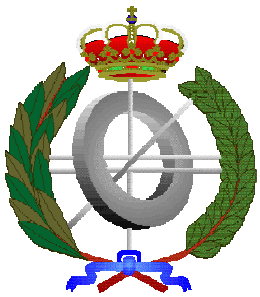


EMULADOR HARDWARE PARA ARQUITECTURAS RECONFIGURABLES DE GRANO GRUESO

Autores: Laura Díaz Escalona
Mª Concepción Fernández Sánchez
Inés González de Miguel

Profesor: Marcos Sánchez-Elez Martín

Curso 2005 – 2006
Proyecto de Sistemas Informáticos

Facultad de Informática
Universidad Complutense de Madrid

1. Resumen

Este proyecto trata de de implementar los algoritmos de codificación y decodificación necesarios para la correcta transmisión del audio y video en la Televisión Digital Terrestre, alcanzando un rendimiento óptimo y un aprovechamiento máximo de los recursos mediante la utilización de segmentación.

Los estándares de Televisión Digital vienen recogidos en la organización DVB (Digital Video Broadcasting), que promueve los estándares aceptados internacionalmente de televisión digital, en especial para HDTV y televisión vía satélite, así como para comunicaciones de datos vía satélite. En particular, el sistema tomado como referencia en este proyecto es el DVB-T (Digital Video Broadcasting – Terrestrial), estándar europeo de DVB para la transmisión de televisión digital terrestre, que transmite un flujo de datos comprimidos de video y audio usando modulación OFDM con codificación de canal encadenada.

En dicho sistema, con vistas a lograr una correcta transmisión de los datos, la codificación (convolucional) previa en el emisor y la decodificación (Viterbi) en el receptor son fundamentales, y como ya se ha dicho, han sido el objetivo de este proyecto.

Hasta alcanzar el resultado final óptimo, el proyecto pasó por varias etapas de implementación. Inicialmente, el desarrollo fue en JAVA, ya que este lenguaje aporta un mayor nivel de abstracción, clarificando el funcionamiento de los distintos algoritmos.

A continuación se segmentó el sistema, con vistas a lograr un mayor aprovechamiento de los recursos existentes, y una consiguiente mejora de rendimiento.

El algoritmo de decodificación Viterbi queda dividido en tres módulos que permiten la ejecución paralela de más de una trama siempre y cuando no se utilicen recursos en común, es decir, mientras cada una de las tramas esté en una etapa distinta de la segmentación.

De este modo, los resultados obtenidos con la versión final del sistema segmentado parecen ofrecer buenos resultados para la transmisión de mensajes desde el emisor al receptor pudiendo éstos haber sufrido modificaciones debido a la presencia de ruido gaussiano blanco (AWGN).

2. Abstract

The aim of this proyect is to implement the encoding and decoding algorithms which are necessary for the correct audio and video data transfer in digital terrestrial television, maximizing the exploitation of resources by the use of pipeline segmentation, which leads to optimal performance.

The standards of digital television are stored by DVB (**Digital Video Broadcasting**), an organization that promotes the internationally accepted open standards for digital television, especially for HDTV and satellite television, as well as for communications through satellite. Particularly, the system this proyect is based on is DVB-T (Digital Video Broadcasting – Terrestrial), European consortium standard for the broadcast transmission of digital terrestrial television. This system transmits a compressed digital audio/video stream, using OFDM modulation with concatenated channel coding.

In this System, in order to achieve a correct data transfer, a previous convolutional encoding process in the transmitter and a Viterbi decoding process in the receiver are fundamental, and as mentioned before, have been the main goals of this proyect.

Before the final optimal version was reached, the project evolved through different stages of implementation. Initially, it was developed in JAVA because this language helped to clarify concepts about the different algorithm's operations.

Next, the system was pipelined to obtain a better exploitation of resources and a consequent better overall performance.

The Viterbi decoding algorithm is divided in three modules allowing a parallel execution of more than one message whenever there are not commonly used resources, i.e., while each of the messages are in a different stage of the pipeline process.

Finally, results obtained from the pipelined version of our proyect seam to give good results for the transfer of messages from a transmitter to a reciever even when these messages have suffered alterations due to the presence of Additive White Gaussian Noise (AWGN).

3. Palabras Clave

- Viterbi
- VHDL
- Grano grueso
- Decodificador
- Codificador convolucional

ÍNDICE

Pág

4. Descripción del problema

DVB (**D**igital **V**ideo **B**roadcast) es un conjunto de estándares internacionales para la transmisión de la televisión digital. Los sistemas DVB distribuyen datos por:

- Satélite (DVB-S)
- Cable (DVB-C)
- Televisión Terrestre (DVB-T)
- Televisión Terrestre para dispositivos móviles (DVB-H)

Nos vamos a centrar en DVB-T. Este sistema transmite un flujo de datos comprimido de audio y video digital. Fue más complejo de desarrollar porque pretendía que fuera soportado por entornos de distinto ancho de banda y ruido.

A continuación mostramos un esquema de los bloques que forman parte del emisor:

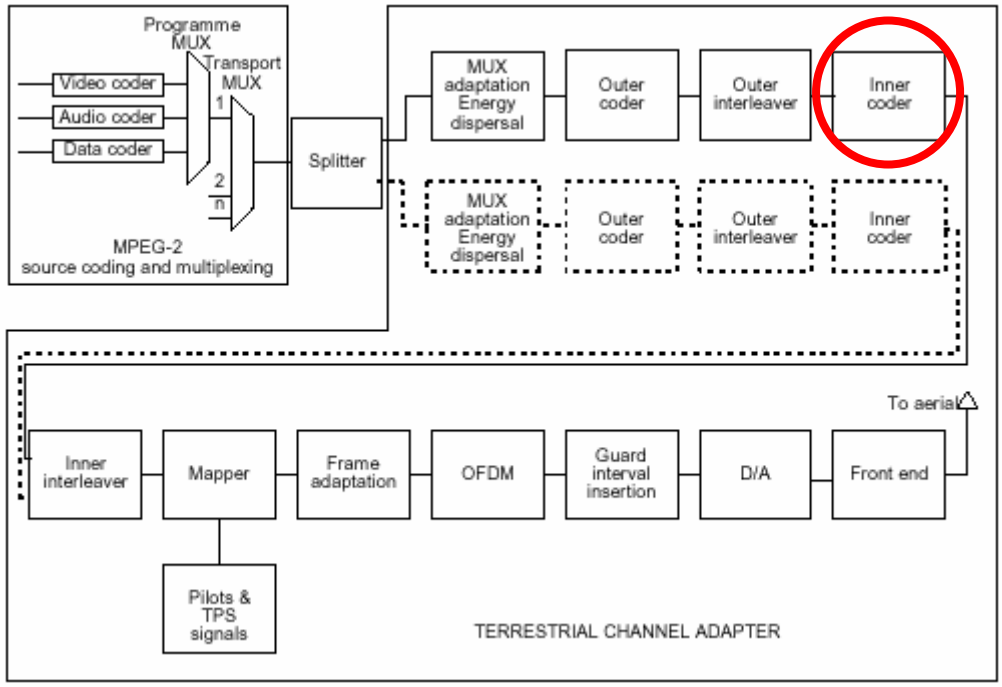


Figura 4.1

En este proyecto, únicamente vamos a ver con detenimiento el codificador interno del emisor y el decodificador interno del receptor.

El codificador interno es el segundo nivel de protección. Esta codificación viene dada a partir de una codificación convolucional. La codificación convolucional es un tipo de código de corrección de errores, donde cada símbolo de k bits es transformado en un símbolo de n bits (ratio k/n).

El esquema general del receptor es el de la figura:

