores anteriores para ser utilizados en la siguiente transición de la señal de reloj RELOJ_OUT_Doble. Las variables *qin_1* y *qin_2* se van a ver reflejas en las señales de salida.

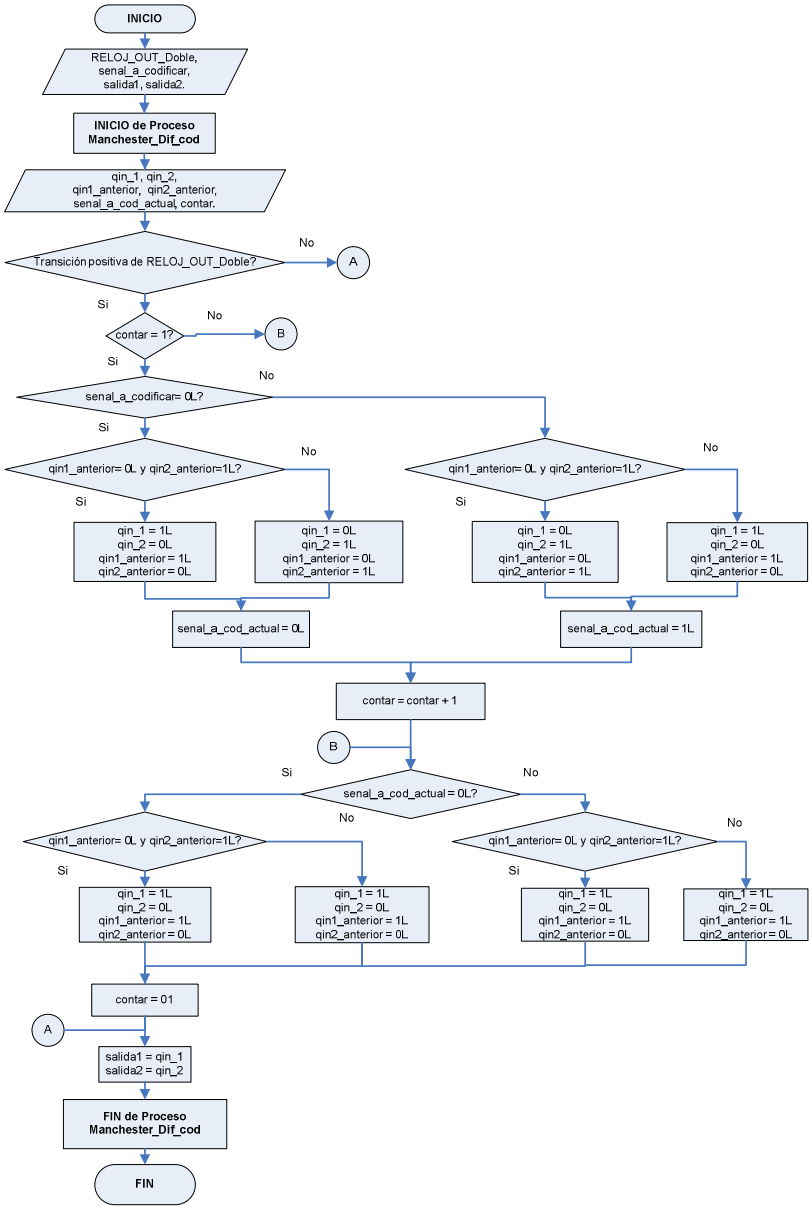


Figura 4.34 Diagrama de Flujo Codificación Manchester Diferencial en VHDL.

La simulación obtenida mediante el Testbench, se indica en la Figura 4.35 para una mejor comprensión del funcionamiento del código.

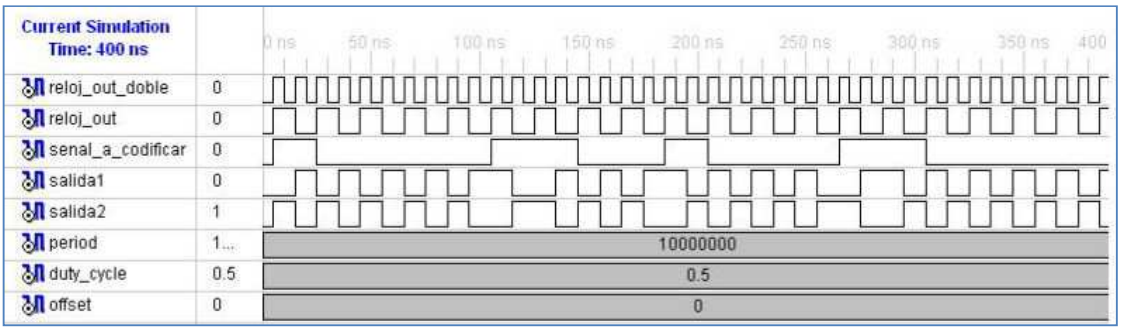


Figura 4.35 Simulación en Testbench de la codificación Manchester Diferencial en VHDL.

En la Figura 4.35 es evidente que cuando la señal de entrada toma el valor de 1L, las salidas reflejan un pulso contrario al precedente, en el caso de ser un 0L, la señal mantiene el pulso precedente, de acuerdo a las reglas de codificación que manifiesta este código.

4.7.5.8 Codificación CMI

La componente *CMI_codificacion* tiene en su entidad las señales de entrada: *RELOJ_OUT_Doble* y *senal_a_codificar*, y las señales de salida: *salida1* y *salida2.*

En esta componente se incluye el proceso *Cmi_cod*. El proceso en mención contiene en la lista de sensibilidad a la señal de reloj *RELOJ_OUT_Doble*, con lo que se puede obtener transiciones a mitad de tiempo de bit cuando la señal de datos, *senal_a_codificar*, es 0L. Las variables que intervienen son: *qin_1* de tipo std_logic y valor inicial 0L, *qin_2* de tipo std_logic y valor inicial 0L, *qin1_anterior* de tipo std_logic y valor inicial 0L, *qin2_anterior* de tipo std_logic y valor inicial 1L, *senal_a_codificar_actual* también de tipo std_logic y valor inicial 0L, *contar* de tipo natural en el rango de 0 a 2 con valor inicial 1. Las variables *qin1_anterior* y *qin2_anterior* permiten almacenar valores lógicos que representan el estado anterior en la señal codificada, debido a que sus valores iniciales son igual a 0L y 1L respectivamente, el voltaje de referencia es -5 [V].

En la Figura 4.36 se aprecia el diagrama de flujo de la componente *CMI_codificacion.*

En cada transición positiva de *RELOJ_OUT_Doble* se actualiza el valor de la variable *contar*, que puede ser 1 si hace referencia al estado en alto de *RELOJ_OUT*, o 2 si hace referencia al estado en bajo de *RELOJ_OUT*. La variable *senal_a_codificar_actual* durante el estado en alto de *RELOJ_OUT* almacena el estado de la señal de datos, para utilizarlo durante el estado en bajo. Las variables *qin_1* y *qin_2* almacenan los valores que deben tomar las salidas: *salida1* y *salida2*, dependiendo de la señal de datos.

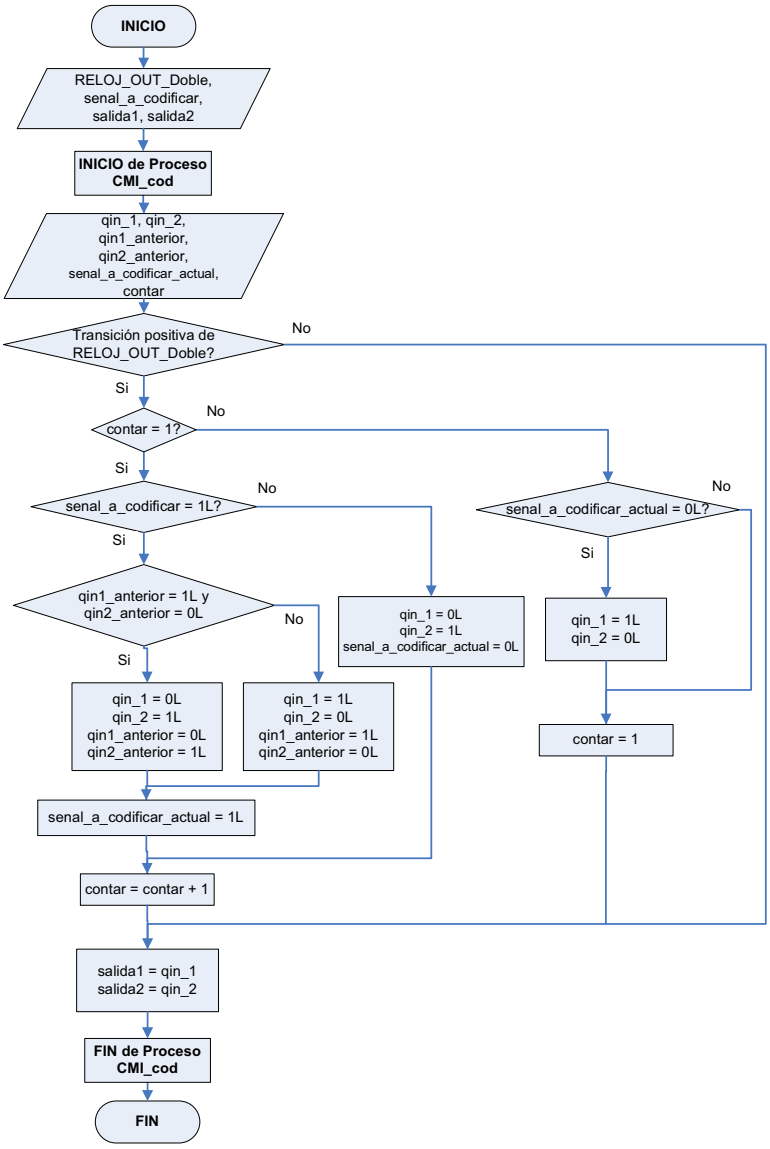


Figura 4.36 Diagrama de Flujo Codificación CMI en VHDL.

La Figura 4.37 presenta la simulación en Testbench del código VHDL de la componente *CMI_codificacion.*

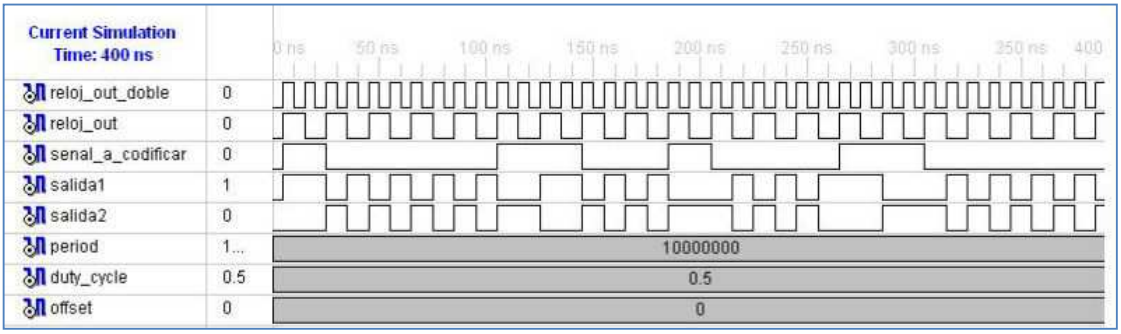


Figura 4.37 Simulación en Testbench de la codificación CMI en VHDL.

Las señales: *salida1* y *salida2* toman los valores 1L y 0L, o 0L y 1L, cuando la señal de datos es 1L, indicando el cambio de polaridad. Cuando se tiene un 0L, las salidas antes mencionadas representan un pulso positivo según el algoritmo de codificación.

4.7.5.9 Codificación AMI

La componente *AMI_codificacion* tiene como señales de entrada: *RELOJ_OUT* y *senal_a_codificar*, y como señales de salida: *salida1* y *salida2*, todas de tipo std_logic. Se utiliza como señal de reloj *RELOJ_OUT*, porque este código únicamente tiene transiciones cada tiempo de bit.

El proceso que se incluye en esta componente, es: *AMI_cod,* sensible a *RELOJ_OUT*. En este proceso se utilizan las variables internas: *qin_1* de tipo std_logic y valor inicial 0L, *qin_2* de tipo std_logic y valor inicial 0L, *qin1_anterior* de tipo std_logic y valor inicial 0L y *qin2_anterior* de tipo std_logic y valor inicial 1L. Los valores iniciales de las variables *qin1_anterior* y *qin2_anterior* se presentan de esa forma ya que el voltaje de referencia que se utiliza para la codificación es -5 [V]. La Figura 4.38 muestra el diagrama de flujo de la componente *AMI_codificacion.*

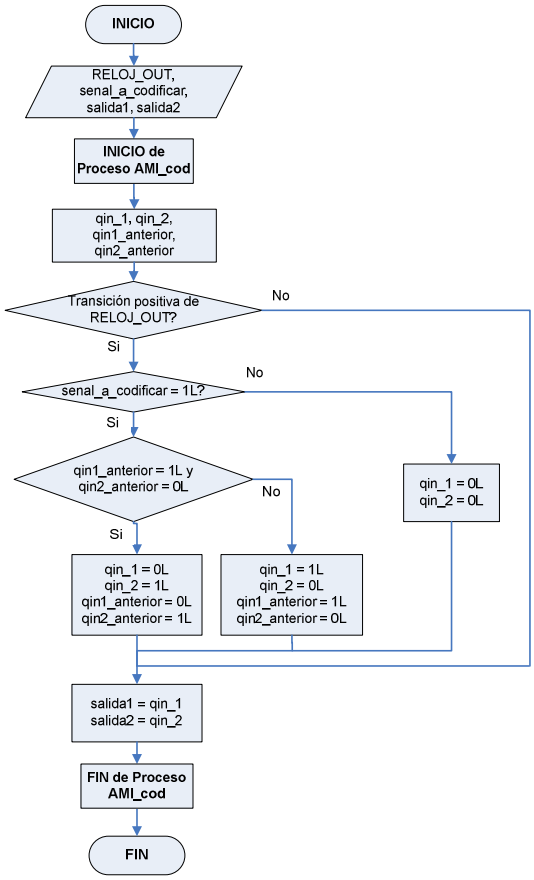


Figura 4.38 Diagrama de Flujo Codificación AMI en VHDL.

Para realizar la codificación, en *qin1_anterior* y *qin2_anterior* se guardan los últimos valores que tomaron las salidas para utilizarlos la siguiente vez que haya una transición positiva de la señal de reloj. En *qin_1 y qin_2* se almacenan los valores que deben tomar las salidas: *salida1* y *salida2.*

La Figura 4.39 muestra la simulación del código VHDL de esta componente en Testbench.

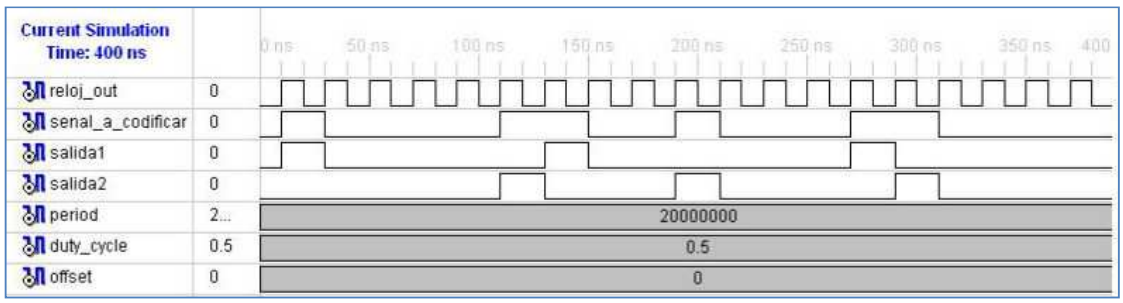


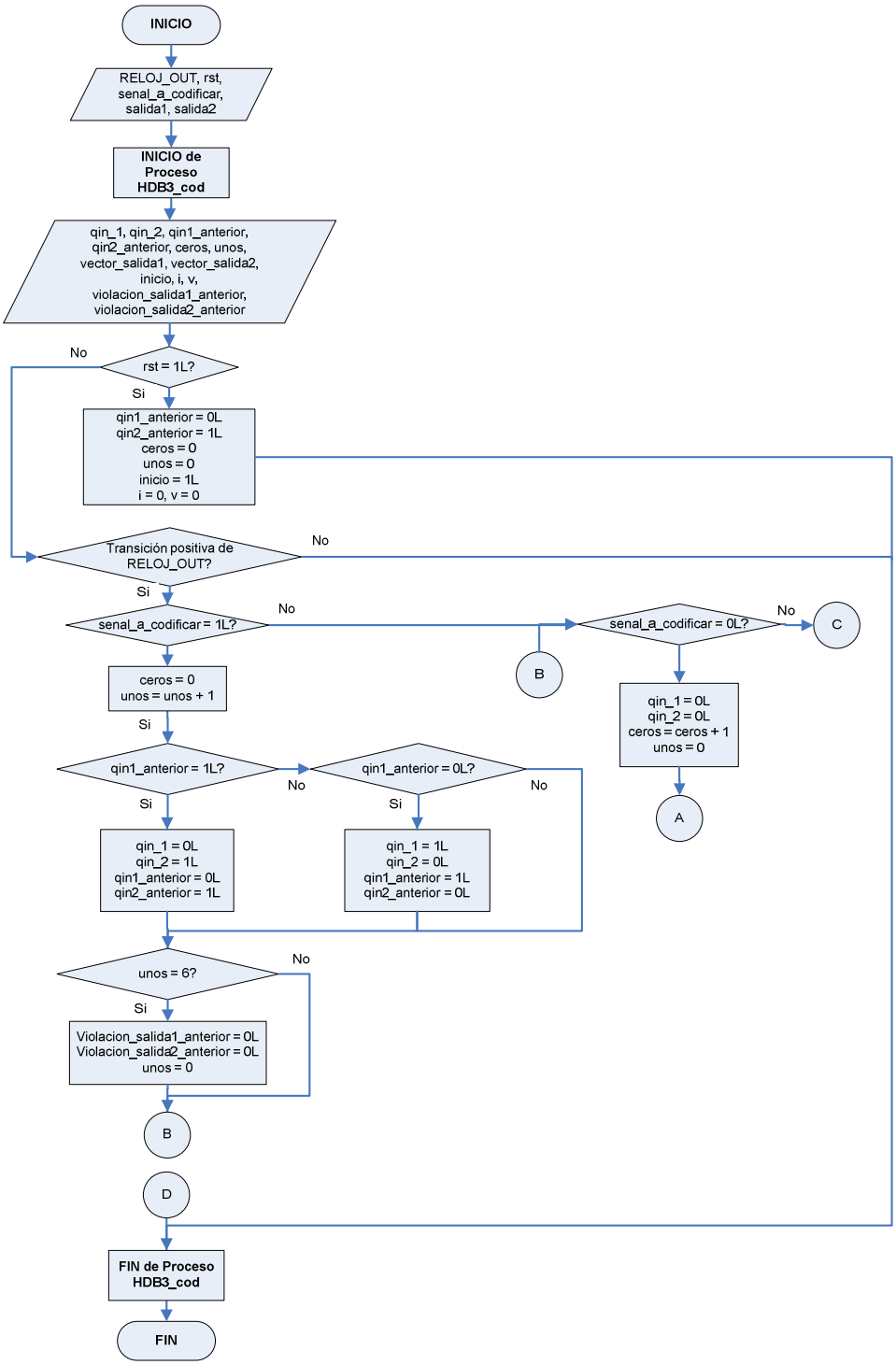
Figura 4.39 Simulación en Testbench de la codificación AMI en VHDL.

En la simulación se observa que la codificación se realiza en base a la señal de reloj *RELOJ_OUT*, tal como lo describe el diagrama de flujo. Cuando la señal de datos, *senal_a_codificar*, es 1L, las salidas: *salida1 y salida2* toman los valores 1L y 0L, o 0L y 1L respectivamente, correspondiendo a los niveles de voltaje + 5 [V] y – 5 [V]. Contrariamente cuando la señal de datos es 0L, las dos salidas, toman el valor 0L, que corresponde al nivel de voltaje 0 [V].

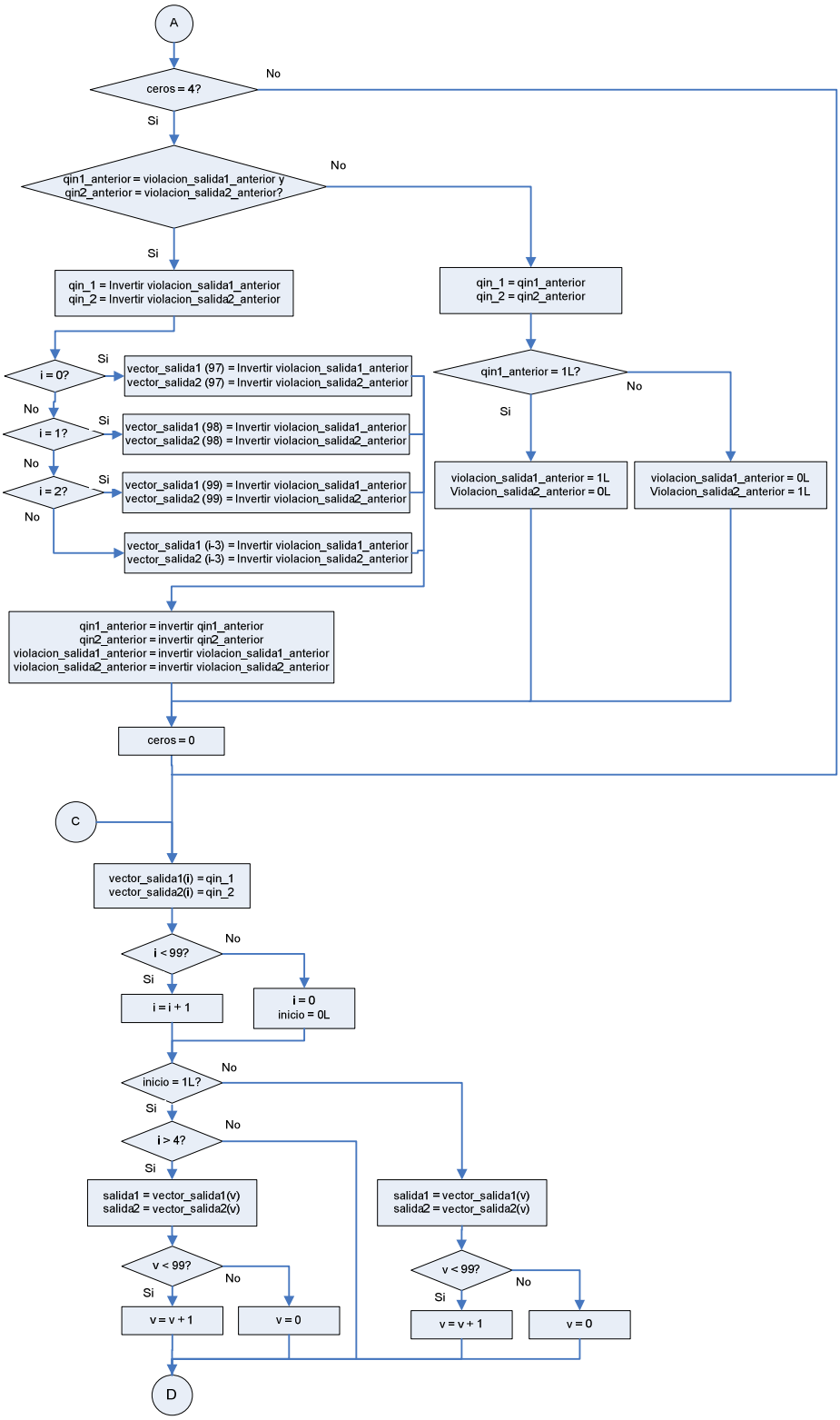
4.7.5.10 Codificación HDB3

La codificación HDB3 se realiza en la componente *HDB3_codificacion*. Las señales de entrada que utiliza esta componente son: *RELOJ_OUT*, *rst* (reset) y *senal_a_codificar*. Las señales que salen de esta componente al código principal, son: *salida1* y *salida2.*

La componente incluye el proceso *HDB3_cod* cuya lista de sensibilidad contiene a la señal de reloj *RELOJ_OUT* y *rst,* debido a que cambia el estado lógico de la señal codificada cada tiempo de bit o cada vez que la señal *rst* se encuentra en esta lógico alto. En este proceso intervienen varias variables de tipo std_logic: *qin_1* de valor inicial 0L, *qin_2* de valor inicial 0L, *qin1_anterior* de valor inicial 0L, *qin2_anterior* de valor inicial 1L, inicio de valor inicial 1L, *violación_salida1_anterior* y *violación_salida2_anterior* de valores iniciales 0L. Además de las variables *vector_salida1* y *vector_salida2* de tipo std_logic_vector de 100 elementos, *unos* y *ceros* de tipo natural y valor inicial 0, *i* y *v* de tipo natural en el rango de 0 a 99 y valor inicial 0. Los valores iniciales de las variables *qin1_anterior* y *qin2_anterior* determinan que el voltaje de referencia es – 5 [V]. En la Figura 4.40 se aprecia el diagrama de flujo de la componente *HDB3_codificacion.*


INICIO
RELOJ_OUT, rst, senal_a_codificar, salida1, salida2
INICIO de Proceso HDB3_cod
qin_1, qin_2, qin1_anterior, qin2_anterior, ceros, unos, vector_salida1, vector_salida2, inicio, i, v, violacion_salida1_anterior, violacion_salida2_anterior
rst = 1L?
No
Si
qin1_anterior = 0L
qin2_anterior = 1L
ceros = 0
unos = 0
inicio = 1L
i = 0, v = 0
Transición positiva de RELOJ_OUT?
No
Si
senal_a_codificar = 1L?
No
B
senal_a_codificar = 0L?
No
C
qin_1 = 0L
qin_2 = 0L
ceros = ceros + 1
unos = 0
A
ceros = 0
unos = unos + 1
Si
qin1_anterior = 1L?
No
Si
qin1_anterior = 0L?
No
Si
qin_1 = 0L
qin_2 = 1L
qin1_anterior = 0L
qin2_anterior = 1L
qin_1 = 1L
qin_2 = 0L
qin1_anterior = 1L
qin2_anterior = 0L
unos = 6?
No
Si
Violacion_salida1_anterior = 0L
Violacion_salida2_anterior = 0L
unos = 0
B
D
FIN de Proceso HDB3_cod
FIN

(a)



(b)

Figura 4.40 Diagrama de Flujo Codificación HDB3 en VHDL.

En el diagrama de flujo se puede apreciar que en cada transición positiva de *RELOJ_OUT* se actualizan los estados lógicos de las salidas.

Para que esto suceda, en cada transición positiva de la señal de reloj, *qin_1* y *qin_2* guardan los valores que deben ser almacenados en los vectores de salida (*vector_salida1* y *vector_salida2*). *qin1_anterior* y *qin2_anterior* almacenan el último estado que adquirieron las salidas cuando no son iguales a 0L, para utilizarlas en la siguiente transición de *RELOJ_OUT*.

La variable *ceros* indica el número de 0L consecutivos que llega en la señal de datos. En forma contraria, si la señal de datos es un 1L, esta variable tomará el valor de 0 para iniciar una nueva cuenta de 0L consecutivos. Cuando la variable *ceros* es igual a cuatro, el código VHDL reconoce que se tiene que realizar un relleno y una violación, o sólo una violación; y obtendrá el valor de *cero* para empezar una nueva cuenta.

La identificación de un relleno y violación, o sólo violación; se realiza con el valor de las variables *violacion_salida1_anterior* y *violación_salida2_anterior* que almacenan el valor de la última violación. Se identifica que se tiene que realizar un relleno y violación, cuando estas dos variables tienen los mismos estados lógicos que *qin1_anterior* y *qin2_anterior,* respectivamente. Caso contrario se le dará el tratamiento de una violación.

La variable *unos* ayuda a reconocer el identificador de inicio para encerar *violacion_salida1_anterior* y *violación_salida2_anterior* e iniciar la codificación sin violación previa.

Los valores de *qin_1* y *qin_2* no se envían directamente a las salidas, sino a los vectores *vector_salida1* y *vector_salida2* donde se almacenan los estados lógicos que deben tomar las salidas. Esto se debe a que cuando se produce relleno se debe cambiar el estado lógico del tercer bit anterior codificado de 0 [V], a + 5 [V] o – 5 [V], dependiendo si *violación_salida1_anterior* y *violación_salida2_anterior* fueron 0L y 1L, o 1L y 0L, respectivamente.

La variable inicio, es igual a 1L cuando identifica si los estados lógicos almacenados en los vectores (*vector_salida1* y *vector_salida2*) corresponden a

los primeros bits codificados, caso contrario es igual a 0L. Esto se realiza para que la señal codificada obtenga un valor desconocido, en este caso 0L, hasta que se codifiquen los cuatro primeros bits de datos, posterior a lo cual se obtendrá en *salida1* y *salida2* los estados correspondientes a la codificación.

Cuando existen rellenos, se debe cambiar el tercer bit anterior, y si se obtuvieran directamente los estados de las salidas (*salida1 y salida2*), sin pasar por vectores, no se pudiera cambiar el estado de este bit porque ya sería publicado tres tiempos de bit antes. Es por esta razón que la señal codificada tiene un retardo de 4 tiempos de bit respecto a la señal de datos.

En la Figura 4.41 se visualiza la simulación del código VHDL de esta componente en Testbench.

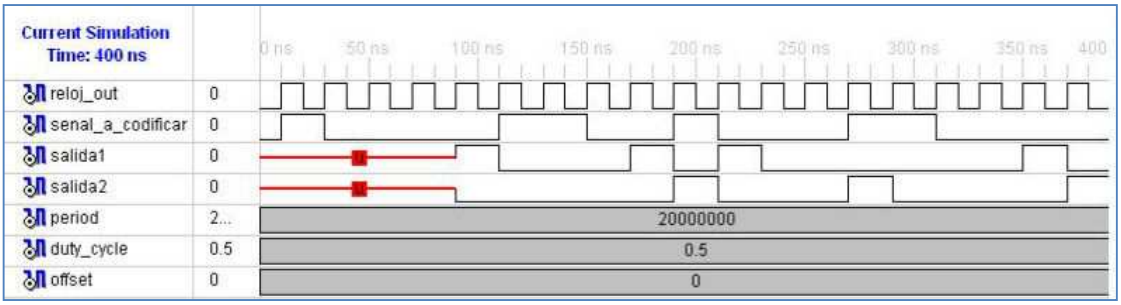


Figura 4.41 Simulación en Testbench de la codificación HDB3 en VHDL.

La simulación muestra el correcto funcionamiento de acuerdo a las reglas de codificación del HDB3. Se aprecia que la señal codificada tiene un retardo de cuatro tiempos de bit, debido al código VHDL de esta componente, tal como se explica en la descripción del diagrama de flujo. Durante este tiempo, las señales *salida1* y *salida2* toman los valores lógicos 'X' (Forcing Unknown), valor desconocido sintetizable en VHDL. Al quinto tiempo de bit se visualiza que *salida1* y *salida2* representan la señal codificada de acuerdo a la señal de datos *senal_a_codificar*, con el respectivo retardo.

Por ejemplo, como el primer bit de la señal de datos, es 1L las salidas: *salida1* y *salida2*, toman los valores 1L y 0L respectivamente, que representa para la señal codificada el nivel de voltaje +5 [V]. Mientras que las dos salidas toman los valores 0L cuando existen menos de tres bits 0L consecutivos.

4.7.5.11 Codificación MLT3

La componente *MLT3_codificacion*, al igual que las precedentemente descritas tiene como señales de entrada: *RELOJ_OUT* y *señal_a_codificar*, y las señales de salida: *salida1* y *salida2*. El diagrama de flujo representado en la Figura 4.42, caracteriza la programación en VHDL del código en cuestión.

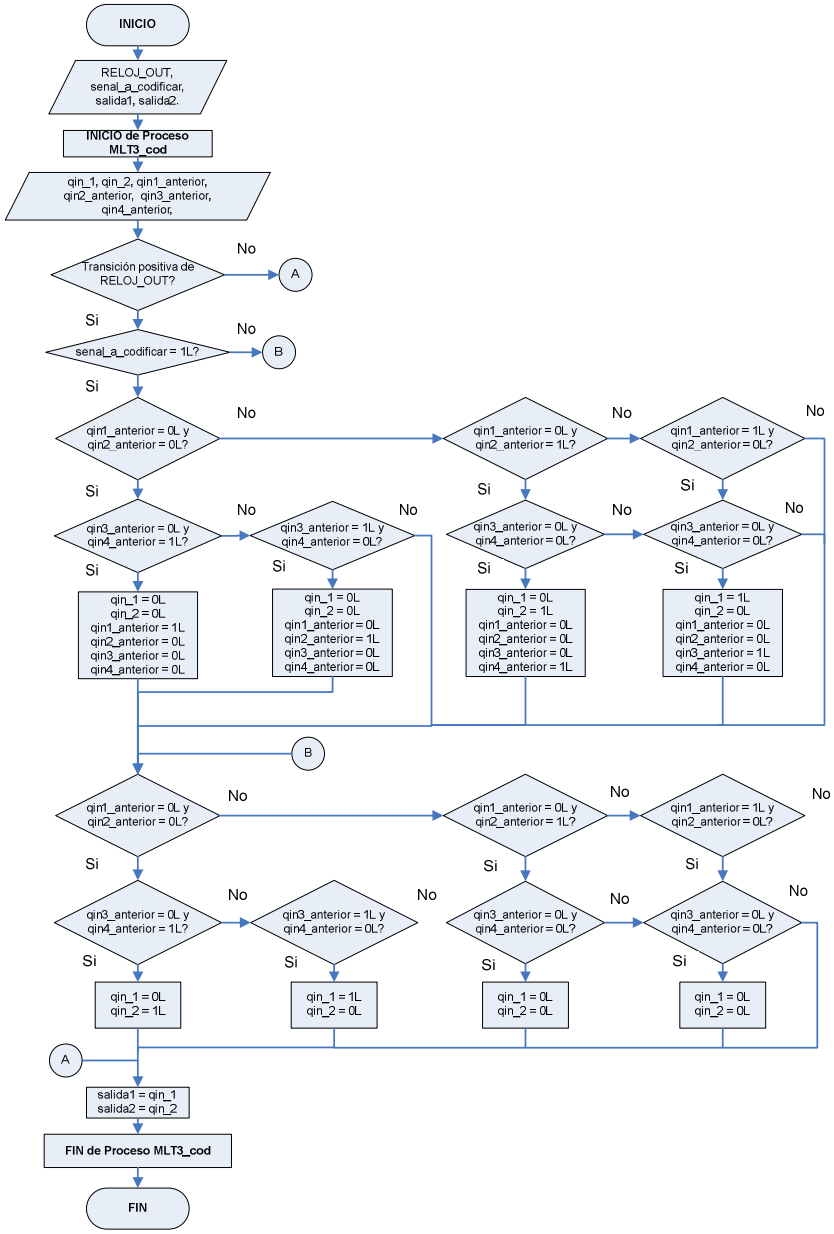


Figura 4.42 Diagrama de Flujo Codificación MLT3 en VHDL.

La componente sostiene el proceso *MLT3_cod*, éste tiene en su lista de sensibilidad la señal *RELOJ_OUT*, este proceso utiliza cuatro variables *qin1_anterior*, *qin2_anterior*, *qin3_anterior* y *qin4_anterior*, con el propósito de recopilar los valores anteriores necesarios para la codificación, para éste código bipolar es trascendente saber los dos estados previos para la interpretación del siguiente bit; cada par de variables representa un estado.

Los valores inciales de *qin1_anterior* y *qin2_anterior* son 0L, esto significa que el primer estado tiene un nivel de voltaje de 0V, las restantes variables, *qin3_anterior* y *qin4_anterior*, toman los valores de 1L y 0L respectivamente, dichos valores figuran el segundo estado anterior con un valor de +5V.

Las variables *qin_1* y *qin_2* se van a reflejar en las señales de salida, según el proceso se ejecute. En la Figura 4.43 se presenta la simulación en Testbench de la componente *MLT3_codificacion*.

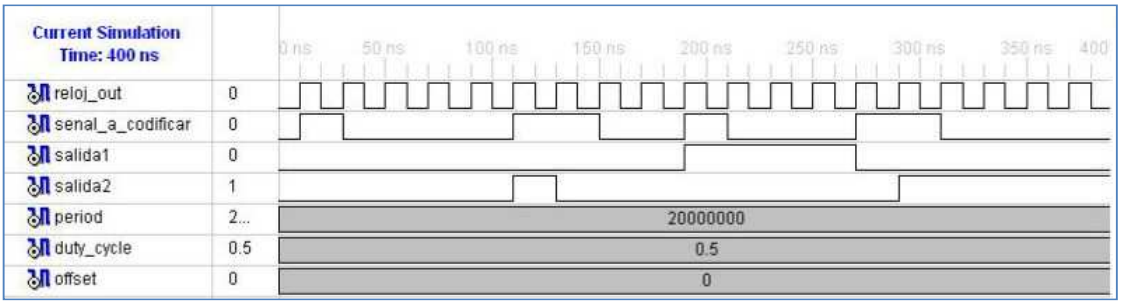


Figura 4.43 Simulación en Testbench de la codificación MLT3 en VHDL.

Recordemos que los estados anteriores eran 0V y +5V; si el primer bit a codificar es un 1L, según las reglas de codificación, el resultado es un valor de 0V, como se muestra en la simulación. Un 0L mantiene el estado anterior. El siguiente 1L va a proporcionar un valor de -5V, es decir, los valores de las señales de salida: *salida1* y *salida2* son: 0L y 1L, respectivamente.

4.7.6 DECODIFICACIÓN

Para continuar la construcción del diseño en forma jerárquica, al igual que en la codificación, la programación en VHDL para la decodificación con cada código de

línea también se la realiza en componentes diferentes, utilizadas por el programa principal cuando sea del caso.

La decodificación debe realizarse con el mismo Código de Línea que se codificó. Entonces se identifica a este, con el mismo Identificador de Código, ID_Codigo, que en la codificación, tal como lo muestra la Tabla 4.6, obteniéndose los resultados de la decodificación con un sólo código de línea.

Código de Línea	Identificador de Codigo		Componente
	Decimal	Binario	
NRZ	192	1100 0000	NRZ_decodificacion
RZ al 50%	194	1100 0010	RZ_decodificacion
4B5B	195	1100 0011	C4B5B_decodificacion
Diferencial Tipo M	196	1100 0100	DIF_M_decodificacion
Diferencial Tipo S	197	1100 0101	DIF_S_decodificacion
Manchester	198	1100 0110	Manchester_decodificacion
Manchester Diferencial	199	1100 0111	Manch_dif_decodificacion
CMI	200	1100 1000	CMI_decodificacion
AMI	202	1100 1010	AMI_decodificacion
HDB3	203	1100 1011	HDB3_decodificacion
MLT3	204	1100 1100	MLT3_decodificacion

Tabla 4.6 Elección de un Código de Línea a ser Implementado de acuerdo al ID_Codigo.

Siempre las señales de entrada provienen del código principal a la componente y se consigue la salida decodificada desde la componente correspondiente hacia el código principal.

Todas las componentes de decodificación contienen tres entradas digitales: *RELOJ_OUT* o *RELOJ_OUT_Doble*, señales de reloj con una de las cuales se procederá a codificar; *entrada_cod1* y *entrada_cod2*, señales de datos a ser decodificadas que provienen del exterior de la tarjeta de entrenamiento. Se utilizan dos señales de entrada de datos que representan la señal codificada debido a que el FPGA no puede recibir niveles de voltaje negativos. Los valores que toman estas señales de entrada al FPGA son las salidas del circuito externo interpretador bipolar - unipolar. Este circuito externo obtiene como resultado las

señales de salida *entrada_cod1* y *entrada_cod2* dependiendo del nivel de voltaje de entrada, tal como se muestra en la Tabla 4.7.

Niveles de Voltaje	entrada_cod1	entrada_cod2
Restringido	0L	0L
- 5 [V]	0L	1L
0 [V]	1L	1L
+ 5 [V]	1L	0L

Tabla 4.7 Niveles de voltaje y los respectivos valores de las señales entrada_cod1 y entrada_cod2.

Debido a que los códigos de línea que se implementan son unipolares, bipolares y polares, los niveles de voltaje de entrada en la decodificación son únicamente: - 5 [V], 0 [V] y +5 [V], por lo que las entradas: *entrada_cod1* y *entrada_cod2* jamás tomarán el valor 0L al mismo tiempo.

En la Figura 4.44 se muestra el diagrama de flujo del proceso decodificación.

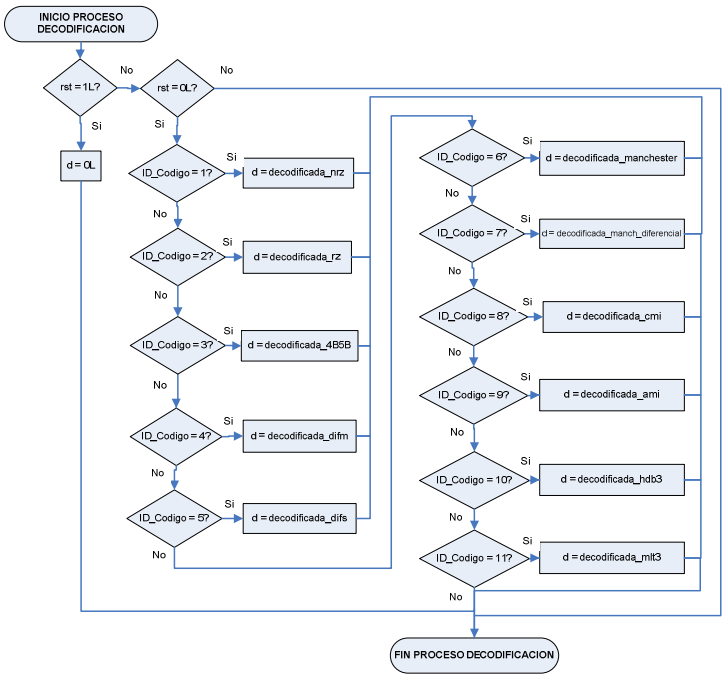


Figura 4.44 Diagrama de Flujo del Proceso Decodificación en VHDL.

La simulación de cada una de las componentes VHDL de decodificación se realiza con la herramienta Testbench del Software Xilinx ISE 10, al igual que en la

codificación nos permite verificar la correcta decodificación con cada uno de los códigos de línea. Cada componente tiene su archivo de simulación .vhw. El patrón que se ingresa en las señales de entrada es la señal codificada correspondiente a los caracteres ASCII de 8 bits a y b, en binario 1000110 y 01000110 respectivamente, con el objetivo de mantener el formato y visualizar que en la decodificación se obtiene como salida la señal de entrada de la simulación de codificación, ya que es el proceso inverso. De igual manera, para continuar con el formato en todas las simulaciones, la señal de reloj tiene una frecuencia de 50 MHz.

4.7.6.1 Decodificación NRZ

La componente que contiene la decodificación del código NRZ, se denomina *NRZ_decodificacion*. Las señales de entrada son: la señal de reloj *RELOJ_OUT*, *entrada_cod1* y *entrada_cod2*; y de salida: la señal *d*. El diagrama de flujo descrito en la Figura 4.45 representa la programación de la componente correspondiente al código NRZ.

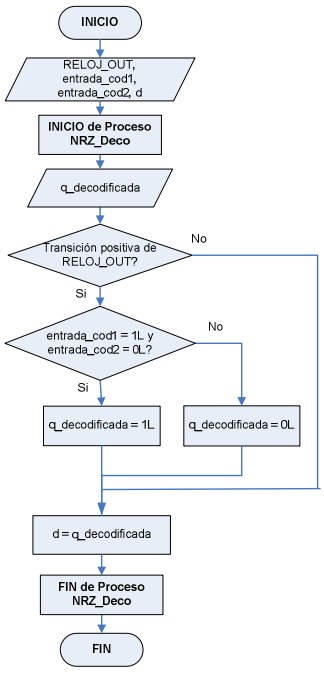


Figura 4.45 Diagrama de Flujo Decodificación NRZ en VHDL.

El proceso de la decodificación del código señalado, es relativamente sencillo, como se puede visualizar en el diagrama de flujo. Una única variable se ha considerado dentro del proceso *NRZ_Deco*, *q_decodificada* que va a permitir almacenar los valores que tome la señal de salida *d*, cada vez que haya una transición positiva de *RELOJ_OUT*.

En la Figura 4.46 se presenta la simulación de la decodificación en la componente *NRZ_decodificacion*.

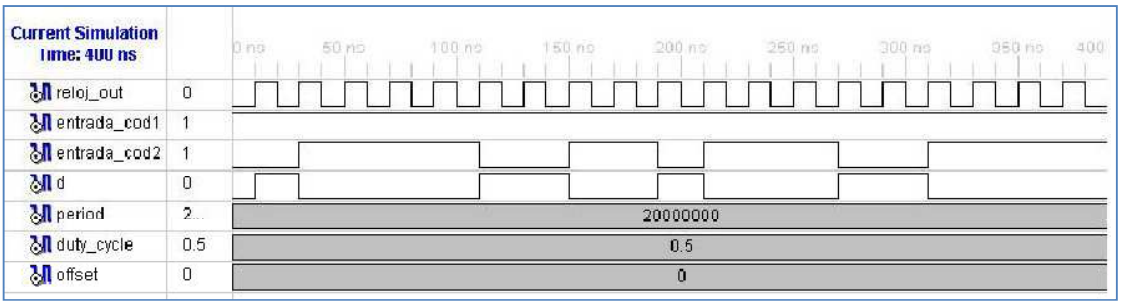


Figura 4.46 Simulación en Testbench de la decodificación NRZ en VHDL.

El código NRZ es de tipo unipolar, en la simulación se observa que cuando se tiene los valores de 1L y 0L para las señales *entrada_cod1* y *entrada_cod2* respectivamente, representan un nivel de voltaje de +5 [V], es decir un 1L, en caso contrario, se tendrá 0 [V].

4.7.6.2 Decodificación RZ al 50%

La decodificación para el código RZ al 50%, se la realiza en la componente *RZ_decodificacion*.

En cuanto a las señales de entrada y salida que presenta esta componente son las mismas que se describen en el código NRZ.

El diagrama de flujo para la decodificación RZ al 50% se exhibe en la Figura 4.47.

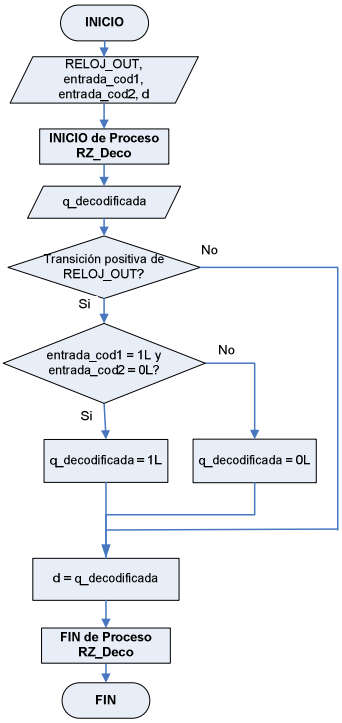


Figura 4.47 Diagrama de Flujo Decodificación RZ al 50% en VHDL.

El proceso que describe el funcionamiento de la componente es *RZ_Deco*, está regido por la señal *RELOJ_OUT*, y contiene una variable que cumple el mismo propósito que se detalla en el código NRZ. La simulación de la componente en Testbench se demuestra en la Figura 4.48.

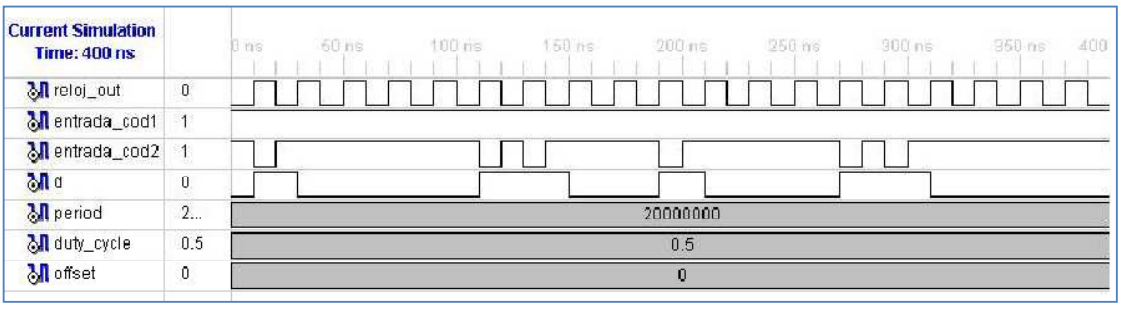


Figura 4.48 Simulación en Testbench de la decodificación RZ al 50% en VHDL.

Se puede advertir en la simulación que RZ al 50% es semejante al código NRZ, la diferencia es que se tiene retorno a cero a mitad del tiempo de bit.

4.7.6.3 Decodificación 4B5B

El diagrama de flujo mostrado en la Figura 4.49, describe el código en VHDL de la componente *C4B5B_decodificacion*.