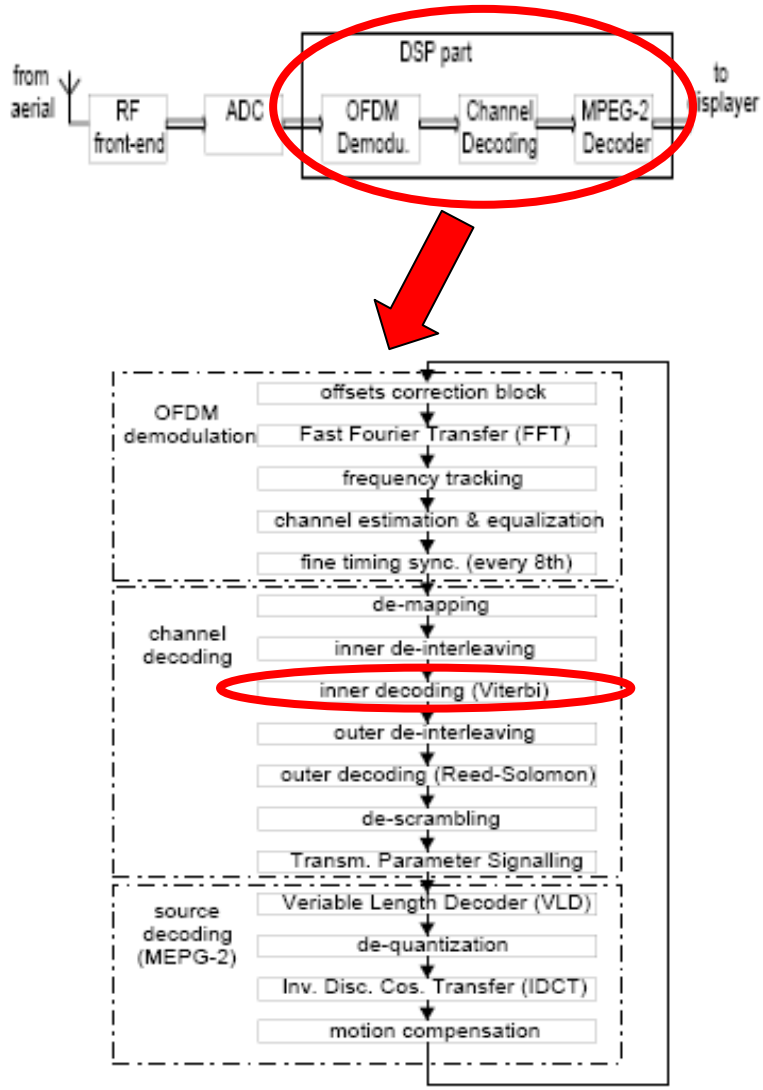


Figura 4.2

El receptor recibe una señal digital y realiza tareas tales como una desmodulación OFDM, una decodificación del cana y una decodificación MPEG-2. De todos estos pasos que tiene que tener el receptor nos centraremos, como hemos dicho, en la decodificación interna, realizada mediante el algoritmo de Viterbi.

El codificador convolucional se ha implementado en JAVA con un ratio de ½ (k/n), pero con la posibilidad de poder ser ampliado fácilmente. Elegimos este ratio puesto que comprobamos que la probabilidad de error disminuía. Cuanto mayor sea n, más segura será la transmisión y menor será la probabilidad de error, sin embargo, será más compleja la decodificación puesto que el diagrama de estados del codificador aumentará notablemente.

Para mejorar el rendimiento del decodificador una vez ya implementado en JAVA, se propuso hacer una implementación segmentada realizada en JAVA y también en VHDL.

5. Descripción del sistema

Cualquier mensaje que deba ser transmitido por un canal de comunicación, debe pasar ciertas etapas que podemos clasificar de la siguiente forma:

- Codificación
- Introducción de ruido blanco
- Decodificación

A continuación pasamos a describir las etapas mencionadas de forma detallada.

5.1. Codificación de datos: Codificador convolucional.

El proceso de codificación, se encarga de convertir los bits del mensaje introducido en símbolos que puedan ser transmitidos a través del canal.

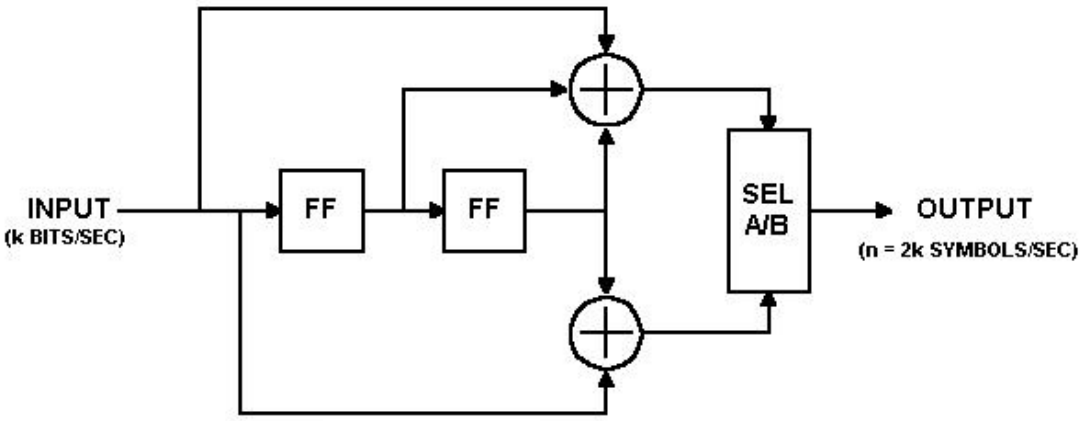


Figura 5.1.1

Un código convolucional queda especificado por tres parámetros (n, k, m):

- n es el número de bits de la palabra codificada.
- k es el número de bits de la palabra de datos.
- m es la memoria del código o longitud restringida

El número de bits por palabra de datos k, cumple: k/n = R (1)

A este cociente se le denomina ratio del codificador. Definimos para este proyecto un codificador convolucional (2,1,3), es decir: un bit para representar la palabra de datos, dos bits de palabra de codificación por cada bit de palabra de datos y tres bits de longitud de registro. En nuestro caso el ratio es ½.

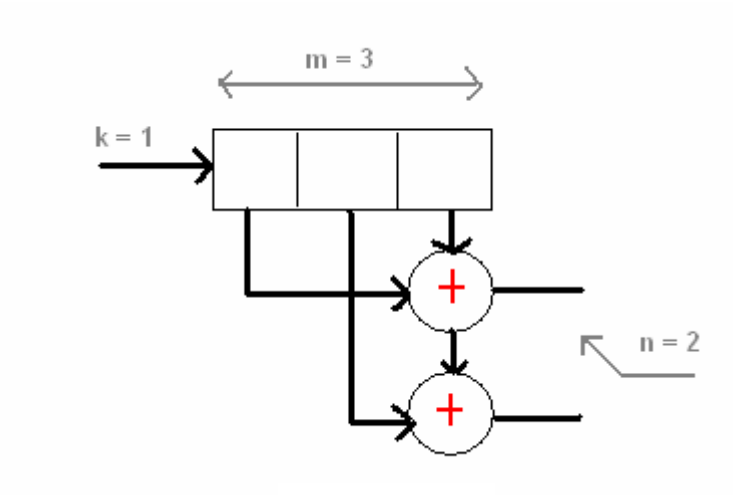


Figura 5.1.2

Cuanto mayor sea n, más segura será la transmisión y menor será la probabilidad de error, eso sí, del mismo modo, más complejo será el diagrama de estados del codificador y más difícil su posterior decodificación. Hay que encontrar, como siempre en transmisión, el compromiso idóneo entre calidad y complejidad.

Un codificador convolucional puede ser fácilmente representado por un diagrama de estados de transición ya que tiene memoria finita. Para ello, procedemos al desarrollo de los diferentes estados de la máquina. Fijamos que en el instante inicial el codificador está cargado con todo ceros, veamos los distintos estados posibles en las figuras 5.1.3 a y 5.1.3 b:

Cargado todo con ceros, le metemos un cero y un uno:

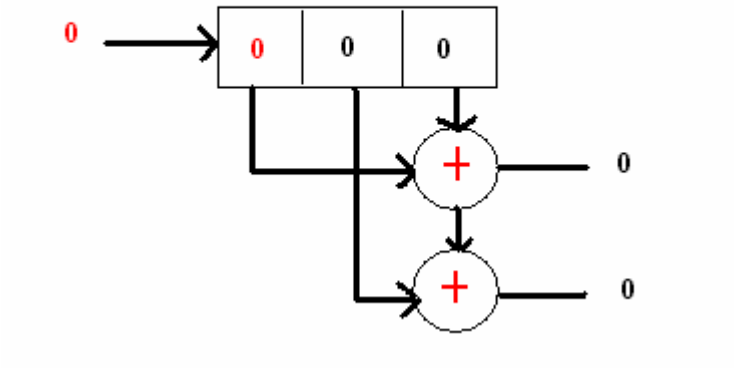


Figura 5.1.3 a

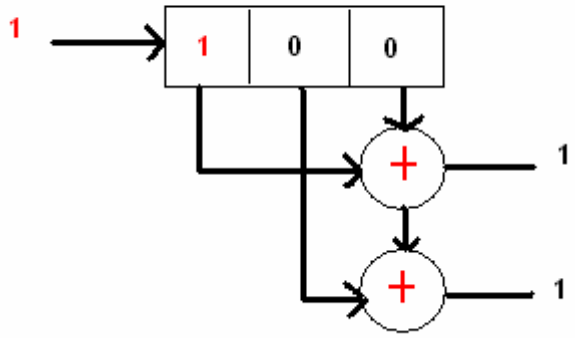


Figura 5.1.3 b

Seguimos con el caso de que hemos introducido un uno porque con el cero observamos que se forma un bucle. Ahora en la situación (1,0,0) veamos cómo codifica el sistema si le introducimos un cero y cómo si es un uno el bit que entra al canal:

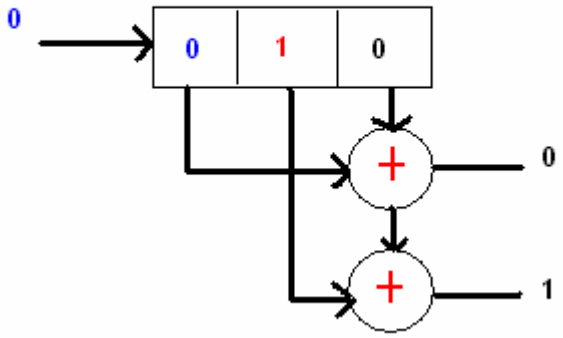


Figura 5.1.4 a

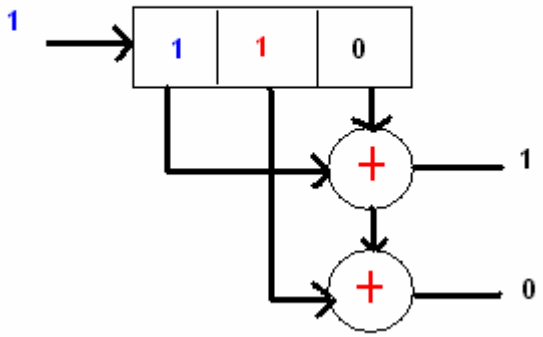


Figura 5.1.4 b

Y así sucesivamente hasta completar un ciclo y con él, la máquina de estados. En este diagrama, cada estado del codificador convolucional se representa mediante una caja y las transiciones entre los estados vienen dadas por líneas que conectan dichas cajas. Para saber de una manera rápida en que estado se encuentra el codificador, basta con observar los dos bits del registro de memoria más alejados de la entrada. Puede comprobarse interpretando la máquina de estados correspondiente.

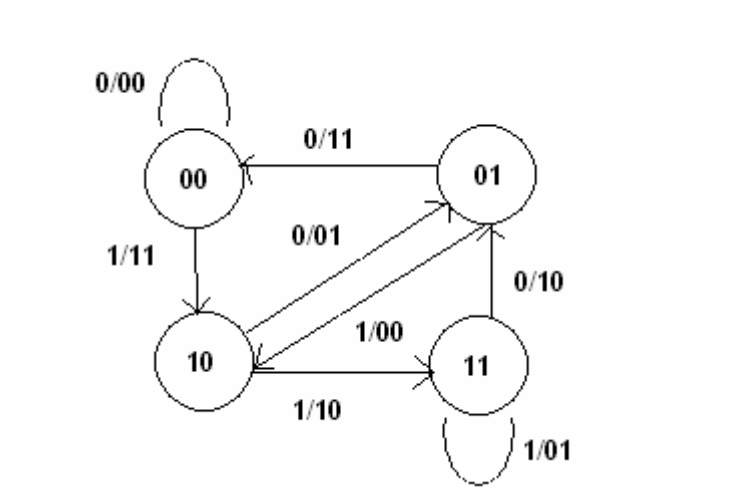


Figura 5.1.5

En cada línea viene representada la salida en función de la entrada. El número de líneas que salen de cada estado es, por lo tanto, igual al número de posibles entradas al codificador en ese estado, que es igual a 2^k.

Una manera de representar las distintas transiciones y los caminos que éstas describen es mediante un ***Diagrama de Trellis***. Una descripción de Trellis de un codificador convolucional muestra cómo cada posible entrada al codificador influye en ambas salidas y a la transición de estado del codificador. Un código de longitud restringida m tiene un Trellis con 2^{m-1} estados en cada intervalo t_i. Así que tendremos cuatro estados en t_i. De cada estado parten otros dos, uno si el bit enviado es un '1' y otro si es un '0'.

5.2. Introducción de ruido blanco AWGN

La finalidad de la transmisión reside en que el receptor reciba exactamente lo que se le envía desde un emisor dado. El principal problema se encuentra en que en el canal se le añade un ruido aleatorio que tomaremos en nuestro caso AWGN (Additive White Gaussian Noise) y, en consecuencia, necesitaremos un proceso mediante el que podamos decidir qué mensaje, de los posibles, ha sido enviado.

El ruido es la señal adicional "no deseada" que interfiere con la señal transmitida. Generalmente, el proceso del ruido se considera “aditivo blanco gaussiano” (AWGN)

– Blanco: espectro de frecuencia plano
– Gaussiano: distribución del ruido

Podemos decir por tanto, que ruido blanco se caracteriza por el hecho de que su valor en dos momentos cualesquiera no es correlativo. Esto motiva a este ruido a tener una potencia de densidad de espectro plana (en potencia de señal por hertzio de ancho de banda) y su pérdida análoga a la "luz blanca" que tiene una potencia de densidad de espectro plana con respecto a la longitud de onda.

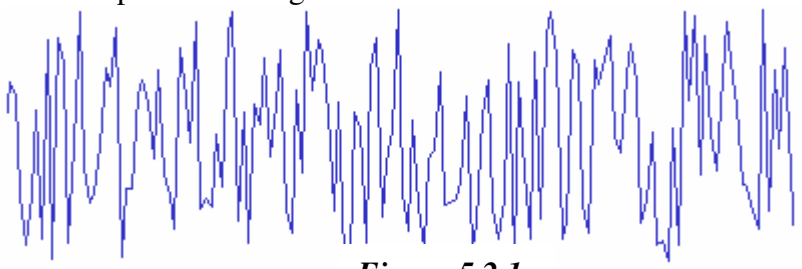


Figura 5.2.1

Hemos simulado el ruido existente en cualquier canal de comunicación, alterando aleatoriamente alguno de los símbolos que se transmiten por el canal.

De esta situación surge la cadena de símbolos que finalmente deberá decodificar el algoritmo de decodificación Viterbi.

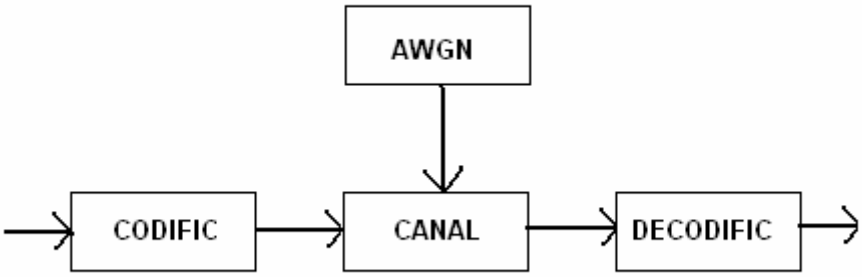


Figura 5.2.2

5.3. Decodificación: Algoritmo de decodificación Viterbi

En esta etapa del proceso, los símbolos transmitidos por el canal, son reconvertidos en el mensaje original una vez han alcanzado su destino.

Para conseguir descifrar el mensaje original a partir de los símbolos que recibimos del codificador pudiendo éstos haber sufrido alteraciones debido a la presencia de ruido blanco, necesitamos hacer uso de las tablas de estado siguiente y de salida mostradas en las figuras 5.3.1 a y 5.3.1 b.

	Estado Siguiente	
Estado actual	**Bit de entrada = 0**	**Bit de entrada = 1**
00	00	10
01	00	10
10	01	11
11	01	11

Figura 5.3.1 a

	Símbolos de salida	
Estado actual	**Bit de entrada = 0**	**Bit de entrada = 1**
00	00	11
01	11	00
10	10	01
11	01	10

Figura 5.3.1 b

La forma de entender la descodificación de manera adecuada se ve detalladamente gracias a los llamados ***diagramas de Trellis***.

La figura 5.3.2 muestra el diagrama de Trellis para un ejemplo de 15 bits de entrada:

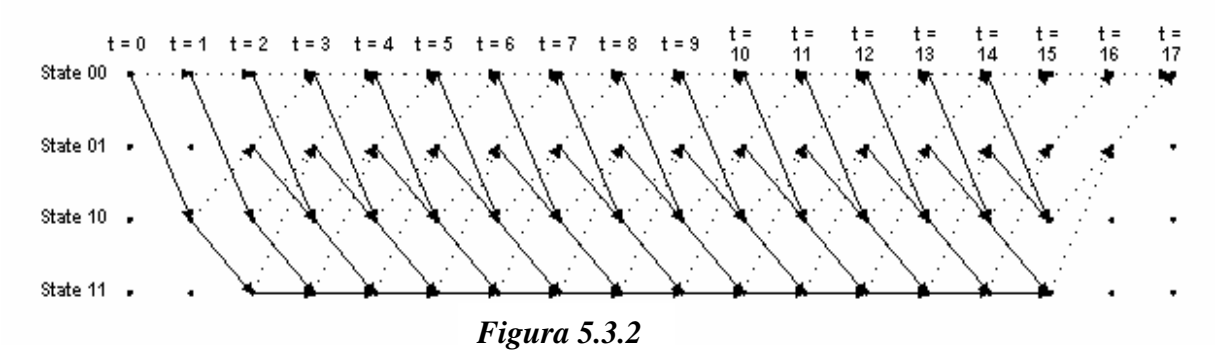


Figura 5.3.2

Como se puede observar, los 4 posibles estados se muestran como filas de puntos horizontales. Encontramos una columna de 4 puntos para el estado inicial del codificador y una mas para cada instante de tiempo durante la transmisión del mensaje.