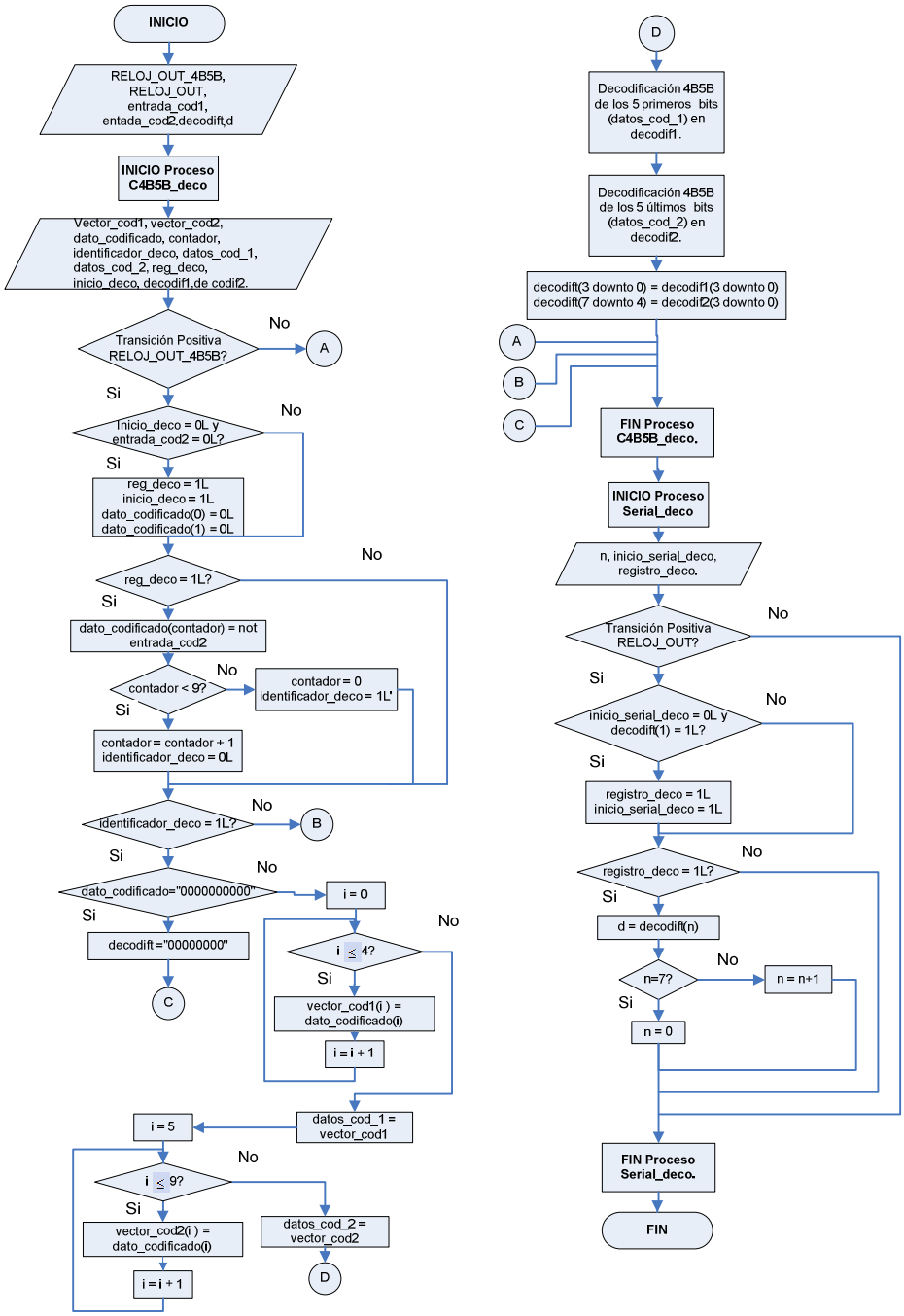


Figura 4.49 Diagrama de Flujo Decodificación 4B5B en VHDL.

Es conocido que la decodificación es el proceso inverso de la codificación, entonces el punto de partida para la implementación de la decodificación 4B5B es el código en VHDL presente en la componente de codificación 4B5B. Esta componente tiene las señales de entrada: *RELOJ_OUT_4B5B*, *RELOJ_OUT*, *entrada_cod1* y *entrada_cod2*. Además, la señal de salida *d*.

La componente de la decodificación contiene dos procesos al igual que la codificación, *C4B5B_deco* y *serial_deco*. El primer proceso está regido por la señal de reloj *RELOJ_OUT_4B5B*, ya que las entradas (*entrada_cod1* y *entrada_cod2*) llegan con un período de bit de la salida de la codificación. El segundo tiene en su lista de sensibilidad la señal *RELOJ_OUT*, con la cual se serializa el vector *decodift*, el cual contiene los datos decodificados que se reflejan en la salida *d*. En la Figura 4.50 se muestra la simulación en Testbench de la componente *C4B5B_decodificacion*.

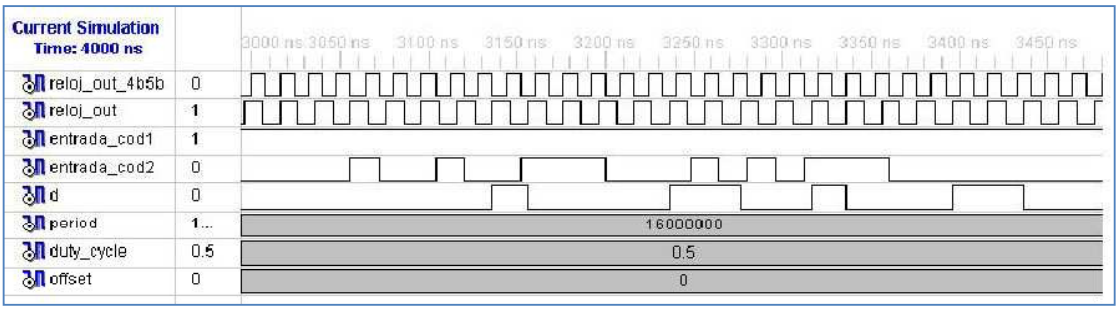


Figura 4.50 Simulación en Testbench de la decodificación 4B5B en VHDL.

En la simulación se puede apreciar dos señales de reloj, la decodificación se la hace con la señal *RELOJ_OUT_4B5B*. Debido a que el código 4B5B es de tipo unipolar, la señal *entrada_cod1* siempre va a ser 1L; en la serialización de los datos decodificados se utiliza la señal *RELOJ_OUT*. Se puede observar que la salida *d* representa los datos originales. Dicha señal se presenta después de diez transiciones de la señal *RELOJ_OUT_4B5B*, esto se debe a que el programa primero almacena diez bits y luego procede a decodificarlos.

4.7.6.4 Decodificación Diferencial Tipo M

La decodificación Diferencial tipo M se realiza en la componente *DIF_M_decodificacion*, que contiene el código VHDL correspondiente. El diagrama de flujo de esta componente se muestra en la Figura 4.51.

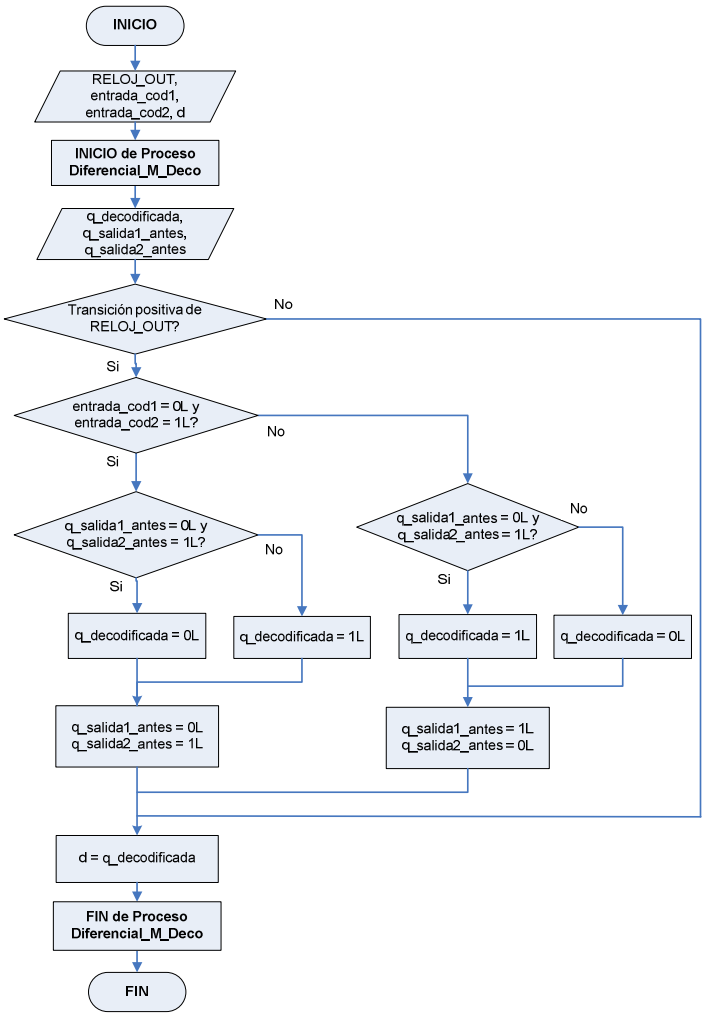


Figura 4.51 Diagrama de Flujo Decodificación Diferencial Tipo M en VHDL.

La componente *DIF_M_decodificacion* que se muestra en el diagrama de flujo tiene tres señales de entrada: *RELOJ_OUT*, *entrada_cod1* y *entrada_cod2*, y la señal de salida *d*. Estas señales ingresan al proceso *Diferencial_M_deco*, sensible a la señal *RELOJ_OUT*, ya que cada transición positiva de esta señal puede cambiar de estado la señal codificada. Las variables que intervienen en este proceso son: *q_decodificada* de tipo *std_logic* y valor inicial 0L,

q_salida1_antes de tipo *std_logic* y valor inicial 0L, y *q_salida2_antes* de tipo std_logic y valor inicial 1L. Como el voltaje de referencia debe ser el mismo en la codificación y decodificación, los valores iniciales de las variables *q_salida1_antes* y *q_salida2_antes* determinan que es – 5 [V] en la decodificación, tal como lo es en la codificación.

Se visualiza que la variable *q_decodificada* se actualiza cada transición positiva de *RELOJ_OUT* y almacena el estado lógico que debe tomar la señal de salida, *d*. *q_salida1_antes* y *q_salida2_antes* guardan los estados lógicos que tienen las entradas, para utilizarlos en el siguiente flanco ascendente de la señal de reloj.

La Figura 4.52 presenta la simulación en Testbench del código VHDL de la componente *Dif_M_decodificacion*.

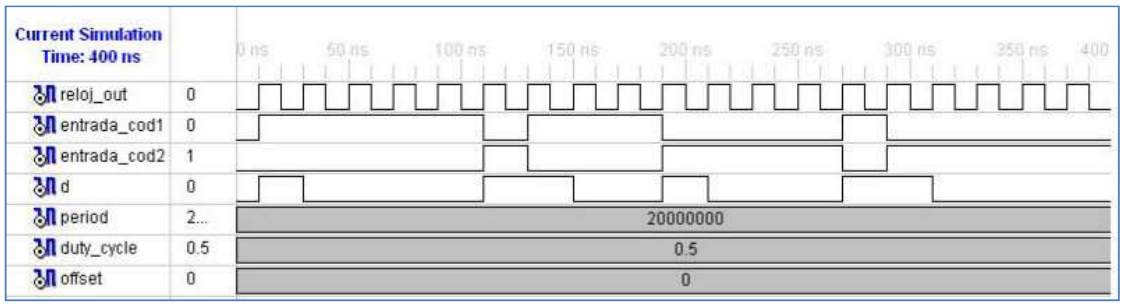


Figura 4.52 Simulación en Testbench de la decodificación Diferencial Tipo M en VHDL.

En la simulación se observa que cada transición positiva de la señal *RELOJ_OUT*, si cambian los niveles de voltaje de las entradas: *entrada_cod1* y *entrada_cod2* (representan la señal a ser decodificada); la señal decodificada, *d*, es 1L, tal como lo establece la regla de codificación de este código.
Las entradas únicamente pueden tomar los valores: *entrada_cod1* = 1L y *entrada_cod2* = 0L, o *entrada_cod1* = 0L y *entrada_cod2* = 1L, representando los niveles de voltaje +5 [V] y – 5 [V] respectivamente, debido a que es un código polar.

4.7.6.5 Decodificación Diferencial Tipo S

La componente *Dif_S_decodificacion* representa la decodificación con el código Diferencial Tipo S en VHDL. Las señales de entrada y salida son las descritas en el Diferencial Tipo M. En la Figura 4.53 se expone el diagrama de flujo de la componente.

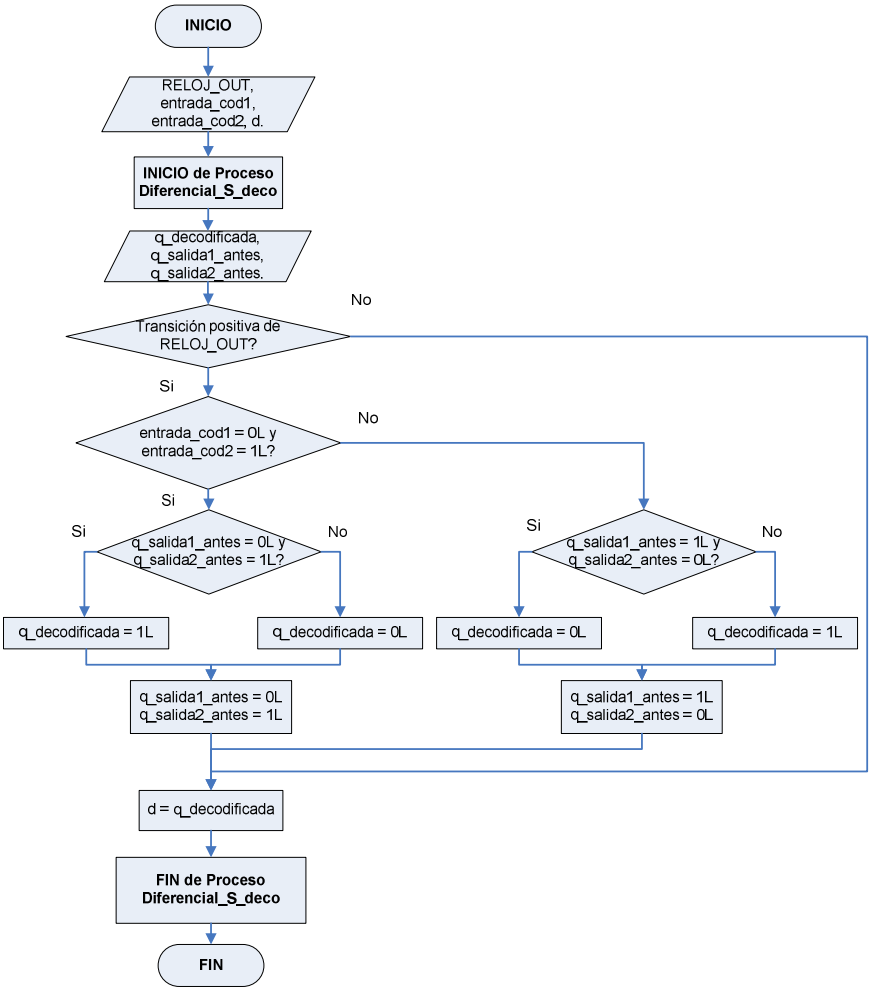


Figura 4.53 Diagrama de Flujo Decodificación Diferencial Tipo S en VHDL.

Se puede apreciar que el proceso involucrado en esta componente es *Diferencial_S_deco*, el cual es sensible a la señal *RELOJ_OUT*; posee dos variables que renuevan sus valores cada transición de la señal de reloj, y otras dos tienen el propósito de advertir el estado anterior para realizar la decodificación, similar al código Diferencial Tipo M.

La simulación en Testbench de la componente se visualiza en la Figura 4.54.

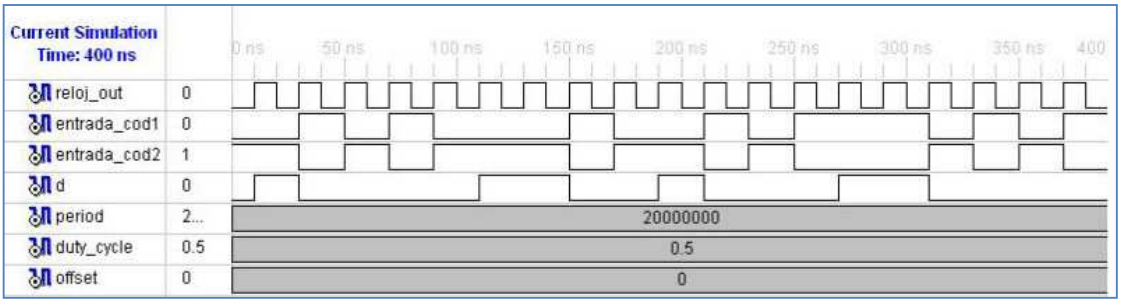


Figura 4.54 Simulación en Testbench de la decodificación Diferencial Tipo S en VHDL.

Cabe mencionar que los valores iniciales de las variables involucradas en el proceso representan un nivel de -5 [V], en la simulación se puede observar que los valores de *entrada_cod1* y *entrada_cod2* en el primer ciclo de reloj representan un nivel de voltaje de -5 [V] (0L y 1L respectivamente), es decir mantiene el nivel, por lo que el primer bit decodificado es un 1L. Un bit decodificado en 0L representa cambio de nivel en la señal de entrada.

4.7.6.6 Decodificación Manchester

La componente *Manchester_decodificacion* tiene tres señales de entrada: *RELOJ_OUT*, *entrada_cod1* y *entrada_cod2*; y la señal de salida *d*. Se utiliza la señal de reloj *RELOJ_OUT*, mas no *RELOJ_OUT_Doble*, debido a que a pesar que es un código que tiene transiciones a mitad de tiempo de bit se puede reconocer si la señal decodificada es 1L o 0L simplemente en el estado en alto de la señal *RELOJ_OUT*, sin tomar en cuenta lo que sucede en el estado en bajo de esta señal.

La componente en mención incluye el proceso *Manchester_deco*, cuya lista de sensibilidad tiene la señal *RELOJ_OUT*. La única variable que interviene en este proceso es *q_decodificada*. Este código no requiere de voltaje de referencia. El diagrama de flujo de esta componente se muestra en la Figura 4.55.

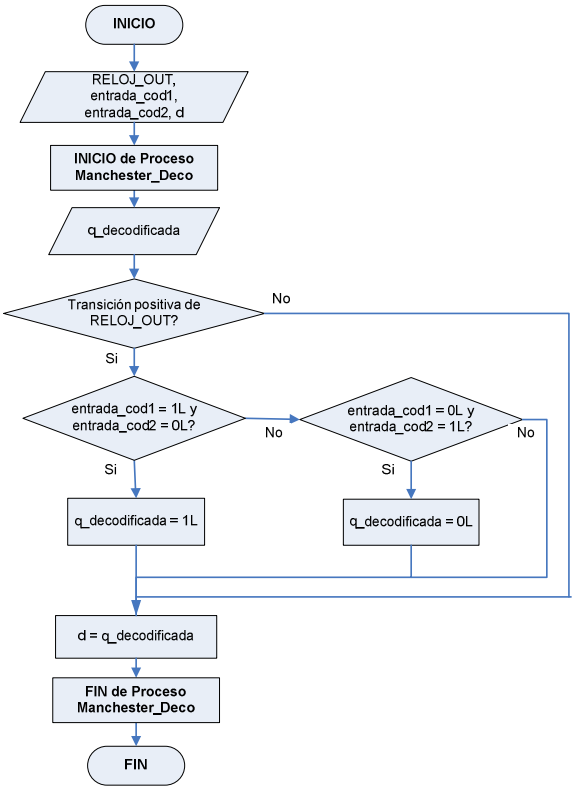


Figura 4.55 Diagrama de Flujo Decodificación Manchester en VHDL.

En el diagrama de flujo se visualiza que en cada transición positiva de *RELOJ_OUT* la variable *q_decodificada* almacena un estado lógico, que finalmente se enviará a la señal de salida, *d*, que corresponde a la señal decodificada. En la Figura 4.56 se visualiza la simulación de la componente *Manchester_decodificacion* realizada en Testbench.

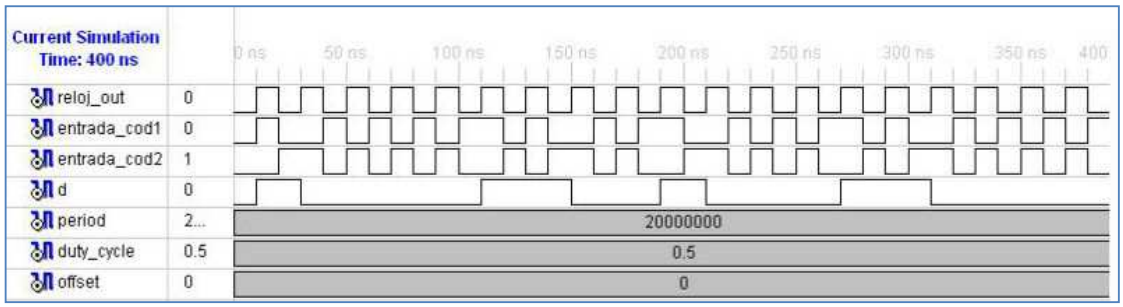


Figura 4.56 Simulación en Testbench de la decodificación Manchester en VHDL.

En la simulación se aprecia que cada transición de la señal *RELOJ_OUT*, cambia el estado lógico de las señales de entrada: *entrada_cod1* y *entrada_cod2*.
Las entradas *entrada_cod1* y *entrada_cod2* únicamente pueden tomar los valores: 1L y 0L, o 0L y 1L, representando los niveles de voltaje +5 [V] y – 5 [V] respectivamente. Por ejemplo la señal decodificada, *d*, es 0L, cuando durante el estado en alto de la señal de reloj, las entradas *entrada_cod1* y *entrada_cod2* tienen el valor 0L y 1L respectivamente.

4.7.6.7 Decodificación Manchester Diferencial

El código Manchester Diferencial se decodifica mediante la componente *Manch_dif_decodificacion*. Las señales de entrada utilizadas son *RELOJ_OUT_Doble*, *entrada_cod1*, *entrada_cod2*, y la salida se especifica en *d*. La Figura 4.57 muestra el diagrama de flujo de la decodificación en VHDL del código en tratamiento.

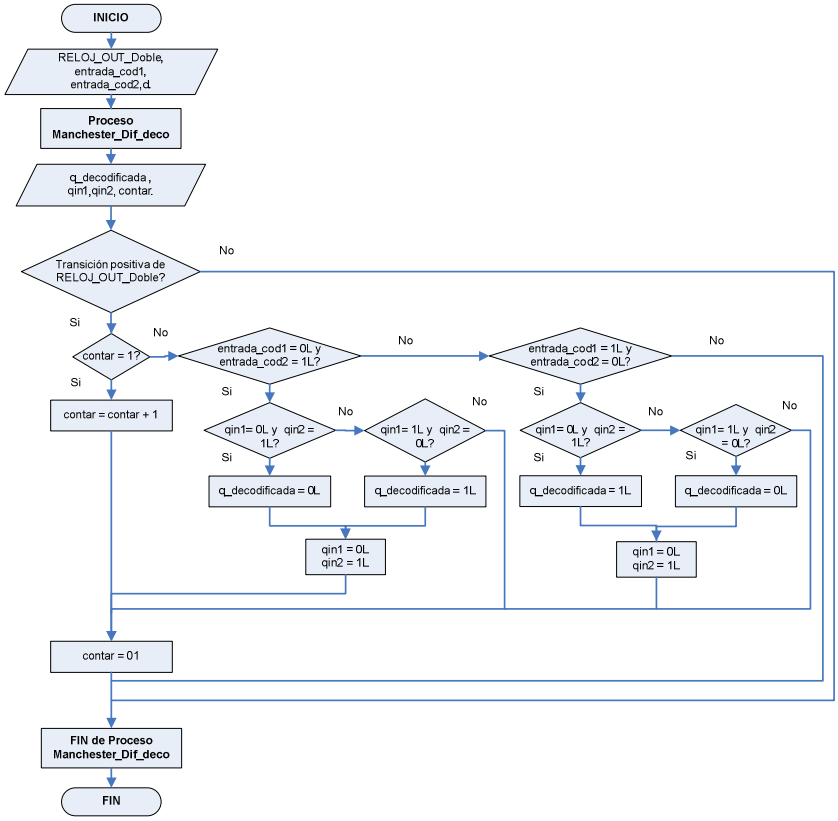


Figura 4.57 Diagrama de Flujo Decodificación Manchester Diferencial en VHDL.

El proceso implicado en la componente presenta en su lista de sensibilidad la señal *RELOJ_OUT_Doble*. A diferencia del Manchester, éste código necesita del estado anterior para realizar la decodificación. Con este propósito se establecen dos variables que van a almacenar los valores de los estados precedentes, y se van actualizando con cada transición positiva de la señal de reloj.

La simulación en Testbench se muestra en la Figura 4.58.

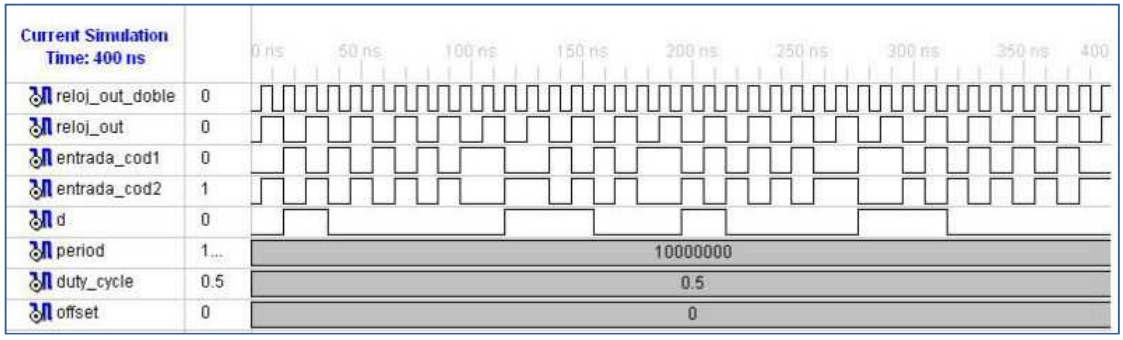


Figura 4.58 Simulación en Testbench de la decodificación Manchester Diferencial en VHDL.

Los valores iniciales de las variables mencionadas anteriormente, representan una transición negativa, los valores de las señales *entrada_cod1* y *entrada_cod2* en el primer ciclo de la señal *RELOJ_OUT* expuestos en la simulación figuran una transición positiva, en consecuencia el primer bit decodificado es un 1L.

4.7.6.8 Decodificación CMI

La componente *CMI_decodificacion*, contiene tres señales de entrada: *RELOJ_OUT_Doble*, *entrada_cod1* y *entrada_cod2*; y la señal de salida *d*.

En esta componente se incluye el proceso *CMI_deco*. Este proceso es sensible a la señal *RELOJ_OUT_Doble*, debido a que se requiere conocer los niveles de voltaje de las señales de entrada en cada transición (positiva y negativa) de la señal *RELOJ_OUT* para determinar la señal decodificada. Las variables que intervienen en este proceso son: *q_decodificada*, *q_mitad_bit* de valor inicial 0L, y *contar* de tipo natural en el rango de 0 a 2 y valor inicial 1. En la codificación con este código se considera como voltaje de referencia – 5 [V]; sin embargo en la

decodificación no hace falta realizar esta consideración ya que cualquiera sea el voltaje de referencia, siempre que los niveles de entrada sean + 5 [V] o – 5 [V] durante un tiempo de bit, indicarán que el estado de la señal decodificada es 1L.

El diagrama de flujo de esta componente se muestra en la Figura 4.59.

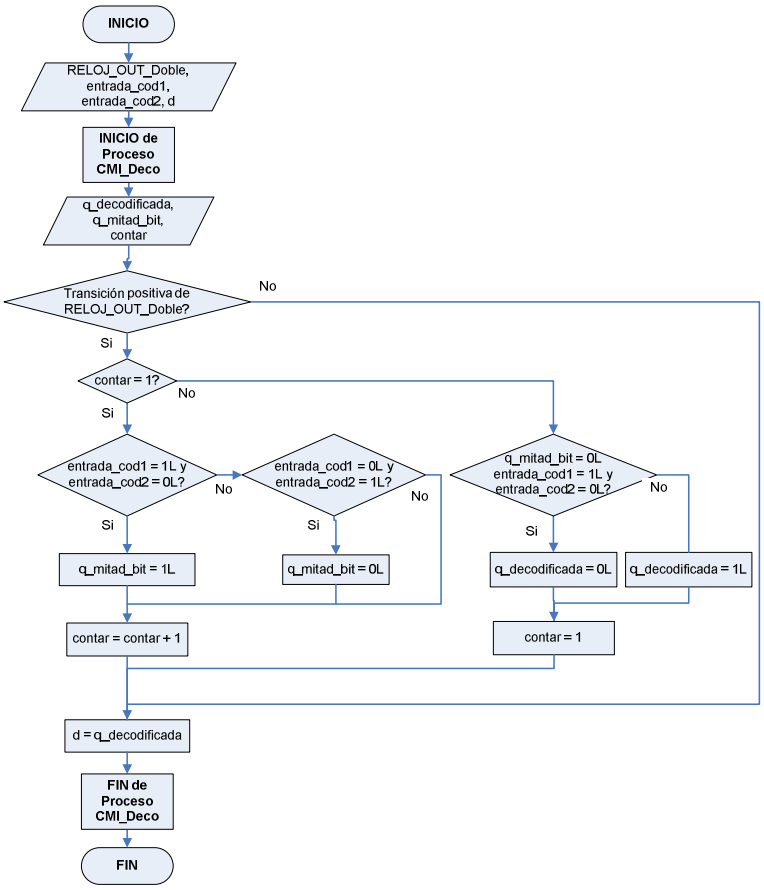


Figura 4.59 Diagrama de Flujo Decodificación CMI en VHDL.

En el diagrama de flujo se visualiza que en cada transición positiva de *RELOJ_OUT_Doble* la variable *contar* cambia de valor, siendo 1 cuando la señal *RELOJ_OUT* se encuentra en estado alto y 2 cuando esta señal se encuentra en estado bajo. La variable *q_mitad_bit* permite identificar cuando hay transición de la señal codificada (señales de entrada) a mitad de tiempo de bit, para almacenar el nivel 0L (cuando hay transición) o 1L (cuando no hay transición) en *q_decodificada*, que finalmente se enviará a la señal de salida, *d*, que corresponde a la señal decodificada.

La simulación de esta componente, *CMI_decodificacion*, se muestra en la Figura 4.60.

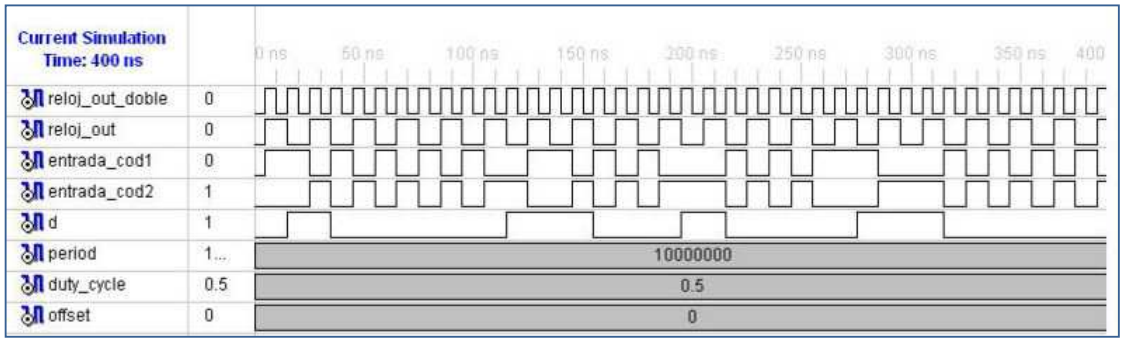


Figura 4.60 Simulación en Testbench de la decodificación CMI en VHDL.

En la simulación se observa que cada transición de la señal *RELOJ_OUT*, pueden cambiar los niveles de voltaje de las entradas: *entrada_cod1* y *entrada_cod2* representando la señal a ser decodificada. Si las entradas: *entrada_cod1* y *entrada_cod2* son: 1L y 0L, o 0L y 1L, respectivamente durante un tiempo de bit, la señal decodificada, *d*, será 1L.

Al igual que en todos los códigos polares, las entradas únicamente pueden tomar los valores: *entrada_cod1* = 1L y *entrada_cod2* = 0L, o *entrada_cod1* = 0L y *entrada_cod2* = 1L.

4.7.6.9 Decodificación AMI

La decodificación AMI se realiza en la componente de VHDL *AMI_decodificacion*. Esta componente contiene las señales de entrada: *RELOJ_OUT*, *entrada_cod1* y *entrada_cod2*, y la señal de salida *d*.

Además la componente contiene el proceso *AMI_deco*, sensible a la señal *RELOJ_OUT* debido a que las señales de entrada cambian de estado cada tiempo de bit requiriendo conocerlas para determinar la señal decodificada. La única variable que interviene en este proceso es *q_decodificada* de valor inicial 0L. En la componente de decodificación no es necesario considerar un voltaje de referencia ya que siempre que los niveles de entrada sean + 5 [V] o – 5 [V], indicarán que el estado lógico de la señal decodificada será 1L.

El diagrama de flujo de esta componente se muestra en la Figura 4.61.

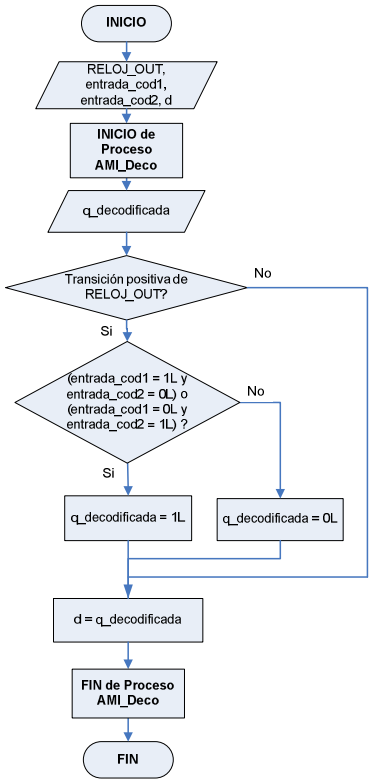


Figura 4.61 Diagrama de Flujo Decodificación AMI en VHDL.

En el diagrama de flujo se observa que en cada transición positiva de la señal de reloj, la variable *q_decodificada* toma el valor 1L en caso de que las entradas correspondan a los niveles de voltaje + 5 [V] o – 5 [V], caso contrario será 0L. Finalmente *q_decodificada* se enviará a la señal de salida de la componente, *d*, señal decodificada. La simulación en Testbench de esta componente se muestra en la Figura 4.62.

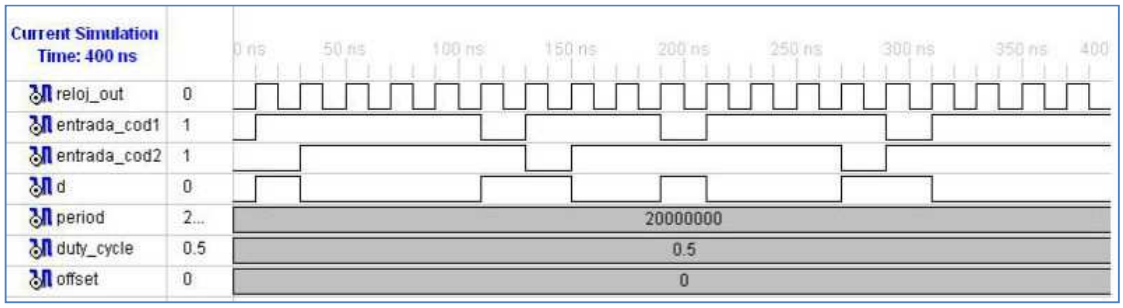


Figura 4.62 Simulación en Testbench de la decodificación AMI en VHDL.

En la simulación se visualiza que cuando las entradas: *entrada_cod1* y *entrada_cod2* son: 1L y 0L, o 0L y 1L, respectivamente durante un tiempo de bit, la señal decodificada, *d*, es 1L. En cambio cuando estas dos entradas toman el valor 1L, representando el nivel de voltaje 0 [V], la señal decodificada, *d*, será 0L.

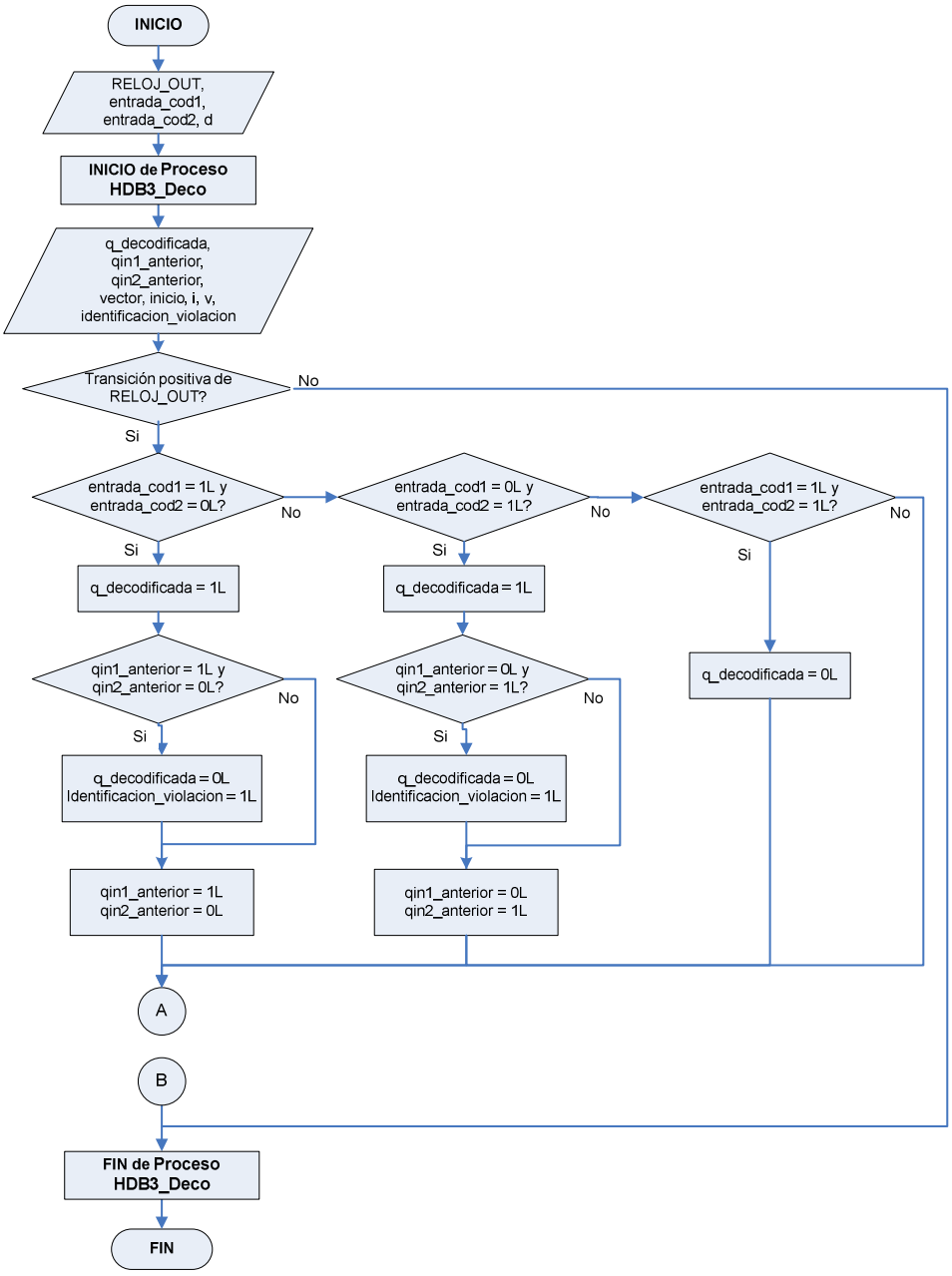
4.7.6.10 Decodificación HDB3

La componente *HDB3_decodificacion* tiene las siguientes señales de entrada: *RELOJ_OUT*, *entrada_cod1* y *entrada_cod2*. La única señal de salida es *d*, señal decodificada. Todas estas señales son de tipo *std_logic*. Se utiliza la señal de reloj *RELOJ_OUT*, ya que las señales de entrada y la señal decodificada tienen transiciones cada tiempo de bit.

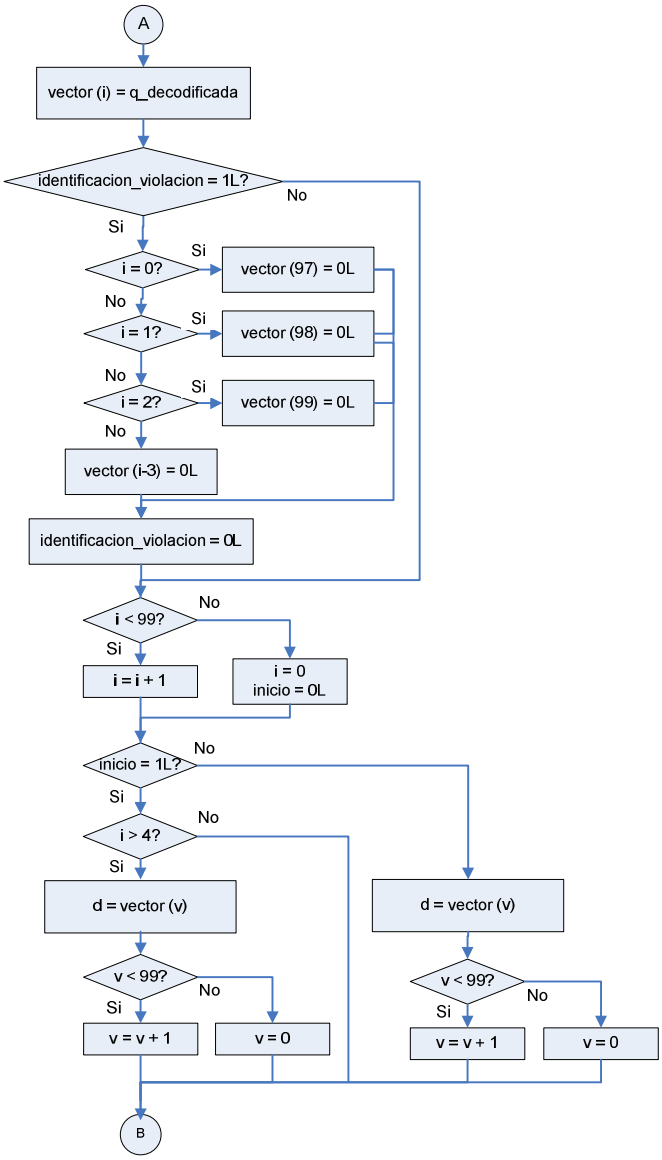
Esta componente incluye el proceso *HDB3_deco*, sensible a la señal de reloj *RELOJ_OUT*. Las variables de tipo *std_logic* que interviene en este proceso son: *q_decodificada* de valor inicial 0L, *qin1_anterior* de valor inicial 0L, *qin2_anterior* de valor inicial 1L, *inicio* de valor inicial 1L, *identificación_violacion* de valor inicial 0L. Además de las variables: *vector* de tipo *std_logic_vector* de 100 elementos, *i* y *v* de tipo natural en el rango de 0 a 99 y valor inicial 0.

El voltaje de referencia está indicado por los valores iniciales de las variables *qin1_anterior* y *qin2_anterior*, es decir – 5 [V], tal como lo es en la componente de codificación HDB3.

El diagrama de flujo de esta componente se muestra en la Figura 4.63.


INICIO
RELOJ_OUT, entrada_cod1, entrada_cod2, d
INICIO de Proceso HDB3_Deco
q_decodificada, qin1_anterior, qin2_anterior, vector, inicio, i, v, identificacion_violacion
Transición positiva de RELOJ_OUT?
No
Si
entrada_cod1 = 1L y entrada_cod2 = 0L?
No
Si
q_decodificada = 1L
qin1_anterior = 1L y qin2_anterior = 0L?
No
Si
q_decodificada = 0L Identificacion_violacion = 1L
qin1_anterior = 1L qin2_anterior = 0L
entrada_cod1 = 0L y entrada_cod2 = 1L?
No
Si
q_decodificada = 1L
qin1_anterior = 0L y qin2_anterior = 1L?
No
Si
q_decodificada = 0L Identificacion_violacion = 1L
qin1_anterior = 0L qin2_anterior = 1L
entrada_cod1 = 1L y entrada_cod2 = 1L?
No
Si
q_decodificada = 0L
A
B
FIN de Proceso HDB3_Deco
FIN

(a)



(b)

Figura 4.63 Diagrama de Flujo Decodificación HDB3 en VHDL.

En el diagrama de flujo se visualiza que en cada transición positiva de *RELOJ_OUT* la variable *q_decodificada* almacena un estado lógico, este estado lógico se lo envía a la variable *vector* donde será tratada en caso que sea producto de una violación o relleno, y finalmente será enviado a la señal decodificada, *d*.

Las variables *qin1_anterior* y *qin2_anterior* guardan los estados lógicos correspondientes al último valor que ingresó por las señales de entrada (excepto cuando la entrada corresponde al nivel 0 [V]) con el objetivo de que sean utilizados cuando se produzca la siguiente transición positiva de *RELOJ_OUT*.

La variable *identificación_violacion*, es 1L cuando identifica una violación o relleno caso contrario es 0L.
La variable *vector* almacena los estados lógicos que deben tomar las salidas, no se las envía directamente a la salida debido a que cuando existe un relleno en la señal a decodificar se debe cambiar el estado lógico del tercer bit anterior decodificado de + 5 [V] o – 5 [V], a 0 [V]. Si se obtuviera los estados de la salida, *d*, sin almacenarlo primero en la variable *vector*, no se pudiera cambiar el estado de los bits anteriores.

i es la variable que identifica en cuál de los 100 elementos de la variable *vector* se almacena el estado lógico proveniente de *q_decodificada*. *v*, es la variable que identifica a uno de los 100 elementos de *vector* para enviarlo a la señal de salida, *d*. *v* siempre va a estar atrasado 4 unidades respecto a *i*.

La variable *inicio*, es igual a 1L cuando identifica si los estados lógicos almacenados en *vector* corresponden a los 100 primeros bits decodificados, caso contrario es igual a 0L. Esto se realiza para que la señal decodificada tenga el valor 0L, hasta que se decodifiquen los cuatro primeros bits de datos, posterior a lo cual se obtendrá en la variable *d* los estados correspondientes a la decodificación. Por esta razón la señal decodificada siempre tendrá un retardo de cuatro bits respecto a la señal a decodificar.

En la Figura 4.64 se visualiza la simulación en Testbench de la componente *HDB3_decodificacion*.

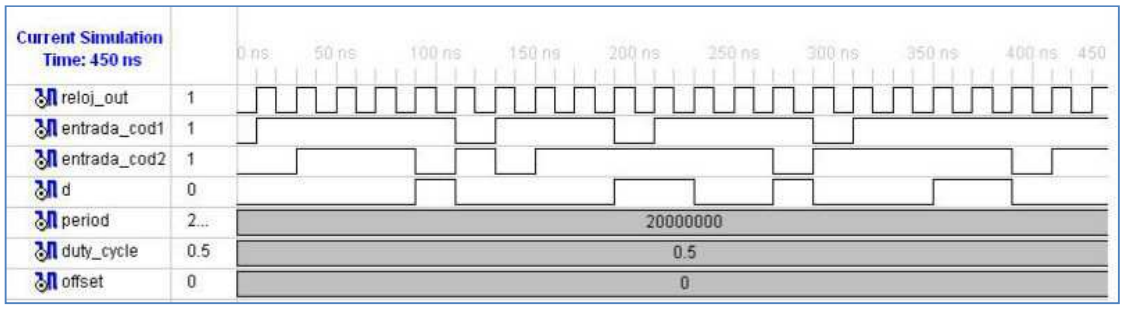


Figura 4.64 Simulación entrada en Testbench de la decodificación HDB3 en VHDL.

En la simulación se puede apreciar que la señal decodificada, *d*, tiene el retardo previsto de cuatro tiempos de bit, debido a lo explicado en la descripción del diagrama de flujo.

Durante este tiempo, la señal de salida *d* toma el valor 0L. En el quinto tiempo de bit se observa que la salida *d*, toma el estado correspondiente a la decodificación del primer nivel de voltaje de entrada. Por ejemplo al tiempo 0 ns las entradas, *entrada_cod1* y *entrada_cod2* tienen los estados 1L y 0L, respectivamente, entonces la salida *d*, toma el valor 1L, después de 4 tiempos de bit.

4.7.6.11 Decodificación MLT3

La decodificación del código bipolar MLT3 se la efectúa en la componente denominada *MLT3_decodificacion*. Las señales de entrada y salida son las utilizadas en la componente *HDB3_decodificacion*. La Figura 4.65 exhibe el diagrama de flujo para esta componente.