Para un mensaje de 15 bits como el mostrado en el ejemplo, con dos "flushing bits" obtenidos en la codificación, habrá 17 instantes de tiempo además del instante t=0 que representa la condición inicial del codificador.

Además, en el ejemplo, las líneas punteadas representan la transición de estado cuando el BIT que llega es un cero, y las líneas continuas, cuando el BIT es un 1.

Se puede observar también que las transiciones de estado del diagrama de trellis se corresponden exactamente con las tablas de estado siguiente y símbolo de salida mostradas anteriormente, y que dado que el estado inicial del codificado es 00 y que los dos "flushing bits" son 0, el diagrama comienza en el estado 00 y acaba en este mismo estado.

La figura 5.3.3 muestra los estados que se alcanzan en cada instante de tiempo para un mensaje de 15 bits como el siguiente:

Mensaje original 0 1 0 1 1 1 0 0 1 0 1 0 0 0 1 **0 0** → marcados en rojo los flushing bits

Salida del codificador: 00 11 10 00 01 10 01 11 11 10 00 10 11 00 11 10 11

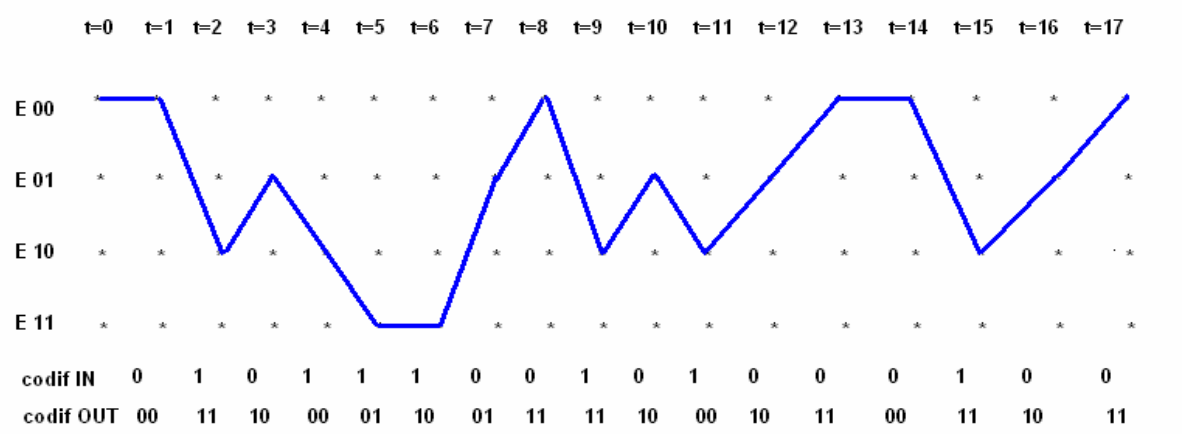


Figura 5.3.3

A continuación, vamos a analizar los pasos de este diagrama de forma más detallada viendo lo que ocurre entre dos instantes de tiempo.

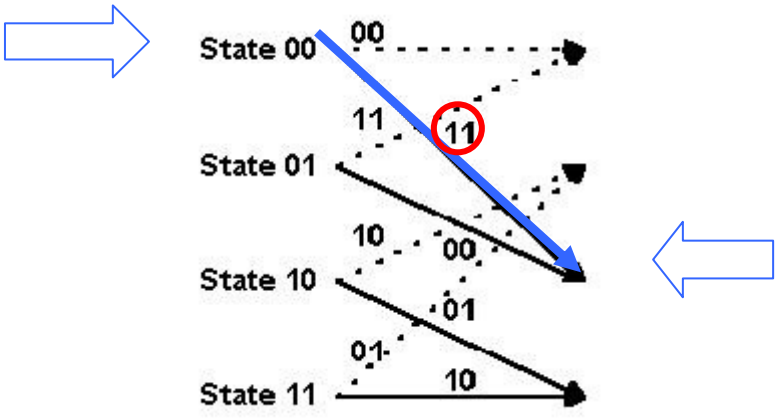


Figura 5.3.4

El numero de dos bits que se encuentra encima de las flechas de transición, es el correspondiente al símbolo de salida que se produce en una transición de un estado X a un estado Y.

Por ejemplo, estando en el estado 0, si el BIT de entrada es un uno, la transición será al estado 2, y el símbolo producido será el 11.Recordemos que las líneas punteadas representan transiciones en las que el bit de entrada es un cero y las líneas continuas en las que le bit de entrada es un uno.

Pero la mejor forma de entender esto, y de implementarlo en nuestro caso, fue utilizando el algoritmo de descodificación para un ejemplo concreto de 15 bits.

Supongamos que recibimos el siguiente mensaje codificado con unos cuantos errores producidos por la presencia de ruido blanco:

Salida del cod: 00 11 10 00 01 10 01 11 11 10 00 10 11 00 11 10 11
Símbolos recibidos: 00 11 11 00 01 10 01 11 11 10 00 00 11 00 11 10 11

Cada vez que recibimos un par de símbolos, vamos a medir la "distancia" entre lo que hemos recibido, y todos los posibles pares de símbolos que podíamos haber recibido.

Por ejemplo: entre los instantes t=0 y t=1, sabiendo que el estado inicial es 00 y gracias a las tabla de salida, sabemos que los únicos dos símbolos que podíamos haber recibido son el 00 si el BIT de entrada es 0 o el 11 si es 1. De esta forma, vamos a medir la distancia entre estos símbolos y el símbolo realmente recibido usando para ello la distancia de Hamming .

La distancia de Hamming de obtiene contando el numero de bits diferentes que hay entre el símbolo recibido y los símbolos que podían haberse recibido en consecuencia de la transición de estado.

Así pues, la distancia de Hamming para t=1 y teniendo en cuanta que el símbolo recibido es el 00 será:

Estado i-1	**Estado i**	**Símbolo recibido**	**Símbolo posible**	**Distancia Hamming**
00	00	00	00	0
00	10	00	11	2

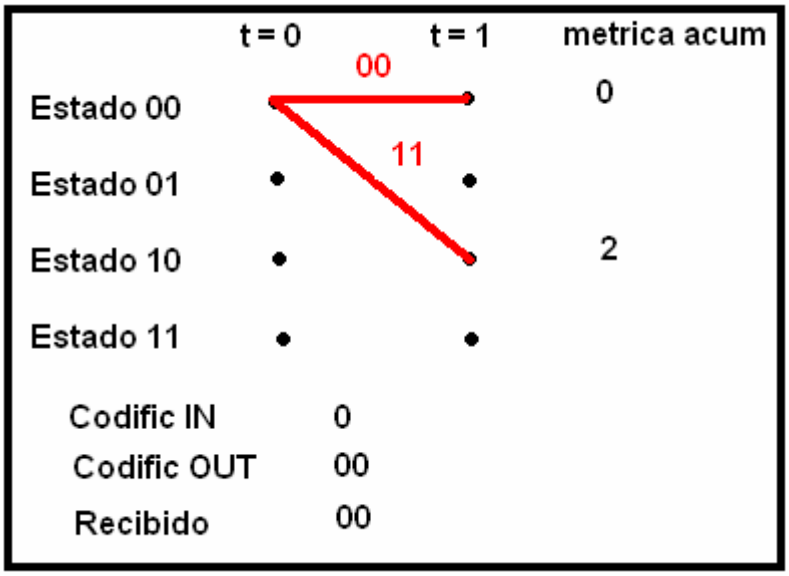


Figura 5.3.5

Haciendo lo mismo para t=2, teniendo en cuanta que el símbolo recibido es 11, y sabiendo que podemos estar tanto en el estado 00 como en el 10, los posibles símbolos serán:

Estado i-1	**Estado i**	**Símbolo recibido**	**Símbolo posible**	**Distancia Hamming**
00	00	00	00	0
00	10	00	11	2
10	01	11	10	1
10	11	11	01	1

Hay que destacar que el error métrico acumulado se corresponde con la suma del error acumulado en los estados anteriores más el error acumulado para el instante actual.