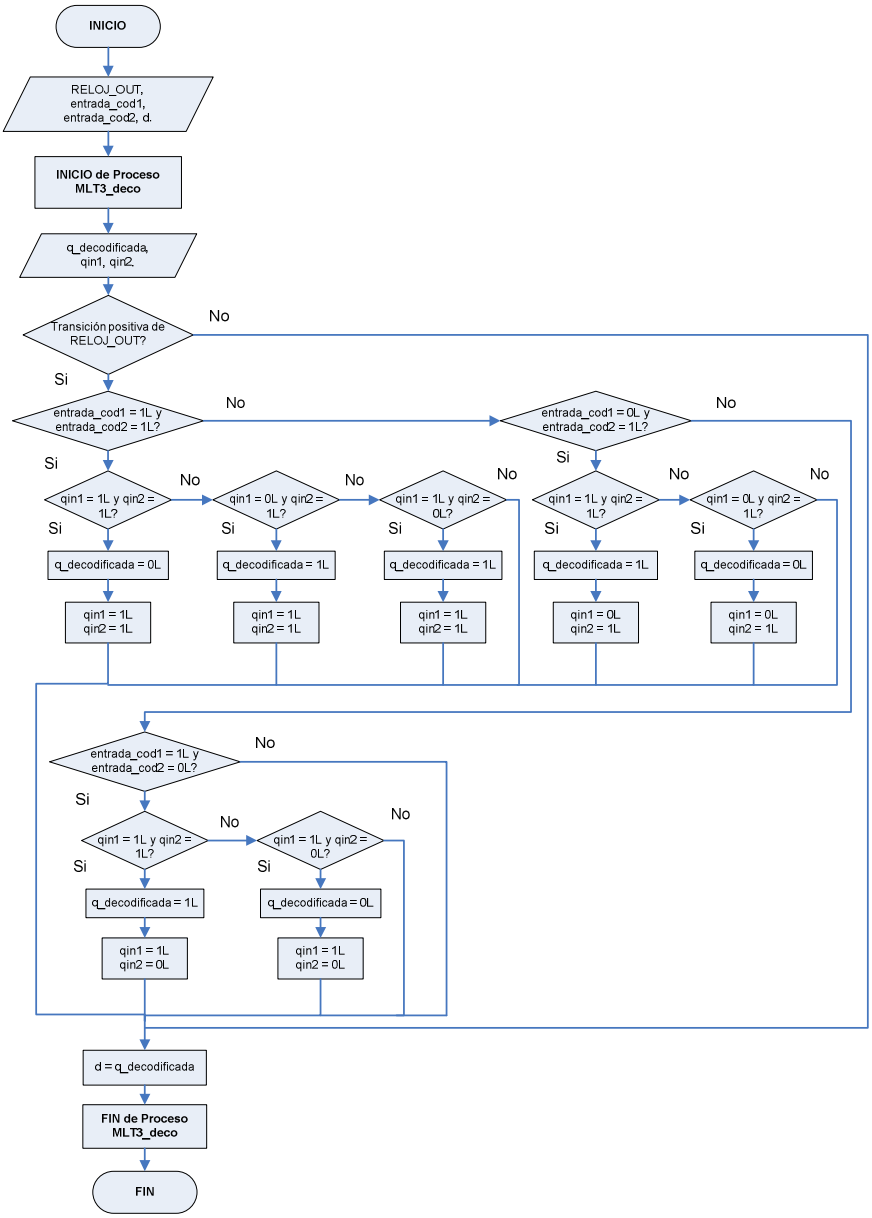


Figura 4.65 Diagrama de Flujo Decodificación MLT3 en VHDL.

La componente en tratamiento contiene el proceso *MLT3_Deco*, el cual está sometido a la señal de reloj *RELOJ_OUT*; dos variables nos van a permitir saber los estados previos para la respectiva decodificación.

La Figura 4.66 presenta la simulación en Testbench de la decodificación MLT3.

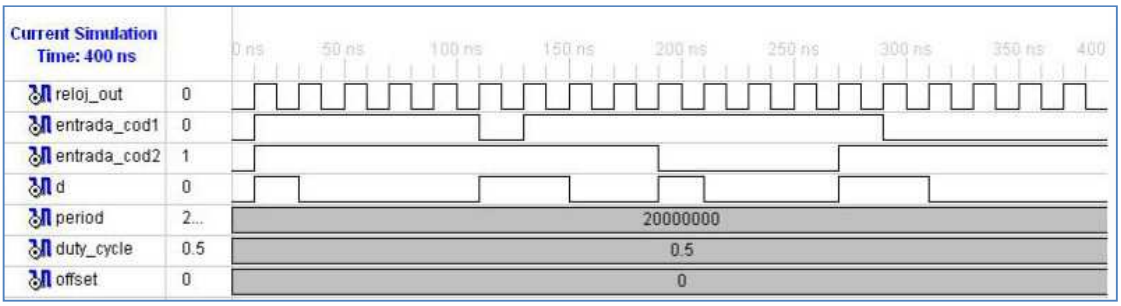


Figura 4.66 Simulación en Testbench de la decodificación MLT3 en VHDL.

La simulación muestra en el primer ciclo de la señal de reloj el valor de 1L para las dos señales de entrada, esto significa la llegada de un nivel de voltaje de 0 [V], el estado anterior cimentado en las dos variables que se encuentran dentro del proceso MLT3_Deco es de +5 [V], con lo que se concluye que el primer bit decodificado es un 1L.

4.7.7 IDENTIFICACIÓN DE BANDERA DE INICIO, ENSAMBLAJE DE DATOS, MATRIZ

Luego de la decodificación, se procede a almacenar los bits de datos de la señal decodificada en la matriz de decodificación de tamaño 100x11, semejante a la implementación de una RAM en VHDL. Esto se realiza en este proceso, *bandera_ensamblaje_matriz*.

El proceso primero reconoce el identificador de inicio (7EH) en la señal decodificada. El identificador de inicio permite establecer cuál es el primer bit de datos decodificado, caso contrario no se podría conocerlo. El bit después de la bandera de inicio es el primer bit de datos decodificado, el mismo que se almacena en el elemento *x_dec*(0)(1) de la matriz de decodificación.

Este proceso incluye en cada fila de la matriz el bit de inicio, bit de paridad y bit de parada, a parte de los 8 bits decodificados; razón por la cual cada fila tiene 11 elementos correspondientes a un carácter a ser transmitido de forma asincrónica. Siempre la primera columna tendrá el bit de inicio, la décima columna el bit de paridad y la última columna el bit de parada.

De esta manera, los caracteres se almacenan listos para ser enviados después de la correspondiente serialización, tal como lo muestra la Figura 4.67.

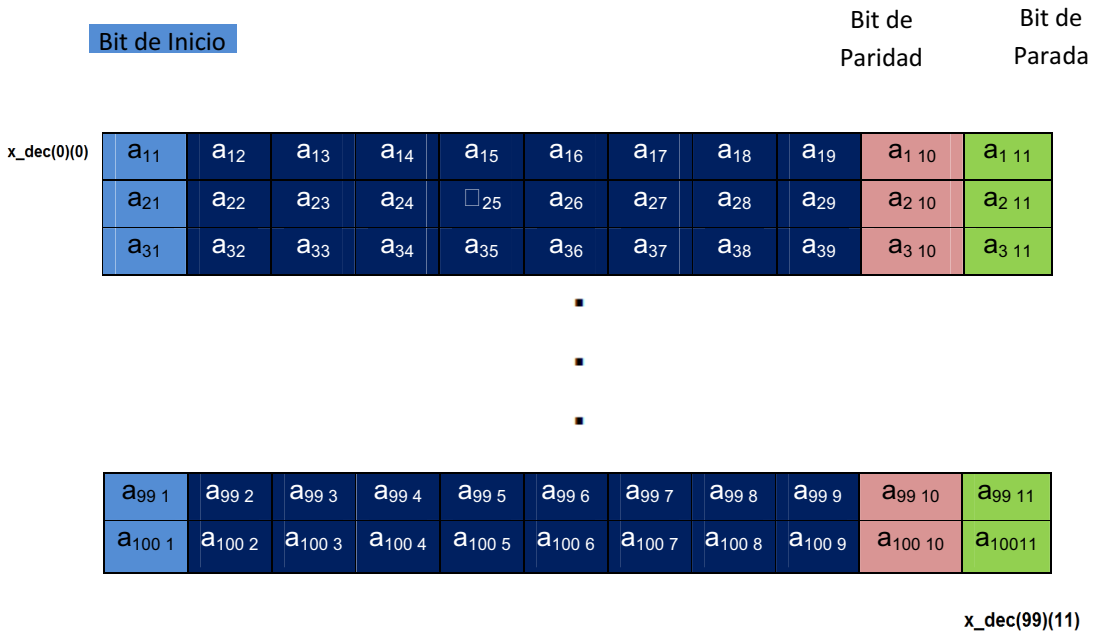


Figura 4.67 Matriz de datos decodificados con formato de un carácter asincrónico.

Cabe indicar que los caracteres no son enviados directamente después de la decodificación debido a que primero se requiere incluir los bits de inicio, paridad y parada para su transmisión.

El diagrama de flujo de este proceso se muestra en la Figura 4.68.

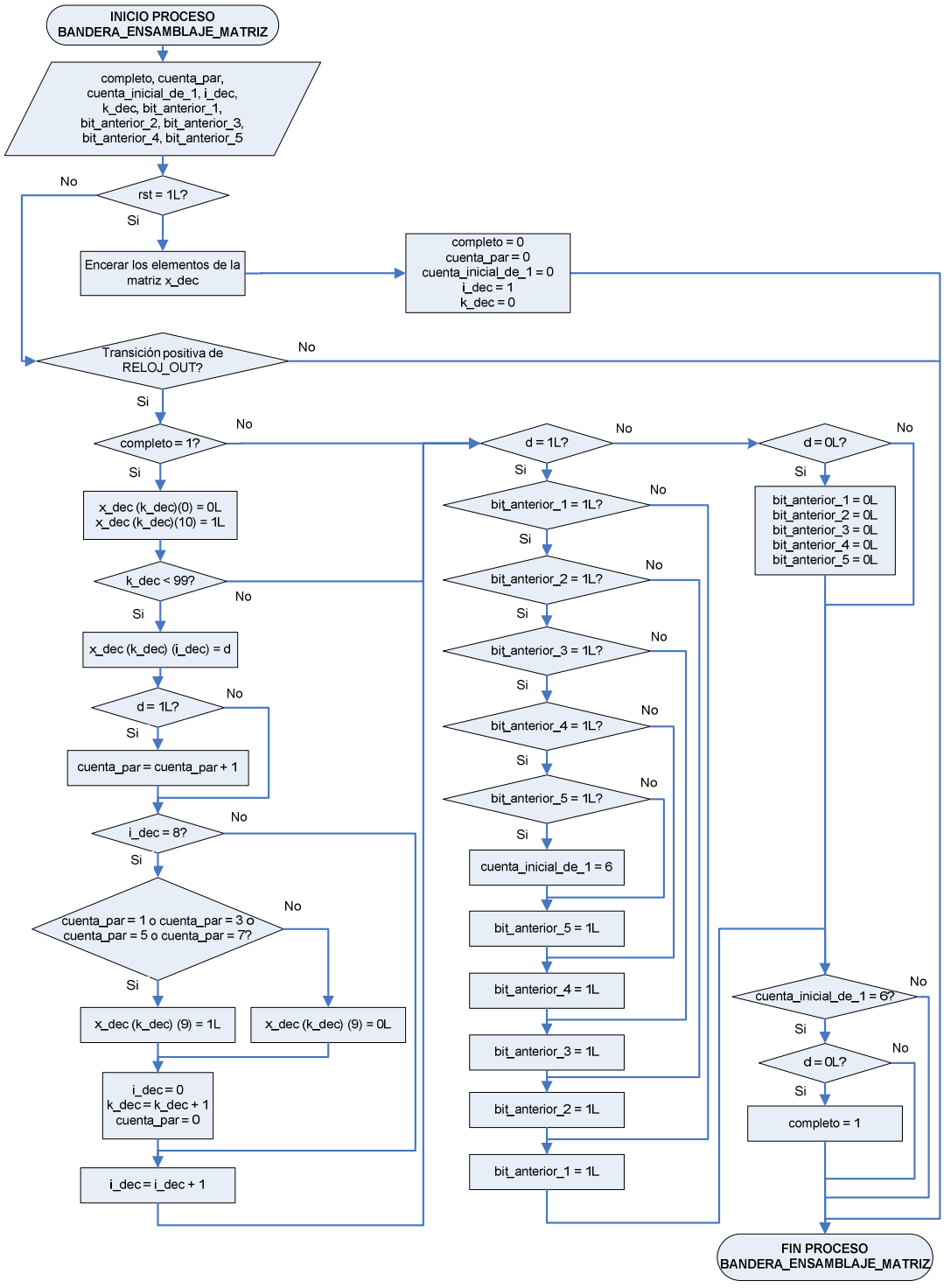


Figura 4.68 Diagrama de Flujo del proceso Identificación de Bandera de Inicio, Ensamblaje de Datos, Matriz en VHDL.

4.7.8 SERIALIZACIÓN Y SALIDA DE DATOS

El proceso Serialización y Salida de datos tiene como objetivo el retorno de los datos decodificados como un stream de once bits por carácter a la interfaz gráfica diseñada en MATLAB para su visualización. El diagrama de flujo de este proceso se puede apreciar en la Figura 4.69.

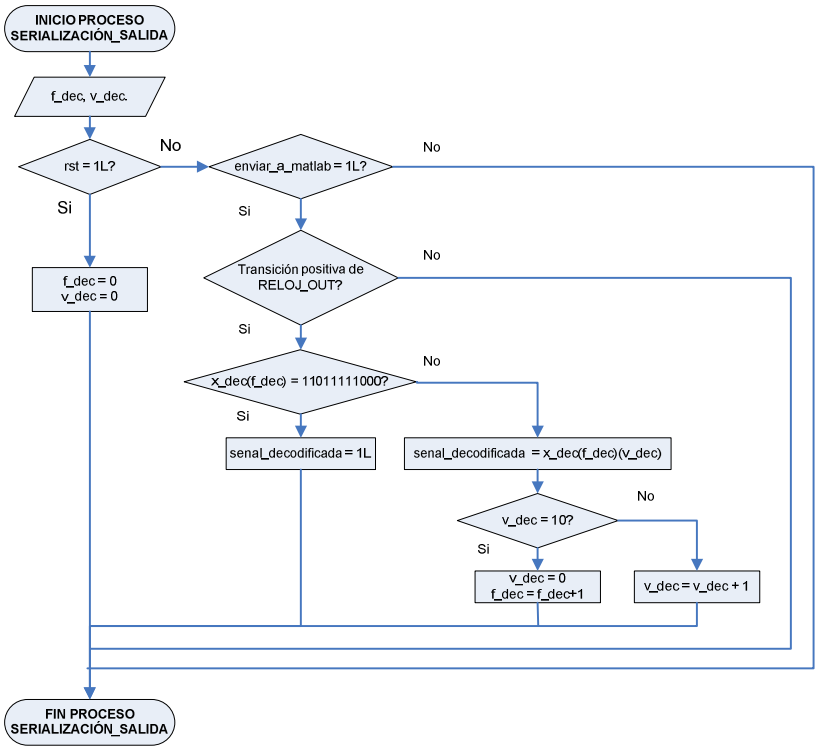


Figura 4.69 Diagrama de Flujo del proceso Serialización y Salida de Datos en VHDL.

Cuando la señal *enviar_a_matlab* tiene el valor 1L, se envía todos los bits de datos de la señal decodificada a la Interfaz Gráfica a través del Puerto Serial RS-232. Las variables *f_dec* y *v_dec* nos permiten identificar cada uno de los elementos de la matriz que contiene los datos decodificados, por ejemplo *x_dec*(0)(0) es el primer elemento donde *f_dec* representa el número de fila y *v_dec* el número de columna.

Con cada transición positiva de la señal *RELOJ_OUT* a través de *senal_decodificada* se envía cada uno de los elementos de la matriz antes mencionada. Si se reconoce el identificador de fin de datos que es 00011111011,

el bit menos significativo a la izquierda, *senal_decodificada* va tomar el valor 1L, que representa reposo en la transmisión asincrónica.

4.8 COMPARACIÓN DE LA PROGRAMACIÓN ENTRE VHDL Y MATLAB PARA LA IMPLEMENTACIÓN DE LOS CÓDIGOS DE LÍNEA.

VHDL es inherentemente concurrente, mientras que el código en MATLAB es secuencial, por consiguiente no habría ninguna equivalencia; sin embargo, el lenguaje de programación VHDL permite realizar las codificaciones con cada uno de los códigos de línea utilizando componentes diferentes que contienen procesos, dichos procesos son secuenciales. En base a este hecho se pueden establecer semejanzas y diferencias.

Semejanzas

El código en MATLAB que realiza la gráfica de la señal codificada con cada código, se asemeja al código VHDL en lo siguiente:

Los dos requieren tener como señal de entrada la señal a codificar, en función de la cual a través de programación se establece la señal codificada.

Para la codificación, tanto en VHDL como en el código en MATLAB se utilizan condicionales.

Diferencias

Las diferencias son numéricamente mayores que las semejanzas, las mismas que se las describen a continuación:

La programación en VHDL de la componente requiere de tres partes: declaración de librerías, entidad y arquitectura. El código en MALTAB no lo necesita.

En VHDL se requiere declarar las señales y variables a utilizarse. Mientras que en MATLAB no se demanda la declaración, es más, no se diferencia entre señales y variables.

En VHDL, en la lista de sensibilidad de cada proceso se requiere utilizar una señal de reloj para que en cada transición positiva de la misma se ejecute el proceso y se actualice la señal codificada; MATLAB no requiere la señal de reloj para la codificación, es más en la función de codificación se genera la señal de reloj para posteriormente graficarla.

En VHDL, cada vez que cambie de estado la señal de reloj, la señal codificada cambia inmediatamente, es decir tiene una respuesta inmediata. En MATLAB hay que almacenar los bits decodificados en un vector para posteriormente graficarlo utilizando la función plot para graficar la señal almacenada en función de otro vector del mismo tamaño que representa el tiempo.

En VHDL no es necesario especificar el valor de voltaje que debería tomar la señal de reloj y la señal codificada cuando se encuentra en estado lógico alto, pues la Spartan 3E Starter Kit Board trabaja con niveles de voltaje LVTTL. Mientras que en MATLAB se tiene que especificar el valor de voltaje que toman las señales cuando se encuentra en estado lógico alto.

En VHDL el nivel de voltaje de la señal codificada se obtiene en cada transición positiva de la señal de reloj ejecutando el proceso respectivo sin la necesidad de crear un vector que represente el tiempo, mientras que para graficar la señal codificada en MATLAB se crea un vector (del mismo tamaño que el vector que contiene la señal codificada) que representa el tiempo.

4.9 ASIGNACIÓN DE TERMINALES

En el código principal de la implementación en VHDL, se utilizan varias señales de entrada y se obtienen otras señales de salida, tal como se puede visualizar en los diagramas de flujo de los procesos.

A las señales de entrada y de salida se les asignan los pines que se detallan en la Tabla 4.8 con el objetivo de utilizar el puerto serial RS-232, la fuente de reloj, los interruptores deslizantes y los conectores de seis pines J1, J2 y J4 de la tarjeta de entrenamiento Spartan 3E Starter Kit Board.

Señal	Modo	Descripción	Asignación de Pin en la Spartan 3E Starter Kit Board
d	Buffer	Señal decodificada	D7
din	Entrada	Ingreso de datos por el pin de recepción del Puerto RS-232	R7
entrada_cod1	Entrada	Conjuntamente con entrada_cod2 representan la señal a ser decodificada proviene del circuito interpretador bipolar unipolar	D5
entrada_cod2	Entrada	Conjuntamente con entrada_cod1 representan la señal a ser decodificada proviene del circuito interpretador bipolar unipolar	C5
enviar_a_matlab	Entrada	Identifica el momento en que se desee enviar la señal decodificada a MATLAB	L14
error	Salida	Identifica si hay error en los bits que representan los datos	F7
RELOJ_BASE	Buffer	Señal de Reloj de 38400 Hz	C7
RELOJ_BASE_4B5B	Buffer	Señal de Reloj de 48000 KHz	F8
RELOJ_IN	Entrada	Fuente de reloj de 50 MHz de la tarjeta de entrenamiento	C9
RELOL_OUT	Buffer	Señal de reloj para codificación y decodificación	A6
RELOJ_OUT_Doble	Buffer	Señal de reloj para codificación y decodificación	B6
RELOJ_OUT_4B5B	Buffer	Señal de reloj para codificación y decodificación con el código de línea 4B5B	E8
Reset	Entrada	Utilizado para el Ingreso de nuevos datos	N17
salida1	Salida	Conjuntamente con salida2 representan la señal codificada enviada al circuito interpretador unipolar bipolar	B4
salida2	Salida	Conjuntamente con salida1 representan la señal codificada enviada al circuito interpretador unipolar bipolar	A4
senal_a_codificar	Buffer	Señal a ser codificada (Solo datos)	E7
senal_decodificada	Salida	Señal decodificada a ser enviada por el pin de transmisión del Puerto RS-232	M14

Tabla 4.8 Asignación de pines a las señales de la entidad en la Tarjeta de Entrenamiento Spartan 3E Starter Kit Board.

En la implementación, a las señales: *datos*, *velocidad* e *ID_Codigo* de modo buffer no se les asignan pines ya que no es de interés su visualización.

En los diagramas de flujo de los procesos del código principal también se observa la intervención de otras señales, estas señales se encuentran declaradas en la arquitectura debido a que no son de entrada o salida, mas bien se crean internamente para utilizarlas entre el código principal y las componentes, o entre procesos del código principal, para obtener las señales de salida. Cabe indicar que a estas señales, no se les debe asignar pines de la tarjeta de entrenamiento.

4.10 DISEÑO Y CONSTRUCCIÓN DE LOS CIRCUITOS INTERPRETADORES

Debido a que la tarjeta de entrenamiento trabaja con niveles de voltaje LVTTL (TTL de bajo voltaje), únicamente se obtienen voltajes positivos; por consiguiente se requiere la construcción de un circuito externo interpretador unipolar – bipolar y un circuito externo interpretador bipolar – unipolar para los códigos de línea polares y bipolares.

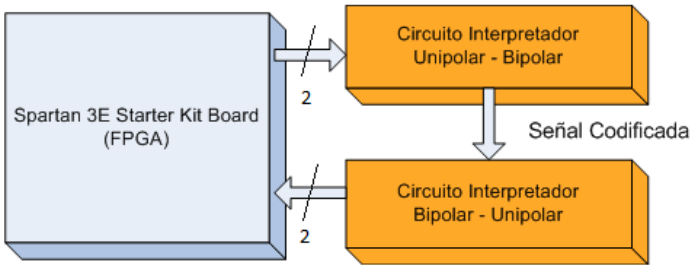


Figura 4.70 Diagrama de Bloques de la conexión entre el FPGA y los circuitos externos.

En la Figura 4.70 se puede apreciar que la tarjeta de entrenamiento, posteriormente a la codificación, envía dos señales al circuito externo interpretador unipolar – bipolar para obtener los niveles de voltaje requeridos; la salida de este circuito, señal codificada, es entrada para el circuito interpretador bipolar – unipolar que envía dos señales a la misma tarjeta de entrenamiento para que se realice la decodificación en el FPGA.

4.10.1 CIRCUITO INTERPRETADOR UNIPOLAR – BIPOLAR

Este circuito externo tiene dos señales de entrada digitales binarias y obtiene una señal digital bipolar de salida, de acuerdo a la Tabla 4.5.

Las dos señales de salida de la tarjeta de entrenamiento Spartan – 3E que representan la señal codificada (*salida1* y *salida2*) constituyen las entradas al circuito interpretador unipolar – bipolar. La señal bipolar de salida de este circuito interpretador es la señal finalmente codificada.

Diseño del circuito

El circuito a diseñar debe obtener una señal digital bipolar de salida, de acuerdo a la Tabla 4.5, esto se puede efectuar con el diagrama de bloques de la Figura 4.71. El diagrama de bloques se lo cumple fácilmente basándose en amplificadores operacionales debido a su bajo costo y a que permiten implementar los bloques necesarios.

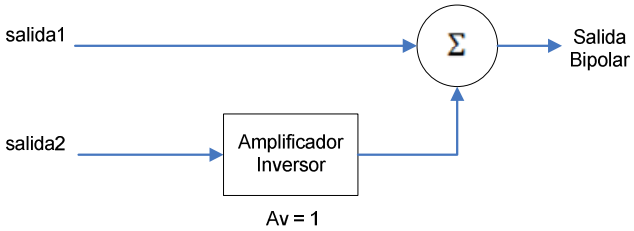


Figura 4.71 Diagrama de bloques del circuito Interpretador Unipolar-Bipolar.

En la Figura 4.71 se aprecia que se deben utilizar dos amplificadores operacionales, uno como amplificador inversor para obtener un nivel de voltaje negativo cuando la entrada al mismo sea un 1L, y el otro como sumador no inversor. Esto nos permite realizar el siguiente análisis:

$$\boldsymbol{Amplificador\ Inversor \Rightarrow Amplificador\ Operacional\ 1}$$

Al observar la Figura 4.72 se puede establecer lo siguiente:

$$A_v = \frac{Vo}{Vin} = -\frac{R_2}{R_1}$$

Sólo se requiere un Amplificador inversor de Ganancia de voltaje = - 1, entonces:

$$A_v = -1 \implies R_2 = R_1$$

$$Sea\ R_2 = R_1 = 68K\Omega$$

$\boldsymbol{Sumador\ no\ inversor \Rightarrow Amplificador\ Operacional\ 2}$

El voltaje de salida del sumador no inversor de la Figura 4.72 está dado por la siguiente expresión:

$$Vo = \left(1 + \frac{R_6}{R_5}\right)\left(\frac{\frac{V_3}{R_3} + \frac{V_4}{R_4}}{\frac{1}{R_3} + \frac{1}{R_4}}\right)$$

$Sea\ R_3 = R_4 = 68K\Omega$

$$\Rightarrow Vo = \left(1 + \frac{R_6}{R_5}\right)\left[\frac{1}{2}(V_3 + V_4)\right]$$

$Si\ Vin1 = 0\ [V]\ y\ Vin2 = 3.3[V]$ $\quad$ $Si\ Vin1 = 3.3\ [V] y\ Vin2 = 0[V]$

$\Rightarrow V_3 = 0\ V_4 = -3.3\ y\ Vo = -5[V]$ $\quad$ $\Rightarrow V_3 = 3.3\ V_4 = 0\ y\ Vo = 5[V]$

$-5 = \frac{1}{2}\left(1 + \frac{R_6}{R_5}\right)(-3.3)$ $\quad$ $5 = \frac{1}{2}\left(1 + \frac{R_6}{R_5}\right)(3.3)$

$1.52 = \frac{1}{2}\left(1 + \frac{R_6}{R_5}\right)$ $\quad$ $1.52 = \frac{1}{2}\left(1 + \frac{R_6}{R_5}\right)$

$3.04 = 1 + \frac{R_6}{R_5}$ $\quad$ $3.04 = 1 + \frac{R_6}{R_5}$

$R_6 = 2.04R_5$ $\quad$ $R_6 = 2.04R_5$

$Sea\ R_5 = 68K\Omega$

$\Rightarrow R_6 = 138.72K\Omega \longrightarrow 150\text{K}\Omega$

$\searrow\ 120K\Omega$

$R_6 = 150K\Omega$

En Resumen:

$R_1 = 68K\Omega$ $\quad$ $R_4 = 68K\Omega$

$R_2 = 68K\Omega$ $\quad$ $R_5 = 68K\Omega$

$R_3 = 68K\Omega$ $\quad$ $R_6 = 150K\Omega$

El amplificador operacional seleccionado para la implementación de este circuito externo es el LF353, debido a su bajo costo y su buena respuesta a altas velocidades (el tiempo de subida y tiempo de bajada está en el orden de las unidades de us), suficiente para que no exista distorsión e ISI (Interferencia inter símbolos) a la mayor velocidad, 19200 bps, en los códigos que tienen transiciones a mitad de bit.

El diagrama esquemático de este circuito realizado en la herramienta computacional PROTEUS 7 PROFESSIONAL se muestra en la Figura 4.72.

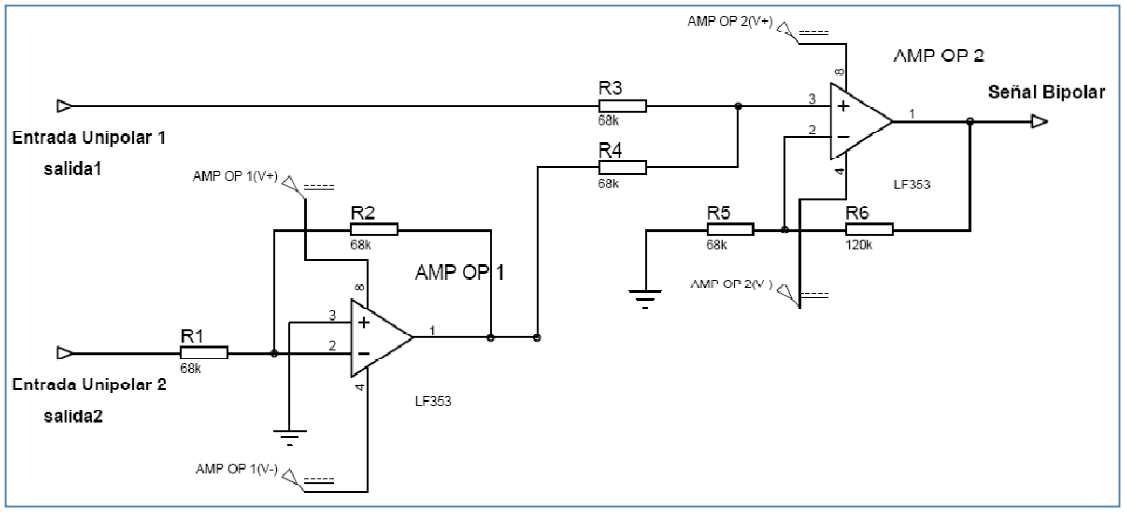


Figura 4.72 Diagrama Esquemático del Circuito interpretador Unipolar Bipolar.

El circuito interpretador nunca obtendrá el valor restringido de la Tabla 4.5 debido a que las señales de entrada al circuito interpretador: *salida1* y *salida2* nunca tomarán los valores 1L al mismo tiempo en el FPGA.

La simulación del Circuito interpretador Unipolar Bipolar se presenta en la Figura 4.73, donde se observa que efectivamente se cumple con la Tabla 4.5.

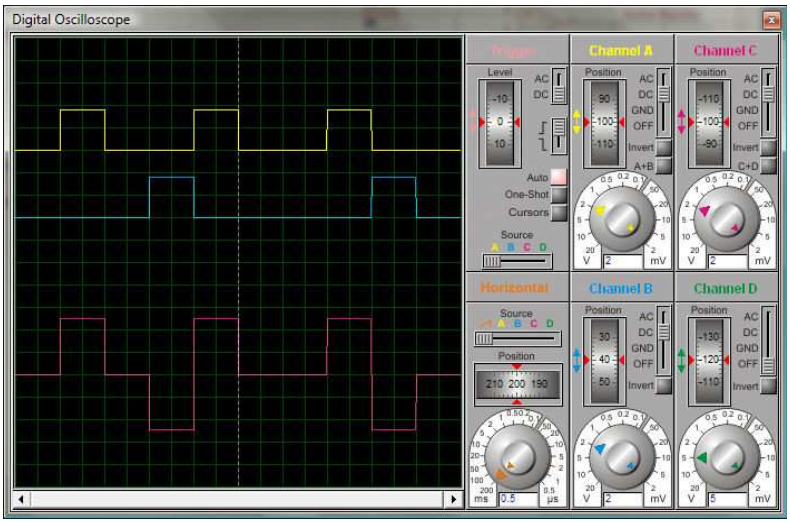


Figura 4.73 Simulación del circuito Interpretador Unipolar - Bipolar en el paquete computacional Proteus Professional.

En la Figura 4.73, en el canal A, señal de color amarillo, ingresa desde la tarjeta de entrenamiento la entrada unipolar 1, *salida1*; en el canal B, señal de color azul, ingresa desde la tarjeta de entrenamiento la entrada unipolar 2, *salida2*; mientras que la salida del circuito se encuentra en el canal C y es la señal de color rojo.

4.10.2 CIRCUITO INTERPRETADOR BIPOLAR – UNIPOLAR

Una vez que se ha interpretado la codificación, el circuito analizado en el item anterior, tiene como salida una señal de tres niveles de voltaje según sea el caso del código. El circuito interpretador bipolar- unipolar va a ser el encargado de traducir los tres niveles de voltaje en dos salidas binarias que van a ser entradas a la tarjeta de entrenamiento para la decodificación.

Diseño del circuito

El circuito tiene como voltaje de entrada una señal bipolar con valores de -5 [V], 0[V] y +5[V]. Se consideró como elemento central a los amplificadores operacionales ya que permiten el manejo de voltajes negativos. Para el diseño se utilizan dos amplificadores operacionales, el primero de ellos va a funcionar como un sumador no inversor, y el restante como sumador inversor. Además, para

enviar voltajes constantes de 3.3 V que acepta la tarjeta de entrenamiento, se utilizan dos diodos zener de 3.3 V.

En la Figura 4.74 se observa el diagrama de bloques que representa la estructura del diseño del circuito en cuestión.

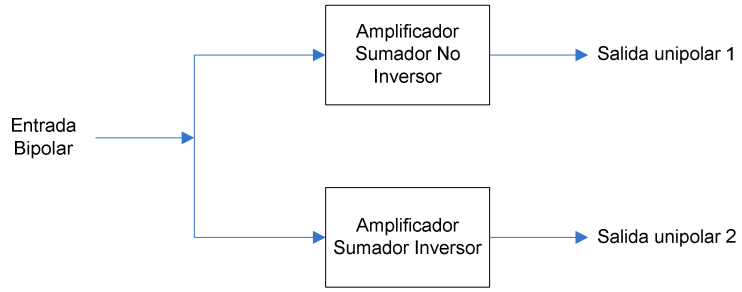


Figura 4.74 Diagrama de bloques del circuito Interpretador Bipolar – Unipolar.

El análisis de este circuito se lo va a iniciar con el sumador no inversor, luego se procederá con el sumador inversor.

$$\boldsymbol{Sumador\ No\ Inversor\ \Rightarrow\ Amplificador\ Operacional\ 1}$$

En la Figura 4.75 se puede establecer lo siguiente:

$$Vo = \left(1 + \frac{R_5}{R_4}\right)\left(\frac{\frac{V_1}{R_1} + \frac{V_2}{R_2} + \frac{V_3}{R_3}}{\frac{1}{R_1} + \frac{1}{R_2} + \frac{1}{R_3}}\right)$$

V_0 constituye el voltaje para la salida unipolar 1, este voltaje debe tomar los siguientes valores: 3.3 [V] o 0 [V], valores que retornan a la tarjeta de entrenamiento que maneja niveles LVTTL.

$$Sea\ R_4 = R_5 = 47K\Omega$$

Donde $V_1 = 0[V]$, $V_2 = 10\,[V]$ y $V_3 = 5[V], 0[V] y - 5[V]$; V_2 es el voltaje de polarización del circuito

Reemplazando los valores de voltaje, R_4 y R_5 en la ecuación anterior se tiene:

$$Vo = \left(1 + \frac{47K\Omega}{47K\Omega}\right)\left(\frac{\frac{10}{R_2} + \frac{V_3}{R_3}}{\frac{1}{R_1} + \frac{1}{R_2} + \frac{1}{R_3}}\right)$$

Si $Vo = 0\ [V]$ y $V_3 = -5\ [V]$

$$0 = 2\left(\frac{\frac{10}{R_2} - \frac{5}{R_3}}{\frac{1}{R_1} + \frac{1}{R_2} + \frac{1}{R_3}}\right)$$

Para que se mantenga la igualdad, el numerador en la fracción presente debe ser cero. Entonces:

$$\frac{10}{R_2} - \frac{5}{R_3} = 0$$

$$\frac{10}{R_2} = \frac{5}{R_3} \rightarrow\ R_2 = 2R_3$$

Sea $R_3 = 75K\Omega,$ Entonces $R_2 = 150K\Omega$

Ahora se considera $Vo = 3{,}3\ [V]$ y $V_3 = 0\ [V]$, reemplazando:

$$3.3 = 2\left(\frac{\frac{10}{2R_3}}{\frac{1}{R_1} + \frac{1}{2R_3} + \frac{1}{R_3}}\right)$$

$$1.65 = \left(\frac{5R_1}{R_3 + 1.5R_1}\right)$$

Despejando R_1 , sabiendo el valor de $R_3 = 75K\Omega,$ se tiene:

$$R_1 = 56.25\ K\Omega\ \rightarrow\ R_1 = 56\ K\Omega$$

$\boldsymbol{Sumador\ Inversor\ \Rightarrow\ Amplificador\ Operacional\ 2}$

El voltaje de salida del sumador inversor de la Figura 4.75 está dado por la expresión:

$$Vo = -\left(\frac{R_8}{R_7}V_1 + \frac{R_8}{R_6}V_2\right)$$

Donde $V_1 = +5[V],\ 0[V],\ -5[V]$ y $V_2 = -10[V]$

En este caso, V_0 es el valor del voltaje para la salida unipolar 2. El valor de V_1 constituye la entrada bipolar, y V_2 tiene un valor de -10 [V], este valor es el voltaje de operación negativo del amplificador operacional.

$Sea\ R_7 = R_8 = 75K\Omega$

Considerando $Vo = 0[V]\ y\ V_1 = 5[V]$ se tiene:

$$0 = -\left(5 + \frac{75}{R_6}(-10)\right) \rightarrow R_6 = 150\ K\Omega$$

En Resumen:

$R_1 = 56\ K\Omega$ $\quad$ $R_5 = 47K\Omega$

$R_2 = 150K\Omega$ $\quad$ $R_6 = 150K\Omega$

$R_3 = 75K\Omega$ $\quad$ $R_7 = 75K\Omega$

$R_4 = 47K\Omega$ $\quad$ $R_8 = 75K\Omega$

En los dos casos restantes de V_1, y en base a los valores de las resistencias obtenidas, se tiene un valor de Vo mayor al que acepta la tarjeta de entrenamiento, por lo que se usó diodos zener de 3.3 V para asegurar un voltaje adecuado en los pines de la tarjeta.

Para la simulación del circuito se utilizó el paquete computacional Proteus Profesional. En la Figura 4.75 se exhibe el el diseño implementado.

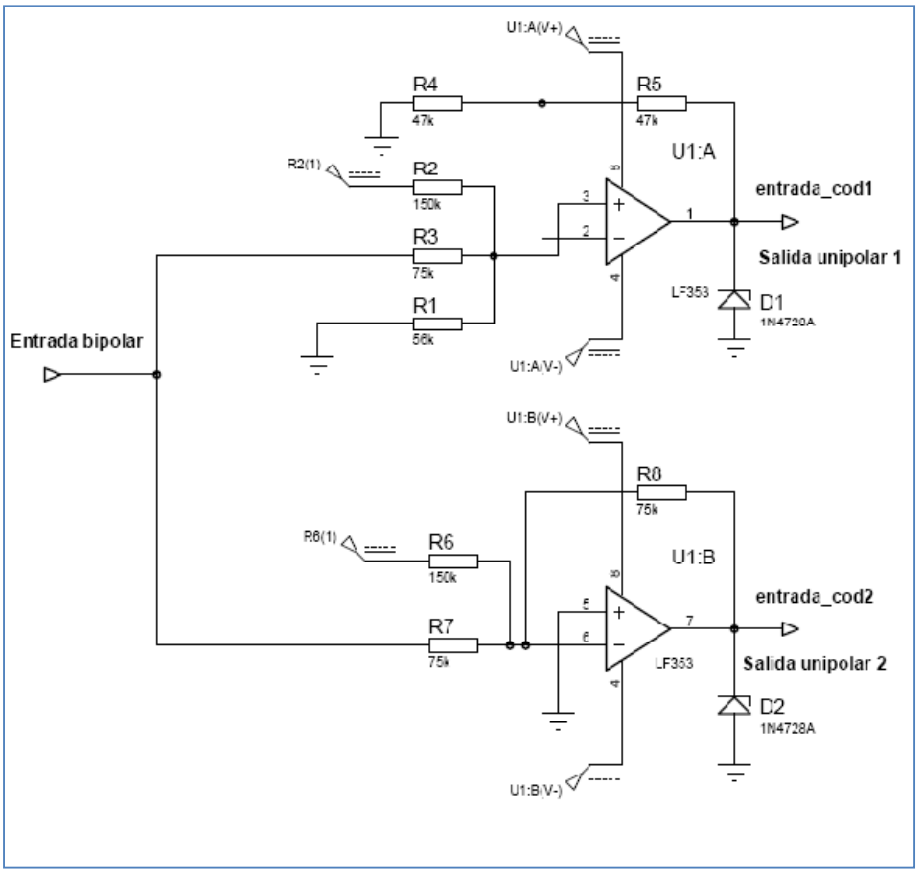


Figura 4.75 Diagrama Esquemático del Circuito interpretador Bipolar – Unipolar.

La simulación del circuito se presenta en la Figura 4.76, donde se observa que se cumple a cabalidad con la Tabla 4.7.

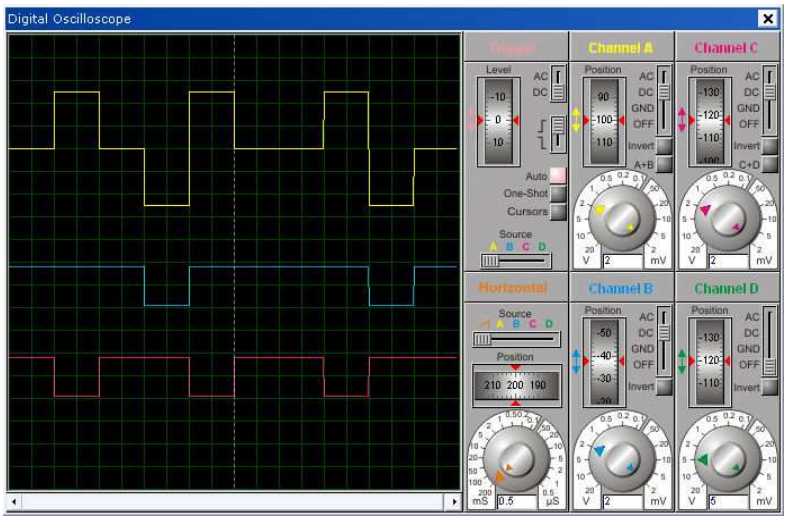


Figura 4.76 Simulación Del circuito Interpretador Bipolar - Unipolar en el paquete computacional Proteus Professional.

De la Figura 4.76 se puede observar, en el canal A la señal en amarillo representa la entrada bipolar, la señal de color azul que está en el canal B es la salida unipolar 1, y por último la señal en rojo figura la salida unipolar 2, presente en el canal C.

4.11 CARACTERES RESTRINGIDOS

En el presente proyecto, los datos a ser codificados y decodificados son caracteres ASCII de ocho bits. Además se han utilizado varios identificadores como el de inicio y fin de datos, de velocidad, y de código de cada uno de los códigos de línea utilizados; los cuales corresponden a caracteres que no se recurren con frecuencia.

En la Tabla 4.9 se exhiben los caracteres restringidos que no deben ser ingresados como datos, correspondientes a los identificadores mencionados.

Tipo de Identificador	Carácter ASCII	Formato Binario	Formato Decimal	Formato Hexadecimal
Identificador de inicio de datos	~	01111110	126	7E
Identificador de fin de datos	|	01111100	124	7C
Velocidad 2400 bps	î	11101110	238	EE
Velocidad 4800 bps	ï	11101111	239	EF
Velocidad 9600 bps	ô	11110100	244	F4
Velocidad 19200 bps	õ	11110101	245	F5
NRZ	À	11000000	192	C0
RZ	Â	11000010	194	C2
4B5B	Ã	11000011	195	C3
Diferencial Tipo M	Ä	11000100	196	C4
Diferencial Tipo S	Å	11000101	197	C5
Manchester	Æ	11000110	198	C6
Manchester Diferencial	Ç	11000111	199	C7
CMI	È	11001000	200	C8
AMI	Ê	11001010	202	CA
HDB3	Ë	11001011	203	CB
MLT3	Ì	11001100	204	CC

Tabla 4.9 Caracteres Restringidos.

CAPÍTULO V

RESULTADOS

Una vez que se ha expuesto en detalle la codificación y decodificación de los códigos de línea con VHDL, así como también su desarrollo en MATLAB; en este capítulo se muestran los resultados obtenidos a nivel de software y hardware.

5.1 HERRAMIENTA UTILIZADA PARA LA VISUALIZACIÓN DE RESULTADOS

Con el objetivo de visualizar y almacenar los resultados producidos en la implementación se utiliza el hardware "DS1M12 Osciloscopio y Generador de Funciones USB". Este instrumento de EasySync puede desenvolverse como osciloscopio y generador de funciones al mismo tiempo. Cabe recalcar que en este proyecto se utiliza el osciloscopio de esta herramienta; para observar, almacenar los resultados, y divulgarlos en este capítulo. Sin embargo los resultados se pueden observar con otros osciloscopios de similares características o superiores.

El hardware DS1M12 se muestra en la Figura 5.1.

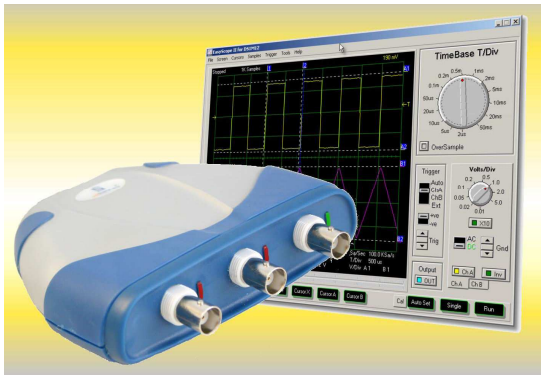


Figura 5.1 DS1M12 Osciloscopio y generador de funciones[16]

Para utilizar el DS1M12, éste viene con el CD de instalación del software y drivers del hardware. Antes de conectar el DS1M12 a un puerto USB, se debe instalar el

[16] Figura tomada de "DS1M12 Getting Starter Guide" de EasySync

EasyScope II (software necesario para el uso de este dispositivo) siguiendo las instrucciones en pantalla una vez insertado el CD en el computador.

Una vez instalado el EasyScope II se debe conectar el DS1M12 a un puerto USB del computador, para lo cual el CD que contiene los instaladores debe estar insertado en el CD ROM. Cuando Windows detecte el nuevo hardware seleccione el CD ROM como fuente de los drivers. Finalmente se podrá usar satisfactoriamente el DS1M12 y el software EasyScope II.

Al utilizar este hardware como osciloscopio se caracteriza por: no requerir alimentación de energía externa ya que usa las líneas de alimentación USB, poseer dos canales de entrada con un ancho de banda de 250 KHz (el tercer canal se desempeña como un generador de funciones), realizar un muestreo simultáneo en los dos canales, tener una velocidad mínima de muestreo de 1 MS/s, permitir un voltaje máximo de entrada por cada canal de 50 V.

El software EasyScope II además de permitir la visualización de las señales en cada canal, permite configurar los siguientes parámetros del osciloscopio: Base de tiempo desde 2us/div a 50ms/div, escala del eje Y desde 10mV/div a 5V/div, cursores de medidas en el eje X y Y en la pantalla, tipos de medidas: mínimo, máximo, medio, V_{RMS} y frecuencia. La Figura 5.2 muestra la pantalla principal del software EasyScope II que permite la manipulación de los parámetros del osciloscopio.

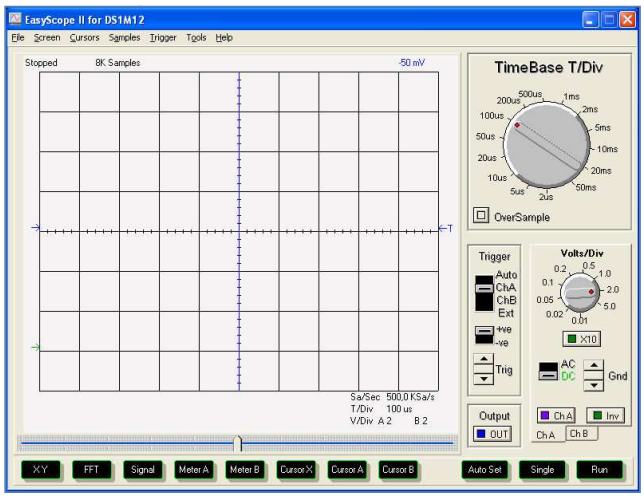


Figura 5.2 Pantalla principal del software Easy Scope II.

5.2 VISUALIZACIÓN DE LAS SEÑALES DE RELOJ

Como se mencionó en el capítulo anterior, ha sido necesario generar cinco señales de reloj para realizar la codificación y decodificación de los once códigos de línea que han sido implementados. Las señales de reloj básicas: *RELOJ_BASE* y *RELOJ_BASE_4B5B*, se presentan en la Figura 5.3. La primera nos permite obtener las señales *RELOJ_OUT y RELOJ_OUT_Doble*, y con la segunda señal se consigue *RELOJ_OUT_4B5B*. Para el caso de los códigos que tienen transición a mitad de tiempo de bit como son el RZ al 50%, Manchester, Manchester Diferencial y CMI, la señal de reloj *RELOJ_OUT_Doble* es apta. En cuanto a los códigos NRZ, Diferencial Tipo M, Diferencial Tipo S, AMI, HDB3 y MLT3 se utiliza la señal *RELOJ_OUT*, cuyo ciclo de reloj es igual a un tiempo de bit. El código 4B5B es un caso especial ya que requiere de una señal de reloj propia para su funcionamiento como es *RELOJ_OUT_4B5B*. Las Figuras 5.4 y 5.5 presentan las tres señales de reloj generadas en base a las dos primeras.