Ejemplo:

El estado 11, tiene como predecesor al estado 10, que tenia un error acumulado de 2. Si sumamos a este error el error actual que es 1, el error medio acumulado es 3, como muestra la figura 5.3.6:

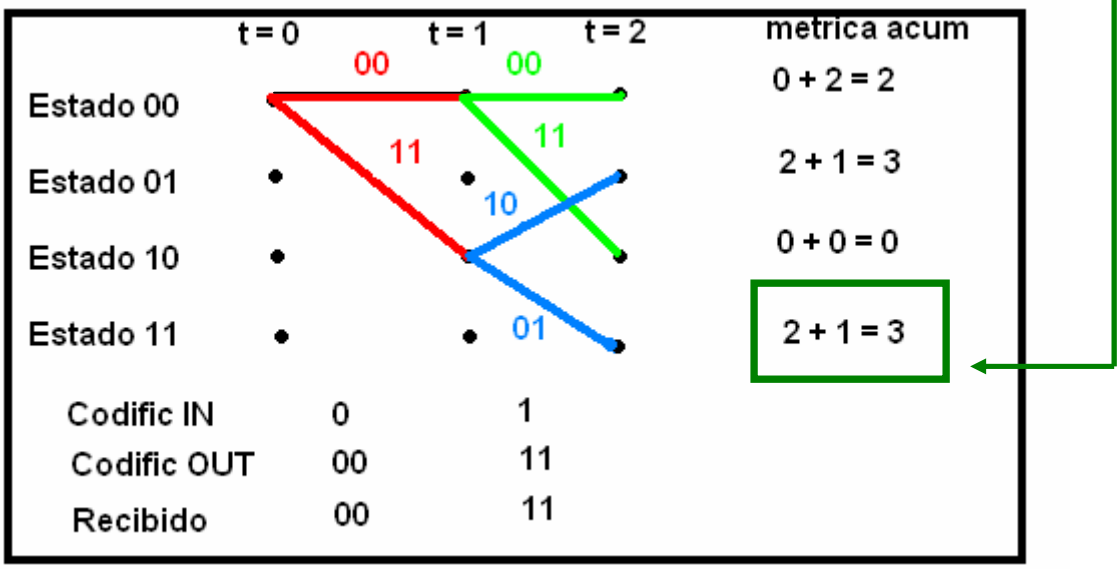


Figura 5.3.6

El Diagrama de Trellis para t = 3 se hace ya mas complicado, ya que ahora tenemos dos caminos diferentes que nos llevan desde cada uno de los cuatro estados validos de t = 2 a los cuatro estados validos de t = 3.

La manera de llevar a cabo esto es, comparar las métricas acumuladas por cada una de las dos ramas, y quedarnos con aquella que sea menor. Si nos encontramos con que los dos valores son iguales, bastara con salvar dicho valor.

Otro problema con el que nos podemos encontrar es que, como para conseguir el mejor camino nos quedamos con la menor de las cuatro metricas de los cuatro estados, haya dos estados que compartan la misma menor metrica. Esto ocurre en el ejemplo que vemos en la figura 5.3.7.

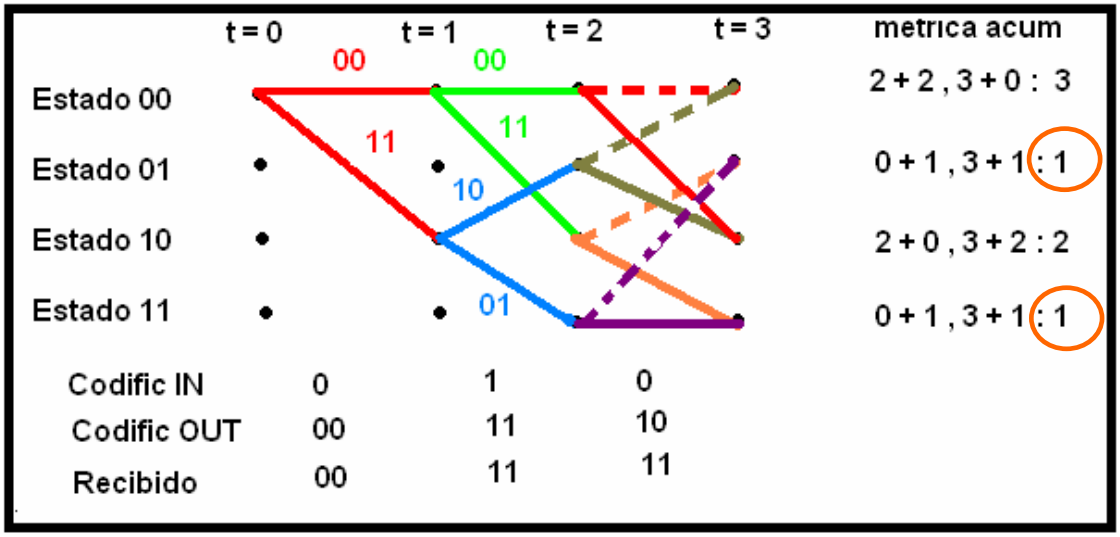


Figura 5.3.7

En este caso, podemos realizar diferentes acciones, como por ejemplo quedarnos siempre con la rama de arriba o al revés. En realidad esta decisión no debería afectar al desarrollo de la decodificación.

Nótese como en este ejemplo, el símbolo codificado no es el mismo que el recibido. Esto es debido a la presencia de ruido.

Si continuásemos desarrollando este diagrama de la misma manera hasta llegar a t = 17 obtendríamos el siguiente resultado final:

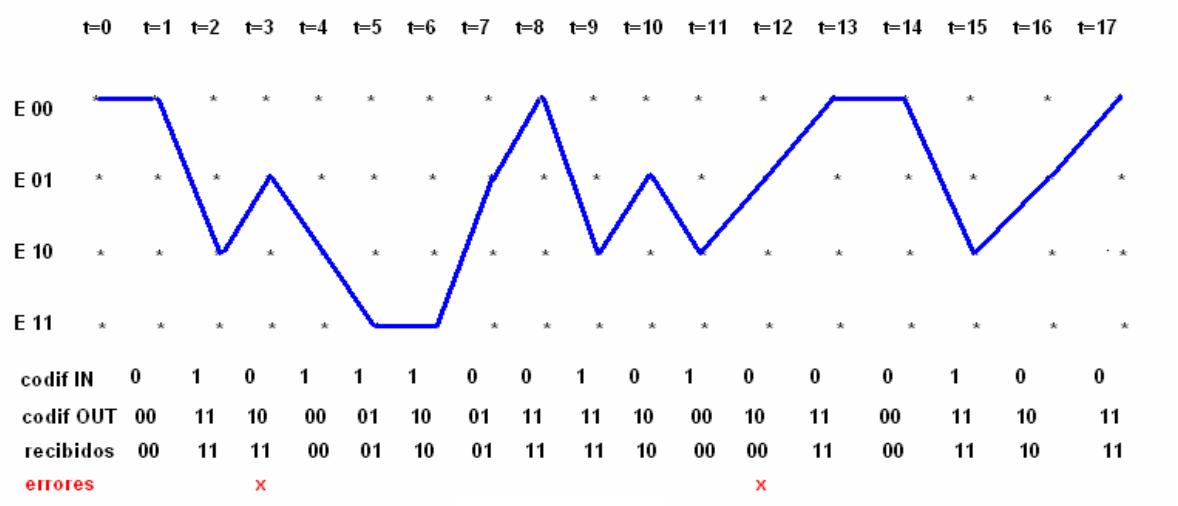


Figura 5.3.8

6. Diseño Hardware

6.1. No segmentado

Una vez comprendidos los conceptos expuestos anteriormente y, por tanto, el funcionamiento tanto del codificador convolucional como del algoritmo de descodificación Viterbi, se comenzó a desarrollar dicho algoritmo en un lenguaje de alto nivel, en este caso JAVA.

La programación en un lenguaje a tan alto nivel, nos aportaba un mayor nivel de abstracción a la hora de desarrollar este algoritmo por primera vez, y nos sirvió como base para su posterior implementación en VHDL.

El desarrollo de la implementación, se baso íntegramente en el algoritmo expuesto anteriormente, y para ello se consideraron necesarias la utilización de varias tablas cuyo significado se expone a continuación.

Como se ha visto, el proceso de decodificación comienza con el cálculo de las métricas de error acumulado para un número de símbolos recibidos a través del canal, y la historia de los símbolos que preceden a estos en cada instante de tiempo t con la menos métrica de error acumulada. Una vez que esto se consigue, el decodificador Viterbi esta preparado para reconstruir el mensaje original que fue la entrada del codificador convolucional cuando este fue codificado para su posterior transmisión.

Pero todo esto requiere llevar a cabo una serie de pasos.

- Primero, seleccionar el estado que tiene la menor métrica acumulada y guardarlo.
- Posteriormente, recorriendo hacia atrás la tabla de historia de estados, para el estado seleccionado, escoger el nuevo estado que aparezca en esa tabla de historia como predecesor del estado actual.
- Finalmnte recorrer la lista de estados seleccionados y guardados en los dos pasos anteriores y reconstruir el mensaje original.

Para poder llevar a cabo estos pasos, es necesaria la construcción de las siguientes tablas:

La primera de ellas es la **tabla de error acumulado**: Esta tabla, contiene el error acumulado para cada estado en cada instante de tiempo.

T=	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Estado 00		0	2	3	3	3	3	4	1	3	4	3	3	2	2	4	5	2
Estado 01			3	1	2	2	3	1	4	4	1	4	2	3	4	4	2	
Estado 10		2	0	2	1	3	3	4	3	1	4	1	4	3	3	2		
Estado 11			3	1	2	1	1	3	4	4	3	4	2	3	4	4		

Figura 6.1.1

La otra tabla es la **tabla de historia**, que te muestra el predecesor de cada estado en el instante t:

T=	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Estado 00	0	0	0	1	0	1	1	0	1	0	0	1	0	1	0	0	0	1
Estado 01	0	0	2	2	3	3	2	3	3	2	2	3	2	3	2	2	2	0
Estado 10	0	0	0	0	1	1	1	0	1	0	0	1	1	0	1	0	0	0
Estado 11	0	0	2	2	3	2	3	2	3	2	2	3	2	3	2	2	0	0

Figura 6.1.2

Una vez obtenida esta tabla y recorriéndola de fin a principio, podemos obtener un vector que nos indica el camino de estados que ha seguido el algoritmo:

T=	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
	0	0	2	1	2	3	3	1	0	2	1	2	1	0	0	2	1	0

Figura 6.1.3

Finalmente con ayuda de la tabla de entrada, de la que obtenemos el bit que es necesario a la entrada para pasar del estado actual al estado siguiente siendo las x transiciones de estado no validas, obtenemos el mensaje original.

	Estado Siguiente			
Estado actual	**00**	**01**	**10**	**11**
00	0	X	1	X
01	0	X	1	X
10	X	0	X	1
11	X	0	X	1

Figura 6.1.4

T=	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	0	1	0	1	1	1	0	0	1	0	1	0	0	0	1

Figura 6.1.5

Una vez construidas estas tablas en JAVA, la resolución del algoritmo de decodificación era instantánea.

La implementación del codificador convolucional no ofrecía grandes problemas, y la introducción de ruido blanco se hacia de forma aleatoria por tanto el algoritmo estaba completo.

Aun así, esta primera representación del codificador-decodificador Viterbi no era del todo eficiente ya que se infrautilizaban recursos y esto repercutía en el tiempo de ejecución.

Ante este problema, surgió la necesidad de proponer una nueva implementación segmentada, que, al dividir la ejecución en etapas, permitía que dos tramas distintas estuviesen ejecutándose en el mismo instante de tiempo mientras no utilizasen recursos en común. Esto ofrecía un mayor rendimiento del sistema y evitaba que ciertas unidades estuviesen inutilizadas durante varios ciclos de reloj.

6.2. Segmentado

6.2.1. Versión Inicial

La primera versión diseñada para la implementación del decodificador segmentada, constaba de distintos módulos dirigidos por un controlador encargado de regir el decodificado de la trama.

La comunicación se realizaría a través de un bus común a los módulos y al controlador. A continuación especificamos las características de los componentes de este diseño preliminar:

- Controlador: en cada flanco de reloj recibiría un símbolo, el cuál depositaría en el bus para que los distintos módulos pudieran acceder a él para conocer dicho símbolo. Dispondría de un contador para conocer el símbolo de la trama que se está procesando en cada instante. Se planteó la necesidad de utilizar un buffer que fuera almacenando los símbolos que recibiría a través del canal. Esto sería necesario en el caso de que la frecuencia del canal y la frecuencia de nuestro decodificador fuesen distintas.

- Tabla error acumulado**:** constaría de una memoria que almacenaría el menor error acumulado para cada estado y en cada uno de los instantes de tiempo. Dispondría de una señal de lectura o escritura así como una señal que indicaría si el chip ha sido seleccionado. Dichas señales las utilizará el controlador para el correcto funcionamiento del procesador.

- Tabla historial: esta memoria almacenaría el estado predecesor superviviente para cada estado y cada instante de tiempo. Al igual que la tabla de error acumulado, este módulo dispondría de una señal de lectura o escritura así como una señal que indicaría si el chip ha sido seleccionado por el controlador para realizar las acciones necesarias.

- Vector estados**:** esta memoria muestra los estados seleccionados con menor error acumulado desde el final de la trama hasta el comienzo de la misma. Este módulo al igual que los anteriores también dispondría de una señal de lectura o escritura así como una señal que indicaría si el chip ha sido seleccionado.

- Buffer de salida**:** en este buffer se iría almacenando el los bits de salida generados después de la decodificación.

En la figura 6.2.1.1 mostramos un esquema del diseño descrito.

En este diseño se pueden diferenciar tres etapas:

- Primera etapa: referente a las acciones realizadas en un ciclo en la tabla historial y en la tabla error acumulado.

- Segunda etapa: referente a las acciones realizadas por el vector de estados.

- Tercera etapa: referente al buffer de salida.

Si nos situáramos en un instante de tiempo en el cual las tres etapas tuvieran los datos necesarios para realizar sus operaciones, es decir, se hubiera procesado por lo menos una trama (t1) además de la que se estaría procesando actualmente (t2), en un mismo ciclo c se estarían realizando las siguientes operaciones en cada una de las etapas:

- Primera etapa: se rellenarían los campos de de las tabla historial y error acumulado correspondientes a ciclo c de la trama t1. Desplazaríamos a la izquierda los datos almacenados en los instantes anteriores.

DIAGRAMA DECODIFICADOR SEGMENTADO

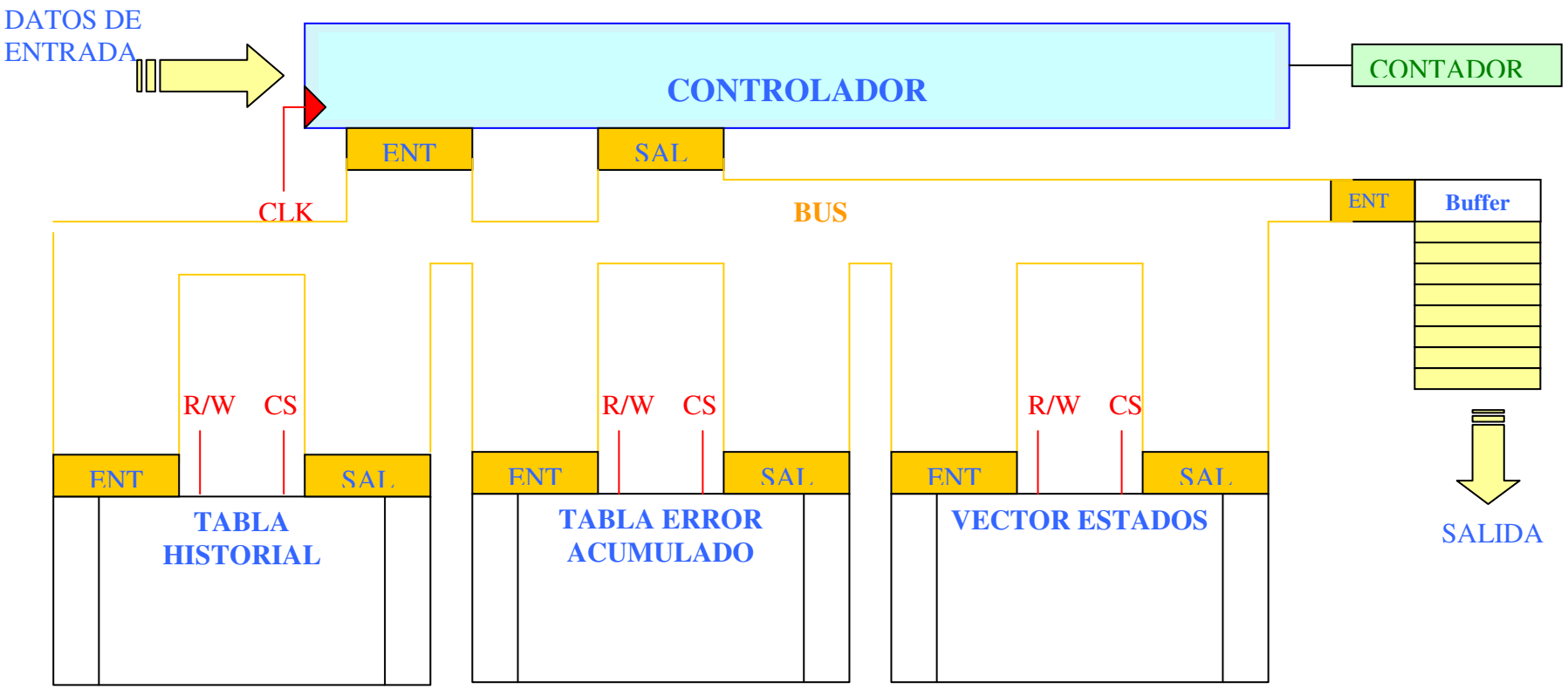


Figura 6.2.1.1

- Segunda etapa: en el instante correspondiente al ciclo c se estarían procesando los datos de la trama t2. En esta etapa es necesario hacer un duplicado del vector, ya que es necesario tener los datos del vector de estados para recorrer dicho vector una vez completada la trama y poder generar la salida en el buffer de salida. De esta forma, tendríamos dos vectores, uno para trama actual y otro para la anterior.

- Tercera etapa: en el mismo ciclo c se estaría almacenando la salida en el buffer de salida.

Diseñada esta primera versión del decodificador segmentado, nos percatamos de que dicho diseño presentaba algunas deficiencias. Una de ellas era la dificultad en la implementación si se quisiera generalizar a un número m de estados a partir de un decodificador de cuatro estados. Otra desventaja era la dificultad de manejar un gran número de señales por parte del controlador.