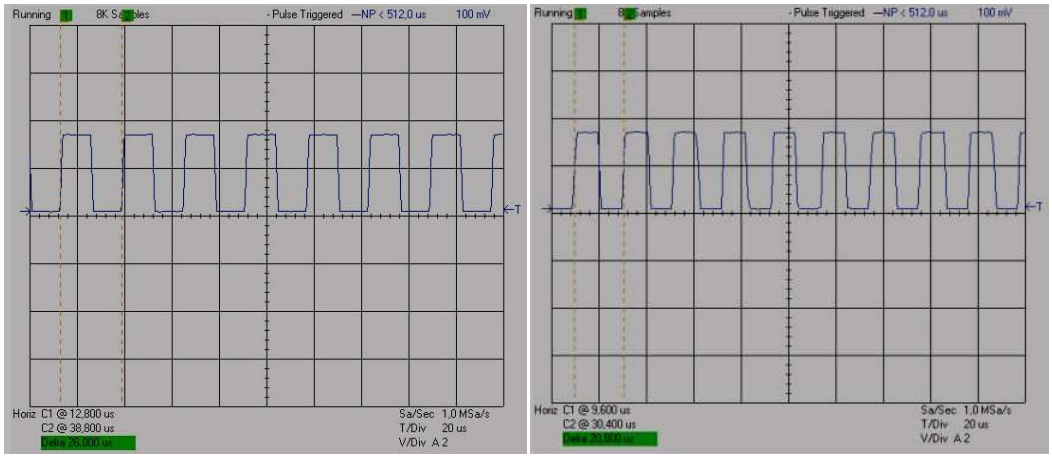


Figura 5.3 Señales de Reloj RELOJ_BASE de 38.4 KHz, y RELOJ_BASE_4B5B de 48 KHz.

La señal de reloj base de 38.4 KHz, tiene un período de 26.04 µs, lo que se comprueba en la figura al observar que un ciclo de reloj tiene una duración de 26 µs aproximadamente. Para la señal de 48 KHz el valor del período es de 20.83 µs. Para una mejor visualización de las señales es conveniente fijar en el osciloscopio una amplitud de 2 voltios por división y en cuanto a tiempo 20 µs por división.

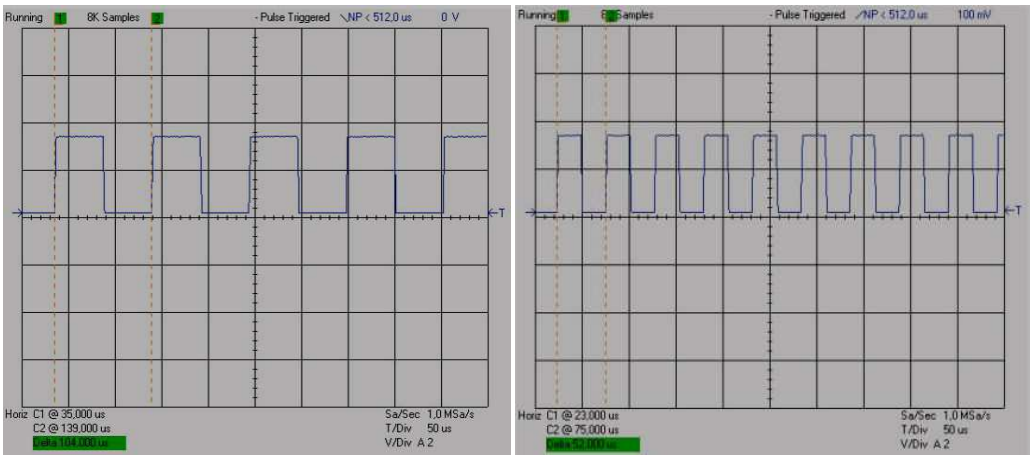


Figura 5.4 Señales de Reloj RELOJ_OUT y RELOJ_OUT_Doble.

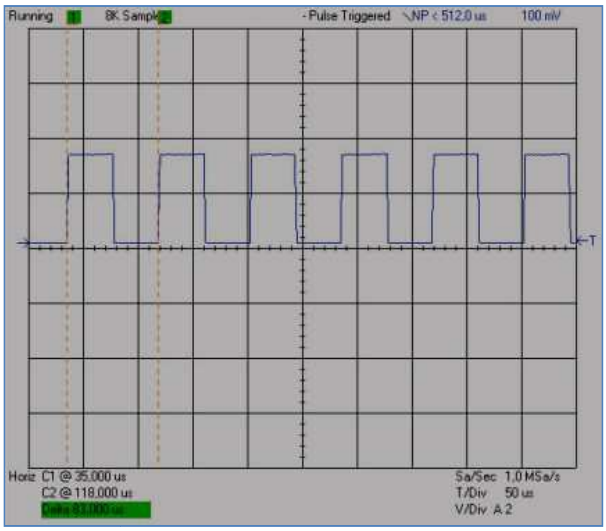


Figura 5.5 Señal de Reloj RELOJ_OUT_4B5B.

Las Figuras 5.4 y 5.5 se producen cuando se ha seleccionado la velocidad estándar de 9600 bps en la interfaz gráfica.

Entonces la señal *RELOJ_OUT* tendrá un valor de 9600 Hz con un período de 104.16 µs. La señal *RELOJ_OUT_Doble* será de 19200 Hz con su respectivo período de 52.08 µs. Estas dos señales se pueden observar en la Figura 5.4.

La señal *RELOJ_OUT_4B5B* será de 12000 Hz en base a lo seleccionado, y el período correspondiente es de 83.33 µs, tal como lo muestra la Figura 5.5.

5.3 VISUALIZACIÓN DEL STREAM DE DATOS ENVIADO DESDE MATLAB

El stream de datos que contiene la información necesaria para realizar los procesos en VHDL, incluye el identificador de inicio de datos, el identificador de código, los datos en sí y el identificador de fin de datos. Estos caracteres son transmitidos asincrónicamente por lo que cada uno tiene un bit de inicio, bit de paridad y bit de parada a demás de los 8 bits de datos, tal como lo muestra la Figura 5.6.

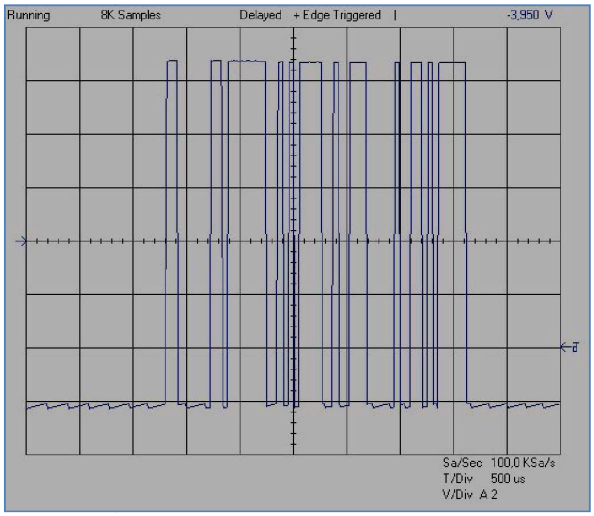


Figura 5.6 Stream de Datos.

En la Figura 5.6 se observa el stream de datos que es enviado luego de configurar la velocidad a 19200 bps y presionar el botón de codificación en la interfaz gráfica. Estos caracteres son transmitidos al FPGA a través del Interfaz RS-232, por lo que los valores de voltaje negativos corresponden al estado lógico 1L, mientras que los valores de voltaje positivos corresponden al estado lógico 0L. En esta figura particular se visualiza el identificador de inicio (7EH), el identificador de código correspondiente al código de línea NRZ (C0H), el carácter “a” de dato (61H), el identificador de fin de datos (7CH) y el terminal (0AH), LF, que MATLAB envía por defecto. De esta manera los bits enviados al FPGA son los que se observan en la Tabla 5.1 y corresponden a la Figura 5.6.

ID Inicio	ID Código	Datos (a)	ID Fin	LF
00111111001	00000001101	01000011011	00011111011	00101000001

Tabla 5.1 Bits del Stream de Datos.

5.4 VISUALIZACIÓN DE LA SEÑAL A CODIFICAR EN EL FPGA

En la Figura 5.7 se muestra la señal a codificar en el FPGA luego de haberse realizado los procesos necesarios para obtener únicamente los bits de datos a ser codificados, es decir sin bits de inicio, paridad y parada. Esta señal incluye los identificadores de inicio de datos y fin de datos, que se usan después de la decodificación.

De forma particular en la Figura 5.7 los bits de datos corresponden al carácter “a”, con el cual se realizan casi todas las pruebas de funcionamiento de los códigos de línea, excepto la prueba de funcionamiento del código HDB3 que se realiza con caracteres que permitan visualizar bits de relleno y violación.

En la Interfaz gráfica cuando se presiona el botón “Graficar señal a codificar y codificada”, se grafica una señal de datos que representa los mismos bits de la Figura 5.7, tal como lo muestran las figuras de los resultados de codificación en MATLAB con cada uno de los códigos de línea.

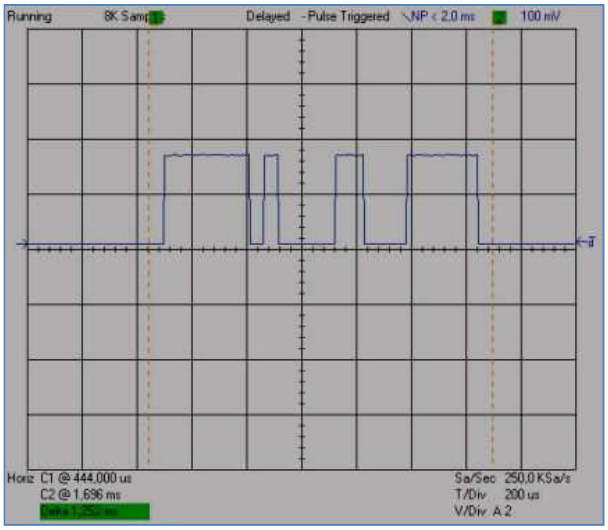


Figura 5.7 Señal de Datos en el FPGA a ser codificada.

La Tabla 5.2 muestra los bits de la señal de datos a ser codificados en el FPGA, los mismos que se visualizan en la Figura 5.7.

ID Inicio	Datos (a)	ID Fin
0 1 1 1 1 1 1 0	1 0 0 0 0 1 1 0	0 0 1 1 1 1 1 0

Tabla 5.2 Bits de la Señal de Datos a ser codificada.

5.5 RESULTADOS DEL FUNCIONAMIENTO DE LOS CÓDIGOS DE LÍNEA

Los resultados de la implementación de los códigos de línea, se los visualiza en el osciloscopio antes mencionado. Las pruebas de funcionamiento para todos los códigos se van a realizar a 19200 bps, que constituye la mayor velocidad y la más crítica. Sólo para el código de línea Diferencial Tipo M se van a efectuar pruebas de funcionamiento a las cuatro velocidades para fines de demostración. Para comparar los resultados del software y hardware, se presenta la codificación tanto en MATLAB, mediante la interfaz gráfica, como la que origina la tarjeta de entrenamiento, reflejada en el osciloscopio.

Todas las pruebas de funcionamiento se realizan con el carácter ASCII “a” de valor binario 1000 0110, el bit menos significativo el de la izquierda, 61H en hexadecimal, excepto en el código HDB3. El carácter de ejemplo va a estar acompañado de dos caracteres adicionales que constituyen los indicadores de inicio y fin de datos tal como se muestra en la Tabla 5.2.

La señal de datos que sale del respectivo pin de la tarjeta de entrenamiento (E7), maneja niveles de voltaje LVTTL, esto es 0V y + 3.3V, y la señal codificada posee niveles de voltaje TTL bipolares, es decir – 5V, 0V y 5V; estos valores se los aprecia a la salida del Circuito Interpretador Unipolar – Bipolar. Los valores que retornan a la tarjeta de entrenamiento para la decodificación deben ser LVTTL, por lo que el Circuito Interpretador Bipolar – Unipolar asegura estos niveles de voltaje.

En cuanto al análisis de la señal codificada, los tres niveles que se maneja van a estar representados según la Tabla 5.3.

Niveles de voltaje	Representación
+ 5 [V]	+A
0 [V]	0
- 5 [V]	-A

Tabla 5.3 Niveles de voltaje y su representación.

Además, hay códigos que presentan transiciones a mitad de tiempo bit, la forma como se representan las transiciones positivas y negativas se visualiza en la Tabla 5.4.

Transición	Representación
Positiva	
Negativa	

Tabla 5.4 Transiciones positiva y negativa.

5.5.1 NRZ

El resultado de la codificación NRZ en MATLAB de la señal de datos a codificar se muestra en la Figura 5.8.

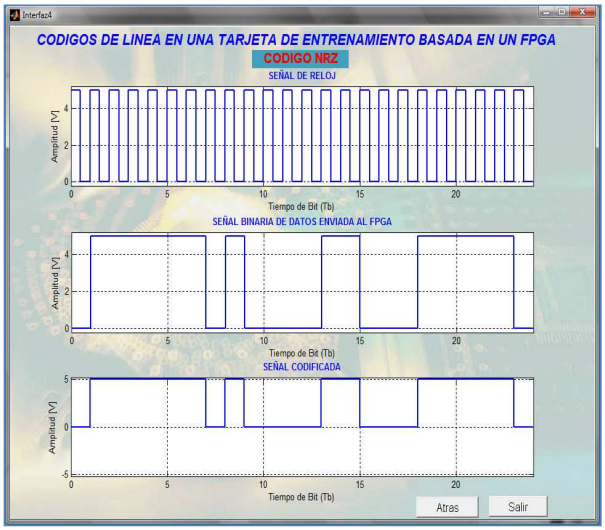


Figura 5.8 Codificación NRZ en MATLAB.

En la Figura 5.9 se muestra en el canal A (color azul) la señal de datos a codificar una vez que se han desarrollado los procesos respectivos en VHDL previos a su obtención; en el canal B (color verde) se observa la señal codificada en VHDL a la salida del circuito interpretador Unipolar – Bipolar. Mediante los cursores en el eje horizontal se determina que un tiempo de bit es 52 µs aproximadamente, correspondiente a la velocidad de 19200 bps con la cual se realizan las pruebas de funcionamiento. Además se aprecia un retardo de un tiempo de bit de la señal

codificada respecto a la señal a codificar, producido por las razones explicadas en el Análisis de tiempos de procesamiento.

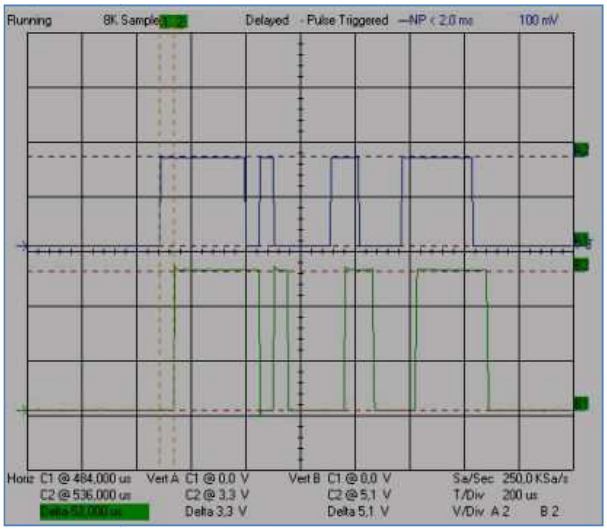


Figura 5.9 Visualización de la Codificación NRZ en VHDL.

Al ser el primer código en ser probado su funcionamiento, se utilizan los cursores del eje vertical, con los mismos que se demuestra, tal como estaba previsto, que la señal de datos en la Spartan-3E Starter Kit Board utiliza los niveles de voltaje LVTTL, característica de la tarjeta de entrenamiento; mientras que la señal codificada, a la salida del circuito interpretador Unipolar - Bipolar obtiene los niveles de voltaje TTL, es decir 0 V para el estado lógico 0L y 5 V para el estado lógico 1L.

En la Tabla 5.5 se detallan los bits de datos y los bits codificados que se muestran en la Figura 5.9.

Señal de datos	0	1	1	1	1	1	1	0	1	0	0	0	0	1	1	0	0	0	1	1	1	1	1	0
Señal codificada	0	+A	+A	+A	+A	+A	+A	0	+A	0	0	0	0	+A	+A	0	0	0	+A	+A	+A	+A	+A	0

Tabla 5.5 Bits de datos y bits codificados con el Código de Línea NRZ.

En la Figura 5.10 se observan la señal codificada y la señal decodificada, en los canales A y B respectivamente. Se puede apreciar en esta figura que la señal decodificada es igual a la señal de datos mostrada en la Figura 5.9. Además se aprecia un retardo de un tiempo de bit de la señal decodificada respecto a la señal a decodificar, lo que se explica en el Análisis de tiempos de procesamiento.

Los cursores paralelos del eje del tiempo, permiten demostrar que la señal a ser decodificada tiene los niveles de voltaje TTL, es decir 0 V para el estado lógico 0L y 5 V para el estado lógico 1L; mientras que la señal decodificada en la Spartan-3E Starter Kit Board tiene niveles de voltaje LVTTL, para este caso 3.3 V.

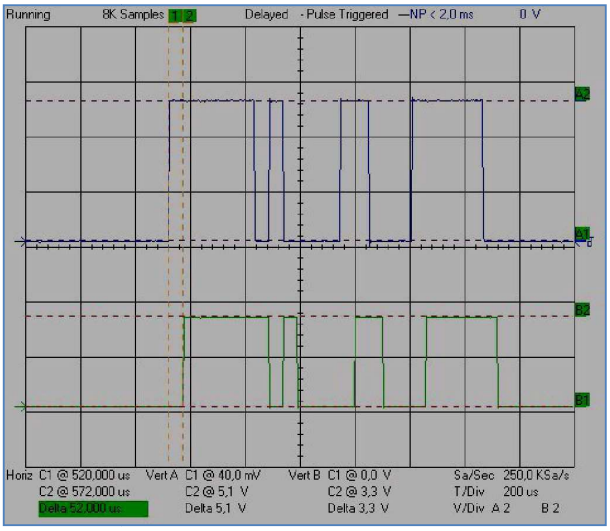


Figura 5.10 Visualización de la Decodificación NRZ en VHDL.

5.5.2 RZ al 50%

El resultado en MATLAB de éste código se presenta en la Figura 5.11, donde se exhibe la señal de reloj, la señal de datos y la señal codificada.

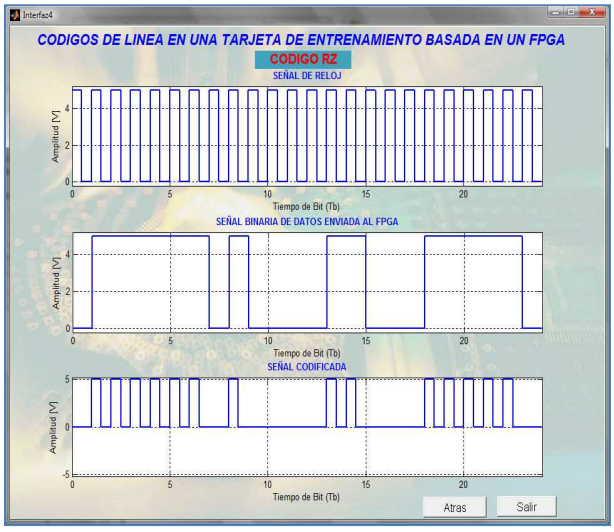
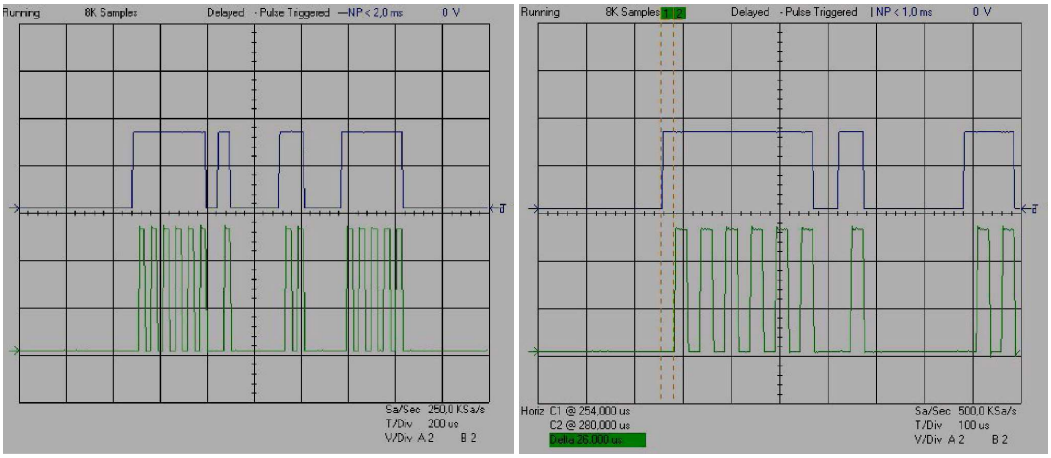


Figura 5.11 Codificación RZ al 50% en MATLAB.

Para el código RZ al 50%, la Figura 5.12 presenta en la parte (a) la señal de datos a codificar en color azul, y la señal codificada en verde, a 200 µs por división de tiempo, lo que nos permite observar las señales en su totalidad. En la parte (b) de la figura, la captura en el osciloscopio se muestra a 100 µs, este tiempo nos faculta a apreciar el retardo entre señal a codificar y codificada, que es de medio tiempo de bit, es válido recordar que la velocidad a la que se hacen las pruebas es de 19200 bps, donde su tiempo de bit es de 52 µs, por lo que dicho retardo se traduce en 26 µs.



(a) (b)

Figura 5.12 Visualización de la Codificación RZ al 50% en VHDL.

En la Tabla 5.6 se pueden observar los bits que representan la señal de datos y la señal codificada. La transición negativa supone el retorno a cero a mitad de tiempo de bit.

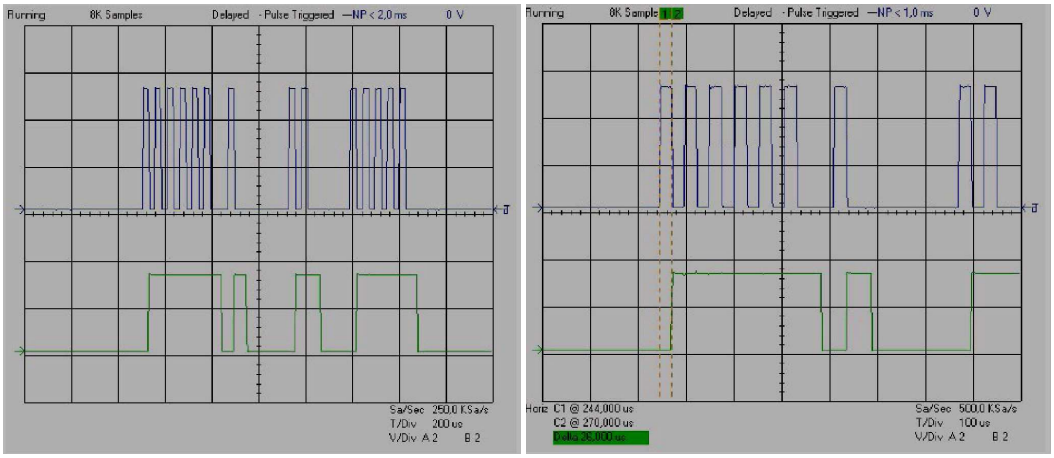
Señal de datos	0	1	1	1	1	1	1	0	1	0	0	0	0	1	1	0	0	0	1	1	1	1	1	0
Señal codificada	0	↓	↓	↓	↓	↓	↓	0	↓	0	0	0	0	↓	↓	0	0	0	↓	↓	↓	↓	↓	0

Tabla 5.6 Bits de datos y bits codificados con el Código de Línea RZ al 50%.

Las dos figuras anteriores están sujetas a comparación, se puede observar que la codificación tanto en el software como en el hardware coinciden, con lo que se concluye el correcto funcionamiento del hardware.

En la parte (a) de la Figura 5.13, presenta la decodificación, donde se muestra la señal que proviene del Circuito Interpretador Unipolar – Bipolar, es decir la señal codificada en azul, y la señal decodificada en verde, tomada desde el pin respectivo en la tarjeta de entrenamiento.

En la parte (b) de esta figura se aprecia que el espacio de tiempo entre la señal codificada y decodificada es de 26 µs.



(a) (b)

Figura 5.13 Visualización de la Decodificación RZ al 50% en VHDL.

5.5.3 4B5B

En cuanto al código 4B5B, su codificación en MATLAB se presenta en la Figura 5.14. Se puede apreciar la señal de reloj, la señal binaria de datos a codificar y la señal codificada en base a la tabla 4B5B que está disponible en la interfaz gráfica.

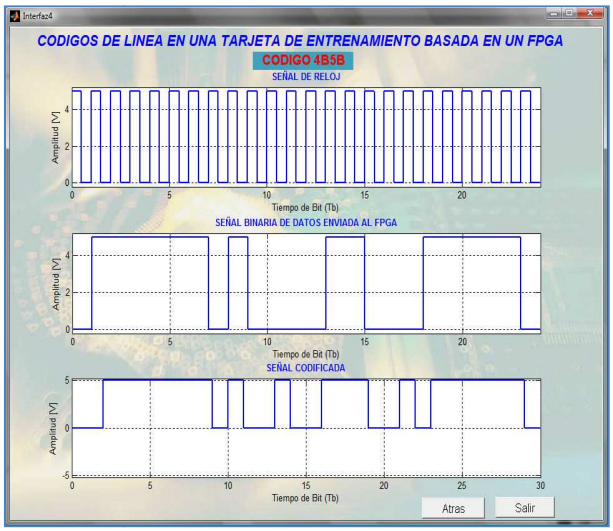


Figura 5.14 Codificación 4B5B en MATLAB.

En la Figura 5.15 se aprecia en color azul la señal de datos a codificar, la misma que se observa en MATLAB, y en color verde la señal codificada proveniente del circuito externo.

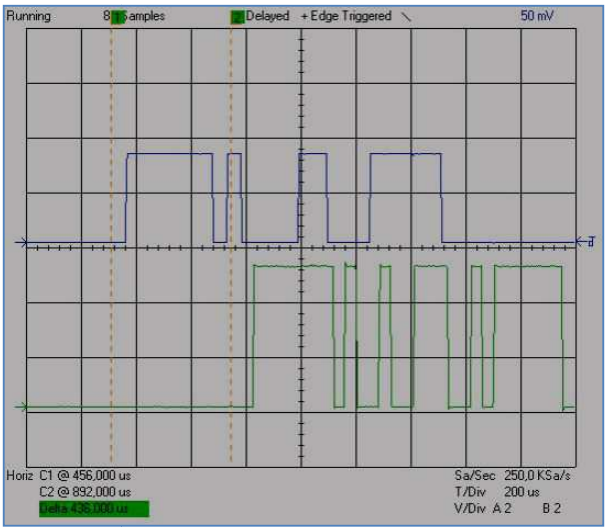


Figura 5.15 Visualización de la Codificación 4B5B en VHDL.

El resumen de la codificación se aprecia en la Tabla 5.7. Se ha agrupado la señal de datos en cuatro bits, con su correspondiente grupo de cinco bits en la señal codificada. Es necesario mencionar que las señales tienen el bit menos significativo a la izquierda.

Señal de datos	0111	1110	1000	0110	0011	1110
Señal codificada	00111	11110	10010	01110	01011	11110

Tabla 5.7 Bits de datos y bits codificados con el Código de Línea 4B5B.

Se puede observar en la figura que el retardo experimentado por las señales a codificar y codificada es de 436 µs. Este hecho se explicará en la sección de tiempos de procesamiento.

La Figura 5.16 presenta la captura de la decodificación en el osciloscopio, donde el canal A representa la señal codificada, y el canal B sostiene la señal decodificada. Las señales especificadas anteriormente están separadas por un espacio de tiempo de 458 µs.

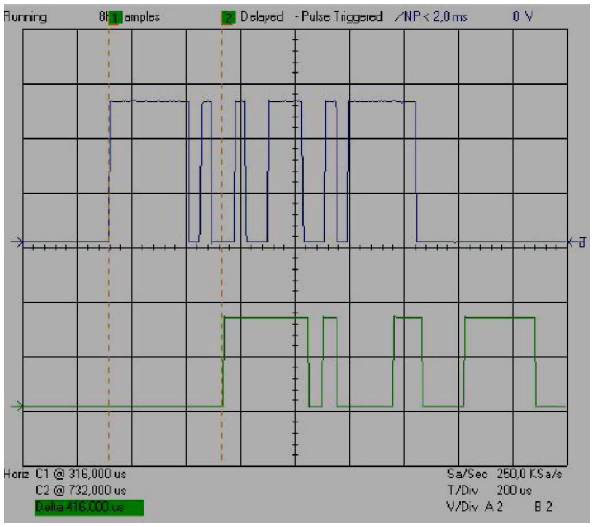


Figura 5.16 Visualización de la Decodificación 4B5B en VHDL.

Es necesario mencionar en este caso, que el retardo total que experimenta la señal de datos en el proceso de codificación y decodificación es de 892 µs, tal como se muestra en la Figura 5.17.

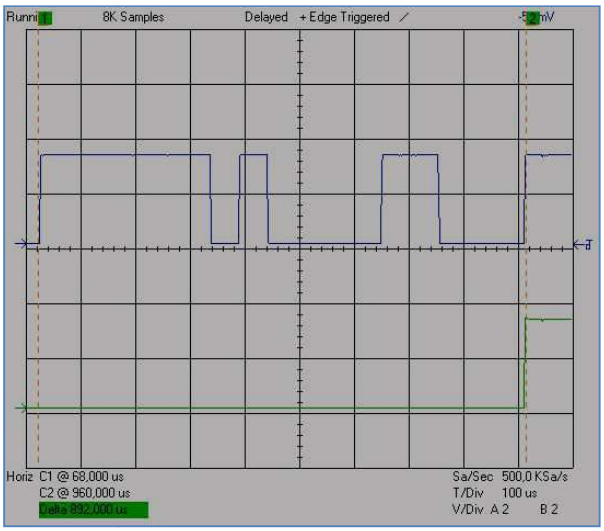


Figura 5.17 Visualización del retardo total del proceso de codificación y decodificación 4B5B en VHDL.

5.5.4 Diferencial tipo M

La Figura 5.18 muestra la codificación Diferencial tipo M en MATLAB.

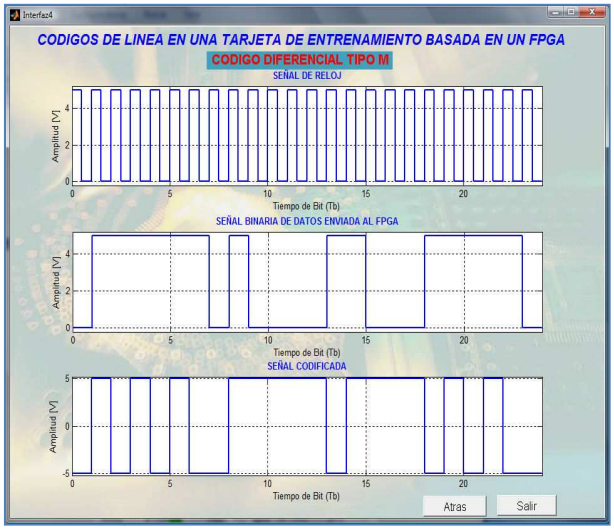


Figura 5.18 Codificación Diferencial tipo M en MATLAB.

En la Figura 5.19 se muestra la señal de datos a codificar y la señal codificada. Los cursores en el eje horizontal demuestran que un tiempo de bit es 52 µs aproximadamente y que existe un retardo de un tiempo de bit en realizarse la codificación.

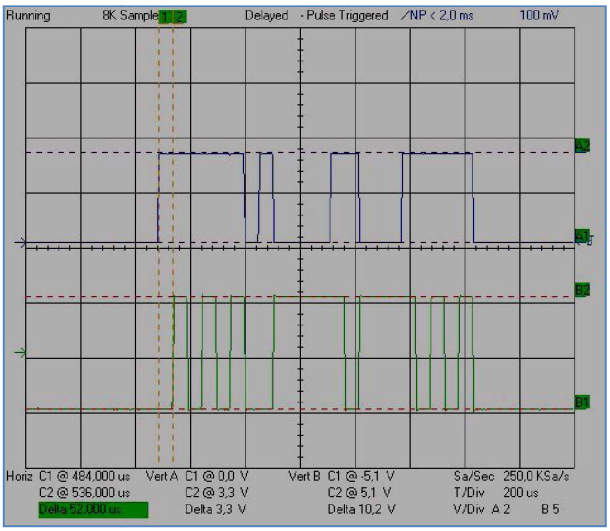


Figura 5.19 Visualización de la Codificación Diferencial tipo M en VHDL.

Los cursores del eje vertical del canal B demuestran que la señal codificada a la salida del circuito interpretador Unipolar – Bipolar es polar y únicamente obtiene los valores de voltaje – 5 V y 5 V.

La Tabla 5.8 demuestra que en la Figura 5.19 se cumple con la regla de codificación del código de línea Diferencial tipo M.

Señal de datos	0	1	1	1	1	1	1	0	1	0	0	0	0	1	1	0	0	0	1	1	1	1	1	0
Señal codificada	-A	+A	-A	+A	-A	+A	-A	-A	+A	+A	+A	+A	+A	-A	+A	+A	+A	+A	-A	+A	-A	+A	-A	-A

Tabla 5.8 Bits de datos y bits codificados con el Código de Línea Diferencial tipo M.

La señal codificada y la señal decodificada se visualizan en la Figura 5.20. Existe un retardo de un tiempo de bit de la señal decodificada respecto a la señal codificada.

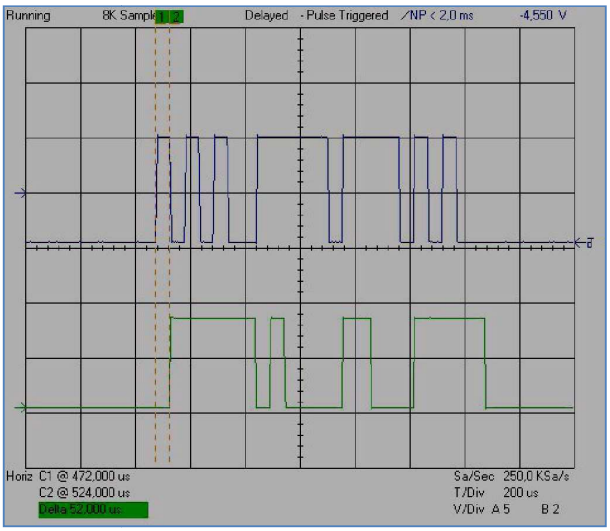


Figura 5.20 Visualización de la Decodificación Diferencial Tipo M en VHDL.

Como es de esperarse, la señal decodificada de la Figura 5.20 representa los mismos bits que la señal de datos mostrada en la Figura 5.19.

5.5.5 Diferencial tipo S

La codificación Diferencial Tipo S en MATLAB, se exhibe en la Figura 5.21, al igual que las demás, está compuesta por las tres señales que muestran su funcionamiento de acuerdo a su regla de codificación.

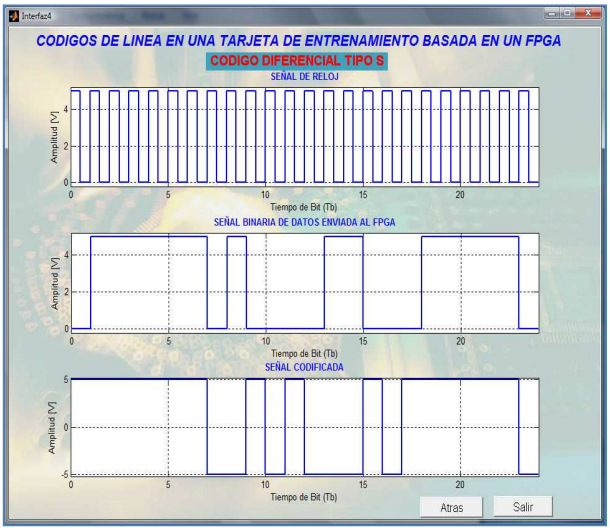


Figura 5.21 Codificación Diferencial Tipo S en MATLAB.

Para la valoración del código, se ha capturado una imagen de su funcionamiento en el osciloscopio, donde se distingue la señal a codificar en azul, y la señal codificada en color verde. En la Figura 5.22 se observa el retardo de un tiempo de bit entre la señales en color azul y verde.

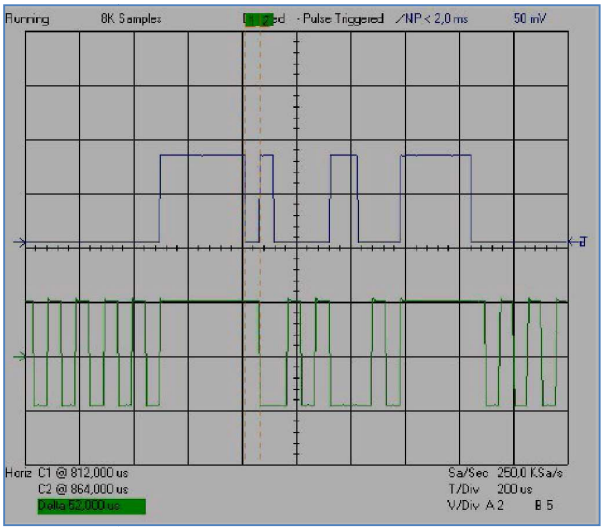


Figura 5.22 Visualización de la Codificación Diferencial Tipo S en VHDL.

La Tabla 5.9 ostenta la codificación del código polar en cuestión con los niveles especificados al inicio del capítulo y su respectiva representación. El valor de referencia es -5 [V].

Señal de datos	0	1	1	1	1	1	1	0	1	0	0	0	0	1	1	0	0	0	1	1	1	1	1	0
Señal codificada	+A	+A	+A	+A	+A	+A	+A	-A	-A	+A	-A	+A	-A	-A	-A	+A	-A	+A	+A	+A	+A	+A	+A	-A

Tabla 5.9 Bits de datos y bits codificados con el Código de Línea Diferencial tipo S.

La Figura 5.23 presenta la decodificación, se observa que en este proceso se obtiene un bit de retardo entre las señales codificada y decodificada.

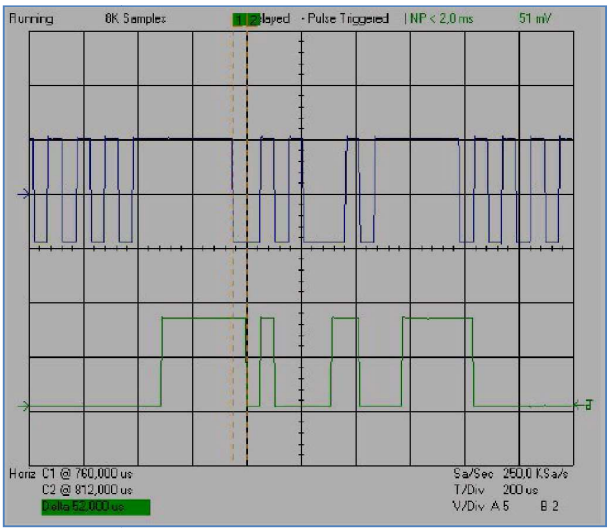


Figura 5.23 Visualización de la Decodificación Diferencial Tipo S en VHDL.

5.5.6 Manchester

La codificación en MATLAB de la señal de datos con el código de línea Manchester se visualiza en la Figura 5.24.

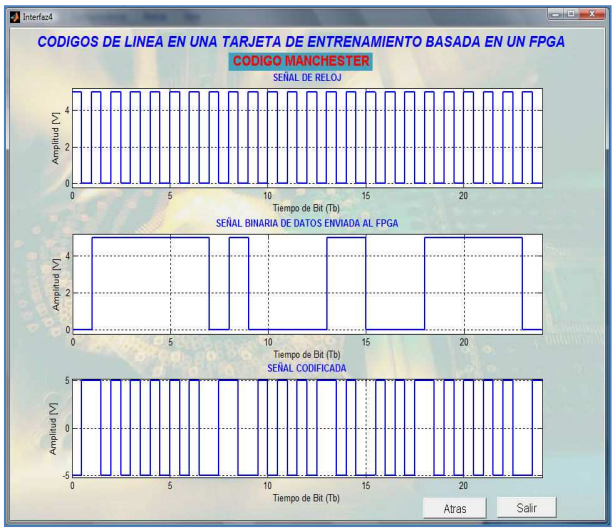


Figura 5.24 Codificación Manchester en MATLAB.

La señal de datos a codificar y la señal codificada en el FPGA a la salida del circuito interpretador Unipolar - Bipolar se muestran en la Figura 5.25.

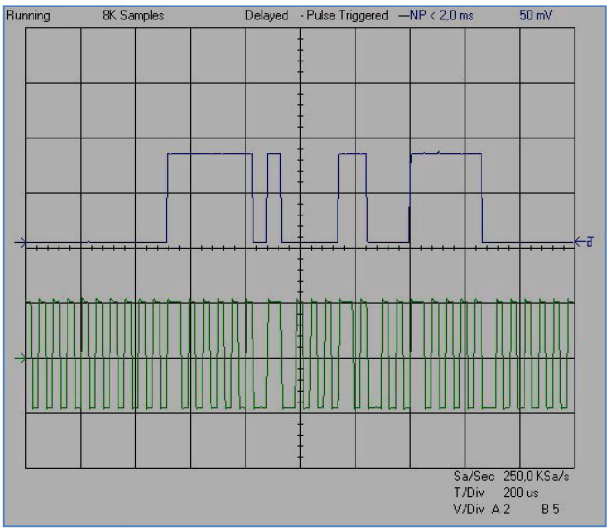


Figura 5.25 Visualización de la Codificación Manchester en VHDL.

En la Tabla 5.10 se demuestra que en la codificación de la señal de datos visualizada en la Figura 5.25 se cumple con la regla del código de línea Manchester.

Señal de datos	0	1	1	1	1	1	1	0	1	0	0	0	0	1	1	0	0	0	1	1	1	1	1	0
Señal codificada																								

Tabla 5.10 Bits de datos y bits codificados con el Código de Línea Manchester.

En la Figura 5.26 se muestra una parte de la señal de datos y de la señal codificada, con el fin de determinar mediante los cursores en el eje horizontal que la señal codificada respecto a la señal de datos tiene un retardo de medio tiempo de bit, 26 μs aproximadamente.

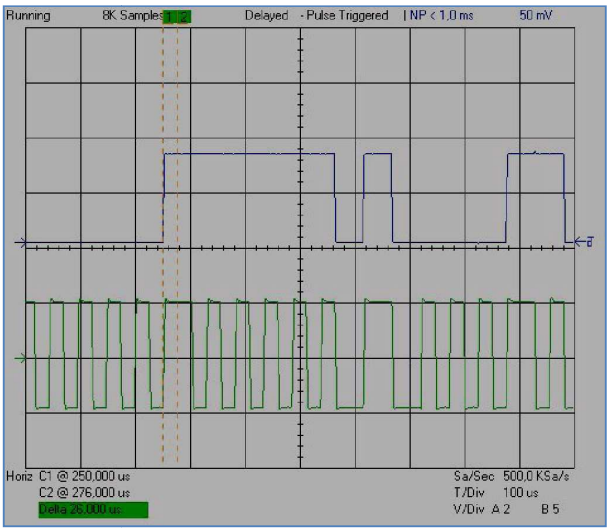


Figura 5.26 Retardo entre la señal de datos y la señal codificada con el Código de Línea Manchester.

La señal a decodificar y la señal decodificada en el FPGA se visualizan en la Figura 5.27.

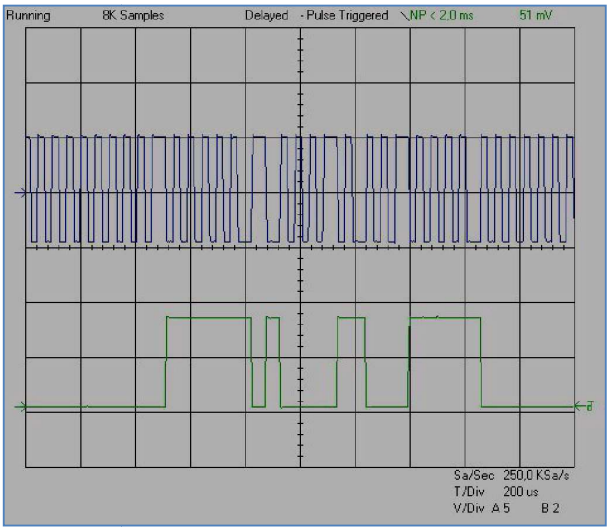


Figura 5.27 Visualización de la Decodificación Manchester en VHDL.

La Figura 5.28 muestra que existe un retardo de un medio tiempo de bit de la señal decodificada respecto a la señal a decodificar.

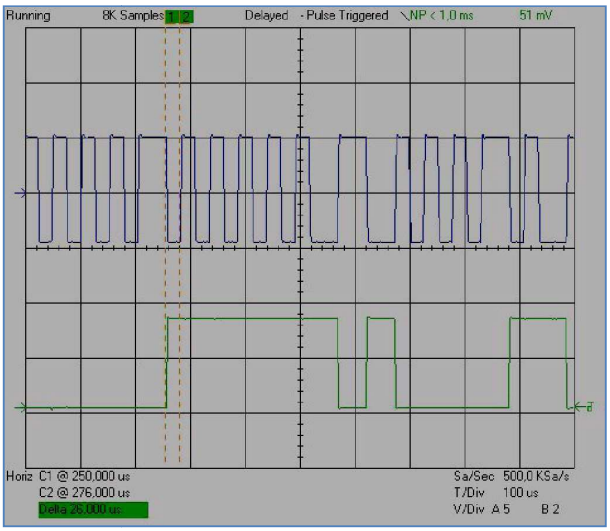


Figura 5.28 Retardo entre la señal a decodificar y la señal decodificada con el Código de Línea Manchester.

5.5.7 Manchester Diferencial

En la Figura 5.29 se muestra la codificación Manchester Diferencial hecha en MATLAB.

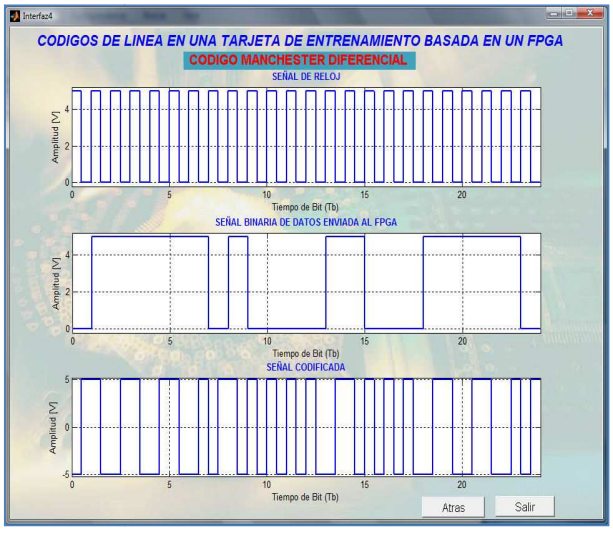
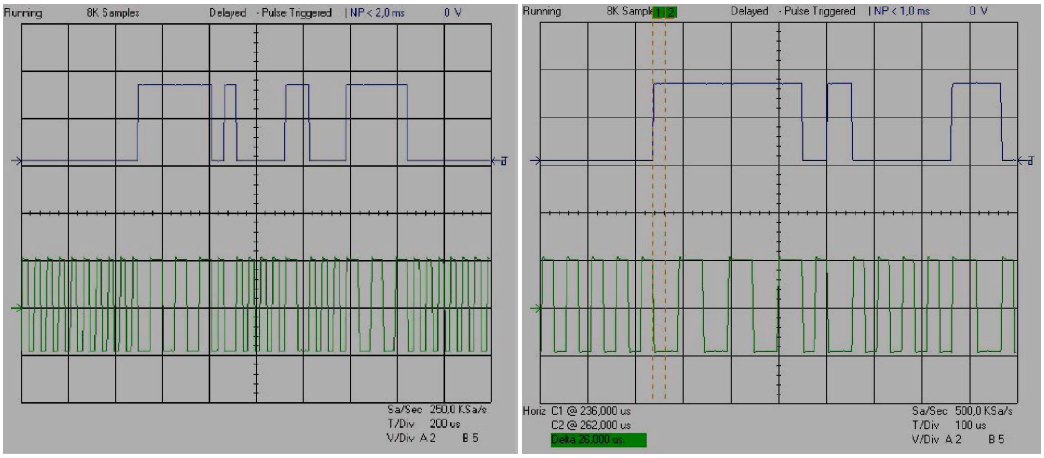


Figura 5.29 Codificación Manchester Diferencial en MATLAB.

Al igual que el código Manchester éste código polar presenta transiciones a mitad de bit para la codificación de ceros y unos. En la Figura 5.30 se presenta en la

parte (a), en el canal A la señal a codificar y en el B la señal codificada. La transición de partida es negativa, por lo que los ceros que acompaña a la señal de datos son traducidos en transiciones negativas, hasta el primer 1L que cambia la transición. La parte (b) permite observar el medio bit de retardo existente entre las señales involucradas.



(a) (b)

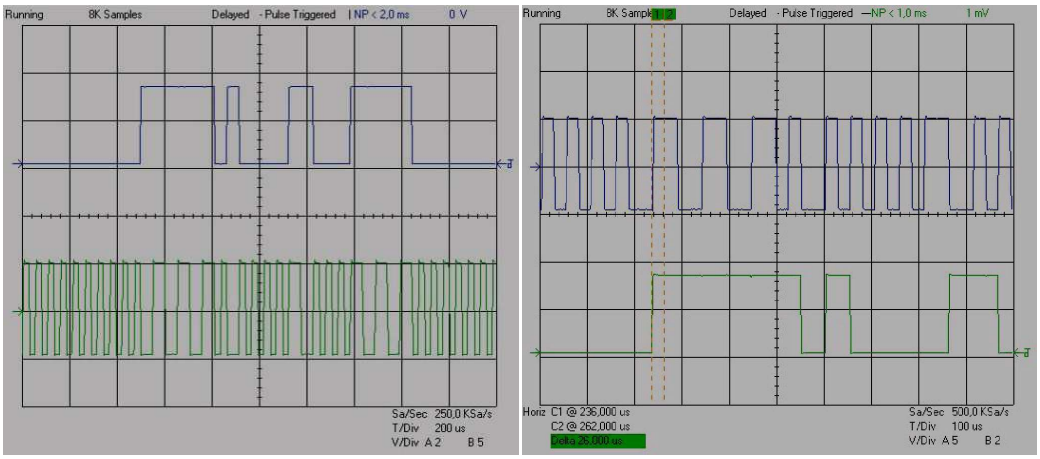
Figura 5.30 Visualización de la Codificación Manchester Diferencial en VHDL.