Estas desventajas se vieron solventadas con el diseño final del sistema segmentado, el cual explicamos a continuación.

6.2.2. Versión Definitiva:

Con la finalidad de aumentar el rendimiento del sistema, se ha implementado el proceso del decodificador Viterbi de forma segmentada. Para simplificar, la implementación se ha hecho a partir de cuatro estados, aunque bien se podría generalizar hasta llegar a los 64 estados del estándar.

Se ha definido una arquitectura hardware con tres módulos independientes: Adición-Comparación-Selección (ACS), Almacenamiento de caminos (AdC) y MáximaVerosimilitud (MV).

A continuación, en la figura 6.2.2.1 mostramos un esquema del decodificador:

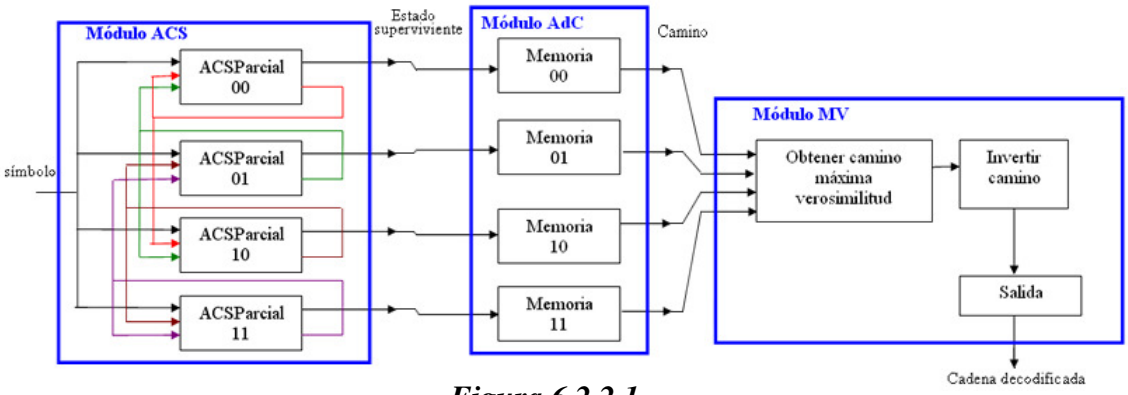


Figura 6.2.2.1

Describiremos cada uno de ellos con mayor detenimiento:

Módulo Adición-Comparación-Selección (ACS):

Esta unidad es la encargada de acumular las métricas de cada estado, seleccionando la menor de las métricas de las ramas que convergen en cada uno de los estados teniendo en cuenta el diagrama de Trellis.

Para ello, podemos dividir el cálculo de cada estado en un submódulo por estado. Así, podremos calcular, para cada símbolo que entra, las métricas de cada estado al mismo tiempo.

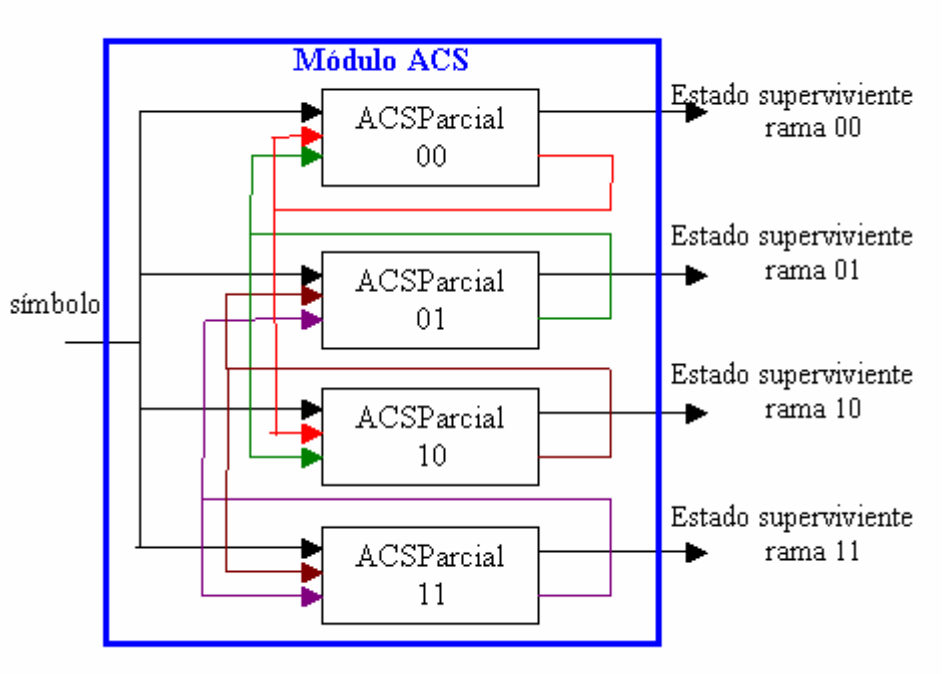


Figura 6.2.2.2

Módulo Almacenamiento de Caminos (AdC):

Este módulo almacena todos los estados supervivientes de cada una de los estados. Consta, por tanto, de una unidad de memoria por cada estado.

El módulo AdC recibe los estados supervivientes en el instante anterior y tiene que almacenar todos estos estados en todos los instantes de tiempo para luego poder recorrerlos empezando por el final, tal y como indica el algoritmo de Viterbi.

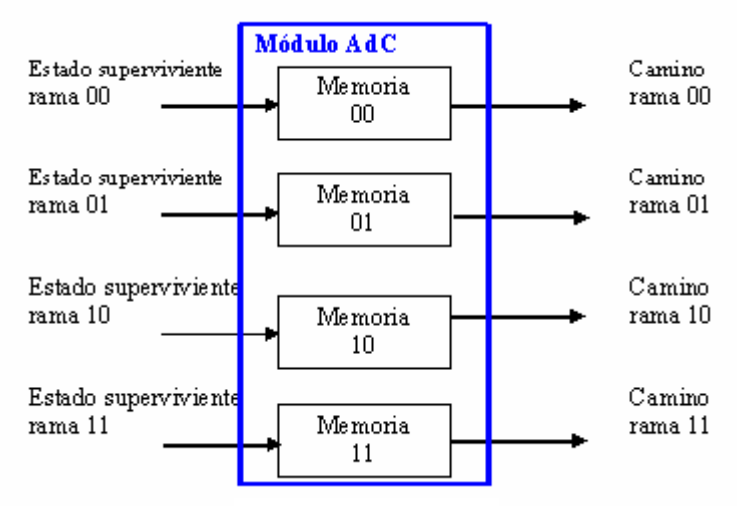


Figura 6.2.2.3

En este módulo y en el siguiente, se puede observar el proceso de segmentación. Realmente, cada submódulo no es una unidad de memoria sino que está compuesto por dos memorias. Mientras en una de ellas se está escribiendo un estado superviviente procedente del módulo ACS que proviene de una trama, de la otra se lee un estado de la trama anterior.

Así, quedaría cada submódulo:

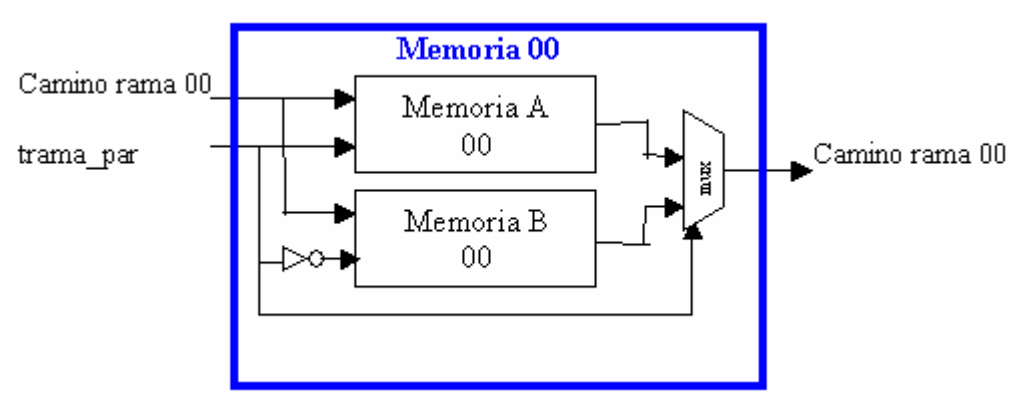


Figura 6.2.2.4

La entrada *control* determina en qué memoria se escribe el dato que se recibe y de qué memoria hay que leer el dato que tiene que recibir el módulo MV.

Módulo Máxima Verosimilitud (MV):

Una vez que el módulo AdC ha almacenado los caminos de todos los estados, el módulo MV debe seleccionar aquel camino que tenga la mayor verosimilitud a partir de la menor métrica acumulada en cada uno de los estados.