En la Tabla 5.11 se puede observar con más claridad la codificación en términos de bits a la señal de datos y la señal codificada interpretada en transiciones.

Señal de datos	0	1	1	1	1	1	1	0	1	0	0	0	0	1	1	0	0	0	1	1	1	1	0
Señal codificada																							

Tabla 5.11 Bits de datos y bits codificados con el Código de Línea Manchester Diferencial.

La Figura 5.31 muestra la captura del osciloscopio donde la señal de color azul es la codificada, y la señal en verde representa la decodificada. Existe medio bit de retardo en el proceso de decodificación.



(a) (b)

Figura 5.31 Visualización de la Decodificación Manchester Diferencial en VHDL.

5.5.8 CMI

La codificación en MATLAB de la señal de datos con el código de línea CMI se muestra en la Figura 5.32.

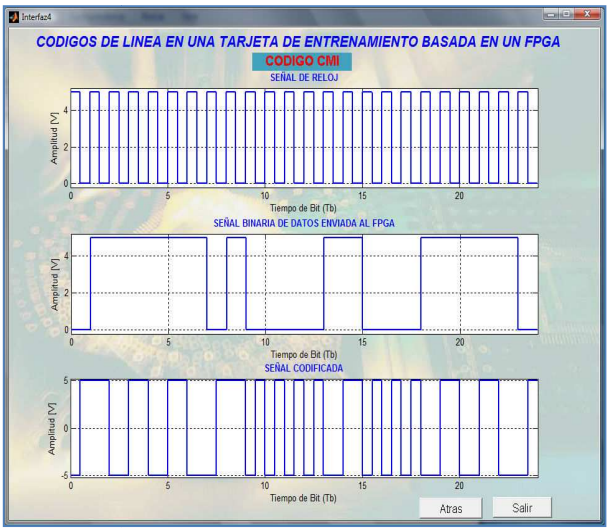


Figura 5.32 Codificación CMI en MATLAB.

En la Figura 5.33 se visualiza la señal de datos a codificar y la señal codificada en VHDL con el código de línea CMI.

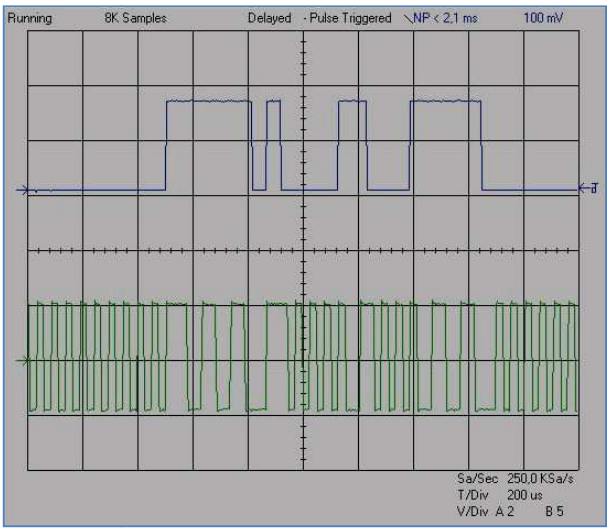


Figura 5.33 Visualización de la Codificación CMI en VHDL.

En la Figura 5.34 se aprecia que existe un retardo de medio tiempo de bit en obtenerse la señal codificada respecto a la señal a codificar.

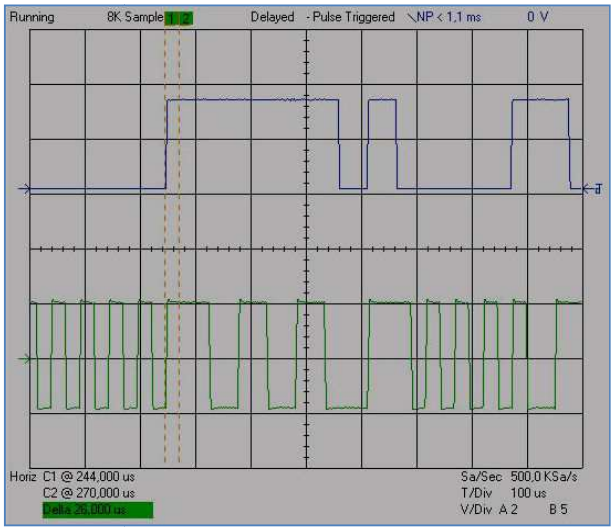


Figura 5.34 Retardo entre la señal a codificar y la señal codificada con el Código de Línea CMI.

En la Tabla 5.12 se detalla la señal de datos y la señal codificada que se muestra en la Figura 5.34.

Señal de datos	0	1	1	1	1	1	1	0	1	0	0	0	0	1	1	0	0	0	1	1	1	1	1	0
Señal codificada		+A	-A	+A	-A	+A	-A		+A					-A	+A				-A	+A	-A	+A	-A	

Tabla 5.12 Bits de datos y bits codificados con el Código de Línea CMI.

La señal a decodificar y la señal decodificada se observan en la Figura 5.35. En esta figura se visualiza un retardo de un tiempo de bit entre las dos señales.

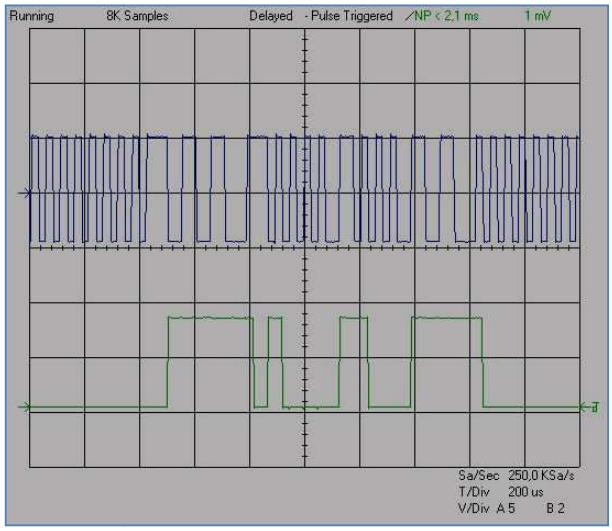


Figura 5.35 Visualización de la Decodificación CMI en VHDL.

5.5.9 AMI

La Figura 5.36 muestra la señal de reloj, señal de datos y la señal codificada en MATLAB luego de haberse realizado la codificación AMI.

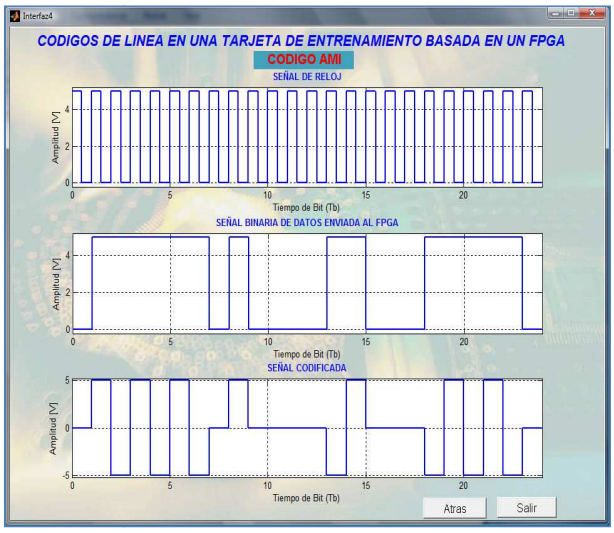


Figura 5.36 Codificación AMI en MATLAB.

La codificación en VHDL de la señal de datos, se muestra en la Figura 5.37. En esta figura se aprecia que la duración del tiempo de bit es aproximadamente 52 us al utilizar la velocidad de 19200 bps. Además con los cursores se determina que la señal codificada se obtiene con un retardo de un tiempo de bit respecto a la señal a codificar.

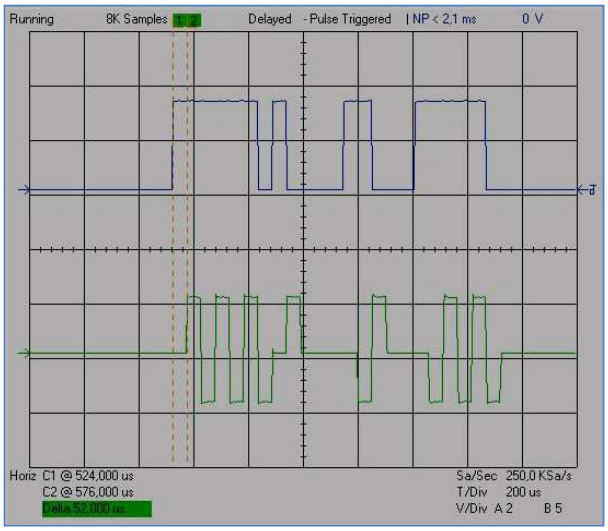


Figura 5.37 Visualización de la Codificación AMI en VHDL.

Los bits de datos y bits codificados con el código de línea AMI se muestran en la Tabla 5.13, los mismos que se obtienen al visualizarlo con el osciloscopio, tal como lo muestra la Figura 5.37.

Señal de datos	0	1	1	1	1	1	1	0	1	0	0	0	0	1	1	0	0	0	1	1	1	1	1	0
Señal codificada	0	+A	-A	+A	-A	+A	-A	0	+A	0	0	0	0	-A	+A	0	0	0	-A	+A	-A	+A	-A	0

Tabla 5.13 Bits de datos y bits codificados con el Código de Línea AMI.

La señal a decodificar que ingresa al circuito interpretador Bipolar – Unipolar y la señal decodificada con el código de línea AMI en el FPGA se observan en la Figura 5.38. La señal decodificada representa los mismos bits que la señal de datos mostrada en la Figura 5.37. Existe un retardo de un tiempo de bit de la señal decodificada con respecto a la señal a decodificar.

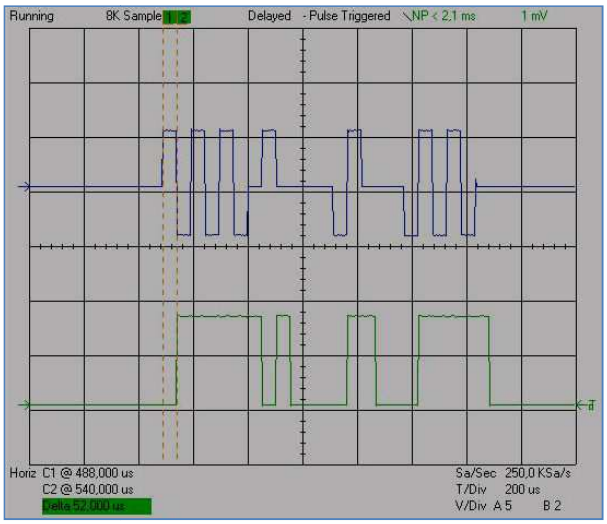


Figura 5.38 Visualización de la Decodificación AMI en VHDL.

5.5.10 HDB3

Para el código de línea HDB3 se realizan dos pruebas de funcionamiento, utilizando los caracteres ASCII de datos “ah” y “papá”, con el objetivo de visualizar bits de violación y relleno, debido a que estas secuencias están representadas por valores binarios que incluyen más de tres estados lógicos 0L consecutivos varias veces, tal como lo muestran la Tabla 5.14 y la Tabla 5.15.

Para los caracteres “ah”, la Figura 5.39 permite visualizar la señal de reloj, la señal a codificar y la señal codificada con el código de línea HDB3 en MATLAB.

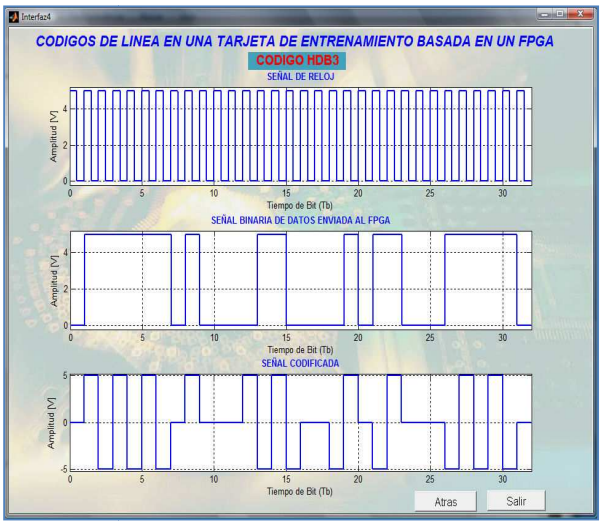


Figura 5.39 Codificación HDB3 de los caracteres “ah” en MATLAB.

En la Figura 5.40 se visualiza en el canal A la señal a codificar, bits correspondientes a “ah”; y en el canal B la señal codificada con el código de línea HDB3 en el FPGA. En VHDL, en la componente de codificación HDB3 se requiere que la señal de salida que representa la señal codificada se retarde 4 tiempos de bit con respecto a la señal a codificar, además del retardo de un tiempo de bit debido a la ejecución del proceso en la componente, razón por la cual se observa que existe un total de 5 tiempos de bit de retardo de la señal codificada con respecto a la señal de datos.

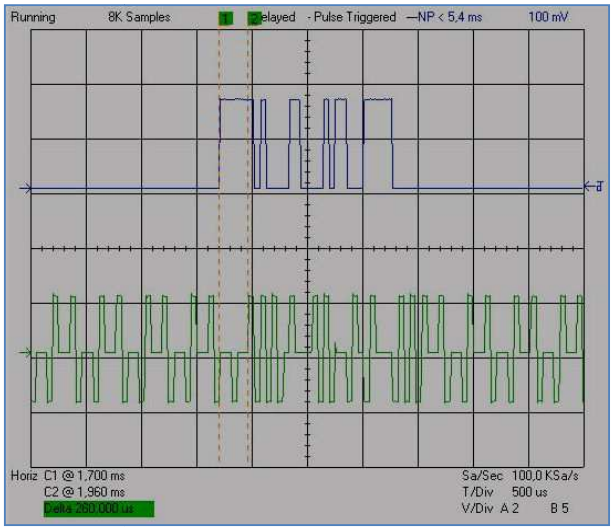


Figura 5.40 Visualización de la Codificación HDB3 de los caracteres “ah” en VHDL.

Los bits de la señal de datos y señal codificada que se muestran en la Figura 5.40, se corroboran en la Tabla 5.14.

Señal de datos	0	1	1	1	1	1	1	0	1	0	0	0	0	1	1	0	0	0	0	1	0	1	1	0	0	0	1	1	1	1	1	0
Señal codificada	0	+A	-A	+A	-A	+A	-A	0	+A	0	0	0	+A	-A	+A	-A	0	0	-A	+A	0	-A	+A	0	0	0	-A	+A	-A	+A	-A	0
													v			r			v													

Tabla 5.14 Bits de datos y bits codificados con el Código de Línea HDB3.

En la figura 5.41 se puede visualizar que la señal decodificada representa los mismos bits que la señal a codificar mostrada en la Figura 5.40, probándose que se realiza de manera correcta la codificación y decodificación. Además se aprecia un retardo de 5 tiempos de bit de la señal decodificada respecto a la señal a decodificar, debido a que la componente de decodificación HDB3 requiere 4

tiempos de bit para obtener la señal decodificada además del retardo de 1 tiempo de bit debido al proceso.

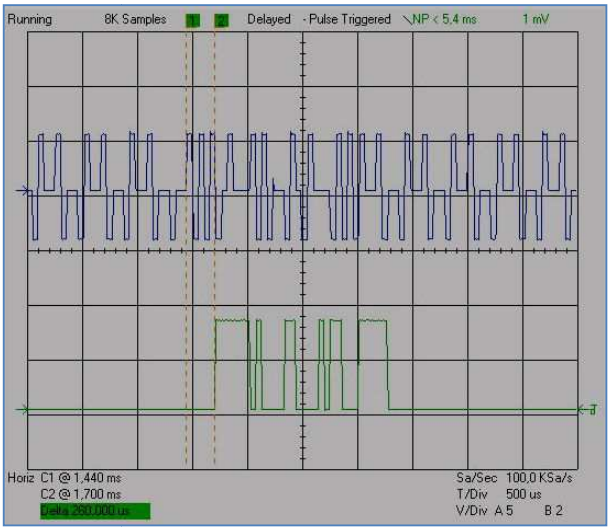


Figura 5.41 Visualización de la Decodificación HDB3 de los caracteres “ah” en VHDL.

Al observar la señal de datos a codificar que se obtiene luego de varios procesos en el FPGA y la señal decodificada que llega al FPGA luego de codificarse, se visualiza que las dos señales representan los mismos bits pero la segunda retardada 10 tiempos de bit respecto a la señal a codificar que es el resultado del retardo de 5 tiempos de bit en la codificación y 5 tiempos de bit en la decodificación, tal como lo muestra la Figura 5.42.

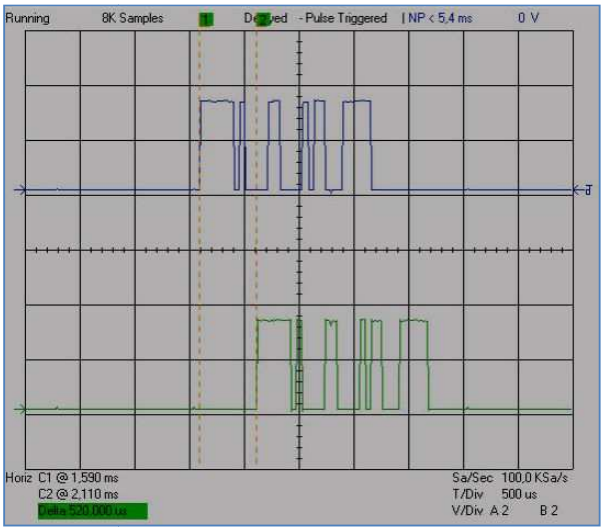


Figura 5.42 Señal de datos a codificar y señal decodificada en el FPGA con el Código de Línea HDB3.

Para los caracteres “papá”; la señal de reloj, la señal a codificar y la señal codificada con el código de línea HDB3 en MATLAB se muestran en la Figura 5.43.

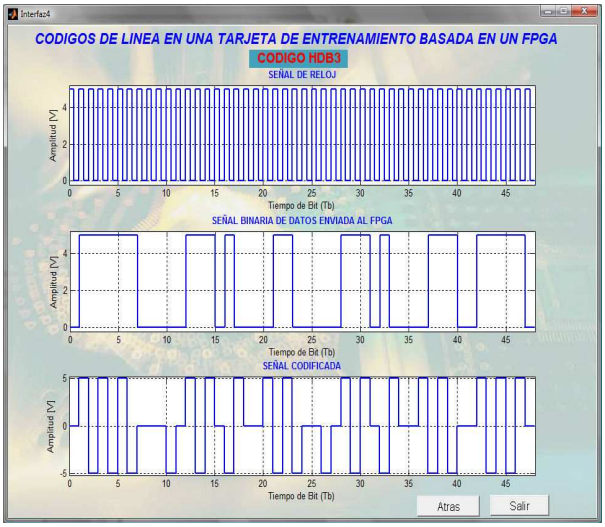


Figura 5.43 Codificación HDB3 de los caracteres “papá” en MATLAB.

El resultado de la codificación en el FPGA con el código de línea HDB3 de los bits correspondientes a “papá” se visualiza en la Figura 5.44; en el canal A la señal a codificar y en el canal B la señal codificada con el correspondiente retardo de 5 tiempos de bit.

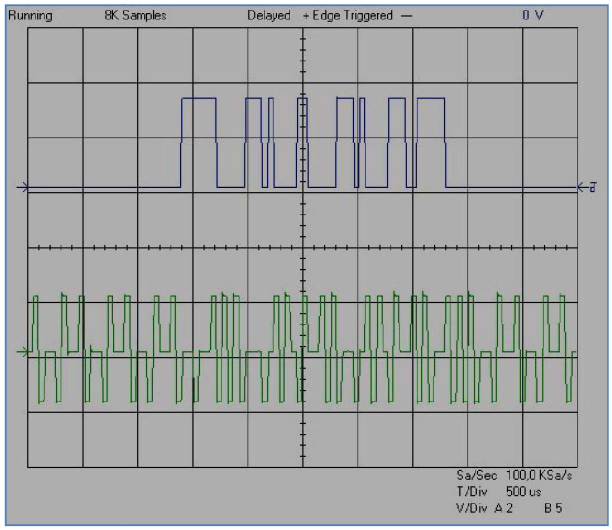


Figura 5.44 Visualización de la Codificación HDB3 de los caracteres “papá” en VHDL.

La Tabla 5.15 muestra los bits de la señal de datos y señal codificada que se observan en la Figura 5.44.

Señal de datos	0 1 1 1 1 1 1 0 0 0 0 0 1 1 1 0 1 0 0 0 0 1 1 0 0 0 0 0 1 1 1 0 1 0 0 0 0 1 1 1 0 0 1 1 1 1 1 0
Señal codificada	0 +A -A +A -A +A -A 0 0 0 v -A 0 +A -A +A 0 -A +A 0 0 +A -A +A -A 0 0 r -A +A 0 0 v +A -A +A -A +A -A 0 0 +A -A +A -A 0 0 +A -A +A -A +A 0 0 +A -A +A -A +A 0

Tabla 5.15 Bits de datos y bits codificados con el Código de Línea HDB3.

La señal a decodificar y la señal decodificada en el FPGA, con el retardo de 5 tiempos de bit, se muestran en la Figura 5.45.

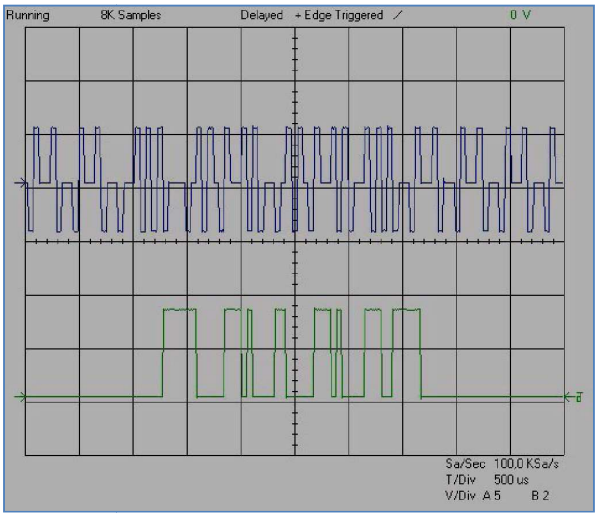


Figura 5.45 Visualización de la Decodificación HDB3 de los caracteres “papá” en VHDL.

5.5.11 MLT3

La codificación MLT3 en MATLAB se presenta en la Figura 5.46.

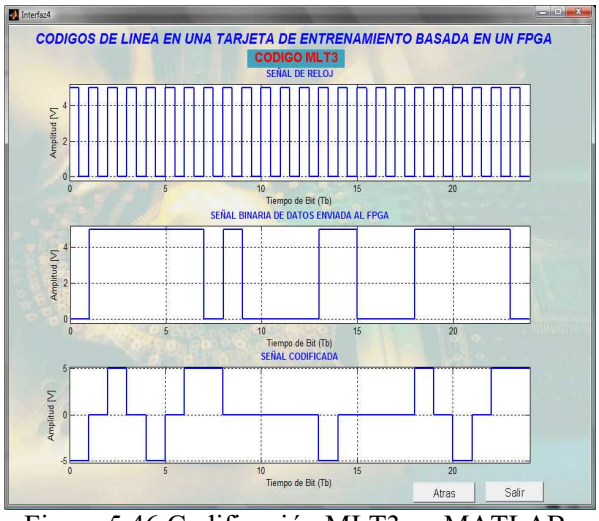


Figura 5.46 Codificación MLT3 en MATLAB.

Este código es el último sujeto a análisis, la Figura 5.47 evidencia la codificación MLT3, en la que se observa la señal a codificar y decodificada. El retardo que toma el proceso de codificación es de 52 µs.

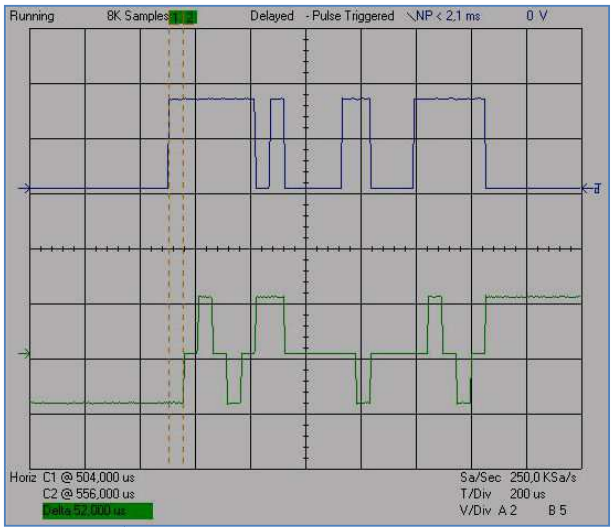


Figura 5.47 Visualización de la Codificación MLT3 en VHDL.

La Tabla 5.16 presenta de una manera más explícita la codificación MLT3 observada en la figura anterior. Hay que mencionar que para este proceso los valores iniciales fueron de 0V y -5V, por lo tanto si llega un 1L, de acuerdo a la regla de codificación, este será igual a 0V.

Señal de datos	0	1	1	1	1	1	1	0	1	0	0	0	0	1	1	0	0	0	1	1	1	1	1	0
Señal codificada	-A	0	+A	0	-A	0	+A	+A	0	0	0	0	0	-A	0	0	0	0	+A	0	-A	0	+A	+A

Tabla 5.16 Bits de datos y bits codificados con el Código de Línea MLT3.

La decodificación se exhibe en la Figura 5.48 se puede apreciar el retardo de un tiempo de bit.

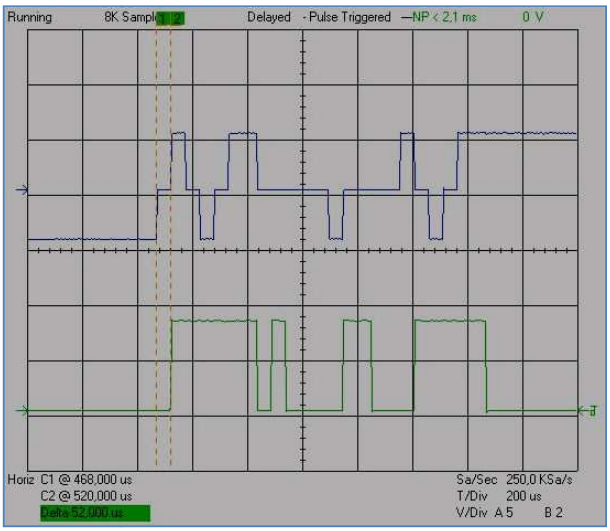


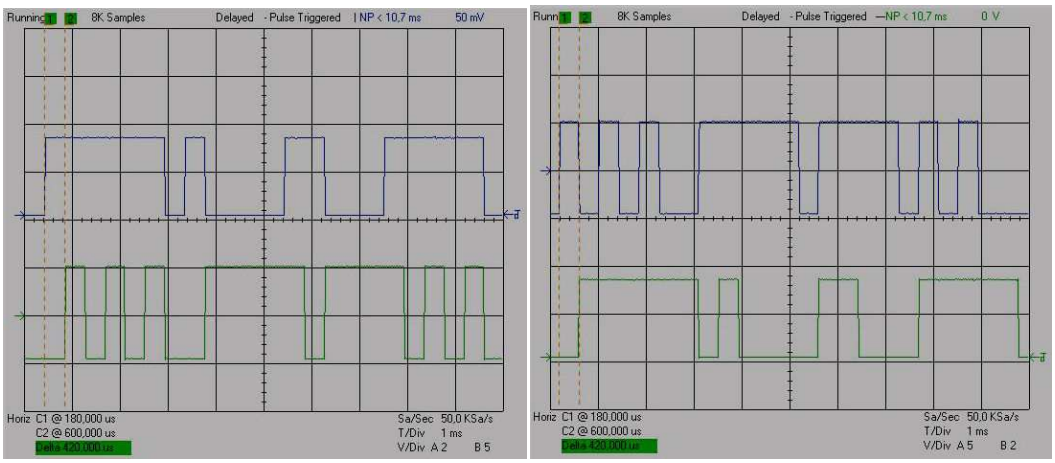
Figura 5.48 Visualización de la Decodificación MLT3 en VHDL.

5.5.12 PRUEBAS DE FUNCIONAMIENTO A TODAS LAS VELOCIDADES

La codificación y decodificación con todos los códigos de línea expuestos se puede realizar a las velocidades 2400, 4800, 9600 y 19200 bps. Sin embargo las pruebas de funcionamiento se han realizado a 19200 bps por ser la condición más crítica, pero en este segmento se realizan pruebas a las tres primeras velocidades descritas con el código de línea Diferencial tipo M para demostrar su comportamiento.

Velocidad de transmisión de 2400 bps

La Figura 5.49 muestra la codificación y decodificación con el Código de Línea Diferencial tipo M cuando se selecciona una velocidad de transmisión de 2400 bps.



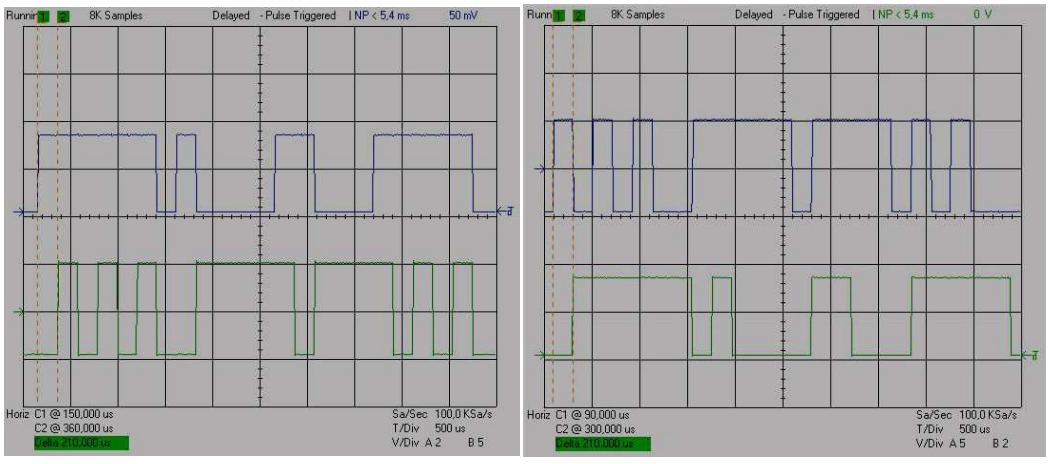
(a) (b)

Figura 5.49 Codificación y decodificación en el FPGA con el código de línea Diferencial tipo M a 2400 bps.

En la Figura 5.49 se puede apreciar que el tiempo de bit es aproximadamente 420 µs. Además en la parte (a) se visualiza que existe un retardo de un tiempo de bit de la señal codificada respecto a la señal a codificar, y en la parte (b) un retardo de un tiempo de bit de la señal decodificada respecto a la señal a decodificar.

Velocidad de transmisión de 4800 bps

La codificación y decodificación con el Código de Línea Diferencial tipo M a una velocidad de transmisión de 4800 bps se muestra en la Figura 5.50.



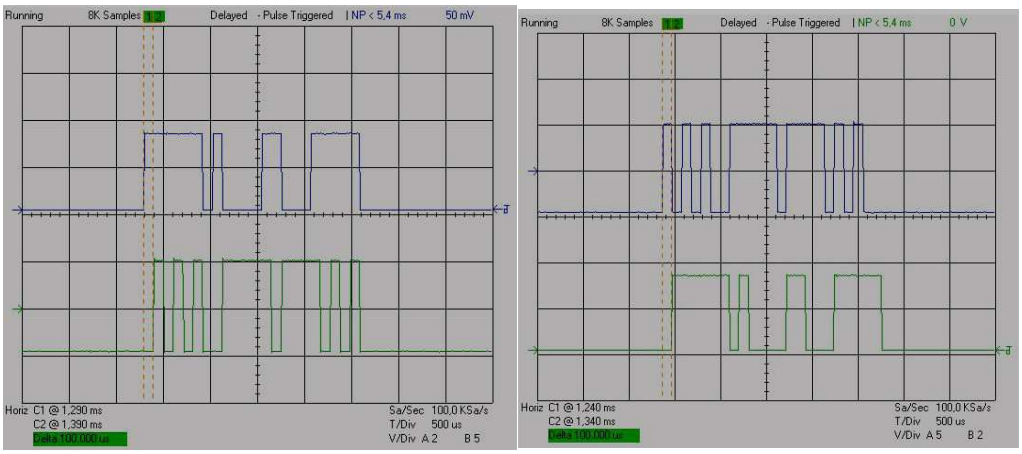
(a) (b)

Figura 5.50 Codificación y decodificación en el FPGA con el código de línea Diferencial tipo M a 4800 bps.

En la Figura 5.50 se aprecia que el tiempo de bit es aproximadamente 208 µs, correspondiente a la velocidad de transmisión de 4800 bps. En la parte (a) se visualiza el retardo de un tiempo de bit de la señal codificada respecto a la señal a codificar, y en la parte (b) el retardo también de un tiempo de bit de la señal decodificada respecto a la señal a decodificar, debido a los procesos.

Velocidad de transmisión de 9600 bps

En la Figura 5.51, la parte (a) muestra la codificación y la parte (b) la decodificación con el Código de Línea Diferencial tipo M a la velocidad de transmisión de 9600 bps.



(a) (b)

Figura 5.51 Codificación y decodificación en el FPGA con el código de línea Diferencial tipo M a 9600 bps.

En la Figura 5.51 se visualiza que el tiempo de bit es aproximadamente 104 µs, además de los respectivos retardos de un tiempo de bit tanto de la señal codificada como de la señal decodificada en las partes (a) y (b) de la figura respectivamente.

5.6 VISUALIZACIÓN DE LAS SEÑALES ENVIADAS DESDE EL FPGA A LA INTERFAZ GRÁFICA

Una vez que llega la señal codificada al FPGA, se realizan los procesos de decodificación mencionados en el capítulo 4, *ensamblaje_bandera_matriz* y *serialización_salida*, para obtener la señal decodificada y lista para ser enviada a la Interfaz Gráfica. Esta señal incluye los bits de inicio, paridad y parada de cada carácter de datos ya que son transmitidos asincrónicamente, como lo muestra la Figura 5.52.

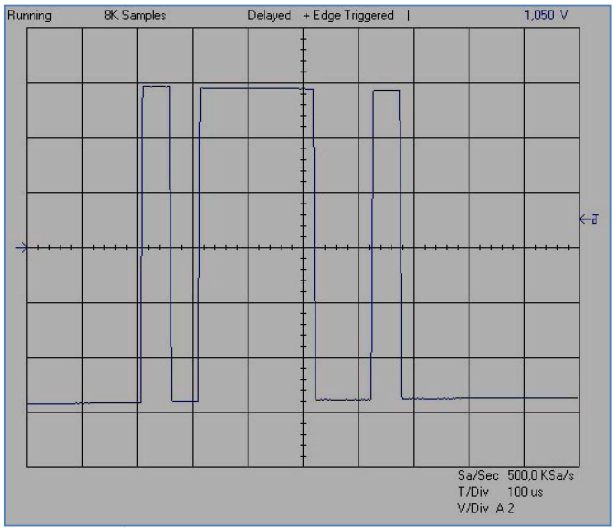


Figura 5.52 Señal de Datos enviada del FPGA a la Interfaz Gráfica.

En la Figura 5.52 se observa que se envía del FPGA a la interfaz gráfica a través del Interfaz RS-232 a 19200 bps el carácter ASCII “a”, cuyo valor hexadecimal es 61H.

En la Interfaz gráfica se interpreta los datos recibidos, sin bits de inicio, paridad y parada como se muestra en la Figura 5.53. Si toda la implementación del proyecto se realiza correctamente la señal recibida desde el FPGA en la Interfaz Gráfica debe ser igual a la señal de datos enviada de la Interfaz Gráfica al FPGA.

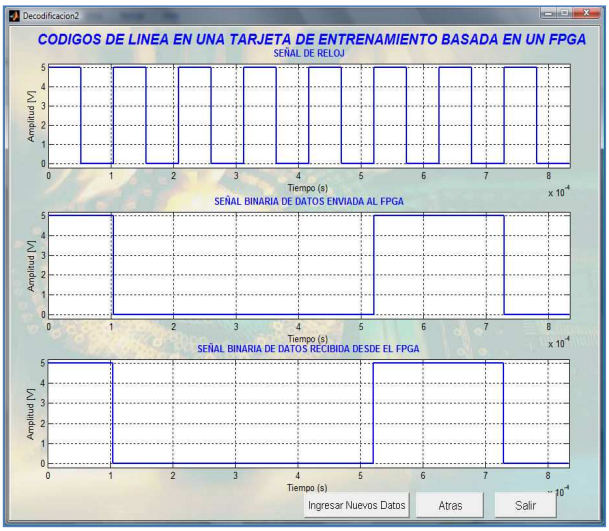


Figura 5.53 Gráfica de los Datos recibidos en la Interfaz Gráfica.

5.7 ANÁLISIS DE TIEMPOS DE PROCESAMIENTO

El análisis de tiempos de procesamiento que se realiza, se divide en dos partes. Primero se analiza el tiempo entre la obtención en el FPGA de la señal a ser codificada y la señal finalmente codificada a la salida del circuito interpretador unipolar – bipolar. Segundo se examina el tiempo entre la señal a ser decodificada a la entrada del circuito interpretador bipolar - unipolar y la obtención de la señal finalmente decodificada en el FPGA.

NRZ, Diferencial tipo M, Diferencial tipo S, AMI y MLT3

En cuanto a tiempo de procesamiento en la implementación, entre la señal de datos a codificar que se obtiene en el FPGA luego de varios procesos y la señal codificada existe un tiempo de bit de retardo, mientras que entre la señal codificada que llega al FPGA para ser decodificada y la señal decodificada se tiene un retardo de otro tiempo de bit. Por esta razón entre la señal de datos y la señal decodificada se tiene una diferencia de dos tiempos de bit. Esto se debe a que en cada proceso de cada componente la señal de salida se obtiene con un retardo de un ciclo de *RELOJ_OUT* (señal que se encuentra en la lista de sensibilidad) con respecto a la señal de entrada.

RZ al 50%, Manchester y Manchester Diferencial

En la implementación, entre la señal de datos a codificar y la señal codificada existe medio tiempo de bit de retardo, mientras que entre la señal codificada que llega al FPGA para ser decodificada y la señal decodificada se tiene un retardo de otro medio tiempo de bit. Por esta razón entre la señal de datos y la señal decodificada se tiene una diferencia de un tiempo de bit. Hay que tener en cuenta que en estos códigos de línea hay transición a mitad de tiempo de bit, lo que implica un retardo en cada proceso de un ciclo de *RELOJ_OUT_Doble*, o sea la mitad de ciclo de la señal *RELOJ_OUT.*

4B5B

En la Figura 5.17 se puede observar que el retardo total entre la señal a codificar y señal decodificada para éste código es de 892 µs, lo que se traduce en aproximadamente 17 tiempos de bit de la señal *RELOJ_OUT*. Este retardo incluye: los tiempos de procesamiento que concierne a la codificación cuando almacena los ocho datos en un buffer para luego ser codificados, los tiempos de procesamiento en la decodificación cuando almacena los 10 datos codificados para luego ser decodificados. Además toma aproximadamente un tiempo de bit en los procesos involucrados en la codificación y decodificación.

CMI

Entre la señal de datos a codificar que se obtiene en el FPGA y la señal codificada existe medio tiempo de bit de retardo, mientras que entre la señal codificada que llega al FPGA para ser decodificada y la señal decodificada se tiene un retardo de un tiempo de bit. Esto se debe a que en el proceso de codificación se utiliza la señal de reloj *RELOJ_OUT_Doble* y en el proceso de decodificación se utiliza la señal *RELOJ_OUT*.
Por esta razón entre la señal de datos y la señal decodificada se tiene una diferencia de tiempo de un tiempo y medio de bit.

HDB3

En cuanto a tiempo de procesamiento, entre la señal de datos a codificar que se obtiene en el FPGA luego de varios procesos y la señal codificada existen cinco tiempos de bit de retardo, mientras que entre la señal codificada que llega al FPGA para ser decodificada y la señal decodificada se tiene un retardo de otros cinco tiempo de bit. Por esta razón entre la señal de datos y la señal decodificada se tiene una diferencia de diez tiempos de bit. Los cinco bits de retardo en el proceso de codificación y decodificación, se resumen en cuatro tiempos de bit que requiere el código HDB3 para obtener las señales: codificada y decodificada. Además un tiempo de bit por cada proceso implicado.

Cabe mencionar que el retardo que origina el circuito externo es de aproximadamente 20 µs, lo que no es muy representativo. Además la corta distancia en la que sucede la comunicación entre el computador y la tarjeta de entrenamiento, hace que el retardo que experimentan las señales no presente mayor variación para todas las velocidades usadas en el proyecto, tal como se probó con el código Diferencial Tipo M.

En la Tabla 5.17 se presenta un resumen de los tiempos de procesamiento totales que le toma a la señal de datos en la codificación y decodificación de cada uno de los códigos de línea analizados.

Código de Línea	Tiempo de Procesamiento a 19200 bps	Tiempo de Procesamiento a otras velocidades
NRZ	104 µs	2 tiempos de bit
RZ 50%	52 µs	1 tiempo de bit
4B5B	892 µs	Aprox. 17 tiempos de bit
Diferencial Tipo M	104 µs	2 tiempos de bit
Diferencial Tipo S	104 µs	2 tiempos de bit
Manchester	52 µs	1 tiempo de bit
Manchester Diferencial	52 µs	1 tiempo de bit
CMI	78 µs	1, 5 tiempos de bit
AMI	104 µs	2 tiempos de bit
HDB3	520 µs	10 tiempos de bit
MLT3	104 µs	2 tiempos de bit

Tabla 5.17 Tiempos de procesamiento totales de los Códigos de Línea.

CAPÍTULO VI

CONCLUSIONES Y RECOMENDACIONES

6.1 CONCLUSIONES

Los algoritmos de codificación de cada uno de los códigos de línea implementados, se desarrollaron mediante el software MATLAB para la visualización de las respectivas simulaciones, lo cual proporciona una base sólida al momento de evaluar el hardware, en este caso, el FPGA.

La interfaz gráfica desarrollada en MATLAB, permite la interacción entre el computador y la tarjeta de entrenamiento mediante el puerto serial RS-232, esta se traduce en el envío y recepción de datos simultáneamente, ya que la comunicación es de tipo full dúplex a las velocidades permitidas por la interfaz.

En el proyecto, el software MATLAB mediante la herramienta GUI Design Environment permite realizar una Interfaz Gráfica que toma los scripts de simulación de codificación, de todos los códigos de línea implementados. Además, admite que el usuario disfrute de un entorno amigable, ingresar datos, seleccionar el código de línea a implementarse en el FPGA, observar los datos binarios recibidos, y presentar información adicional.

El DFS (Sintetizador de Frecuencia) del DCM, es una herramienta que cumple un papel importante en la elaboración de este proyecto, ya que permitió obtener la señal de reloj básica a partir de la cual se desarrollaron otras señales de reloj ocupadas en la transmisión, recepción, codificación, y decodificación de datos en la tarjeta de entrenamiento.

La programación en VHDL debe ser óptima, es decir, no debe contener señales y/o variables que pueden ser omitidas, las listas de sensibilidad deben contener

todas las señales que al cambiar de estado permiten que se ejecute un proceso, las señales y variables de los tipos entero y natural deben declararse en rangos numéricos en los cuales se desenvuelven, y al utilizar segmentos secuenciales saber las formas de utilización de condicionales. Caso contrario al momento de sintetizar el XST no permitirá obtener resultados esperados, se ocuparán segmentos (slices) del FPGA de forma excesiva incluso llegando a desbordar su capacidad, o a pesar de tener una buena sintaxis se producirán errores en el proceso de síntesis.

Los scripts de MATLAB y los códigos fuente con el lenguaje de programación VHDL son totalmente diferentes. Los scripts de MATLAB son secuenciales, en su lugar VHDL es inherentemente concurrente. La concurrencia permite que todos los procesos se ejecuten al mismo tiempo, es decir ocurran en paralelo en el FPGA, mientras el código fuente de MATLAB y la mayoría de lenguajes de programación ejecutan un conjunto de instrucciones en una manera secuencial. VHDL tiene segmentos de código que pueden ser secuenciales internamente y concurrentes entre sí. Entonces una ventaja de usar FPGAs es que puede ejecutar operaciones en paralelo y de manera secuencial.

En los resultados obtenidos se visualiza que para cada código de línea, desde que se obtiene la señal a codificar en el FPGA hasta que se la codifica, al igual desde que ingresa la señal codificada hasta que se la decodifica, el tiempo de procesamiento se mantiene en función de tiempos de bit, es decir no se tiene un tiempo de procesamiento fijo sino depende de la velocidad de transmisión escogida por el usuario. Esto se debe a que los procesos en VHDL, en los que se realiza la codificación o decodificación con cada código de línea, se ejecutan cada vez que cambia de estado la señal de reloj que se encuentra en la lista de sensibilidad y se procesa en función del último bit de la señal de datos que ingresó a la componente.

Se concluye que los resultados obtenidos en el hardware son satisfactorios, ya que las gráficas obtenidas en el osciloscopio de la codificación son iguales a las conseguidas mediante MATLAB. La señal de datos y la señal decodificada, al

final de todos los procesos implicados, son similares. Para este logro, el diseño del circuito externo jugó un papel muy importante, ya que permitió interpretar los códigos polares y bipolares, los cuales no se podían visualizar directamente desde los pines de la Spartan-3E Starter Kit Board debido a las restricciones que ésta presenta.

6.2 RECOMENDACIONES

Se recomienda realizar proyectos de sistemas digitales implementados en VHDL con FPGAs, incluso se podría elaborar un laboratorio de circuitos digitales con FPGAs; ya que hoy en día en otras regiones del mundo se realizan desde circuitos simples con puertas lógicas, pasando por circuitos combinacionales y contadores, hasta circuitos digitales complejos a través de lógica programable. Además el uso de FPGAs implicaría tener un hardware reconfigurable y un bajo costo en relación al número de compuertas lógicas que se pueden implementar.

Este proyecto de titulación se basa en el uso de la tarjeta de entrenamiento Spartan 3E Starter Kit Board y el lenguaje de descripción de hardware VHDL. Sin embargo para definir el comportamiento del FPGA existen otros métodos, los cuales se recomienda investigar a través de otros proyectos, entre ellos: el lenguaje de descripción de hardware Verilog, la herramienta System Generator de Simulink que en forma gráfica puede desarrollar bloques de código, los IP Cores en especial soft cores para FPGAs, la herramienta EDK (Embedded Development Kit) que ayuda en el uso de periféricos de las tarjetas de entrenamiento. En cuanto a hardware se sugiere investigar la familia de FPGAs Virtex 5 debido a que tiene la misma arquitectura pero atributos diferentes que la familia Spartan 3E; además se encuentran embebidos en tarjetas de desarrollo que poseen más periféricos.

Al alcanzar las investigaciones propuestas en el enunciado anterior, se recomienda plantearse una meta mayor que es el realizar proyectos complejos en

telecomunicaciones con FPGAs, utilizando todas las herramientas descritas y acoplarlas con el objetivo de optimizar tiempo y recursos.

El presente proyecto de titulación está implementado con sólo una tarjeta de entrenamiento, donde el FPGA hace las funciones de codificación y decodificación. Como una ampliación de este proyecto, se podría proponer la ejecución de la implementación de los códigos de línea en base al esquema tradicional de comunicación de datos, con dos tarjetas de entrenamiento, en conjunto con dos computadores que constituyan un transmisor y receptor conectados por un canal de transmisión, y además hacer las pruebas a largas distancias, para evaluar de mejor manera el rendimiento de los FPGAs.

REFERENCIAS BIBLIOGRÁFICAS

LIBROS

- ✓ MEYER-BAESE, Uwe; Digital Signal Processing with Field Programmable Gate Arrays. Springer-Verlag Berlin Heidelberg. Alemania. 2001

- ✓ PEDRONI, Volnei A.; Circuit Design with VHDL. 1a Edición. 2004

- ✓ STALLINGS, William; Data and Computer Communications. 5ª Edición.

- ✓ WOODS, Roger; McAllister, John; LIGHTBODY, Gaye; YI, Ying; FPGA-based Implementation of Signal Processing Systems. 1a Edición. 2008

- ✓ XILINX, ISE Design Suite 10.1 Release Notes and Installation Guide. 2008.

- ✓ XILINX, Spartan-3 Generation FPGA User Guide

- ✓ XILINX, Spartan-3E FPGA Family

- ✓ XILINX, Spartan-3E Starter Kit Board User Guide

ARTÍCULOS

- ✓ BROWN, Stephen; ROSE, Jonathan; Architecture of FPGAs and CPLDs: A Tutorial.

DIRECCIONES ELECTRÓNICAS

- ✓ http://www.opencores.org/

- ✓ http://www.fpga4fun.com/

- ✓ http://www.isa.cie.uva.es/proyectos/codec/marco1.html
- ✓ http://www.itescam.edu.mx/principal/sylabus/fpdb/recursos/r67142.ppt
- ✓ http://samkerr.wordpress.com/2009/09/21/first-steps-with-an-fpga/#more-207

SCRIPTS PARA LA SIMULACIÓN DE CÓDIGOS DE LÍNEA EN MATLAB

NRZ

```
function NRZ(senal_a_enviar)

%Numero de bits de la señal de datos
num_bits=length(senal_a_enviar);
%Vector tiempo para graficar
tb=0:1/1000:num_bits-1/1000;

%Vectores que vamos a concatenar
senal_de_reloj = [];
senal_de_datos=[];
senal_codificada=[];

%Amplitud
a=5;

%Para inicializar while
n=1;

while n<=num_bits;

    % Señal de Reloj
    reloj(n,:)=[ones(1,500) zeros(1,500)];

    if senal_a_enviar(n) == 0 %Si el bit de datos es 0L
        datos(n,:)=[zeros(1,1000)];
        codificado(n,:)= [zeros(1,1000)]; %Bit codificado es 0L
    else %Si el bit de datos es 1L
        datos(n,:)=[ones(1,1000)];
        codificado(n,:) = [ones(1,1000)]; %Bit codificado es 1L
    end

    %Concatenar señales de reloj, datos y codificada
    senal_de_reloj=cat(2,senal_de_reloj,reloj(n,:));
    senal_de_datos=cat(2,senal_de_datos,datos(n,:));
    senal_codificada=cat(2,senal_codificada,codificado(n,:));

    n=n+1;

end

%Graficar las señales
figure

subplot(3,1,1)
plot(tb,a*senal_de_reloj,'LineWidth',1.5)
grid on
axis([0 num_bits -0.2 1.04*a]) %Límites x1 x2 y1 y2
```

```
title('SEÑAL DE RELOJ','color','b','Fontweight','Bold');
xlabel('Tiempo de Bit (Tb)');
ylabel('Amplitud [V]');

subplot(3,1,2)
plot(tb,a*senal_de_datos,'LineWidth',1.5)
grid on
axis([0 num_bits -.2 1.04*a]); %Límites x1 x2 y1 y2
title('SEÑAL BINARIA DE DATOS ENVIADA AL
FPGA','color','b','Fontweight','Bold');
xlabel('Tiempo de Bit (Tb)');
ylabel('Amplitud [V]');

subplot(3,1,3)
plot(tb,a*senal_codificada,'LineWidth',1.5)
grid on
axis([0 num_bits -1.04*a 1.04*a]); %Límites x1 x2 y1 y2
title('SEÑAL CODIFICADA','color','b','Fontweight','Bold');
xlabel('Tiempo de Bit (Tb)');
ylabel('Amplitud [V]');
```

RZ al 50%

```
function RZ(senal_a_enviar)

%Número de bits de la señal de datos
num_bits=length(senal_a_enviar);
%Vector tiempo para graficar
tb=0:1/1000:num_bits-1/1000;

%Vectores que vamos a concatenar
senal_de_reloj = [];
senal_de_datos=[];
senal_codificada=[];

%Amplitud
a=5;

%Para inicializar while
n=1;

while n<=num_bits;

    % Señal de Reloj
    reloj(n,:)=[ones(1,500) zeros(1,500)];
    if senal_a_enviar(n) == 0 %Si el bit a codificar es 0L
        datos(n,:)=[zeros(1,1000)];
        codificado(n,:)= [zeros(1,1000)]; %Bit codificado es 0L
    else %Si el bit a codificar es 1L
        datos(n,:)=[ones(1,1000)];
        codificado(n,:) = [ones(1,500) zeros(1,500)]; %BitCodificado es 1L al
50%
    end
```

```
    %Concatenar señales de reloj, datos y codificada
    senal_de_reloj=cat(2,senal_de_reloj,reloj(n,:));
    senal_de_datos=cat(2,senal_de_datos,datos(n,:));
    senal_codificada=cat(2,senal_codificada,codificado(n,:));

    n=n+1;

end

%Graficar las señales
figure

subplot(3,1,1)
plot(tb,a*senal_de_reloj,'LineWidth',1.5)
grid on
axis([0 num_bits -0.2 1.04*a]) %Limites x1 x2 y1 y2
title('SEÑAL DE RELOJ','color','b','Fontweight','Bold');
xlabel('Tiempo de Bit (Tb)');
ylabel('Amplitud [V]');

subplot(3,1,2)
plot(tb,a*senal_de_datos,'LineWidth',1.5)
grid on
axis([0 num_bits -.2 1.04*a]); %Limites x1 x2 y1 y2
title('SEÑAL BINARIA DE DATOS ENVIADA AL
FPGA','color','b','Fontweight','Bold');
xlabel('Tiempo de Bit (Tb)');
ylabel('Amplitud [V]');

subplot(3,1,3)
plot(tb,a*senal_codificada,'LineWidth',1.5)
grid on
axis([0 num_bits -1.04*a 1.04*a]); %Limites x1 x2 y1 y2
title('SEÑAL CODIFICADA','color','b','Fontweight','Bold');
xlabel('Tiempo de Bit (Tb)');
ylabel('Amplitud [V]');
```

4B5B

```
function C4B5B(senal_a_enviar)

%Numero de bits de la señal de datos
num_bits=length(senal_a_enviar);

%Amplitud
a=5;

%La senal de datos debe ser múltiplo de 4 bits, caso contrario se rellena
%con 0L hasta que el número de bits sea múltiplo de 4
%Mientras el residuo de num_bits/4 no sea igual a 0
while (rem (num_bits,4) ~= 0)
    senal_a_enviar(num_bits+1)=0;
    num_bits=num_bits+1;
end
```

```
%Vector tiempo para graficar señal de datos
tb=0:1/1000:num_bits-1/1000;
%Vector tiempo para graficar la señal codificada 4B5B
tb4b5b=0:1/1000:(5*num_bits/4)-1/1000;

%Señales que vamos a concatenar
senal_de_reloj=[];
senal_de_datos=[];
senal_codificada=[];

%Para inicializar el vector suma
m = 1;

%En el vector suma se almacenan los valores en decimal del 0 al 15
%de cada 4 bits que deben ser codificados
for i = 1:4:num_bits
    suma(m) = 8*senal_a_enviar(i+3) + 4*senal_a_enviar(i+2) +
    2*senal_a_enviar(i+1) + senal_a_enviar(i);
    m = m + 1;
end

%Para identificar el primer elemento del vector codigo
j = 1;

%Lazo para obtener los bits codificados en el vector codigo
for i = 1:length(suma)
    if suma(i) == 0
        codigo(j:j+4) = [0 1 1 1 1];
    end
    if suma(i) == 1
        codigo(j:j+4) = [1 0 0 1 0];
    end
    if suma(i) == 2
        codigo(j:j+4) = [0 0 1 0 1];
    end
    if suma(i) == 3
        codigo(j:j+4) = [1 0 1 0 1];
    end
    if suma(i) == 4
        codigo(j:j+4) = [0 1 0 1 0];
    end
    if suma(i) == 5
        codigo(j:j+4) = [1 1 0 1 0];
    end
    if suma(i) == 6
        codigo(j:j+4) = [0 1 1 1 0];
    end
    if suma(i) == 7
        codigo(j:j+4) = [1 1 1 1 0];
    end
    if suma(i) == 8
        codigo(j:j+4) = [0 1 0 0 1];
    end
    if suma(i) == 9
        codigo(j:j+4) = [1 1 0 0 1];
    end
    if suma(i) == 10
        codigo(j:j+4) = [0 1 1 0 1];
    end
```

```
    if suma(i) == 11
        codigo(j:j+4) = [1 1 1 0 1];
    end
    if suma(i) == 12
        codigo(j:j+4) = [0 1 0 1 1];
    end
    if suma(i) == 13
        codigo(j:j+4) = [1 1 0 1 1];
    end
    if suma(i) == 14
        codigo(j:j+4) = [0 0 1 1 1];
    end
    if suma(i) == 15
        codigo(j:j+4) = [1 0 1 1 1];
    end
    j=j+5;
end


%Para inicializar while
n = 1;

while n<=num_bits;

    % Señal de Reloj
    reloj(n,:)=[ones(1,500) zeros(1,500)];

    if senal_a_enviar(n) == 0 %Si el bit de datos es 0L
        datos(n,:)=[zeros(1,1000)];
    else %Si el bit de datos es 1L
        datos(n,:) = [ones(1,1000)];
    end

    %Concatenar señales de reloj y datos
    senal_de_reloj=cat(2,senal_de_reloj,reloj(n,:));
    senal_de_datos=cat(2,senal_de_datos,datos(n,:));

    n=n+1;

end

%Para inicializar while
p = 1;

%Mientras m sea menor o igual que el número de símbolos codificados
while p<=(5*num_bits/4);

    if codigo(p) == 0 %Si el bit codificado es 0L
        codificado(p,:)= [zeros(1,1000)];
    else %Si el bit codificado es 1L
        codificado(p,:) = [ones(1,1000)];
    end

    %Concatenar señal codificada
    senal_codificada=cat(2,senal_codificada,codificado(p,:));

    p=p+1;
```

```
end

%Graficar las señales
figure

subplot(3,1,1)
plot(tb,a*senal_de_reloj,'LineWidth',1.5)
grid on
axis([0 num_bits -0.2 1.04*a]) %Limites x1 x2 y1 y2
title('SEÑAL DE RELOJ','color','b','Fontweight','Bold');
xlabel('Tiempo de Bit (Tb)');
ylabel('Amplitud [V]');

subplot(3,1,2)
plot(tb,a*senal_de_datos,'LineWidth',1.5)
grid on
axis([0 num_bits -.2 1.04*a]); %Limites en x1 x2 y1 y2
title('SEÑAL BINARIA DE DATOS ENVIADA AL
FPGA','color','b','Fontweight','Bold');
xlabel('Tiempo de Bit (Tb)');
ylabel('Amplitud [V]');

subplot(3,1,3)
plot(tb4b5b,a*senal_codificada,'LineWidth',1.5)
grid on
axis([0 (5*num_bits/4) -1.04*a 1.04*a]); %Limites en x1 x2 y1 y2
title('SEÑAL CODIFICADA','color','b','Fontweight','Bold');
xlabel('Tiempo de Bit (Tb)');
ylabel('Amplitud [V]');
```

Diferencial tipo M

```
function Diferencial_M(senal_a_enviar)

%Número de bits de la señal de datos
num_bits=length(senal_a_enviar);
% Vector tiempo para graficar
tb=0:1/1000:num_bits-1/1000;

% Vectores que vamos a concatenar
senal_de_reloj=[];
senal_de_datos=[];
senal_codificada=[];

%Amplitud
a=5;

%Para inicializar while
n=1;

while n<=num_bits;

    % Señal de Reloj
```

```
    reloj(n,:)=[ones(1,500) zeros(1,500)];

    if senal_a_enviar(n) == 0 %Si el bit a codificar es 0L
        datos(n,:)=[zeros(1,1000)];
        if n == 1 %Si el bit a codificar es el primero
            %El bit codificado es -1L porque referencia es -1L
            codificado(n,:) = [-ones(1,1000)];
        else %Si el bit a codificar no es el primero
            codificado(n,:) = codificado(n-1,:); %Se mantiene el nivel
        end
    else %Si el bit a codificar es 1L
        datos(n,:)=[ones(1,1000)];
        if n == 1 %Si el bit a codificar es el primero
            %El bit codificado es 1L porque referencia es -1L
            codificado(n,:)= [ones(1,1000)];
        else %Si el bit a codificar no es el primero
            codificado(n,:)=-codificado(n-1,:); %Cambia el nivel
        end
    end

    %Concatenar señales de reloj, datos y codificada
    senal_de_reloj=cat(2,senal_de_reloj,reloj(n,:));
    senal_de_datos=cat(2,senal_de_datos,datos(n,:));
    senal_codificada=cat(2,senal_codificada,codificado(n,:));

    n=n+1;

end

%Graficar las señales
figure

subplot(3,1,1)
plot(tb,a*senal_de_reloj,'LineWidth',1.5)
grid on
axis([0 num_bits -0.2 1.04*a]) %Limites en x1 x2 y1 y2
title('SEÑAL DE RELOJ','color','b','Fontweight','Bold');
xlabel('Tiempo de Bit (Tb)');
ylabel('Amplitud [V]');

subplot(3,1,2)
plot(tb,a*senal_de_datos,'LineWidth',1.5)
grid on
axis([0 num_bits -.2 1.04*a]); %Limites en x1 x2 y1 y2
title('SEÑAL BINARIA DE DATOS ENVIADA AL
FPGA','color','b','Fontweight','Bold');
xlabel('Tiempo de Bit (Tb)');
ylabel('Amplitud [V]');

subplot(3,1,3)
plot(tb,a*senal_codificada,'LineWidth',1.5)
grid on
axis([0 num_bits -1.04*a 1.04*a]); %Limites en x1 x2 y1 y2
title('SEÑAL CODIFICADA','color','b','Fontweight','Bold');
xlabel('Tiempo de Bit (Tb)');
ylabel('Amplitud [V]');
```

Diferencial tipo S

```
function Diferencial_S(senal_a_enviar)

%Número de bits de la señal de datos
num_bits=length(senal_a_enviar);
% Vector tiempo para graficar
tb=0:1/1000:num_bits-1/1000;

%Vectores que se van a concatenar
senal_de_reloj=[];
senal_de_datos=[];
senal_codificada=[];

%Amplitud
a=5;

%Para inicializar while
n=1;

while n<=num_bits;

    % Generación de la Señal de Reloj
    reloj(n,:)=[ones(1,500) zeros(1,500)];

    if senal_a_enviar(n) == 0 %Si el bit a codificar es 0L
        datos(n,:)=[zeros(1,1000)];
        if n == 1 %Si el bit a codificar es el primero
            %El bit codificado es 1L porque referencia es -1L
            codificado(n,:)= [ones(1,1000)];
        else %Si el bit a codificar no es el primero
            codificado(n,:)=-codificado(n-1,:); %Cambia el nivel
        end
    else %Si el bit a codificar es 1L
        datos(n,:)=[ones(1,1000)];
        if n == 1 %Si el bit a codificar es el primero
            %El bit codificado es -1L porque referencia es -1L
            codificado(n,:) = [-ones(1,1000)];
        else
            codificado(n,:) = codificado(n-1,:); %Se mantiene el nivel
        end
    end

    %Concatenar señales de reloj, datos y codificada
    senal_de_reloj=cat(2,senal_de_reloj,reloj(n,:));
    senal_de_datos=cat(2,senal_de_datos,datos(n,:));
    senal_codificada=cat(2,senal_codificada,codificado(n,:));

    n=n+1;

end

%Graficar las señales
figure

subplot(3,1,1)
```

```
plot(tb,a*senal_de_reloj,'LineWidth',1.5)
grid on
axis([0 num_bits -0.2 1.04*a]) %Limites en x1 x2 y1 y2
title('SEÑAL DE RELOJ','color','b','Fontweight','Bold');
xlabel('Tiempo de Bit (Tb)');
ylabel('Amplitud [V]');

subplot(3,1,2)
plot(tb,a*senal_de_datos,'LineWidth',1.5)
grid on
axis([0 num_bits -.2 1.04*a]); %Limites en x1 x2 y1 y2
title('SEÑAL BINARIA DE DATOS ENVIADA AL
FPGA','color','b','Fontweight','Bold');
xlabel('Tiempo de Bit (Tb)');
ylabel('Amplitud [V]');

subplot(3,1,3)
plot(tb,a*senal_codificada,'LineWidth',1.5)
grid on
axis([0 num_bits -1.04*a 1.04*a]); %Limites en x1 x2 y1 y2
title('SEÑAL CODIFICADA','color','b','Fontweight','Bold');
xlabel('Tiempo de Bit (Tb)');
ylabel('Amplitud [V]');
```

Manchester

```
function MANCHESTER(senal_a_enviar)

%Numero de bits de la señal de datos
num_bits=length(senal_a_enviar);
%Vector tiempo para graficar
tb=0:1/1000:num_bits-1/1000;

%Vectores que se van a concatenar
senal_de_reloj=[];
senal_de_datos=[];
senal_codificada=[];

%Amplitud
a = 5;

%Para inicializar while
n=1;

while n<=num_bits;

    % Generación de la Señal de Reloj
    reloj(n,:)=[ones(1,500) zeros(1,500)];

    if senal_a_enviar(n) == 0 %Si el bit a codificar es 0L
        datos(n,:)=[zeros(1,1000)];
        codificado(n,:)= [-ones(1,500) ones(1,500)]; %Transición positiva
    else %Si el bit a codificar es 1L
        datos(n,:)=[ones(1,1000)];
```

```
        codificado(n,:) = [ones(1,500) -ones(1,500)]; %Transición
negativa
    end

    %Concatenar señales de reloj, datos y codificada
    senal_de_reloj = cat (2,senal_de_reloj,reloj(n,:));
    senal_de_datos=cat(2,senal_de_datos,datos(n,:));
    senal_codificada=cat(2,senal_codificada,codificado(n,:));

    n=n+1;

end

%Graficar las señales
figure

subplot(3,1,1)
plot(tb,a*senal_de_reloj,'LineWidth',1.5)
grid on
axis([0 num_bits -0.2 1.04*a]) %Límites en x1 x2 y1 y2
title('SEÑAL DE RELOJ','color','b','Fontweight','Bold');
xlabel('Tiempo de Bit (Tb)');
ylabel('Amplitud [V]');

subplot(3,1,2)
plot(tb,a*senal_de_datos,'LineWidth',1.5)
grid on
axis([0 num_bits -.2 1.04*a]); %Límites en x1 x2 y1 y2
title('SEÑAL BINARIA DE DATOS ENVIADA AL
FPGA','color','b','Fontweight','Bold');
xlabel('Tiempo de Bit (Tb)');
ylabel('Amplitud [V]');

subplot(3,1,3)
plot(tb,a*senal_codificada,'LineWidth',1.5)
grid on
axis([0 num_bits -1.04*a 1.04*a]); %Límites en x1 x2 y1 y2
title('SEÑAL CODIFICADA','color','b','Fontweight','Bold');
xlabel('Tiempo de Bit (Tb)');
ylabel('Amplitud [V]');
```

Manchester Diferencial

```
function Manchester_Diferencial (senal_a_enviar)

%Numero de bits de la señal de datos
num_bits=length(senal_a_enviar);
%Vector tiempo para graficar
tb=0:1/1000:num_bits-1/1000;

%Vectores que se van a concatenar
senal_de_reloj=[];
senal_de_datos=[];
senal_codificada=[];
```

```
%Amplitud
a=5;

%Para inicializar while
n=1;

while n<=num_bits;

    % Generación de la Señal de Reloj
    reloj(n,:)=[ones(1,500) zeros(1,500)];

    if senal_a_enviar(n) == 0 %Si el bit a codificar es 0L
        datos(n,:)=[zeros(1,1000)];
        if n == 1 %Si el bit a codificar es el primero
           %Transición positiva porque referencia es Transición positiva
            codificado(n,:)= [-ones(1,500) ones(1,500)];
        else %Si el bit a codificar no es el primero
            %Se mantiene la Transición
            codificado(n,:) = codificado(n-1,:);
        end
    else % Si el bit a codificar es 1L
        datos(n,:)=[ones(1,1000)];
        if n == 1 %Si el bit a codificar es el primero
           %Transición negativa porque referencia es Transición positiva
            codificado(n,:) = [ones(1,500) -ones(1,500)];
        else
            %Cambia la Transición
            codificado(n,:) =  -1*codificado(n-1,:);
        end
    end

    %Concatenar señales de reloj, datos y codificada
    senal_de_reloj = cat (2,senal_de_reloj,reloj(n,:));
    senal_de_datos=cat(2,senal_de_datos,datos(n,:));
    senal_codificada=cat(2,senal_codificada,codificado(n,:));

    n=n+1;

end

%Graficar las señales
figure

subplot(3,1,1)
plot(tb,a*senal_de_reloj,'LineWidth',1.5)
grid on
axis([0 num_bits -0.2 1.04*a]) %Límites en x1 x2 y1 y2
title('SEÑAL DE RELOJ','color','b','Fontweight','Bold');
xlabel('Tiempo de Bit (Tb)');
ylabel('Amplitud [V]');

subplot(3,1,2)
plot(tb,a*senal_de_datos,'LineWidth',1.5)
grid on
axis([0 num_bits -.2 1.04*a]); %Límites en x1 x2 y1 y2
title('SEÑAL BINARIA DE DATOS ENVIADA AL
FPGA','color','b','Fontweight','Bold');
```

```
xlabel('Tiempo de Bit (Tb)');
ylabel('Amplitud [V]');

subplot(3,1,3)
plot(tb,a*senal_codificada,'LineWidth',1.5)
grid on
axis([0 num_bits -1.04*a 1.04*a]); %Límites en x1 x2 y1 y2
title('SEÑAL CODIFICADA','color','b','Fontweight','Bold');
xlabel('Tiempo de Bit (Tb)');
ylabel('Amplitud [V]');
```

CMI

```
function CMI(senal_a_enviar)

%Numero de bits de la señal de datos
num_bits=length(senal_a_enviar);
%Vector tiempo para graficar
tb=0:1/1000:num_bits-1/1000;

%Vectores que vamos a concatenar
senal_de_reloj=[];
senal_de_datos=[];
senal_codificada=[];

%Amplitud
a = 5;
%Voltaje de Referencia
codificado_1L_anterior=-1;

%Para inicializar while
n=1;

while n<=num_bits;

    % Señal de Reloj
    reloj(n,:)=[ones(1,500) zeros(1,500)];

    if senal_a_enviar(n) == 0 %Si el bit a codificar es 0L
        datos(n,:)=[zeros(1,1000)];
        codificado(n,:) = [-ones(1,500) ones(1,500)]; %Transicion
positiva
    else %Si el bit a codificar es 1L
        datos(n,:)=[ones(1,1000)];
        %Bit codificado es el inverso del bit codificado con 1L anterior
        codificado_1L_anterior=-codificado_1L_anterior;
        codificado(n,:)=[codificado_1L_anterior*ones(1,1000)];
    end

    %Concatenar señales de reloj, datos y codificada
    senal_de_reloj=cat(2,senal_de_reloj,reloj(n,:));
    senal_de_datos=cat(2,senal_de_datos,datos(n,:));
    senal_codificada=cat(2,senal_codificada,codificado(n,:));

    n=n+1;
```

```
end

%Graficar las señales
figure
subplot(3,1,1)
plot(tb,a*senal_de_reloj,'LineWidth',1.5)
grid on
axis([0 num_bits -0.2 1.04*a]) %Limites en x1 x2 y1 y2
title('SEÑAL DE RELOJ','color','b','Fontweight','Bold');
xlabel('Tiempo de Bit (Tb)');
ylabel('Amplitud [V]');

subplot(3,1,2)
plot(tb,a*senal_de_datos,'LineWidth',1.5)
grid on
axis([0 num_bits -.2 1.04*a]); %Limites en x1 x2 y1 y2
title('SEÑAL BINARIA DE DATOS ENVIADA AL
FPGA','color','b','Fontweight','Bold');
xlabel('Tiempo de Bit (Tb)');
ylabel('Amplitud [V]');

subplot(3,1,3)
plot(tb,a*senal_codificada,'LineWidth',1.5)
grid on
axis([0 num_bits -1.04*a 1.04*a]); %Limites en x1 x2 y1 y2
title('SEÑAL CODIFICADA','color','b','Fontweight','Bold');
xlabel('Tiempo de Bit (Tb)');
ylabel('Amplitud [V]');
```

AMI

```
function AMI(senal_a_enviar)

%Numero de bits de la señal de datos
num_bits=length(senal_a_enviar);
%Vector tiempo para graficar
tb=0:1/1000:num_bits-1/1000;

%Vectores que vamos a concatenar
senal_de_reloj=[];
senal_de_datos=[];
senal_codificada=[];

%Amplitud
a=5;
%Nivel de Referencia es -1L
codificado_1L_anterior=-1;

%Para inicializar while
n=1;

while n<=num_bits;

    % Señal de Reloj
```

```
    reloj(n,:)=[ones(1,500) zeros(1,500)];

    if senal_a_enviar(n) == 0 %Si el bit a codificar es 0L
        datos(n,:)=[zeros(1,1000)];
        codificado(n,:)= [zeros(1,1000)]; %Bit codificado es 0L
    else %Si el bit a codificar es 1L
        datos(n,:)=[ones(1,1000)];
        %Bit codificado es el inverso del bit codificado con 1L anterior
        codificado_1L_anterior=-codificado_1L_anterior;
        codificado(n,:) = [codificado_1L_anterior*ones(1,1000)];
    end

    %Concatenar señales de reloj, datos y codificada
    senal_de_reloj = cat(2,senal_de_reloj,reloj(n,:));
    senal_de_datos=cat(2,senal_de_datos,datos(n,:));
    senal_codificada=cat(2,senal_codificada,codificado(n,:));

    n=n+1;

end

%Graficar señales
figure

subplot(3,1,1)
plot(tb,a*senal_de_reloj,'LineWidth',1.5)
grid on
axis([0 num_bits -0.2 1.04*a]) %Limites en x1 x2 y1 y2
title('SEÑAL DE RELOJ','color','b','Fontweight','Bold');
xlabel('Tiempo de Bit (Tb)');
ylabel('Amplitud [V]');

subplot(3,1,2)
plot(tb,a*senal_de_datos,'LineWidth',1.5)
grid on
axis([0 num_bits -.2 1.04*a]); %Limites en x1 x2 y1 y2
title('SEÑAL BINARIA DE DATOS ENVIADA AL
FPGA','color','b','Fontweight','Bold');
xlabel('Tiempo de Bit (Tb)');
ylabel('Amplitud [V]');

subplot(3,1,3)
plot(tb,a*senal_codificada,'LineWidth',1.5)
grid on
axis([0 num_bits -1.04*a 1.04*a]); %Limites en x1 x2 y1 y2
title('SEÑAL CODIFICADA','color','b','Fontweight','Bold');
xlabel('Tiempo de Bit (Tb)');
ylabel('Amplitud [V]');
```

HDB3

```
function HDB3(senal_a_enviar)

%Número de bits de la señal de datos
num_bits=length(senal_a_enviar);
%Vector tiempo para graficar
tb=0:1/1000:num_bits - 1/1000;

%Amplitud
a=5;
%Nivel de referencia -1L
Uno_codificado_anterior = -1;
%Violacion inicial
ultima_violacion = 0;

%Para inicializar while
n=1;

while n<=num_bits;

    if senal_a_enviar(n)==1 %Si el bit a codificar es 1L
        % Señal de Reloj y datos
        reloj(n,:)=[ones(1,500) zeros(1,500)];
        datos(n,:)=[ones(1,1000)];
        if Uno_codificado_anterior == 1
            %El bit codificado es -1L
            codificado(n,:)= -[ones(1,1000)];
            Uno_codificado_anterior = -1;
        else
            %El bit codificado es +1L
            codificado(n,:)= [ones(1,1000)];
            Uno_codificado_anterior = 1;
        end
        n = n +1;

    else %Si el bit a codificar es 0L
        %El número de 0L = 1
        ceros = 1;
        if (n+3) <= num_bits %Si el 0L actual no es uno de los 3 últimos bits
            for j = 1: 3
                if senal_a_enviar(n+j) == 0 %Si el siguiente bit de datos es
0L
                    ceros = ceros + 1; %Se suma 1 al número de 0L
                else %Si el siguiente bit de datos no es 0L
                    break %Sale del lazo sin sumar más ceros
                end
            end
        end
        if ceros < 4 %Si el número de 0L es menor que cuatro
            for k = 0:ceros-1
                reloj(n+k,:)=[ones(1,500) zeros(1,500)];
                %Bits de datos y bits codificados son 0L
                datos(n+k,:)=[zeros(1,1000)];
                codificado(n+k,:)= [zeros(1,1000)];
            end
            n = n + ceros;
        else %Si el número de 0L es igual a cuatro
            % Siempre la primera palabra va a ser 01111110
            %Señal de reloj
            for l = 0:3
```

```
                reloj(n+1,:)=[ones(1,500) zeros(1,500)];
            end
            %Si hay violacion y relleno
            if ultima_violacion == Uno_codificado_anterior
                datos(n,:)=[zeros(1,1000)];
                %El primero de los cuatro bits codificados es el inverso
                %del último bit de violación
                codificado(n,:)=-ultima_violacion.*[ones(1,1000)];
                for k = 1:2 %Codificando segundo bit y tercer bit de
cuatro
                    %Bits de datos y bits codificados son 0L
                    datos(n+k,:)=[zeros(1,1000)];
                    codificado(n+k,:)= [zeros(1,1000)];
                end
                datos(n+3,:)=[zeros(1,1000)];
                %Codificando el cuarto bit de cuatro
                codificado(n+3,:)= codificado(n,:);
                %El bit codificado es el inverso del bit codificado como
                %1L o -1L anterior y de la violación anterior
                Uno_codificado_anterior = -Uno_codificado_anterior;
                ultima_violacion =-ultima_violacion;
            else %Si solo hay violacion
                %Señal de datos
                datos(n+3,:)=[zeros(1,1000)];
                %Señal codificada
                for k = 0:2 %El 1, 2 y 3 de la señal codificada son 0L
                    datos(n+k,:)=[zeros(1,1000)];
                    codificado(n+k,:)= [zeros(1,1000)];
                end
                %El cuarto bit codificado es el mismo que el último bit
                %codificado como 1L o -1L (Uno_codificado_anterior)

codificado(n+3,:)=Uno_codificado_anterior.*[ones(1,1000)];
                %Si el bit codificado anteriormente como 1L o -1L fue 1L
                if Uno_codificado_anterior == 1
                    ultima_violacion =1;
                else %Si el bit codificado anteriormente como 1L o -1L fue -1L
                    ultima_violacion = -1;
                end
            end
            n = n+4;
        end
    end

end

%Para inicializar while
i=1;

% Vectores que vamos a concatenar
senal_de_reloj=[];
senal_de_datos=[];
senal_codificada=[];

%Concatenar señales de reloj, datos y codificada
while i<=num_bits;
    senal_de_reloj=cat(2,senal_de_reloj,reloj(i,:));
    senal_de_datos=cat(2,senal_de_datos,datos(i,:));
    senal_codificada=cat(2,senal_codificada,codificado(i,:));
```

```
    i=i+1;
end

%Graficar las señales
figure

subplot(3,1,1)
plot(tb,a*senal_de_reloj,'LineWidth',1.5)
grid on
axis([0 num_bits -0.2 1.04*a]) %Límites x1 x2 y1 y2
title('SEÑAL DE RELOJ','color','b','Fontweight','Bold');
xlabel('Tiempo de Bit (Tb)');
ylabel('Amplitud [V]');

subplot(3,1,2)
plot(tb,a*senal_de_datos,'LineWidth',1.5)
grid on
axis([0 num_bits -0.2 1.04*a]); %Limites x1 x2 y1 y2
title('SEÑAL BINARIA DE DATOS ENVIADA AL
FPGA','color','b','Fontweight','Bold');
xlabel('Tiempo de Bit (Tb)');
ylabel('Amplitud [V]');

subplot(3,1,3)
plot(tb,a*senal_codificada,'LineWidth',1.5)
grid on
axis([0 num_bits -1.04*a 1.04*a]); %Limites x1 x2 y1 y2
title('SEÑAL CODIFICADA','color','b','Fontweight','Bold');
xlabel('Tiempo de Bit (Tb)');
ylabel('Amplitud [V]');
```

MLT3

```
function MLT3 (senal_a_enviar)

%Numero de bits de la señal de datos
num_bits=length(senal_a_enviar);
%Vector tiempo para graficar
tb=0:1/1000:num_bits-1/1000;

%Vectores que se van a concatenar
senal_de_reloj=[];
senal_de_datos=[];
senal_codificada=[];

%Amplitud
a=5;

%Para inicializar while
n=1;

%Condiciones iniciales para la codificación
variable1=-1;
variable2=0;
```

```
while n<=num_bits;

    % Generación de la Señal de Reloj
    reloj(n,:)=[ones(1,500) zeros(1,500)];

    if senal_a_enviar(n) == 1 %Si el bit a codificar es 1L
        datos(n,:)=[ones(1,1000)];
        if variable1==1 %Si el penúltimo bit codificado fue 1L
            if variable2==0 %Si el último bit codificado fue 0L
                %Bit codificado es -1L
                codificado(n,:)= [-ones(1,1000)];
                variable1 = 0;
                variable2 =-1; %Último bit codificado es -1L
            end
        elseif variable1==-1 %Si el penúltimo bit codificado fue -1L
            if variable2==0 %Si el último bit codificado fue 0L
                %Bit codificado es 1L
                codificado(n,:)= [ones(1,1000)];
                variable1 = 0;
                variable2 = 1; %Último bit codificado es 1L
            end
        elseif variable1==0 %Si el penúltimo bit codificado fue 0L
            if variable2==1 %Si el último bit codificado fue 1L
                %Bit codificado es 0L
                codificado(n,:)= [zeros(1,1000)];
                variable1 = 1;
                variable2 = 0; %Último bit codificado es 0L
            elseif variable2==-1 %Si el último bit codificado fue -1L
                %Bit codificado es 0L
                codificado(n,:)= [zeros(1,1000)];
                variable1 = -1;
                variable2 = 0; %Último bit codificado es 0L
            end
        end
    else %Si el bit a codificar es 0L
        datos(n,:)=[zeros(1,1000)];
        if n == 1 %Si el bit a codificar es el primero
            %El bit codificado es 0L porque empieza en -1L, 0L
            codificado(n,:) = [zeros(1,1000)];
        else %Si el bit a codificar no es el primero
            codificado(n,:) = codificado(n-1,:); %Se mantiene el nivel
        end
    end

    %Concatenar señales de reloj, datos y codificada
    senal_de_reloj = cat(2,senal_de_reloj,reloj(n,:));
    senal_de_datos=cat(2,senal_de_datos,datos(n,:));
    senal_codificada=cat(2,senal_codificada,codificado(n,:));

    n=n+1;

end

%Graficar las señales
figure

subplot(3,1,1)
```

```
plot(tb,a*senal_de_reloj,'LineWidth',1.5)
grid on
axis([0 num_bits -0.2 1.04*a]) %Límites x1 x2 y1 y2
title('SEÑAL DE RELOJ','color','b','Fontweight','Bold');
xlabel('Tiempo de Bit (Tb)');
ylabel('Amplitud [V]');

subplot(3,1,2)
plot(tb,a*senal_de_datos,'LineWidth',1.5)
grid on
axis([0 num_bits -.2 1.04*a]); %Límites en x1 x2 y1 y2
title('SEÑAL BINARIA DE DATOS ENVIADA AL
FPGA','color','b','Fontweight','Bold');
xlabel('Tiempo de Bit (Tb)');
ylabel('Amplitud [V]');

subplot(3,1,3)
plot(tb,a*senal_codificada,'LineWidth',1.5)
grid on
axis([0 num_bits -1.04*a 1.04*a]); %Límites en x1 x2 y1 y2
title('SEÑAL CODIFICADA','color','b','Fontweight','Bold');
xlabel('Tiempo de Bit (Tb)');
ylabel('Amplitud [V]');
```

CÓDIGO VHDL PARA LA IMPLEMENTACIÓN DE CÓDIGOS DE LÍNEA EN EL FPGA

Componente reloj_38400

```
----------------------------------------------------------------------------------
-- Universidad: Escuela Politécnica Nacional
-- Creadores: Daniel Velásquez y Viviana Checa
-- Director: Robin Alvarez PhD.
--
-- Module Name:    reloj_38400 - Behavioral
-- Create Date:    02/01/2010
-- Quito - Ecuador
----------------------------------------------------------------------------------

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

```
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity reloj_38400 is

	port (Reloj_Entrada: in std_logic;
		Reloj_Salida: buffer std_logic);

end reloj_38400;

architecture Behavioral of reloj_38400 is

--Señales internas de la componente
signal contador: std_logic_vector(9 downto 0):="0000000000";
signal dividir_para: natural range 0 to 625 :=625;
signal reloj: std_logic:='0';

begin

Reloj_38400 : process(Reloj_Entrada)

begin

	--Si ocurre una transición positiva de RELOJ_Entrada
	if rising_edge(Reloj_Entrada) then
		if contador = (dividir_para - 1) then --Si contador = 624
			contador <= (others => '0');  --Se inicializa contador
			reloj<= not reloj; --reloj se invierte
		else --contador de 0 a 623
			contador <= contador + 1;
		end if;
	end if;

end process Reloj_38400;

Reloj_Salida <= reloj;

end Behavioral;
```

Componente reloj_48000

```
----------------------------------------------------------------------------------
-- Universidad: Escuela Politécnica Nacional
-- Creadores: Daniel Velásquez y Viviana Checa
-- Director: Robin Alvarez PhD.
--
-- Module Name:    reloj_48000 - Behavioral
-- Create Date:    09/10/2010
-- Quito - Ecuador
----------------------------------------------------------------------------------

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

entity reloj_48000 is

	port (Reloj_Entrada: in std_logic;
		Reloj_Salida: buffer std_logic);
```

```
end reloj_48000;

architecture Behavioral of reloj_48000 is

--Señales internas de la componente
signal contador: std_logic_vector(9 downto 0):="0000000000";
signal reloj: std_logic:='0';

begin

Reloj_48000 : process(Reloj_Entrada)

begin

	--Si ocurre una transición positiva de RELOJ_Entrada
	if rising_edge(Reloj_Entrada) then
		if contador = (500 - 1) then --Si contador = 499
			contador <= (others => '0'); --Se inicializa contador
			reloj <= not reloj; --reloj se invierte
		else --contador de 0 a 498
			contador <= contador + 1;
		end if;
	end if;

end process Reloj_48000;

Reloj_Salida <= reloj;

end Behavioral;
```

Componente NRZ_codificacion

```
----------------------------------------------------------------------------------
-- Universidad: Escuela Politécnica Nacional
-- Creadores: Daniel Velásquez y Viviana Checa
-- Director: Robin Alvarez PhD.
--
-- Module Name:    NRZ_codificacion - Behavioral
-- Create Date:    15:10:46 07/10/2010
-- Quito - Ecuador
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity NRZ_codificacion is

	--Señales externas de la componente

	port (RELOJ_OUT: in std_logic;
		senal_a_codificar: in std_logic;
		salida1: buffer std_logic;
		salida2: buffer std_logic);

end NRZ_codificacion;

architecture Behavioral of NRZ_codificacion is

begin

nrz_cod : process (RELOJ_OUT)

--Variables que almacenan los valores lógicos de las salidas
```

```
--Valor inicial 0V
variable qin_1: std_logic := '0';
variable qin_2: std_logic := '0';

begin

	--Si ocurre una transición positiva de RELOJ_OUT
	if rising_edge (RELOJ_OUT) then
		if (senal_a_codificar = '1') then --Si senal a codificar es 1L
			--Senal codificada es 5V
			qin_1 := '1';
			qin_2 := '0';
		else --Si senal a codificar es 0L
			--Senal codificada es 0V
			qin_1 := '0';
			qin_2 := '0';
		end if;
	end if;

	--Las salidas toman los valores lógicos de las variables
	salida1 <= qin_1;
	salida2 <= qin_2;

end process nrz_cod;

end Behavioral;
```

Componente RZ_codificacion

```
----------------------------------------------------------------------------------
-- Universidad: Escuela Politécnica Nacional
-- Creadores: Daniel Velásquez y Viviana Checa
-- Director: Robin Alvarez PhD.
--
-- Module Name:    RZ_codificacion - Behavioral
-- Create Date:    15:10:46 07/10/2010
-- Quito - Ecuador
----------------------------------------------------------------------------------

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

entity RZ_codificacion is

	--Señales externas de la componente

	port (RELOJ_OUT_Doble: in std_logic;
		senal_a_codificar: in std_logic;
		salida1: buffer std_logic;
		salida2: buffer std_logic);

end RZ_codificacion;

architecture Behavioral of RZ_codificacion is

begin

--Codificación RZ al 50%

rz_cod: process (RELOJ_OUT_Doble)

--Variables utilizadas en este proceso
```

```
--La combinación de las variables qin_1 y qin_2
--permiten obtener los valores de -5V, 0V y +5V
--Si qin_1=0 y qin_2=0, van a representar 0V
--Si qin_1=1 y qin_2=0, van a representar +5V
--Si qin_1=0 y qin_2=1, van a representar -5V

--Variables que almacenan los valores lógicos de las salidas
--Valor inicial 0V
variable qin_1: std_logic := '0';
variable qin_2: std_logic := '0';
--Variable que identifica la transición (positiva o negativa) de RELOJ_OUT
--Valor inicial identifica transición positiva
variable contar: std_logic_vector(1 downto 0) :="01";

begin

	--Para este proceso es necesario utilizar la señal RELOJ_OUT_DOble,
	--ya que el algoritmo de codificación RZ al 50% requiere
	--una transición a la mitad de bit cuando se codifica un 1L

	--Si ocurre una transición positiva de RELOJ_OUT_Doble
	if rising_edge(RELOJ_OUT_Doble) then
		if (contar = 1) then --Transicion positiva de RELOJ_OUT
			--Si el bit a codificar es 1L
			if (senal_a_codificar = '1') then
				--Senal codificada es 5V
				qin_1 := '1';
				qin_2 := '0';
			else--Si el bit a codificar es 0L
				--Senal codificada es 0V
				qin_1 := '0';
				qin_2 := '0';
			end if;
			contar := contar + 1;
		else --Transicion negativa de RELOJ_OUT
			--Senal codificada es 0V
			qin_1 := '0';
			qin_2 := '0';
			contar := "01";
		end if;
	end if;

	--Las salidas toman los valores lógicos de las variables qin_1 y qin_2
	salida1 <= qin_1;
	salida2 <= qin_2;

end process rz_cod;

end Behavioral;
```

Componente c4B5B_codificacion

```
----------------------------------------------------------------------------------
-- Universidad: Escuela Politécnica Nacional
-- Creadores: Daniel Velásquez y Viviana Checa
-- Director: Robin Alvarez PhD.
--
-- Module Name:    C4B5B_codificacion - Behavioral
-- Create Date:    15:10:46 07/10/2010
-- Quito - Ecuador
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
```

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

entity cod_4B5B is

	--Señales externas  de la componente

	Port (RELOJ_OUT: in std_logic;
		RELOJ_OUT_4B5B: in std_logic;
		rst : in std_logic;
		senal_a_codificar: in std_logic;
		salida1: buffer std_logic;
		salida2: buffer std_logic);

end cod_4B5B;

architecture Behavioral of cod_4B5B is

--Señal interna de la componente
signal codift: std_logic_vector (9 downto 0);

--La codificación 4B5B implica dos procesos
--El primero de ellos c4B5B, hace la codificación
--en terminos de vectores de 4 a 5 bits.
--El segundo proceso da paso a la serialización de los
--datos para la visualización

begin

--En el primer porceso se utiliza la señal de RELOJ_OUT
--a la cual viene los datos desde el programa principal

c4B5B: process (RELOJ_OUT)

--Variables utilizadas en este proceso
variable dato: std_logic_vector (7 DOWNTO 0);
variable vector1: std_logic_vector (3 DOWNTO 0);
variable vector2: std_logic_vector (7 DOWNTO 4);
variable count: integer range 0 to 7 := 1;
variable     identifica: std_logic := '0';
variable datos_1: std_logic_vector (3 DOWNTO 0);
variable datos_2: std_logic_vector (3 DOWNTO 0);
variable reg:  std_logic :='0';
variable inicio:  std_logic :='0';
variable codif1:  std_logic_vector (4 DOWNTO 0);
variable codif2:  std_logic_vector (4 DOWNTO 0);

begin

	if (rst = '1')then
		-- La señal rst permite resetear ciertas variables implicadas en el
		-- proceso, para dar paso a una nueva codificación y decodificación
		count:= 1;
		inicio:='0';
		reg:='0';
		identifica:= '0';

	--Si ocurre una transición positiva de RELOJ_OUT
	elsif rising_edge (RELOJ_OUT) then

		if (inicio = '0' and senal_a_codificar = '1') then
			--El delimitador de inicio de datos es siempre 0111110,
			--en base a esto, se identifica en la señal a codificar
			--el primer 1L.
			--Si ha llegado un 1L a la senal_a_codificar, eso quiere
			--decir que el primer bit a codificar es un 0L,
			--debido a la bandera de inicio de datos
```

```
        reg := '1';
        inicio := '1';
        dato(0) := '0';
end if;

if (reg = '1') then

        --El vector dato se llena con los 7  bits restantes de la
        --senal_a_codificar a partir de count = 1 (al inicio)
        dato(count) := senal_a_codificar;

        IF (count < 7) THEN --Cuenta de 0 a 6
                count := count + 1;
                identifica := '0';--Todavia no se puede codificar
        ELSE -- si count = 7
                --quiere decir que en el vector dato se llenó con 8 bits
                --Se encera la variable count para dar paso al
                --siguiente caracter de información
                count := 0;
                identifica := '1';--dato está listo para la codificación
        END IF;

end if;

if (identifica = '1') then--Iniciar de la codificación
        --Para no causar confusión con los ceros que anteceden a los
        --datos, se codifica una secuencia de 8 ceros por 10 ceros
        if (dato="00000000")then

                codift<="0000000000";

        else--Si los datos no son 8 0Ls
                --Se extrae 4 primeros bits del vector dato en vector1(i)
                FOR i IN 0 to 3 LOOP
                        vector1(i):=dato(i);
                end loop;
                --La información de vector1 se pasa a datos_1
                datos_1:= vector1;

                --Se extrae 4 ultimos bits del vector dato en vector2(i)
                FOR i IN 4 to 7 LOOP
                        vector2(i):=dato(i);
                end loop;
                --La información de vector2 se pasa a datos_2
                datos_2:= vector2;

                --Se hace la codificación de los cuatro primeros
                --bits que constituye datos_1, en base
                --a la tabla 4B5B descrita a continuación
                CASE datos_1 IS
                        --TABLA 4B5B
                        WHEN "0000" => codif1:="11110";--0
                        WHEN "0001" => codif1:="01001";--1
                        WHEN "0010" => codif1:="10100";--2
                        WHEN "0011" => codif1:="10101";--3
                        WHEN "0100" => codif1:="01010";--4
                        WHEN "0101" => codif1:="01011";--5
                        WHEN "0110" => codif1:="01110";--6
                        WHEN "0111" => codif1:="01111";--7
                        WHEN "1000" => codif1:="10010";--8
                        WHEN "1001" => codif1:="10011";--9
                        WHEN "1010" => codif1:="10110";--A
                        WHEN "1011" => codif1:="10111";--B
                        WHEN "1100" => codif1:="11010";--C
                        WHEN "1101" => codif1:="11011";--D
                        WHEN "1110" => codif1:="11100";--E
                        WHEN "1111" => codif1:="11101";--F
```

```
                              WHEN OTHERS =>codif1:="00000";

                        END CASE;

                        --Se hace la codificación de los cuatro últimos bits
                        --que constituye datos_2, en base
                        --a la tabla 4B5B descrita a continuación
                        CASE datos_2 IS
                              --TABLA 4B5B
                              WHEN "0000" => codif2:="11110";--0
                              WHEN "0001" => codif2:="01001";--1
                              WHEN "0010" => codif2:="10100";--2
                              WHEN "0011" => codif2:="10101";--3
                              WHEN "0100" => codif2:="01010";--4
                              WHEN "0101" => codif2:="01011";--5
                              WHEN "0110" => codif2:="01110";--6
                              WHEN "0111" => codif2:="01111";--7
                              WHEN "1000" => codif2:="10010";--8
                              WHEN "1001" => codif2:="10011";--9
                              WHEN "1010" => codif2:="10110";--A
                              WHEN "1011" => codif2:="10111";--B
                              WHEN "1100" => codif2:="11010";--C
                              WHEN "1101" => codif2:="11011";--D
                              WHEN "1110" => codif2:="11100";--E
                              WHEN "1111" => codif2:="11101";--F

                              WHEN OTHERS =>codif2:="00000";

                        END CASE;
                        --La señal codift va a estar constituida por los
                        --valores codif1 y codif2, los mismos
                        --que son los valores coficados
                        codift(4 downto 0) <= codif1(4 downto 0);
                        codift(9 downto 5) <= codif2(4 downto 0);

                  end if;

            end if;

      end if;

end process c4B5B;

--El segundo proceso utiliza la señal de RELOJ_OUT_4B5B, para
--serializar la señal codift, que contiene la codificación

serial: process (RELOJ_OUT_4B5B)

--Variables utilizadas en este proceso
variable m : natural range 0 to 9:=0;
variable inicio_serial:  std_logic :='0';
variable registro:  std_logic :='0';

begin

      if (rst = '1')then
            -- La señal rst permite resetear ciertas variables implicadas,
            --para dar paso a una nueva serialización de los datos codificados
            m:= 0;
            inicio_serial:='0';
            registro :='0';
      --Si ocurre una transición positiva de RELOJ_OUT_4B5B
      elsif rising_edge (RELOJ_OUT_4B5B) then
            --La codificación correspondiente a el delimitador
            --de inicio 01111110 es 0011111110, por lo que
            --se toma en cuenta el primer bit 1L que esta en la posicion 3
```

```
            --desde la izquierda, codift(2).
            --Las dos condiciones indican que se ha identificado el id de inicio
            if (inicio_serial = '0' and codift(2) = '1') then
                  registro := '1'; --Iniciar la serialización
                  inicio_serial := '1'; --Para que no regrese a esta condición
            end if;
            if (registro = '1') then--Inicio de la serialización
                  --salida1 va a contener la información de codift
                  salida1<=codift(m);
                  if (m=9)then--Si han llegado los 10 primeros bits
                        m:=0;--Se da paso a los siguientes 10 bits codificados
                  else--De 0 a 8
                        m:=m+1;
                  end if;
            end if;
      end if;

      --Al ser código unipolar, la señal salida2 siempre va a ser 0L
      salida2<='0';

end process serial;

end Behavioral;
```

Componente DIF_M_codificacion

```
----------------------------------------------------------------------------------
-- Universidad: Escuela Politécnica Nacional
-- Creadores: Daniel Velásquez y Viviana Checa
-- Director: Robin Alvarez PhD.
--
-- Module Name:    DIF_M_codificacion - Behavioral
-- Create Date:    15:10:46 07/10/2010
-- Quito - Ecuador
----------------------------------------------------------------------------------

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

entity DIF_M_codificacion is

      --Señales externas de la componente

      Port (RELOJ_OUT: in std_logic;
                  senal_a_codificar: in std_logic;
                  salida1: buffer std_logic;
                  salida2: buffer std_logic);

end DIF_M_codificacion;

architecture Behavioral of DIF_M_codificacion is

begin

Diferencial_M_cod : process (RELOJ_OUT) --nrzi

--Variables que almacenan los valores lógicos de las salidas
--Valor inicial 0V, en realidad -5V
variable qin_1 : std_logic := '0';
variable qin_2 : std_logic := '0';

begin
```

```
	if rising_edge (RELOJ_OUT) then --Si ocurre una transición positiva de RELOJ_OUT
		if (senal_a_codificar='1') then --Si senal a codificar es 1L
			--Si el último estado de la senal codificada fue -5V,
			--o condición inicial
			if (qin_1= '0' and qin_2='1') or (qin_1= '0' and qin_2='0') then
				--Senal codificada es 5V
				qin_1 := '1';
				qin_2 := '0';
			else --Si el último estado de la senal codificada fue 5V
				--Senal codificada es -5V
				qin_1 :='0';
				qin_2 :='1';
			end if;
		else --Si senal a codificar es 0L
			--Para cumplir condición inicial
			if (qin_2='0' and qin_1 = '0') then
				--Senal codificada es -5V
				qin_1 := '0';
				qin_2 := '1';
			end if;
		end if;
	end if;

	--Las salidas toman los valores lógicos de las variables
	salida1<=qin_1;
	salida2<=qin_2;

end process Diferencial_M_cod;

end Behavioral;
```

Componente DIF_S_codificacion

```
----------------------------------------------------------------------------------
-- Universidad: Escuela Politécnica Nacional
-- Creadores: Daniel Velásquez y Viviana Checa
-- Director: Robin Alvarez PhD.
--
-- Module Name:    DIF_S_codificacion - Behavioral
-- Create Date:    15:10:46 07/10/2010
-- Quito - Ecuador
----------------------------------------------------------------------------------

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

entity DIF_S_codificacion is

	--Señales externas de la componente

	Port (RELOJ_OUT: in std_logic;
		senal_a_codificar: in std_logic;
		salida1: buffer std_logic;
		salida2: buffer std_logic);

end DIF_S_codificacion;

architecture Behavioral of DIF_S_codificacion is

begin
```

```
difs_cod: process(RELOJ_OUT)

--Variables utilizadas en este proceso

--La combinación de las variables qin_1 y qin_2
--permiten obtener los valores de -5V, 0V y +5V
--Si qin_1=0 y qin_2=0, van a representar 0V
--Si qin_1=1 y qin_2=0, van a representar +5V
--Si qin_1=0 y qin_2=1, van a representar -5V

--Variables que almacenan los valores lógicos de las salidas
--Valor inicial para empezar la codificaión es -5V
variable qin_1 : std_logic := '0';
variable qin_2 : std_logic := '1';

begin

      --Si ocurre una transición positiva de RELOJ_OUT
      if rising_edge (RELOJ_OUT) then
            if (senal_a_codificar='0') then--Si el bit a codificar es 0L
                  --Si el último estado de la senal codificada fue -5V o 0V
                  if (qin_1='0' and qin_2= '1') or (qin_1='0' and qin_2= '0') then
                        --Senal codificada es 5V
                        qin_1 := '1';
                        qin_2 := '0';
                  else --Si el último estado de la senal codificada fue 5V
                        --Senal codificada es -5V
                        qin_1 :='0';
                        qin_2 :='1';
                  end if;
            else --Si el bit a codificar es 1L
                  --Si el último estado de la senal codificada fue -5V
                  if (qin_1='0' and qin_2 = '1') then
                        --mantiene el nivel negativo
                        qin_1 := '0';
                        qin_2 := '1';
                  --Si el último estado de la senal codificada fue 5V
                  elsif (qin_1='1' and qin_2 = '0') then
                        --mantiene el nivel positivo
                     qin_1 := '1';
                        qin_2 := '0';
                  end if;
            end if;
      end if;
      --Las salidas toman los valores lógicos de las variables qin_1 y qin_2
      salida1<=qin_1;
      salida2<=qin_2;

end process difs_cod;

end Behavioral;
```

Componente Manchester_codificacion

```
----------------------------------------------------------------------------------
-- Universidad: Escuela Politécnica Nacional
-- Creadores: Daniel Velásquez y Viviana Checa
-- Director: Robin Alvarez PhD.
--
-- Module Name:    Manchester_codificacion - Behavioral
-- Create Date:    15:10:46 07/10/2010
-- Quito - Ecuador
```

```
----------------------------------------------------------------------------------

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

entity Manchester_codificacion is

	--Señales externas  de la componente

	Port (RELOJ_OUT_Doble: in std_logic;
		senal_a_codificar: in std_logic;
		salida1: buffer std_logic;
		salida2: buffer std_logic);

end Manchester_codificacion;

architecture Behavioral of Manchester_codificacion is

begin

manchester_cod: process (RELOJ_OUT_Doble)

--Variables que almacenan los valores lógicos de las salidas
--Valor inicial 0V
variable qin_1: std_logic := '0';
variable qin_2: std_logic := '0';

--Variable que identifica el último estado lógico de la senal a codificar
variable senal_a_codificar_actual: std_logic := '0';
--Variable que identifica la transición (positiva o negativa) de RELOJ_OUT
--Valor inicial identifica transición positiva
variable contar: std_logic_vector(1 downto 0) :="01";

begin

	--Si ocurre una transición positiva de RELOJ_OUT_Doble
	if rising_edge(RELOJ_OUT_Doble) then
		if (contar = 1) then -- Si Transicion positiva de RELOJ_OUT
			if (senal_a_codificar='0') then --Si senal a codificar es 0L
				--Senal codificada es -5V durante medio tiempo de bit
				qin_1 := '0';
				qin_2 := '1';
				--Almacenar el estado 0L de senal a codificar actual
				senal_a_codificar_actual:='0';
			else --Si senal a codificar es 1L
				--Senal codificada es 5V durante medio tiempo de bit
				qin_1 :='1';
				qin_2 :='0';
				--Almacena el estado 1L de senal a codificar actual
				senal_a_codificar_actual:='1';
			end if;
			contar := contar + 1;
		else --Transicion negativa de RELOJ_OUT
			--Si senal a codificar actual es 0L
			if (senal_a_codificar_actual='0') then
				--Senal codificada es 5V durante medio tiempo de bit
				qin_1 := '1'; --1534
				qin_2 := '0';
			else --Si senal a codificar actual es 1L
				--Senal codificada es -5V durante medio tiempo de bit
				qin_1 :='0';
				qin_2 :='1';
			end if;
			contar := "01";
		end if;
```

```
	end if;

	--Las salidas toman los valores lógicos de las variables qin_1 y qin_2
	salida1<=qin_1;
	salida2<=qin_2;

end process manchester_cod;

end Behavioral;
```

Componente Manch_Dif_codificacion

```
----------------------------------------------------------------------------------
-- Universidad: Escuela Politécnica Nacional
-- Creadores: Daniel Velásquez y Viviana Checa
-- Director: Robin Alvarez PhD.
--
-- Module Name:    Manch_dif_codificacion - Behavioral
-- Create Date:    15:10:46 07/10/2010
-- Quito - Ecuador
----------------------------------------------------------------------------------

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

entity Manch_dif_codificacion is

	--Señales externas  de la componente

	Port (RELOJ_OUT_Doble: in std_logic;
		senal_a_codificar: in std_logic;
		salida1: buffer std_logic;
		salida2: buffer std_logic);

end Manch_dif_codificacion;

architecture Behavioral of Manch_dif_codificacion is

begin

manchester_dif_cod : process(RELOJ_OUT_Doble)

--Variables utilizadas en este proceso

--La combinación de las variables qin_1 y qin_2
--permiten obtener los valores de -5V, 0V y +5V
--Si qin_1=0 y qin_2=0, van a representar 0V
--Si qin_1=1 y qin_2=0, van a representar +5V
--Si qin_1=0 y qin_2=1, van a representar -5V

--Variables que almacenan los valores lógicos de las salidas
--Valor inicial 0V
variable qin_1 : std_logic := '0';
variable qin_2 : std_logic := '0';

--Variables que almacenan el estado lógico anterior de la senal codificada
--Valor inicial -5V, transición negativa
variable qin1_anterior : std_logic := '0';
variable qin2_anterior : std_logic := '1';
--Variable que identifica el último estado lógico de la senal a codificar
variable senal_a_codificar_actual: std_logic := '0';
--Variable que identifica la transición (positiva o negativa) de RELOJ_OUT
```

```
--Valor inicial identifica transición positiva
variable contar: std_logic_vector(1 downto 0) :="01";

begin

	-- Si ocurre una transición positiva de RELOJ_OUT_Doble
	if rising_edge (RELOJ_OUT_Doble) then
		if (contar = 1) then --Si Transicion positiva de RELOJ_OUT
			if (senal_a_codificar='0') then--Si el bit a codificar es 0L
				--Si la transición anterior fue negativa
				if (qin1_anterior= '0' and qin2_anterior='1') then
					--Senal codificada es 5V durante 1/2 Tb
					qin_1 := '1';
					qin_2 := '0';
					--Se almacena valor para el siguiente
					--medio tiempo de bit
					qin1_anterior := '1';
					qin2_anterior := '0';
				else --si la transición anterior fue positiva
					--Senal codificada es -5V durante 1/2 Tb
					qin_1 :='0';
					qin_2 :='1';
					--Se almacena valor para el siguiente 1/2 Tb
					qin1_anterior := '0';
					qin2_anterior := '1';
				end if;
				--Almacenar el estado 0L de senal a codificar actual
				senal_a_codificar_actual:= '0';
			else --Si el bit a codificar es 1L
				--Si el valor anterior fue -5V,
				--la transición anterior fue negativa
				if (qin1_anterior= '0' and qin2_anterior='1') then
					--Senal codificada es -5V durante 1/2 Tb
					qin_1 := '0';
					qin_2 := '1';
					--Se almacena valor para el siguiente 1/2 Tb
					qin1_anterior := '0';
					qin2_anterior := '1';
				else --Si la transición anterior fue positiva
					--Senal codificada es 5V durante 1/2 Tb
					qin_1 :='1';
					qin_2 :='0';
					--Se almacena valor para el siguiente 1/2 Tb
					qin1_anterior := '1';
					qin2_anterior := '0';
				end if;
				--Almacenar el estado 1L de senal a codificar actual
				senal_a_codificar_actual := '1';
			end if;
			contar := contar + 1;
		else --Transicion negativa de RELOJ_OUT
			--Si el bit a codificar es 0L
			if (senal_a_codificar_actual='0') then
				--Si el valor anterior fue -5V
				if (qin1_anterior= '0' and qin2_anterior='1') then
					--Senal codificada es 5V durante segundo 1/2 Tb
					qin_1 := '1';
					qin_2 := '0';
					--Se almacena valor para el siguiente 1/2 Tb
					qin1_anterior := '1';
					qin2_anterior := '0';
				else --Si el valor anterior fue 5V
					--Senal codificada es -5V durante segundo 1/2 Tb
					qin_1 :='0';
					qin_2 :='1';
					--Se almacena valor para el siguiente 1/2 Tb
					qin1_anterior := '0';
```

```
                              qin2_anterior := '1';
                        end if;
                  else --Si el bit a codificar es 1L
                        --Si el valor anterior fue -5V
                        if (qin1_anterior= '0' and qin2_anterior='1') then
                              --Senal codificada es 5V el segundo 1/2 Tb
                              qin_1 := '1';
                              qin_2 := '0';
                              qin1_anterior := '1';
                              qin2_anterior := '0';
                        else --Si el valor anterior fue 5V
                              --Senal codificada es -5V el segundo 1/2 Tb
                              qin_1 :='0';
                              qin_2 :='1';
                              --Se almacena valor para el siguiente 1/2 Tb
                              qin1_anterior := '0';
                              qin2_anterior := '1';
                        end if;
                  end if;
                  contar := "01";
            end if;
      end if;

      --Las salidas toman los valores lógicos de las variables qin_1 y qin_2
      salida1 <= qin_1;
      salida2 <= qin_2;

end process manchester_dif_cod;

end Behavioral;
```

Componente CMI_codificacion

```
----------------------------------------------------------------------------------
-- Universidad: Escuela Politécnica Nacional
-- Creadores: Daniel Velásquez y Viviana Checa
-- Director: Robin Alvarez PhD.
--
-- Module Name:    CMI_codificacion - Behavioral
-- Create Date:    15:10:46 07/10/2010
-- Quito - Ecuador
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity CMI_codificacion is

      --Señales externas  de la componente

      port (RELOJ_OUT_Doble: in std_logic;
                  senal_a_codificar: in std_logic;
                  salida1: buffer std_logic;
                  salida2: buffer std_logic);

end CMI_codificacion;

architecture Behavioral of CMI_codificacion is

begin

cmi_cod: process (RELOJ_OUT_Doble)

--Variables que almacenan los valores lógicos de las salidas
```

```
variable qin_1: std_logic := '0';
variable qin_2: std_logic := '0';

--Variables que almacenan el estado lógico anterior de la senal codificada
--Valor inicial -5V
variable qin1_anterior: std_logic := '0';
variable qin2_anterior: std_logic := '1';

--Variable que identifica el último estado lógico de la senal a codificar
variable senal_a_codificar_actual: std_logic := '0';
--Variable que identifica la transición (positiva o negativa) de RELOJ_OUT
--Valor inicial identifica transición positiva
variable contar: std_logic_vector(1 downto 0) :="01";

begin

	--Si ocurre una transición positiva de RELOJ_OUT_Doble
	if rising_edge (RELOJ_OUT_Doble) then
		if (contar = 1) then --Si Transicion positiva de RELOJ_OUT
			--Si senal a codificar es 1L
			if (senal_a_codificar='1') then
				--Si senal codificada fue 5V
				if (qin1_anterior= '1' and qin2_anterior='0') then
					--Senal codificada es -5V
					qin_1 := '0';
					qin_2 := '1';
					qin1_anterior := '0';
					qin2_anterior := '1';
				else --Si senal codificada no fue 5V, fue -5V
					--Senal codificada es 5V
					qin_1 :='1';
					qin_2 :='0';
					qin1_anterior := '1';
					qin2_anterior := '0';
				end if;
				--Almacena el estado 1L de senal a codificar actual
				senal_a_codificar_actual := '1';
			else --Si senal a codificar es 0L
				--Senal codificada es -5V durante 1/2 Tb
				qin_1 := '0';
				qin_2 := '1';
				--Almacenar el estado 0L de senal a codificar actual
				senal_a_codificar_actual := '0';
			end if;
			contar := contar + 1;
		else --Si Transicion negativa de RELOJ_OUT
			--Si senal a codificar actual es 0L
			if (senal_a_codificar_actual='0') then
				--Senal codificada es 5V durante el segundo 1/2 Tb
				qin_1 :='1';
				qin_2 :='0';
			end if;
			contar := "01";
		end if;
	end if;

	--Las salidas toman los valores lógicos de las variables qin_1 y qin_2
	salida1<=qin_1;
	salida2<=qin_2;

end process cmi_cod;

end Behavioral;
```

Componente AMI_codificacion

```
----------------------------------------------------------------------------------
-- Universidad: Escuela Politécnica Nacional
-- Creadores: Daniel Velásquez y Viviana Checa
-- Director: Robin Alvarez PhD.
--
-- Module Name:    AMI_codificacion - Behavioral
-- Create Date:    09/11/2010
-- Quito - Ecuador
----------------------------------------------------------------------------------

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

entity AMI_codificacion is

	--Señales externas de la componente

	port (RELOJ_OUT: in std_logic;
		senal_a_codificar: in std_logic;
		salida1: buffer std_logic;
		salida2: buffer std_logic);

end AMI_codificacion;

architecture Behavioral of AMI_codificacion is

begin

ami_cod: process (RELOJ_OUT)

--Variables que almacenan los valores lógicos de las salidas
variable qin_1 : std_logic := '0';
variable qin_2 : std_logic := '0';
--Variables que almacenan el estado lógico anterior
--Valor inicial -5V
variable qin1_anterior : std_logic := '0';
variable qin2_anterior : std_logic := '1';
begin

	--Si ocurre una transición positiva de RELOJ_OUT
	if rising_edge(RELOJ_OUT) then
		--Si senal a codificar es 1L
		if (senal_a_codificar='1') then
				--Si senal codificada fue 5V
			if (qin1_anterior= '1' and qin2_anterior='0') then
				--Senal codificada es -5V
				qin_1 := '0';
				qin_2 := '1';
				qin1_anterior := '0';
				qin2_anterior := '1';
			else --Si senal codificada fue -5V
				--Senal codificada es 5V
				qin_1 :='1';
				qin_2 :='0';
				qin1_anterior := '1';
				qin2_anterior := '0';
			end if;
		else --Si senal a codificar es 0L
				--Senal codificada es 0V
				qin_1 := '0';
				qin_2 := '0';
		end if;
	end if;
```

```
      --Las salidas toman los valores lógicos de las variables qin_1 y qin_2
      salida1<=qin_1;
      salida2<=qin_2;

end process ami_cod;

end Behavioral;
```

Componente HDB3_codificacion

```
----------------------------------------------------------------------------------
-- Universidad: Escuela Politécnica Nacional
-- Creadores: Daniel Velásquez y Viviana Checa
-- Director: Robin Alvarez PhD.
--
-- Module Name:    HDB3_codificacion - Behavioral
-- Create Date:    15:10:46 07/10/2010
-- Quito - Ecuador
----------------------------------------------------------------------------------

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

entity HDB3_codificacion is

      port (RELOJ_OUT: in std_logic;
            rst : in std_logic;
            senal_a_codificar: in std_logic;
            salida1: buffer std_logic;
            salida2: buffer std_logic);

end HDB3_codificacion;

architecture Behavioral of HDB3_codificacion is

begin

hdb3_cod: process (RELOJ_OUT)

--Variables que almacenan los valores lógicos de las salidas
variable qin_1 : std_logic := '0';
variable qin_2 : std_logic := '0';

--Variables que almacenan el estado lógico anterior
--Valor inicial -5V
variable qin1_anterior : std_logic := '0';
variable qin2_anterior : std_logic := '1';

--Variables que almacenan el número de 0L y 1L consecutivos
variable ceros: natural range 0 to 4 := 0;
variable unos: natural range 0 to 6 := 0;

--Vectores que almacenan los estados lógicos que deben tomar
--las salidas
variable vector_salida1: std_logic_vector(99 downto 0);
variable vector_salida2: std_logic_vector(99 downto 0);
--Variable que identifica los primeros bits codificados
variable inicio : std_logic :=  '1';
--Variables para denotar el elemento de los vectores de salida
variable i : natural range 0 to 99 := 0;
variable v: natural range 0 to 99 := 0;
```

```
--Variables que identifican el valor de la última violación
--Violacion inicial no debe ser +5 ni -5 al inicio porque todavía no hay
violaciones
variable violacion_salida1_anterior: std_logic := '0';
variable violacion_salida2_anterior: std_logic := '0';

begin

	if (rst = '1')then --Si rst = 1L
		--Inicializar todas las variables implicadas en este proceso
		qin1_anterior:= '0';
		qin2_anterior := '1';
		ceros := 0;
		unos := 0;
		inicio :=  '1';
		i := 0;
		v := 0;

	--Si ocurre una transición positiva de RELOJ_OUT
	elsif rising_edge(RELOJ_OUT) then
		--Si senal a codificar es 1L
		if (senal_a_codificar='1') then
			ceros := 0; --Este bit y los anteriores no fueron 0L
			unos := unos + 1; --Incrementa en 1 el valor de unos
			--Si senal codificada fue 5V
			if (qin1_anterior= '1') then --and qin2_anterior='0'
				--Senal codificada es -5V
				qin_1 := '0';
				qin_2 := '1';
				qin1_anterior := '0';
				qin2_anterior := '1';
			--Si senal codificada fue -5V
			elsif (qin1_anterior= '0') then --and qin2_anterior='1'
				--Senal codificada es 5V
				qin_1 :='1';
				qin_2 :='0';
				qin1_anterior := '1';
				qin2_anterior := '0';
			end if;
			if (unos = 6) then --Si el número de 1L es igual a 6
				--Ha pasado el identificador de inicio
				--Inicializar violaciones
				violacion_salida1_anterior := '0';
				violacion_salida2_anterior := '0';
				unos := 0;
			end if;
		end if;
		if (senal_a_codificar = '0')then --Si senal a codificar es 0L
			--Senal codificada es 0V
			qin_1 := '0';
			qin_2 := '0';
			ceros := ceros + 1; --Incrementa en 1 el valor de ceros
			unos := 0; --Este bit y los anteriores no fueron 1L
			if (ceros = 4) then --Si han pasado 4 ceros seguidos
			  --Violación y Relleno
			  --Si bit codificado anterior = última violación
			  if   ((qin1_anterior = violacion_salida1_anterior) and
			       (qin2_anterior = violacion_salida2_anterior)) then
				--Senal codificada es igual a la
				--última violación invertida de signo
				qin_1 := not violacion_salida1_anterior;
				qin_2 := not violacion_salida2_anterior;
				--Invertir el signo de última violación, 1001 o -100-1,
				--y almacenarlo en vector_salida (actual - 3)
				if (i=0) then
				vector_salida1(97) := not violacion_salida1_anterior;
```

```
                vector_salida2(97) := not violacion_salida2_anterior;
                elsif (i=1) then
                vector_salida1(98) := not violacion_salida1_anterior;
                vector_salida2(98) := not violacion_salida2_anterior;
                elsif (i=2) then
                vector_salida1(99) := not violacion_salida1_anterior;
                vector_salida2(99) := not violacion_salida2_anterior;
                else
                vector_salida1(i-3) := not violacion_salida1_anterior;
                vector_salida2(i-3) := not violacion_salida2_anterior;
                end if;
                --Almacenar el estado actual de senal codificada
                qin1_anterior := not qin1_anterior;
                qin2_anterior := not qin2_anterior;
                --Almacenar el estado actual de violación
                violacion_salida1_anterior:=not violacion_salida1_anterior;
                violacion_salida2_anterior:=not violacion_salida2_anterior;
          --Violación
          else
                      --Bit de Senal codificada es igual al último
                      --bit codificado como 5V o -5V
                      qin_1 := qin1_anterior;
                      qin_2 := qin2_anterior;
                      --Si último pulso de senal codificada fue 5V
                      if (qin1_anterior= '1') then
                      --and qin2_anterior='0'
                            violacion_salida1_anterior := '1';
                            violacion_salida2_anterior := '0';
                      else --Si último pulso de senal codificada fue -5V
                            violacion_salida1_anterior := '0';
                            violacion_salida2_anterior := '1';
                      end if;
                end if;
                ceros := 0; --Encerar cuenta de 0L
      end if;
end if;
vector_salida1(i) := qin_1; --Almacenar en los vectores los valores
vector_salida2(i) := qin_2; --lógicos de las variables qin_1 y qin_2
if (i < 99) then --Si el elemento de los vectores es menor que 99
      i := i + 1; --Pase al siguiente elemento del vector
elsif (i = 99) then --Si el elemento de los vectores es el 99
      i := 0; --Pase al primer elemento de los vectores
      --identificar que han pasado los primeros 99 bits codificados
      inicio := '0';
end if;
if (inicio = '1') then --si aún no pasan los primeros 99
      --Si se han codificados más de 4 bits
      --bits codificados y almacenado en los vectores
      if (i > 4) then
            --Las salidas toman los valores lógicos
            --almacenados en los vectores
            salida1 <= vector_salida1(v);
            salida2 <= vector_salida2(v);
            if (v < 99) then
                  --Pase al siguiente elemento de los vectores
                  v := v + 1;
            elsif (v = 99) then
                  --Pase al primer elemento de los vectores
                  v := 0;
            end if;
      end if;
else --Luego de los primeros bits codificados
      --Las salidas toman los valores lógicos
      --almacenados en los vectores
      salida1 <= vector_salida1(v);
      salida2 <= vector_salida2(v);
      if (v < 99) then
```

```
                        --Pase al siguiente elemento de los vectores
                        v := v + 1;
                  elsif (v = 99) then
                        --Pase al primer elemento de los vectores
                        v := 0;
                  end if;
            end if;
      end if;

end process hdb3_cod;

end Behavioral;
```

Componente MLT3_codificacion

```
----------------------------------------------------------------------------------
-- Universidad: Escuela Politécnica Nacional
-- Creadores: Daniel Velásquez y Viviana Checa
-- Director: Robin Alvarez PhD.
--
-- Module Name:    MLT3_codificacion - Behavioral
-- Create Date:    15:10:46 07/10/2010
-- Quito - Ecuador
----------------------------------------------------------------------------------

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

entity MLT3_codificacion is

--Señales externas  de la componente
      Port (RELOJ_OUT: in std_logic;
            senal_a_codificar: in std_logic;
            salida1: buffer std_logic;
            salida2: buffer std_logic);

end MLT3_codificacion;

architecture Behavioral of MLT3_codificacion is

begin


mlt3_cod: process(RELOJ_OUT)

--Variables utilizadas en este proceso

--Variables utilizadas en este proceso

--La combinación de las variables qin_1 y qin_2
--permiten obtener los valores de -5V, 0V y +5V
--Si qin_1=0 y qin_2=0, van a representar 0V
--Si qin_1=1 y qin_2=0, van a representar +5V
--Si qin_1=0 y qin_2=1, van a representar -5V

--Variables que almacenan los valores lógicos de las salidas
--Valor inicial 0V
variable qin_1 : std_logic := '0';
variable qin_2 : std_logic := '0';

--Las variables qin1_anterior,qin2_anterior, qin3_anterior y qin4_anterior
--utilizan las mimas combinaciones descritas arriba.
```

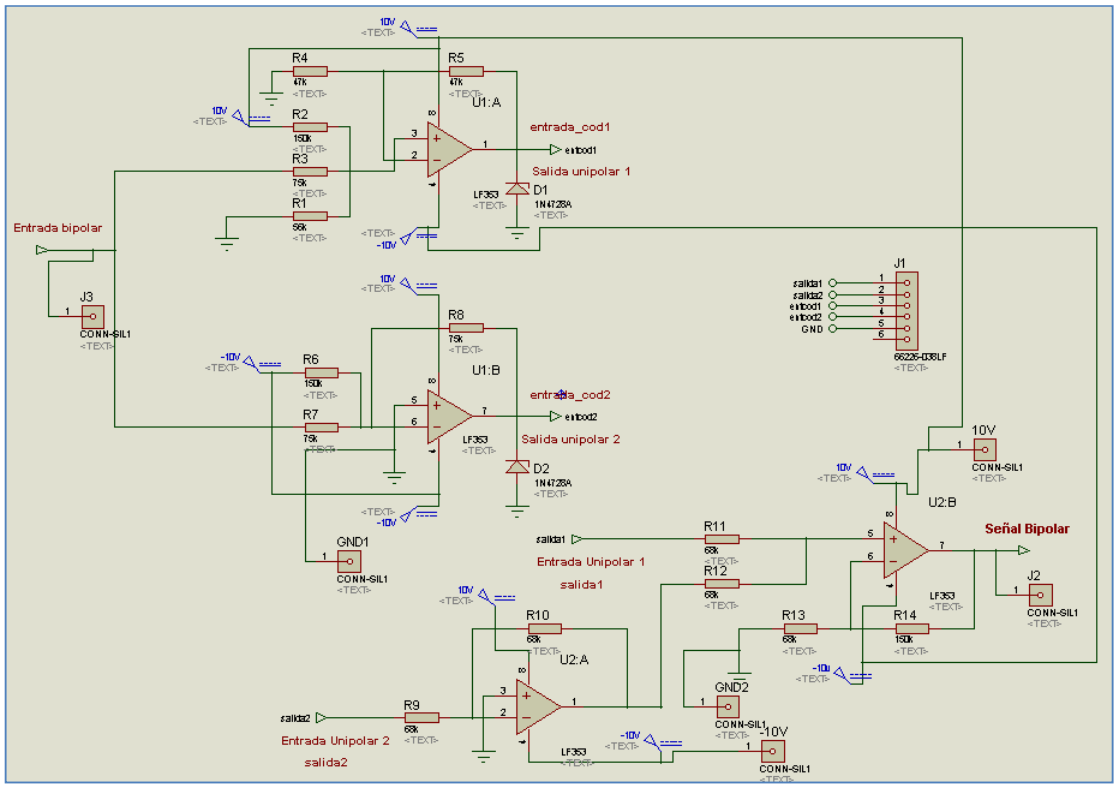
```
--Estas variables permiten almacenar los dos valores anteriores
--que son necesarios para la codificación de un 1L
--Valores iniciales 0 y +V
variable qin1_anterior : std_logic := '0'; --0V
variable qin2_anterior : std_logic := '0';
variable qin3_anterior : std_logic := '1'; --5V
variable qin4_anterior : std_logic := '0';

begin

	--Si ocurre una transición positiva de RELOJ_OUT
	if rising_edge (RELOJ_OUT) then

		if (senal_a_codificar='1') then--Si el bit a codificar es 1L
			--Si el penúltimo valor anterior fue 0
			if (qin1_anterior='0' and qin2_anterior='0')then
				--Si el último valor anterior fue -5V
				if (qin3_anterior='0' and qin4_anterior='1')then
					--Senal codificada es 0V
					qin_1:='0';
					qin_2:='0';
					--Almacenar en las 4 variables
					--los valores anteriores
					--Almacenar en penúltimo valor anterior -5V
					qin1_anterior := '0';
					qin2_anterior := '1';
					--Almacenar en último valor anterior fue 0V
					qin3_anterior := '0';
					qin4_anterior := '0';
				--Si el último valor anterior fue 5V
				elsif (qin3_anterior='1' and qin4_anterior='0')then
					--Senal codificada es 0V
					qin_1:='0';
					qin_2:='0';
					--Almacenar en penúltimo valor anterior -5V
					qin1_anterior := '1';
					qin2_anterior := '0';
					--Almacenar en último valor anterior fue 0V
					qin3_anterior := '0';
					qin4_anterior := '0';
				end if;
			--Si el penúltimo valor anterior fue -5V
			elsif (qin1_anterior='0' and qin2_anterior='1')then
				--Si el último valor anterior fue 0V
				if (qin3_anterior='0' and qin4_anterior='0')then
					--Senal codificada es 5V
					qin_1:='1';
					qin_2:='0';
					--Almacenar en penúltimo valor anterior 0V
					qin1_anterior := '0';
					qin2_anterior := '0';
					--Almacenar en último valor anterior 5V
					qin3_anterior := '1';
					qin4_anterior := '0';
				end if;
			--Si el penúltimo valor anterior fue 5V
			elsif (qin1_anterior='1' and qin2_anterior='0')then
				--Si el último valor anterior fue 0V
				if (qin3_anterior='0' and qin4_anterior='0')then
					--Senal codificada es -5V
					qin_1:='0';
					qin_2:='1';
					--Almacenar en penúltimo valor anterior 0V
					qin1_anterior := '0';
					qin2_anterior := '0';
					--Almacenar en último valor anterior -5V
					qin3_anterior := '0';
```

```
                        qin4_anterior := '1';
                    end if;
                end if;

            else --Si el bit a codificar es 0L

                --Si el penúltimo valor anterior fue 0V
                if (qin1_anterior='0' and qin2_anterior='0')then
                    --Si el último valor anterior fue -5V
                    if (qin3_anterior='0' and qin4_anterior='1')then
                        --Se mantiene el nivel anterior, es decir, -5V
                        qin_1:='0';
                        qin_2:='1';
                    --Si el último valor anterior fue 5V
                    elsif (qin3_anterior='1' and qin4_anterior='0')then
                        --Se mantiene el nivel anterior, es decir, 5V
                        qin_1:='1';
                        qin_2:='0';
                    end if;
                --Si el penúltimo valor anterior fue -5V
                elsif (qin1_anterior='0' and qin2_anterior='1')then
                    --Si el último valor anterior fue 0V
                    if (qin3_anterior='0' and qin4_anterior='0')then
                        --Se mantiene el nivel anterior, es decir, 0V
                        qin_1:='0';
                        qin_2:='0';
                    end if;
                --Si el penúltimo valor anterior fue 5V
                elsif (qin1_anterior='1' and qin2_anterior='0')then
                    --Si el último valor anterior es 0V
                    if (qin3_anterior='0' and qin4_anterior='0')then
                        --Senal codificada es 0V
                        qin_1:='0';
                        qin_2:='0';
                    end if;
                end if;

            end if;

        end if;
        --Las salidas toman los valores lógicos de las variables qin_1 y qin_2
        salida1<=qin_1;
        salida2<=qin_2;

end process mlt3_cod;

end Behavioral;
```

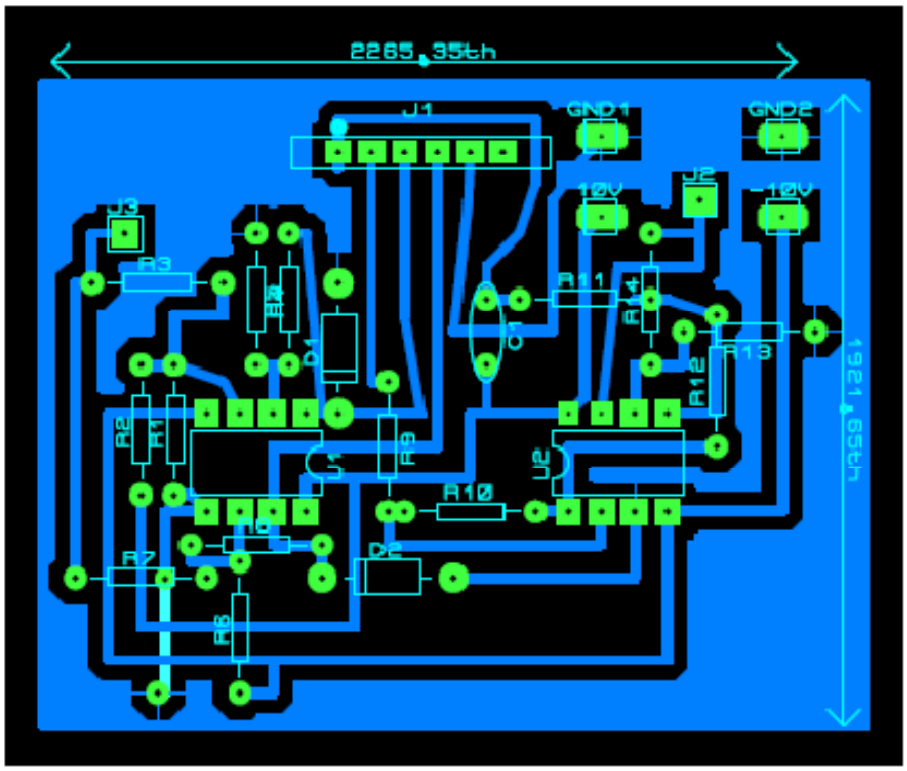
Nota.- El código del programa principal y las componentes de decodificación, que no se encuentran en este Anexo, se localizan únicamente en el CD entregado a la BIEE, debido a su gran extensión; tomando en cuenta que el propósito de este proyecto de titulación no es mostrar una gran extensión de código VHDL.

CIRCUITO INTERPRETADOR UNIPOLAR – BIPOLAR Y CIRCUITO INTERPRETADOR BIPOLAR - UNIPOLAR

DIAGRAMA ESQUEMÁTICO



CIRCUITO IMPRESO



National Semiconductor

February 1995

LF353 Wide Bandwidth Dual JFET Input Operational Amplifier

General Description

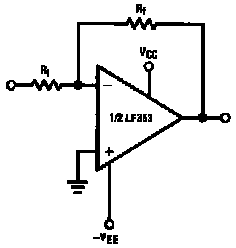
These devices are low cost, high speed, dual JFET input operational amplifiers with an internally trimmed input offset voltage (BI-FET II™ technology). They require low supply current yet maintain a large gain bandwidth product and fast slew rate. In addition, well matched high voltage JFET input devices provide very low input bias and offset currents. The LF353 is pin compatible with the standard LM1558 allowing designers to immediately upgrade the overall performance of existing LM1558 and LM358 designs.

These amplifiers may be used in applications such as high speed integrators, fast D/A converters, sample and hold circuits and many other circuits requiring low input offset voltage, low input bias current, high input impedance, high slew rate and wide bandwidth. The devices also exhibit low noise and offset voltage drift.

Features

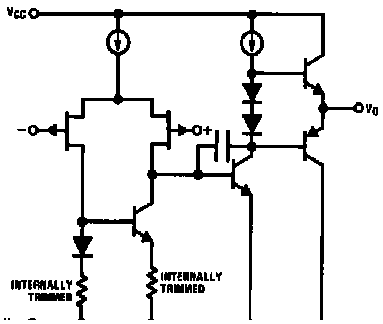
- Internally trimmed offset voltage 10 mV
- Low input bias current 50pA
- Low input noise voltage 25 nV/$\sqrt{\text{Hz}}$
- Low input noise current 0.01 pA/$\sqrt{\text{Hz}}$
- Wide gain bandwidth 4 MHz
- High slew rate 13 V/ms
- Low supply current 3.6 mA
- High input impedance $10^{12}\Omega$
- Low total harmonic distortion $A_V = 10$, $R_L = 10k$, $V_O = 20V_{p-p}$, BW = 20 Hz-20 kHz $\leq 0.02\%$
- Low 1/f noise corner 50 Hz
- Fast settling time to 0.01% 2 ms

Typical Connection



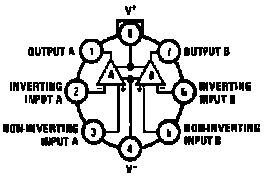
Simplified Schematic

1/2 Dual



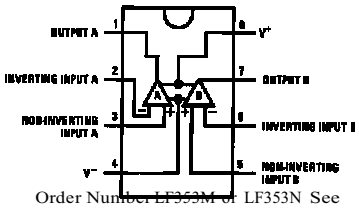
Connection Diagrams

Metal Can Package (Top View)



Order Number LF353H
See NS Package Number H08A

Dual-In-Line Package (Top View)



Order Number LF353M or LF353N See
NS Package Number M08A or N08E

TL/H/5649–1

BI-FET II™ is a trademark of National Semiconductor Corporation.

Absolute Maximum Ratings

If Military/Aerospace specified devices are required, please contact the National Semiconductor Sales Office/Distributors for availability and specifications.

Supply Voltage	± 18V
Power Dissipation	(Note 1)
Operating Temperature Range	0§C to +70§C
T_j(MAX)	150§C
Differential Input Voltage	±30V
Input Voltage Range (Note 2)	± 15V
Output Short Circuit Duration	Continuous
Storage Temperature Range	−65§C to +150§C
Lead Temp. (Soldering, 10 sec.)	260§C
Soldering Information	
Dual-In-Line Package	
Soldering (10 sec.)	260§C
Small Outline Package	
Vapor Phase (60 sec.)	215§C
Infrared (15 sec.)	220§C

See AN-450 ''Surface Mounting Methods and Their Effect on Product Reliability'' for other methods of soldering surface mount devices.

ESD Tolerance (Note 7)	1700V
θ_{JA} M Package	TBD

DC Electrical Characteristics (Note 4)

Symbol	Parameter	Conditions	LF353			Units
			MIn	Typ	Max	
V_{OS}	Input Offset Voltage	R_S = 10kX, T_A = 25§C Over Temperature		5	10 13	mV mV
DV_{OS}/DT	Average TC of Input Offset Voltage	R_S = 10 kX		10		mV/§C
I_{OS}	Input Offset Current	T_j = 25§C, (Notes 4, 5) T_j ≤ 70§C		25	100 4	pA nA
I_B	Input Bias Current	T_j = 25§C, (Notes 4, 5) T_j ≤ 70§C		50	200 8	pA nA
R_{IN}	Input Resistance	T_j = 25§C		10^{12}		X
A_{VOL}	Large Signal Voltage Gain	V_S = ±15V, T_A = 25§C V_O = ±10V, R_L = 2 kX Over Temperature	25 15	100		V/mV V/mV
V_O	Output Voltage Swing	V_S = ±15V, R_L = 10kX	±12	±13.5		V
V_{CM}	Input Common-Mode Voltage Range	V_S = ±15V	±11	+15 −12		V V
CMRR	Common-Mode Rejection Ratio	R_S ≤ 10kX	70	100		dB
PSRR	Supply Voltage Rejection Ratio	(Note 6)	70	100		dB
I_S	Supply Current			3.6	6.5	mA

AC Electrical Characteristics (Note 4)

Symbol	Parameter	Conditions	LF353			Units
			Min	Typ	Max	
	Amplifier to Amplifier Coupling	T_A = 25§C, f = 1 Hz−20 kHz (Input Referred)		−120		dB
SR	Slew Rate	V_S = ±15V, T_A = 25§C	8.0	13		V/ms
GBW	Gain Bandwidth Product	V_S = ±15V, T_A = 25§C	2.7	4		MHz
e_n	Equivalent Input Noise Voltage	T_A = 25§C, R_S = 100X, f = 1000 Hz		16		nV/$\sqrt{Hz}$
i_n	Equivalent Input Noise Current	T_j = 25§C, f = 1000 Hz		0.01		pA/$\sqrt{Hz}$

Note 1: For operating at elevated temperatures, the device must be derated based on a thermal resistance of 115§C/W typ junction to ambient for the N package, and 158§C/W typ junction to ambient for the H package.

Note 2: Unless otherwise specified the absolute maximum negative input voltage is equal to the negative power supply voltage.

Note 3: The power dissipation limit, however, cannot be exceeded.

Note 4: These specifications apply for V_S = ±15V and 0§C ≤ T_A ≤ +70§C. V_{OS}, I_B and I_{OS} are measured at V_{CM} = 0.

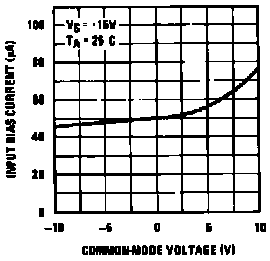
Note 5: The input bias currents are junction leakage currents which approximately double for every 10§C increase in the junction temperature, T_j. Due to the limited production test time, the input bias currents measured are correlated to junction temperature. In normal operation the junction temperature rises above the ambient temperature as a result of internal power dissipation, P_D. $T_j = T_A + \theta_{jA} P_D$ where θ_{jA} is the thermal resistance from junction to ambient. Use of a heat sink is recommended if input bias current is to be kept to a minimum.

Note 6: Supply voltage rejection ratio is measured for both supply magnitudes increasing or decreasing simultaneously in accordance with common practice. V_S = ±6V to ±15V.

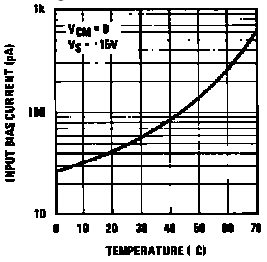
Note 7: Human body model, 1.5 kX in series with 100 pF.

Typical Performance Characteristics

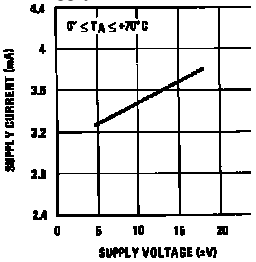
Input Bias Current



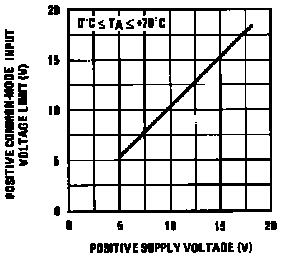
Input Bias Current



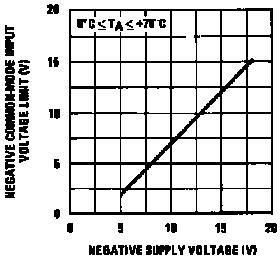
Supply Current



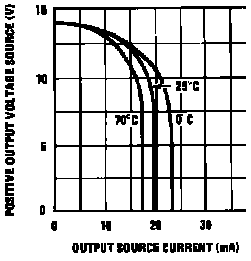
Positive Common-Mode Input Voltage Limit



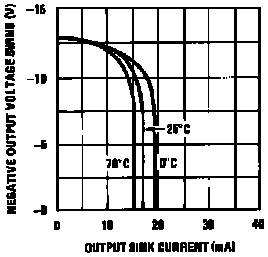
Negative Common-Mode Input Voltage Limit



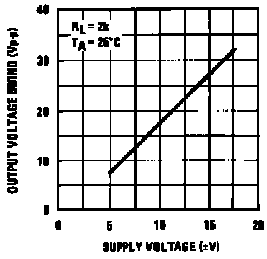
Positive Current Limit



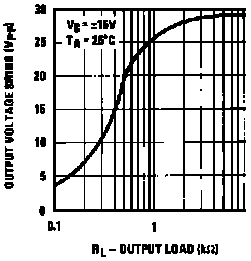
Negative Current Limit



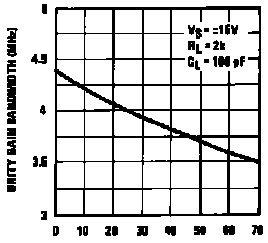
Voltage Swing



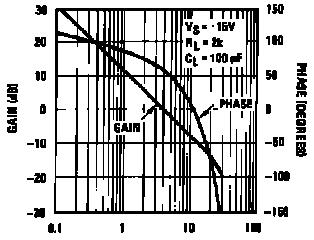
Output Voltage Swing



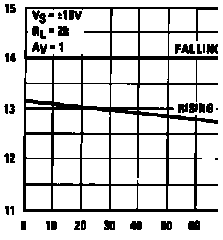
Gain Bandwidth



Bode Plot



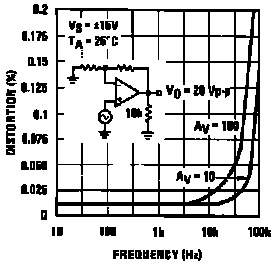
Slew Rate



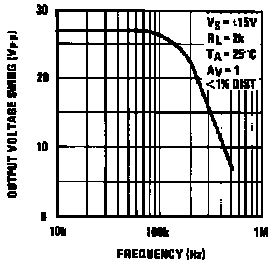
TL/H/5649–2

Typical Performance Characteristics (Continued)

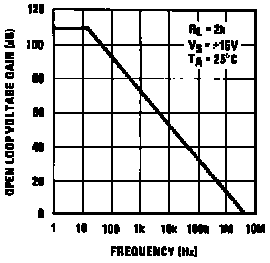
Distortion vs Frequency



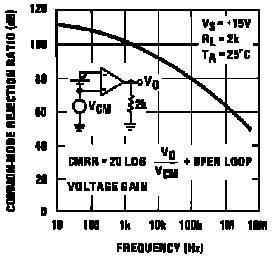
Undistorted Output Voltage Swing



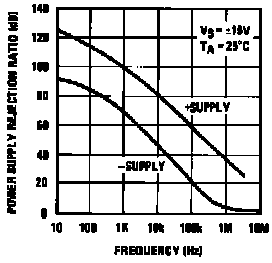
Open Loop Frequency Response



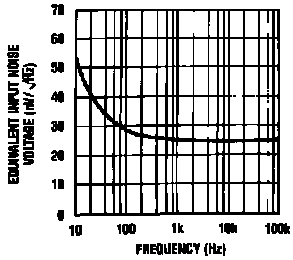
Common-Mode Rejection Ratio



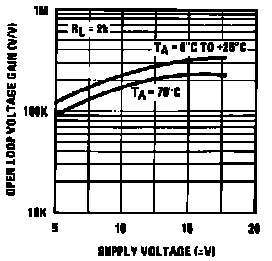
Power Supply Rejection Ratio



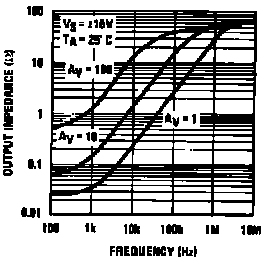
Equivalent Input Noise Voltage



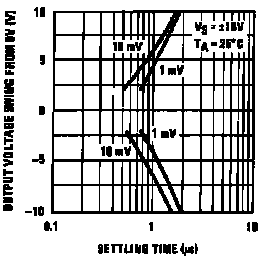
Open Loop Voltage Gain (V/V)



Output Impedance



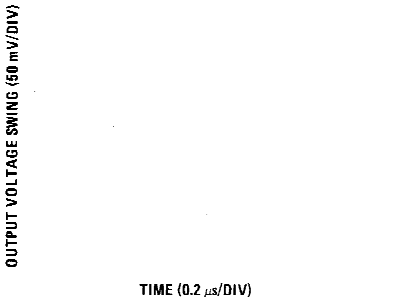
Inverter Settling Time



TL/H/5649–3

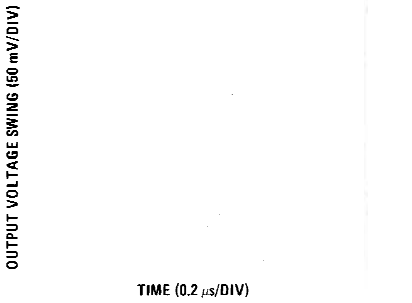
Pulse Response

Small Signaling Inverting



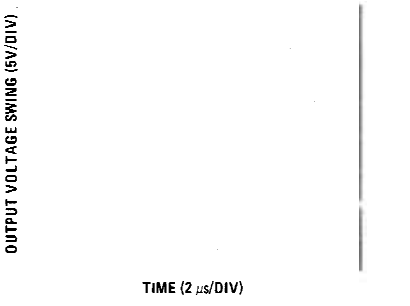
TL/H/5649–4

Small Signal Non-Inverting



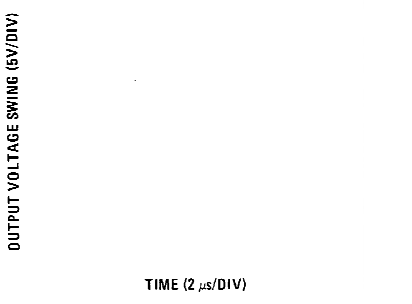
TL/H/5649–5

Large Signal Inverting



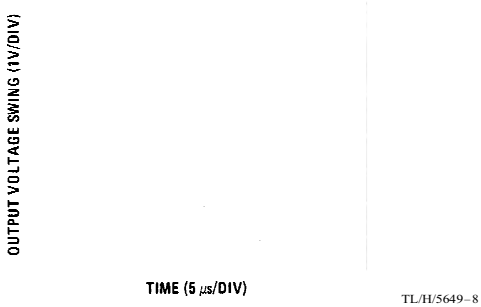
TL/H/5649–6

Large Signal Non-Inverting



TL/H/5649–7

Current Limit (R_L = 100X)



TL/H/5649–8

Application Hints

These devices are op amps with an internally trimmed input offset voltage and JFET input devices (BI-FET II). These JFETs have large reverse breakdown voltages from gate to source and drain eliminating the need for clamps across the inputs. Therefore, large differential input voltages can easily be accommodated without a large increase in input current. The maximum differential input voltage is independent of the supply voltages. However, neither of the input voltages should be allowed to exceed the negative supply as this will cause large currents to flow which can result in a destroyed unit.

Exceeding the negative common-mode limit on either input will force the output to a high state, potentially causing a reversal of phase to the output. Exceeding the negative common-mode limit on both inputs will force the amplifier output to a high state. In neither case does a latch occur since raising the input back within the common-mode range again puts the input stage and thus the amplifier in a normal operating mode.

Application Hints (Continued)

Exceeding the positive common-mode limit on a single input will not change the phase of the output; however, if both inputs exceed the limit, the output of the amplifier will be forced to a high state.

The amplifiers will operate with a common-mode input voltage equal to the positive supply; however, the gain bandwidth and slew rate may be decreased in this condition. When the negative common-mode voltage swings to within 3V of the negative supply, an increase in input offset voltage may occur.

Each amplifier is individually biased by a zener reference which allows normal circuit operation on ±6V power supplies. Supply voltages less than these may result in lower gain bandwidth and slew rate.

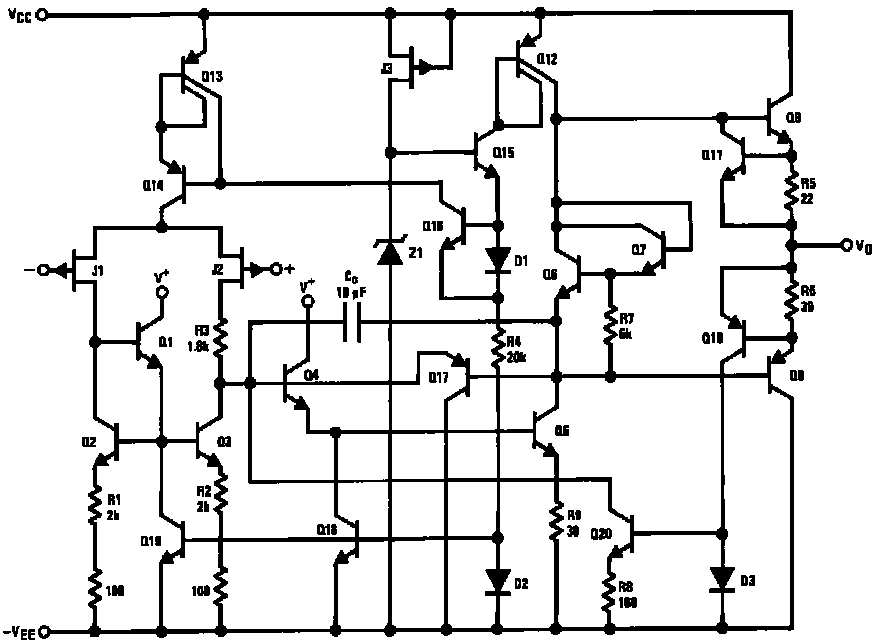
The amplifiers will drive a 2 kX load resistance to ±10V over the full temperature range of 0°C to a70°C. If the amplifier is forced to drive heavier load currents, however, an increase in input offset voltage may occur on the negative voltage swing and finally reach an active current limit on both positive and negative swings.

Precautions should be taken to ensure that the power supply for the integrated circuit never becomes reversed in polarity or that the unit is not inadvertently installed backwards in a socket as an unlimited current surge through the resulting forward diode within the IC could cause fusing of the internal conductors and result in a destroyed unit.

As with most amplifiers, care should be taken with lead dress, component placement and supply decoupling in order to ensure stability. For example, resistors from the output to an input should be placed with the body close to the input to minimize ''pick-up'' and maximize the frequency of the feedback pole by minimizing the capacitance from the input to ground.

A feedback pole is created when the feedback around any amplifier is resistive. The parallel resistance and capacitance from the input of the device (usually the inverting input) to AC ground set the frequency of the pole. In many instances the frequency of this pole is much greater than the expected 3 dB frequency of the closed loop gain and consequently there is negligible effect on stability margin. However, if the feedback pole is less than approximately 6 times the expected 3 dB frequency a lead capacitor should be placed from the output to the input of the op amp. The value of the added capacitor should be such that the RC time constant of this capacitor and the resistance it parallels is greater than or equal to the original feedback pole time constant.

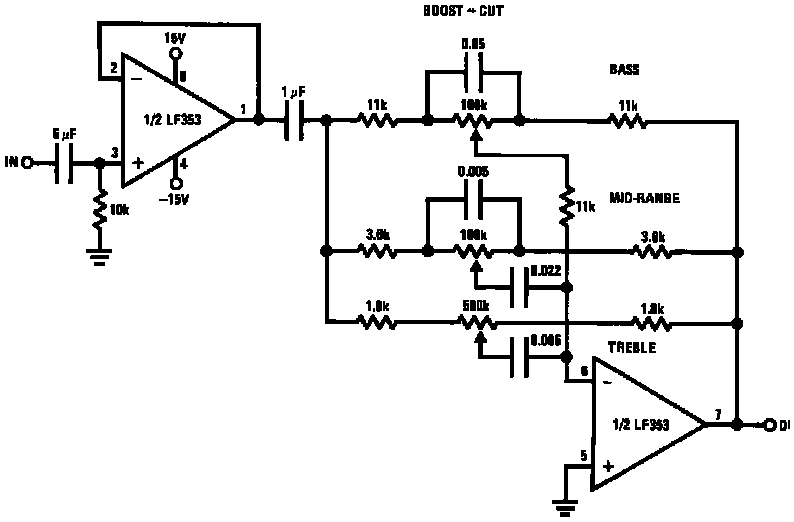
Detailed Schematic

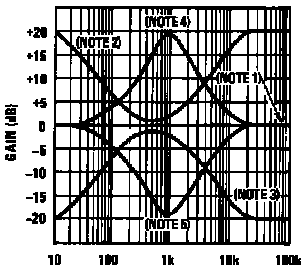


TL/H/5649–9

Typical Applications

Three-Band Active Tone Control





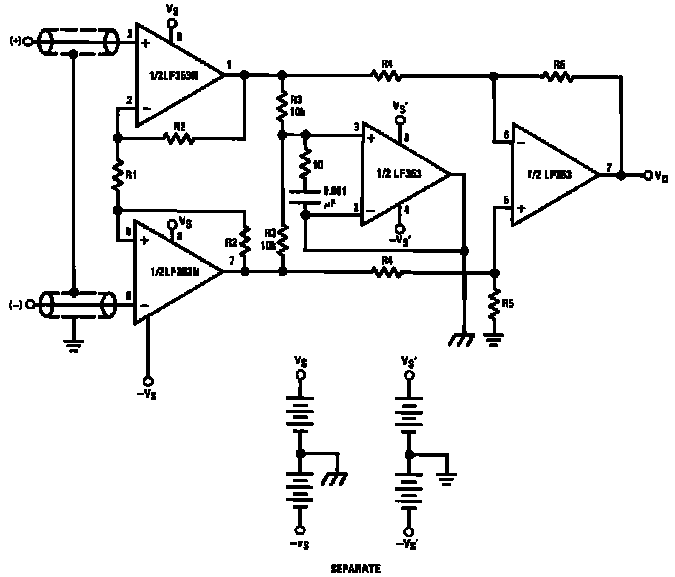
TL/H/5649–10

Note 1: All controls flat.
Note 2: Bass and treble boost, mid flat.
Note 3: Bass and treble cut, mid flat.
Note 4: Mid boost, bass and treble flat.
Note 5: Mid cut, bass and treble flat.

All potentiometers are linear taper
Use the LF347 Quad for stereo applications

Typical Applications (Continued)

Improved CMRR Instrumentation Amplifier



$$A_V = \left(\frac{2R2}{R1} + 1\right)\frac{R5}{R4}$$

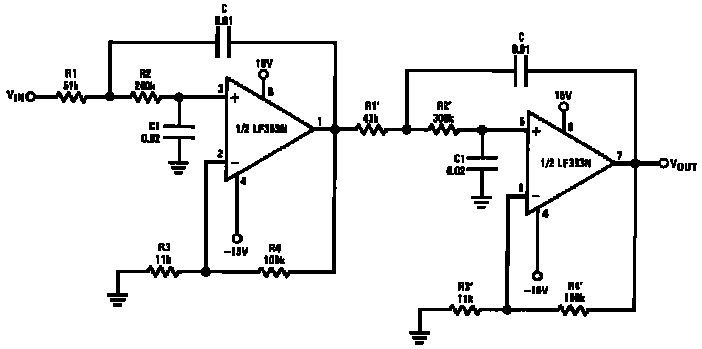
and are separate isolated grounds

Matching of R2's, R4's and R5's control CMRR

With A_{V_T} = 1400, resistor matching = 0.01%: CMRR = 136 dB

- Very high input impedance
- Super high CMRR

Fourth Order Low Pass Butterworth Filter

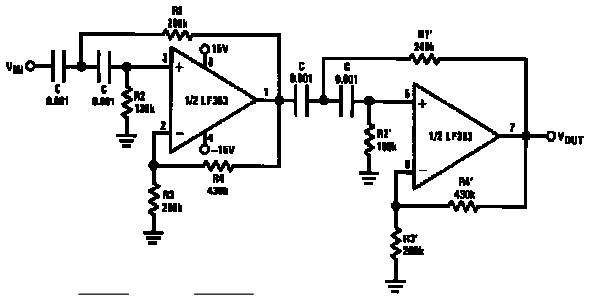


- Corner frequency (f_c) = $\sqrt{\frac{1}{R1R2CC1}} \cdot \frac{1}{2\pi} = \sqrt{\frac{1}{R1'R2'CC1}} \cdot \frac{1}{2\pi}$
- Passband gain (H_0) = (1 + R4/R3) (1 + R4'/R3')
- First stage Q = 1.31
- Second stage Q = 0.541
- Circuit shown uses nearest 5% tolerance resistor values for a filter with a corner frequency of 100 Hz and a passband gain of 1

TL/H/5649–11

Typical Applications (Continued)

Fourth Order High Pass Butterworth Filter



Corner frequency (f_c) = $\frac{0}{2\pi\sqrt{R1R2C^2}}$ = $\frac{0}{2\pi\sqrt{R1'R2'C^2}}$ = 1 kHz

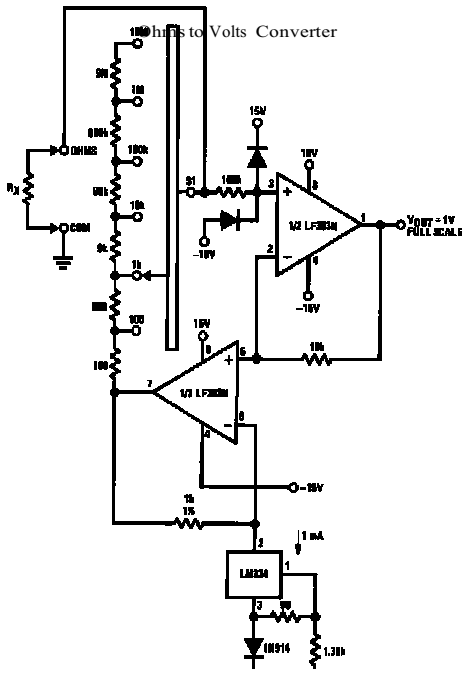
Passband gain (H_O) = (1 + R4/R3) (1 + R4'/R3') = 1.31 and gain of 10.

First stage Q = $\frac{1}{2}$

Second stage Q = $\frac{1}{1}$

Circuit shown uses closest 5% tolerance resistor values for a filter with a corner frequency of 1 kHz and a passb

Ohms to Volts Converter

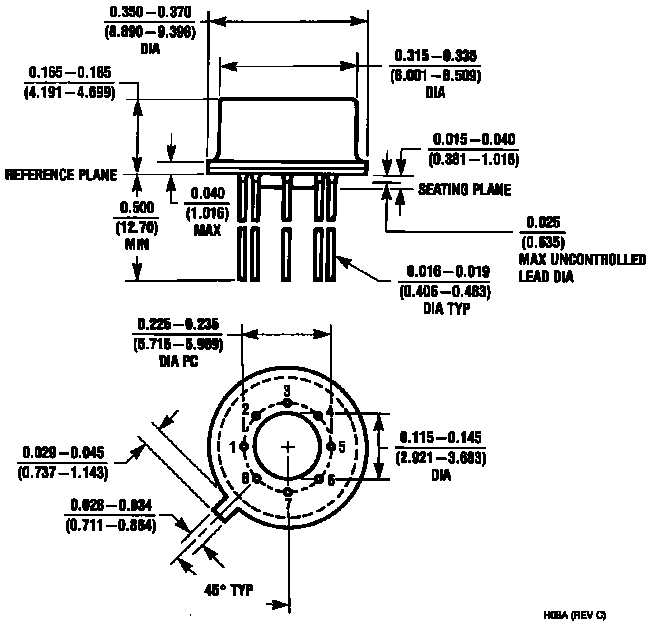


TL/H/5649–13

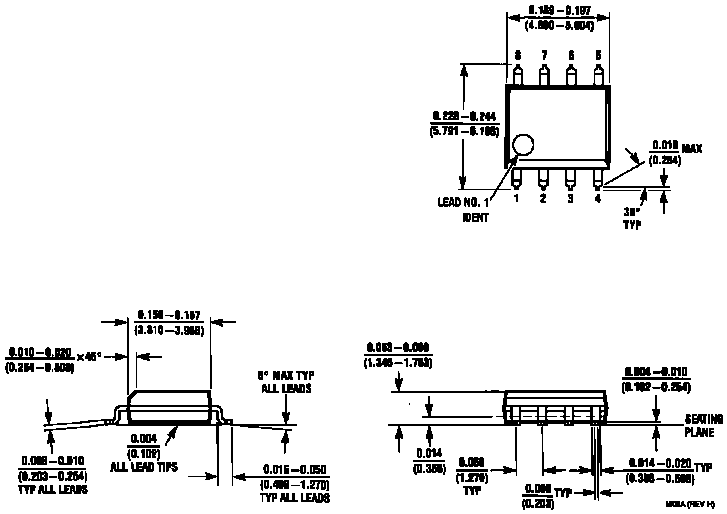
$$V_O = \frac{1V}{R_{LADDER}} \times R_X$$

Where R_{LADDER} is the resistance from switch S1 pole to pin 7 of the LF353.

Physical Dimensions inches (millimeters)

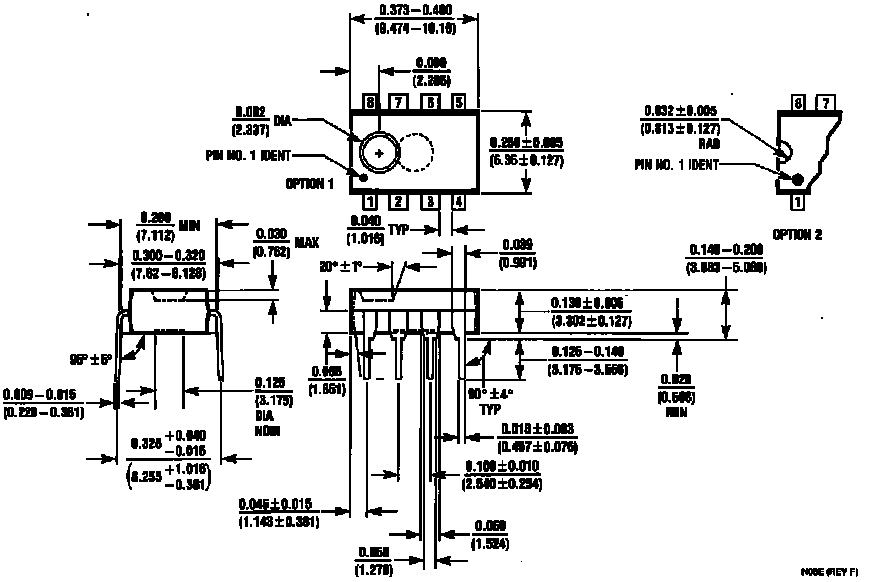


Metal Can
Package (H) Order
Number LF353H
NS Package Number
H08A



Order Number
LF353M NS Package
Number M08A

Physical Dimensions inches (millimeters) (Continued)



Molded Dual-In-Line Package
Order Number LF353N
NS Package N08E

LIFE SUPPORT POLICY

NATIONAL'S PRODUCTS ARE NOT AUTHORIZED FOR USE AS CRITICAL COMPONENTS IN LIFE SUPPORT DEVICES OR SYSTEMS WITHOUT THE EXPRESS WRITTEN APPROVAL OF THE PRESIDENT OF NATIONAL SEMICONDUCTOR CORPORATION. As used herein:

1. Life support devices or systems are devices or systems which, (a) are intended for surgical implant into the body, or (b) support or sustain life, and whose failure to perform, when properly used in accordance with instructions for use provided in the labeling, can be reasonably expected to result in a significant injury to the user.
2. A critical component is any component of a life support device or system whose failure to perform can be reasonably expected to cause the failure of the life support device or system, or to affect its safety or effectiveness.

National Semiconductor Corporation
1111 West Bardin Road
Arlington, TX 76017
Tel: 1(800) 272-9959
Fax: 1(800) 737-7018

National Semiconductor Europe
Fax: (a49) 0-180-530 85 86
Email: cnjwge @ tevm2.nsc.com
Deutsch Tel: (a49) 0-180-530 85 85
English Tel: (a49) 0-180-532 78 32
Fran3ais Tel: (a49) 0-180-532 93 58
Italiano Tel: (a49) 0-180-534 16 80

National Semiconductor Hong Kong Ltd.
13th Floor, Straight Block,
Ocean Centre, 5 Canton Rd.
Tsimshatsui, Kowloon
Hong Kong
Tel: (852) 2737-1600
Fax: (852) 2736-9960

National Semiconductor Japan Ltd.
Tel: 81-043-299-2309
Fax: 81-043-299-2408

MANUAL DE USUARIO PARA LA IMPLEMENTACIÓN DE CÓDIGOS DE LÍNEA EN LA SPARTAN – 3E STARTER KIT BOARD

CONTENIDO

MANUAL DE USUARIO PARA LA IMPLEMENTACIÓN DE CÓDIGOS DE LÍNEA EN LA SPARTAN – 3E STARTER KIT BOARD

I. REQUERIMIENTOS DE HARDWARE Y SOFTWARE

Es necesario tener a disposición los siguientes elementos en cuanto a Hardware y Software se refiere, para la implementación del proyecto.

Hardware

Todos los elementos descritos a continuación, constituyen el banco de trabajo planteado para la implementación y visualización de resultados, se presenta en la Figura E.1.

Figura E.1 Banco de Trabajo.

- Tarjeta de entrenamiento SPARTAN 3E STARTER KIT BOARD
- Circuito externo (contiene un Interpretador Unipolar-Bipolar, un Interpretador Bipolar-Unipolar y un bus de datos con conector tipo Molex macho.)
- 2 fuentes DC (positiva y negativa para la polarización del circuito)
- 2 cables de alimentación para el circuito externo
- Osciloscopio Tektronix TDS 1002B
- 2 puntas de prueba
- Computador con las siguientes características
    - ✓ Sistema operativo Windows XP con Service Pack 3
    - ✓ Dos puerto USB
    - ✓ Memoria RAM mínimo de 512 Mbyte

- ✓ Procesador Pentium 4 o superior

- 1 cable USB-serial.

Software

El computador utilizado para el proyecto debe tener instalado en su sistema operativo los siguientes programas:

- MATLAB 7.1
- Xilinx ISE 10.1

II. INSTALACIÓN DEL HARDWARE

Para conectar la tarjeta de entrenamiento al PC por primera vez, dos controladores deben ser instalados: el Xilinx Embedded Platform USB Firmware Loader y el Xilinx USB Cable. Para la instalación de los dos controladores es necesario seguir los siguientes pasos:

1. Conecte el cable USB de programación de la tarjeta al puerto USB de la PC y enciéndala.

2. Cuando el hardware se conecta por primera vez, el sistema operativo le solicitará al usuario el software para el nuevo dispositivo USB.

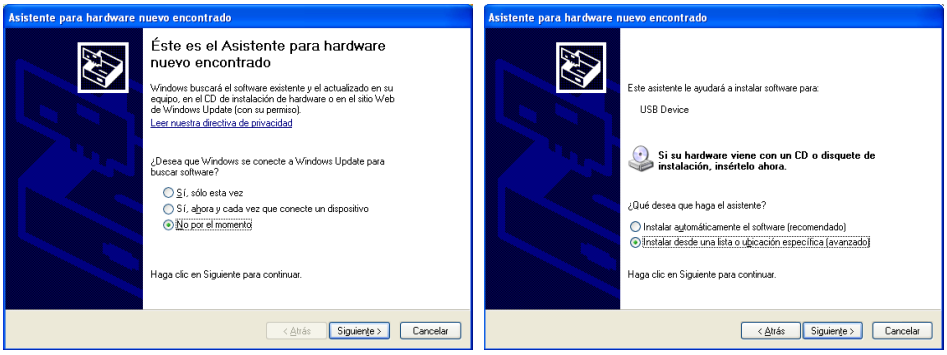


Figura E. 2 Asistente para la instalación de Xilinx Embedded Platform USB Firmware Loader.

3. Seleccionamos la opción de instalar desde una ubicación específica, para esta acción, el CD del instalador o una imagen del programa Xilinx ISE 10.1 debe estar presente en el computador. El controlador se encuentra en la ubicación i:\ise\bin\nt, en este caso ***i*** es la unidad de DVD. Una vez que

se presiona *Siguiente*, el asistente buscará el instalador, y posterior Windows instalará el software.

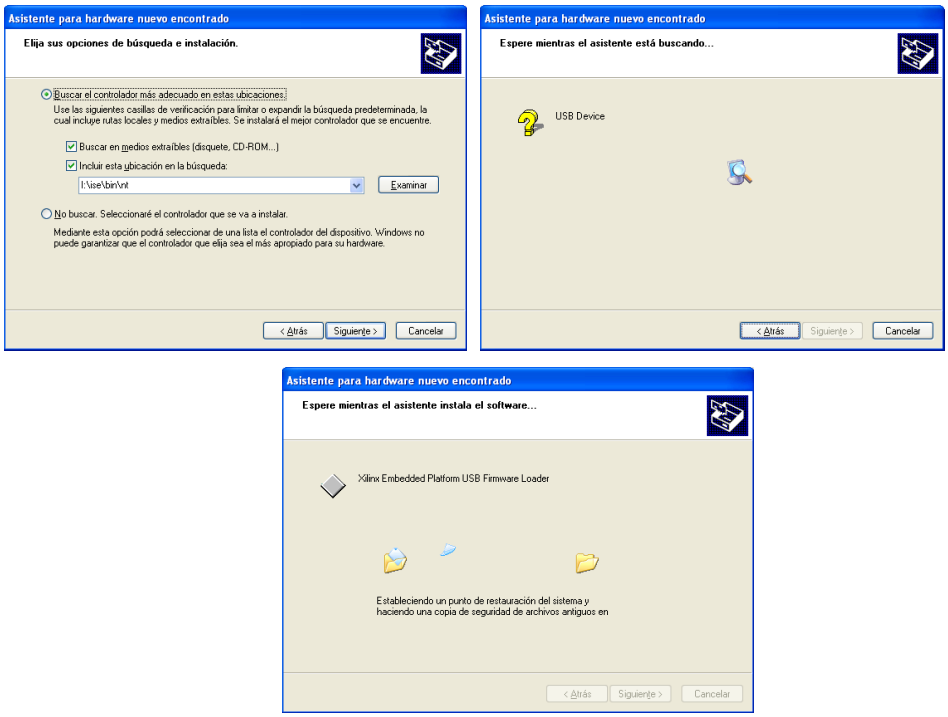


Figura E. 3 Ubicación e instalación de los controladores del hardware.

4. Presionamos Finalizar para terminar la instalación del controlador.

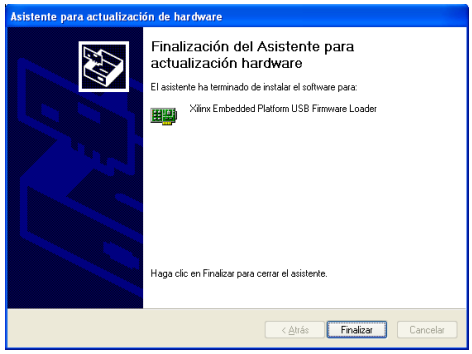


Figura E.4 Finalización de la instalación de Xilinx Embedded Platform USB Firmware Loader.

5. Ahora el sistema operativo solicitará al usuario el software para el Xilinx USB Cable.

6. Repetimos los pasos 3 y 4. Y por último damos click en *Finalizar* para dar por terminado la instalación del Xilinx USB Cable.

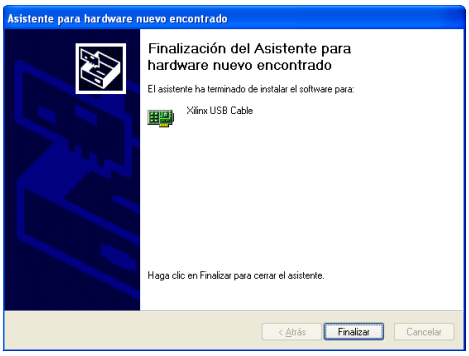


Figura E.5 Finalización de la instalación de Xilinx USB Cable.

III. CONEXIONES

Una vez que se tiene a disposición todos los elementos descritos anteriormente, se procede a hacer las siguientes conexiones:

- Conectar el cable de alimentación de la tarjeta de entrenamiento a la red eléctrica. Alimentación

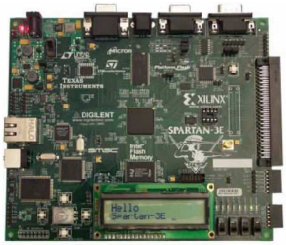


Figura E.6 Alimentación de la Spartan-3E Starter Kit Board.

- Conectar el cable de programación de la tarjeta a un puerto USB del computador (Este cable es de uso exclusivo para programación).

Programación

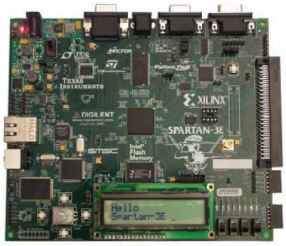


Figura E.7 Puerto de programación de la Spartan-3E Starter Kit Board.

- La tarjeta de entrenamiento descrita, cuenta con un conector DB9 hembra que se conecta a cualquier puerto USB del computador mediante el cable

USB-Serial (para transmisión y recepción de datos). Es necesario instalar el software del cable USB - Serial.

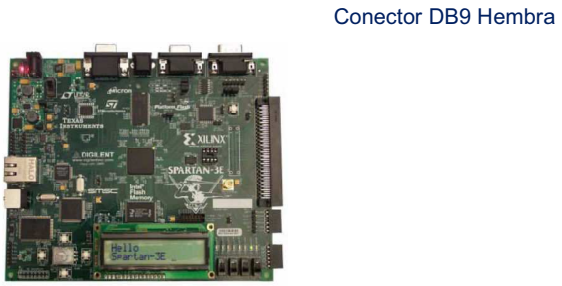


Figura E.8 Conector DB9 hembra de la Spartan-3E Starter Kit Board.

- El bus de datos que viene del circuito externo contiene un conector Molex macho que va a ser conectado al conector J1 de la tarjeta de entrenamiento para la codificación y decodificación de datos. Cada pin del conector representa las siguientes señales de izquierda a derecha tal como se muestra en la Figura E.9: *salida1, salida2, entrada_cod1, entrada_cod2 y GND*.

Figura E.9 Conector tipo Molex macho y Conector J1 de la Spartan-3E Starter Kit Board.

- Por último, conectar los cables de alimentación desde las fuentes positiva y negativa, al circuito externo según corresponda.

IV. VISUALIZACIÓN

- Encienda la tarjeta de entrenamiento y proceda a instalar el hardware (siga los pasos que se encuentran en **INSTALACIÓN DEL HARDWARE**), si es que es la primera vez que se conecta. Las siguientes ocasiones que se use se debe verificar que esté conectado en el *Administrador de Dispositivos* del PC.

- Una vez que esté instalado el hardware, en el computador, abrir la herramienta IMPACT del software ISE 10.1, para esto se ubica en *Inicio, Todos los Programas, Xilinx ISE Design Suite 10.1, Accesories, iMPACT*, tal como se muestra en la Figura E.10.

Figura E.10 Acceso a la herramienta iMPACT de Xilix ISE 10.1.

- Al abrir la herramienta iMPACT, se abre una interfaz donde se despliegan ventanas que ayudarán a la configuración de la programación del dispositivo. En la primera ventana se debe seleccionar *create a new Project (.ipf),* seguido de la opción *Configure devices using Boundary-Scan* en la siguiente ventana y finalizar.

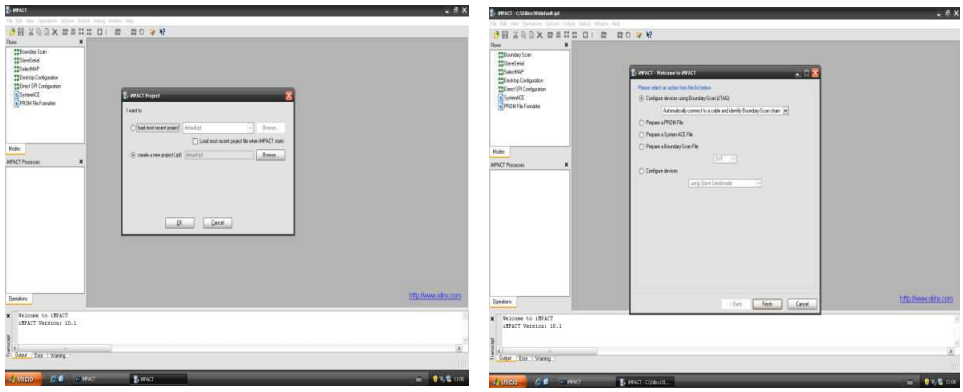


Figura E.11 Configuración de la herramienta iMPACT de Xilix ISE 10.1 para la programación.

- Después de haber seleccionado las configuraciones descritas, esta herramienta despliega los elementos de la tarjeta de entrenamiento que pueden ser programados, como son: el FPGA, una memoria Flash PROM y un CPLD. El proyecto sólo va a hacer uso del FPGA, éste va a ser programado con el archivo ***codigos_de_linea.bit*** generado por el software ISE 10.1; para los restantes elementos que no tienen un archivo de programación se simplemente se salta la configuración seleccionando ***Bypass***. En la Figura E.12 se aprecia el proceso de cargar el archivo en el FPGA.

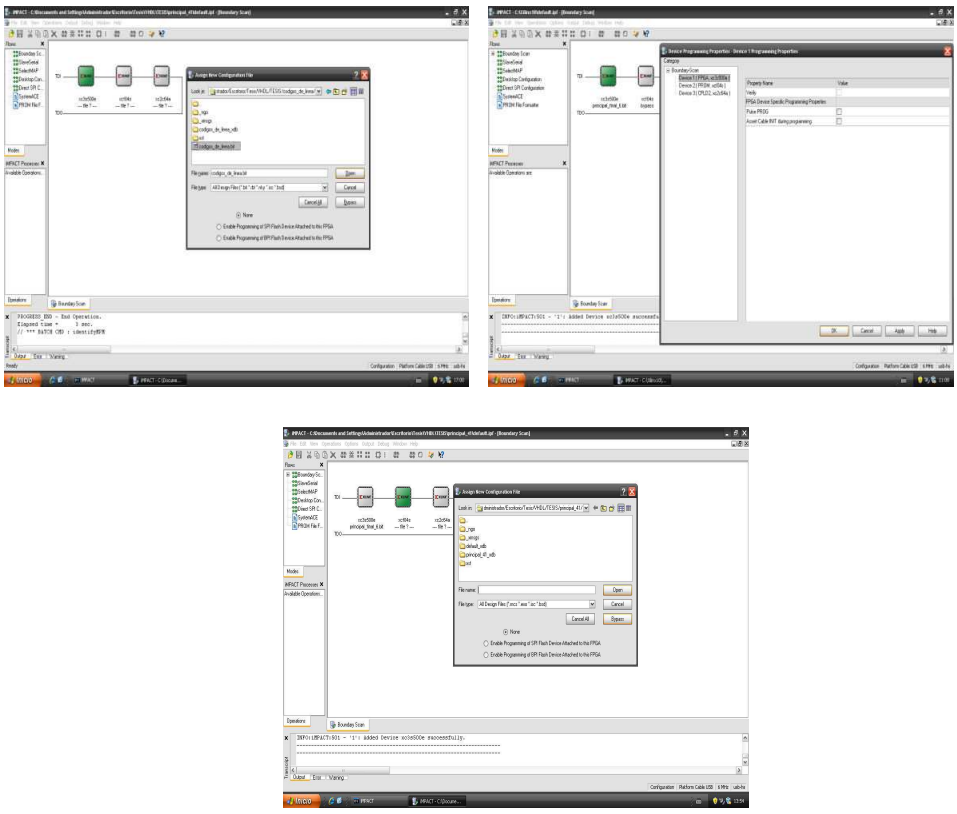


Figura E.12 Selección del archivo para la programación del FPGA.

- Una vez que el archivo de programación está cargado en el FPGA, se hace un click derecho sobre este elemento y se escoge la opción P*rogram.* La programación es exitosa cuando se observa el mensaje en azul *Program Succeeded.*

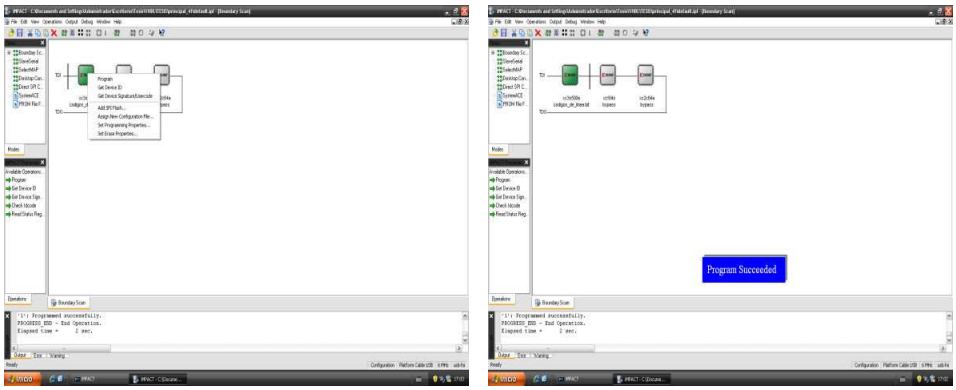


Figura E.13 Programación del FPGA.

- Encender las fuentes DC y el osciloscopio. El osciloscopio utilizado para la visualización de los resultados del proyecto es un Tektronix TDS 1002B.

- Para observar la *señal a codificar* y la *señal codificada* (esto aplica para todos los códigos), las puntas de prueba del osciloscopio deben posicionarse de la siguiente forma: el canal A en el pin E7 (*señal a codificar*), que se encuentra en la tercera posición del conector J2 de la tarjeta de entrenamiento desde el interior hacia el exterior; y el canal B en la salida del circuito Interpretador Unipolar-Bipolar (*señal codificada)*.

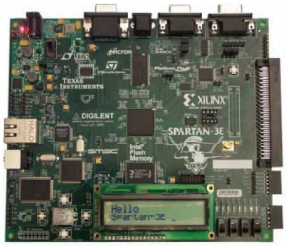


Figura E.14 Conector J2 de la Spartan-3E Starter Kit Board.

- Para visualizar la *señal codificada* y la *señal decodificada*, el canal A debe estar en la salida del circuito Interpretador Unipolar-Bipolar (*señal codificada)*; en tanto que el canal B, se posiciona en el pin D7 (*señal decodificada*), localizado en la primera posición del conector J4 de la tarjeta de entrenamiento.

Conector J4

Figura E.15 Conector J4 de la Spartan-3E Starter Kit Board.

- En cuanto a la visualización de las diferentes señales de reloj, en la Tabla E1 se obtiene la información de los pines donde se encuentran dichas señales. Los pines utilizados se localizan en los conectores J2 y J4 de la tarjeta de entrenamiento. La señal RELOJ_IN se encuentra en la sección de fuentes de reloj de la tarjeta.

Fuentes de Reloj

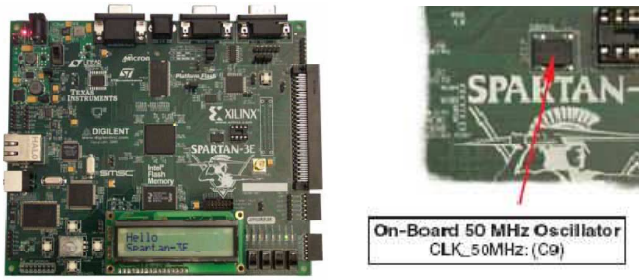


Figura E. 16 Fuentes de reloj de la Spartan-3E Starter Kit Board.

Señales de Reloj	Pines	Descripción
RELOJ_IN	C9 (Oscilador)	Fuente de reloj de 50 MHz de la Spartan-3E Starter Kit Board.
RELOJ_BASE	C7 (Conector J4)	Señal de Reloj de 38400 Hz.
RELOJ_BASE_4B5B	F8 (Conector J4)	Señal de Reloj de 48000 KHz.
RELOL_OUT	A6 (Conector J2)	Señal de reloj para codificación y decodificación de los códigos NRZ, Diferencial Tipo M y S, AMI, HDB3 y MLT3.
RELOJ_OUT_Doble	B6 (Conector J2)	Señal de reloj para codificación y decodificación de los códigos con transición a mitad de tiempo de bit, tales como: RZ, Manchester, Manchester Diferencial y CMI.
RELOJ_OUT_4B5B	E8 (Conector J4)	Señal de reloj para codificación y decodificación del código de línea 4B5B.

Tabla E.1 Ubicación de las Señales de Reloj y su descripción.

V. MANEJO DE LA INTERFAZ GRÁFICA

VERIFICAR PUERTO COM PARA LA COMUNICACIÓN SERIAL

Antes de iniciar la codificación y decodificación con códigos de línea, se debe identificar el puerto de comunicación (COM) que se utiliza para la comunicación serial entre el PC y la Spartan-3E Starter Kit Board. Esto se observa en la Pantalla de Administración de Dispositivos, tal como lo muestra la Figura E.17. En Windows, para observar la Pantalla de Administración de Dispositivos, se selecciona Administrar después de hacer un clic derecho en Mi PC.

El puerto de comunicación habilitado que se conecta a la Spartan-3E Starter Kit Board debe ser el mismo que el usuario selecciona en la Interfaz Gráfica, para este caso el puerto habilitado es *COM4*.

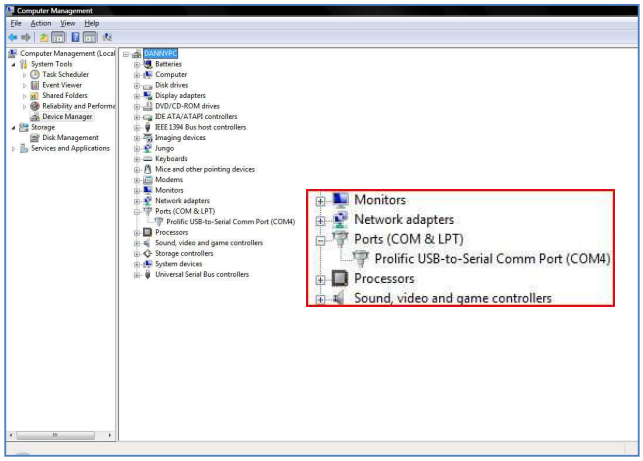


Figura E.17 Ventana de Administración de Dispositivos de Windows Vista.

ACCESO A LAS VENTANAS DE LA INTERFAZ GRÁFICA DE USUARIO

Para acceder a la Interfaz Gráfica se debe:

1. Abrir MATLAB 7.1
2. En la Ventana de Comandos escribir “guide”
3. Seleccionar la pestaña Abrir GUI existente

4. Buscar la carpeta Interfaz Gráfica y abrir el archivo Interfaz1.fig.
5. Seleccionar el botón Run (Correr).

La Interfaz Gráfica de Usuario está conformada de varias GUIs, las mismas que se describen a continuación.

1. Presentación

Esta pantalla, contiene dos botones:

- **Continuar**, permite desplegar la pantalla de configuración e ingreso de datos.
- **Salir**, permite cerrar la Interfaz Gráfica.

Figura E.18 Presentación de la Interfaz Gráfica.

2. Configuración e Ingreso de Datos

Esta pantalla consta de dos secciones, **Configuración** e **Introducción de Datos**, tal como lo muestra la Figura E.19.

Figura E.19 Configuración e Ingreso de Datos.

En la sección **Configuración**, en **Puerto COM** se selecciona el puerto de comunicación (COM) con el cual el PC se comunica con la Tarjeta de Entrenamiento; en **Velocidad de Transmisión** se elige la velocidad de transmisión a la que se enviarán los datos a ser codificados a la Spartan 3E Starter Kit Board, entre: 2400, 4800, 9600 o 19200 bps. El usuario puede seleccionar cualquiera de las velocidades antes mencionadas. Al **Aceptar** estos dos parámetros, se envía a la Tarjeta de Entrenamiento la configuración seleccionada. Para verificar que las señales de reloj se encuentran debidamente configuradas se las puede observar de acuerdo a la Tabla E.1, por ejemplo la señal RELOJ_OUT en el pin A6.

Si se requiere cambiar la velocidad de transmisión o el puerto de comunicación se debe accionar el interruptor deslizante Reset (pin N17) en la Spartan-3E Starter Kit Board y regresarlo a su posición inicial, antes de cambiar los parámetros requeridos.

En la sección **Introducción de Datos** se debe ingresar los caracteres cuyos valores binarios van a ser codificados en la Tarjeta de Entrenamiento. Al presionar el botón **Tabla ASCII** se despliega una pantalla de caracteres ASCII con su correspondiente valor binario, tal como lo muestra la Figura E.20. El botón **Mostrar Binario** permite aceptar los caracteres introducidos.

Bits				5	0	1	0	1	0	1	0	1
				6	0	0	1	1	0	0	1	1
				7	0	0	0	0	1	1	1	1
1	2	3	4									
0	0	0	0		NUL	DEL	SP	0	@	P	‘	p
1	0	0	0		SOH	DC1	¡	1	A	Q	a	q
0	1	0	0		STX	DC2	“	2	B	R	b	r
1	1	0	0		ETX	DC3	#	3	C	S	c	s
0	0	1	0		EOT	DC4	%	4	D	T	d	t
1	0	1	0		ENQ	NAK	$	5	E	U	e	u
0	1	1	0		ACK	SYN	&	6	F	V	f	v
1	1	1	0		BEL	ETB	‘	7	G	W	g	w
0	0	0	1		BS	CAN	(	8	H	X	h	x
1	0	0	1		HT	EM	)	9	I	Y	i	y
0	1	0	1		LF	SUB	*	:	J	Z	j	z
1	1	0	1		VT	ESC	+	;	K	[	k	{
0	0	1	1		FF	FS	,	<	L	\	l	|
1	0	1	1		CR	GS	-	=	M	]	m	}
0	1	1	1		SO	RS	.	>	N	^	n	~
1	1	1	1		SI	US	/	?	O	-	o	DEL

Figura E.20 Tabla ASCII.

El botón **Atrás** cierra la pantalla actual y abre la pantalla de presentación de la Figura E.18. El botón **Selección de Código** permite abrir la pantalla de la Figura E.21.

3. Selección de Código de Línea

En esta ventana se debe escoger el código de línea que se implementará en el FPGA. Existen tres botones de selección: Unipolar, Polar y Bipolar. Al seleccionar **Unipolar** se puede escoger uno de los siguientes códigos de línea: NRZ, RZ al 50% o 4B5B. Al seleccionar **Polar** se puede escoger: Diferencial tipo M, Diferencial tipo S, Manchester, Manchester Diferencial o CMI. Al elegir **Bipolar** se puede escoger: AMI, HDB3 o MLT3.

Figura E.21 Selección del Código de Línea

Se puede observar tres marcos de botones: Generales, Simulación y Hardware.

En el marco **Generales**, una vez seleccionado el Código de Línea, se puede presionar los siguientes botones:

- ✓ **Atrás**, abre la pantalla de Configuración e Ingreso de Datos.
- ✓ **Información del Código**, visualiza una reseña del código de línea escogido.
- ✓ **Espectro Teórico,** abre una pantalla que muestra el espectro teórico luego de la codificación.
- ✓ **Salir,** cierra la interfaz gráfica.

En el marco **Simulación**, se puede presionar el botón:

- ✓ **Graficar Señales a Codificar y Codificada,** presenta una ventana que incluye tres gráficas: la señal de reloj, la señal a codificar (incluye el identificador de inicio y fin de datos) y la señal codificada con el código de línea seleccionado.

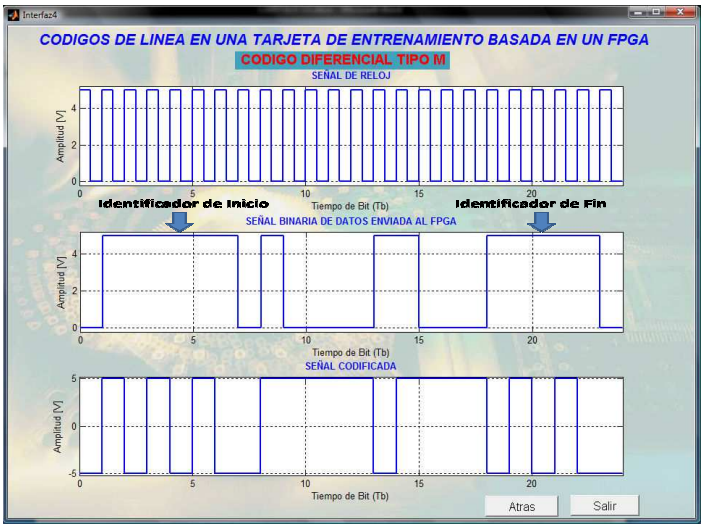


Figura E.22 Gráfica de la Señal de Reloj, Señal a Codificar y Señal Codificada.

En el marco **Hardware**, se puede presionar los botones:

- ✓ **Botón Enviar Datos al FPGA para la Codificación:** envía la información introducida por el usuario mediante el puerto RS-232 a la Tarjeta de Entrenamiento.
- ✓ **Botón Ver datos recibidos Decodificados:** abre la ventana Decodificación.

4. Decodificación

En la ventana de la Figura E.24, se mostrará los caracteres enviados para la codificación, los caracteres recibidos después de la decodificación siempre que se hayan recibido los datos decodificados desde la Spartan 3E Starter Kit Board. Además se visualiza la velocidad de transmisión y la velocidad de señal.

Para enviar la señal decodificada de la Spartan 3E Starter Kit Board a la Interfaz Gráfica, y visualizar los resultados tal como se muestra en la Figura E.24, se debe accionar el interruptor deslizante L14 de la Tarjeta de Entrenamiento.

Interruptor Deslizante L14

Figura E.23 Interruptor Deslizante L14.

Figura E.24 Visualización de Caracteres Enviados y Recibidos del FPGA.

Esta ventana contiene tres botones:

- **Atrás**, permite abrir la ventana Selección de Código de Línea, Figura E.21.
- **Siguiente**, permite visualizar la ventana de la Figura E.25.
- **Salir,** cierra la interfaz gráfica.

Figura E.25 Gráfica de la Señal de Reloj, Señal Enviada y Señal Recibida del FPGA.

La ventana de la Figura E.25 muestra tres gráficas: la señal de reloj, la señal enviada, y la señal recibida del FPGA. Esta ventana tiene tres botones:

- **Ingresar Nuevos Datos**, abre la ventana de **Configuración e Ingreso de Datos** con el propósito de iniciar una nueva codificación.
- **Atrás**, abre la ventana Decodificación, Figura E.24.
- **Salir**, cierra la interfaz gráfica.

UNA NUEVA CODIFICACIÓN Y DECODIFICACIÓN

Cada vez que se inicie una nueva codificación, se debe accionar el interruptor deslizante Reset (pin N17) de la tarjeta de entrenamiento y regresarlo a su posición inicial (0L), verificar que el interruptor deslizante de envío de datos a Matlab (pin L14) esté en 0L; posterior a esto se puede repetir todo el procedimiento que concierne al Manejo de la Interfaz Gráfica de Usuario.

Interruptor Deslizante N17

Figura E.26 Interruptor Deslizante N17.