Como el módulo AdC almacena el camino de forma inversa, este módulo debe invertir el camino de máxima verosimilitud y obtener, con la ayuda de la tabla de verdad de salida, los bits que forman parte de la cadena decodificada.

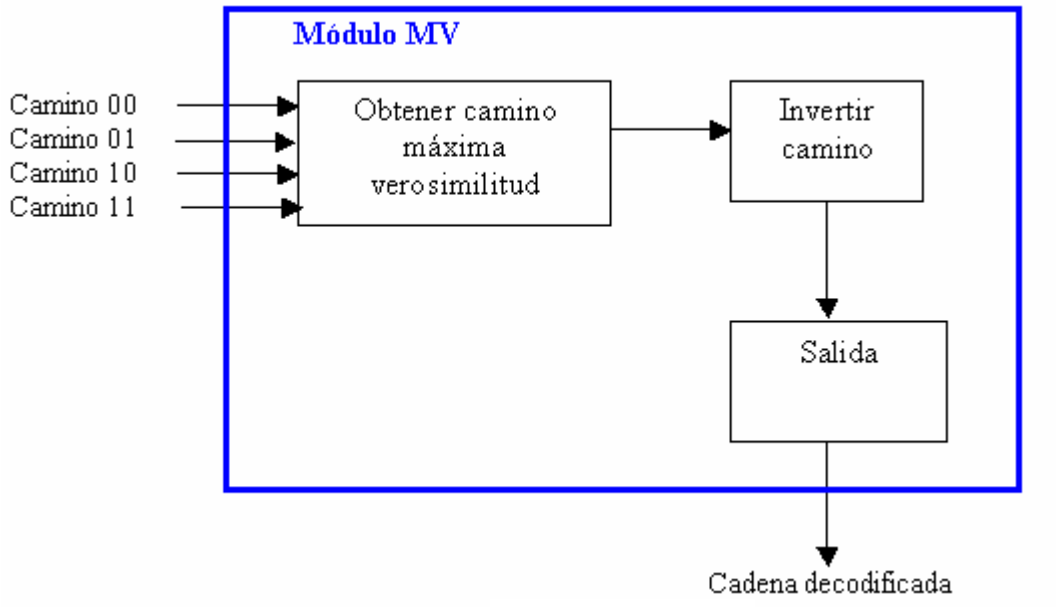


Figura 6.2.2.5

Para obtener el camino de máxima verosimilitud, debemos guardar el estado con la menor métrica en el último instante de tiempo. A partir de este estado, podremos recomponer un camino y obtener la cadena de salida del decodificador.

El submódulo que se utiliza para invertir el camino seleccionado se comporta del mismo modo que el módulo AdC. Está compuesto de dos unidades de memoria en las que en cada instante de tiempo se escribe en una y se lee de la otra.

Tal y como hicimos anteriormente, hemos implementado el decodificador Viterbi segmentado en Java y, además esta vez, también hemos utilizado el lenguaje de descripción hardware VHDL, con el fin de que, en un futuro, pueda probarse en una arquitectura.

Ahora, trataremos de explicar con más detalle cada una de las implementaciones, según el lenguaje de programación elegido.

6.2.3. JAVA

6.2.3.1. Implementación

El lenguaje Java es un lenguaje de alto nivel, debido a esto, la implementación del sistema en este lenguaje nos facilita la comprensión del funcionamiento sin necesidad de bajar a un nivel de implementación físico como nos permite el lenguaje de VHDL. De esta forma, hemos implementado en Java el codificador/decodificador Viterbi segmentando, ayudándonos de la simulación gráfica para poder observar y comprender el funcionamiento real del sistema.

La implementación del sistema codificador/decodificador Viterbi consta principalmente del codificador, la simulación de un canal de comunicación, el decodificador Viterbi y el buffer de salida que contiene los bits decodificados.

A continuación exponemos el sistema de clases en Java de nuestro codificador/decodificador Viterbi tal y como se puede observar en la figura 6.2.3.1.1. En el mismo se puede observar tres bloques bien diferenciados: Codificador, decodificador e interfaz.

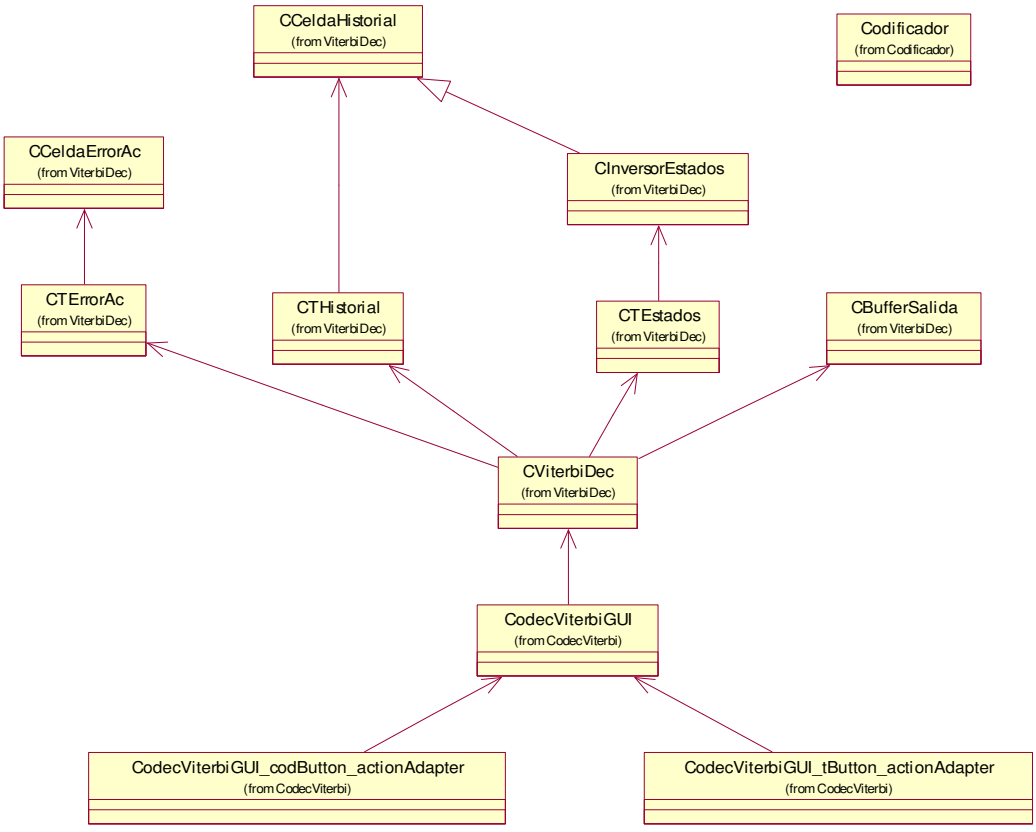


Figura 6.2.3.1.1

El codificador no ha sido segmentado debido a su simplicidad, ya que no aportaba ninguna ventaja la segmentación del mismo. Así pues, el diseño del codificador es el mismo que el descrito en el desarrollo del sistema codificador – decodificador Viterbi sin segmentación. En consecuencia omitiremos la implementación del codificador pues ya ha sido explicado. En la figura 6.2.3.1.2 podemos observar los atributos y métodos de la clase Codificador:

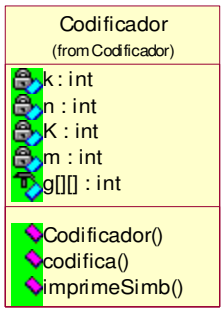


Figura 6.2.3.1.2

Hemos simulado un canal de comunicación incorporando ruido blanco al canal, para representar las posibles alteraciones que puede sufrir la información que circule a través del mismo. De esta forma, los símbolos que le lleguen al decodificador habrán sido alterados y será el propio decodificador el encargado de reconstruir correctamente la cadena de bits de entrada.

La implementación del decodificador Viterbi se ha realizado en concordancia a la última versión segmentada del decodificador Viterbi, pues era la mejor de las especificadas. A continuación especificamos cada una de las clases implementadas:

- CTErrorAc: esta clase se corresponde con el módulo ACS general explicado anteriormente. Este módulo se encarga esencialmente de enviar a las celdas parciales de CTErrorAc el símbolo que recibe del canal junto con los predecesores correspondientes a un estado dado y el error acumulado de sus posibles predecesores. Este módulo almacena el menor error acumulado hasta el momento pues es imprescindible en nuestro módulo MV para el proceso de decodificación.

- CCeldaErrorAc: esta clase se corresponde con el módulo ACS parcial expuesto anteriormente. Este submódulo asume la tarea de calcular el nuevo error acumulado y el predecesor más probable en base al símbolo recibido y el error anterior de sus predecesores, para ello calculamos la distancia de Hamming entre el símbolo recibido y el símbolo teórico que es realmente el que debería haber recibido. A partir de esta clase se crearían el número de objetos necesarios en función del número de estados, en nuestro caso se crearían cuatro objetos, los cuales se corresponden con el ACS parcial.

En la figura 6.2.3.1.3 mostramos el diagrama para estas dos clases:

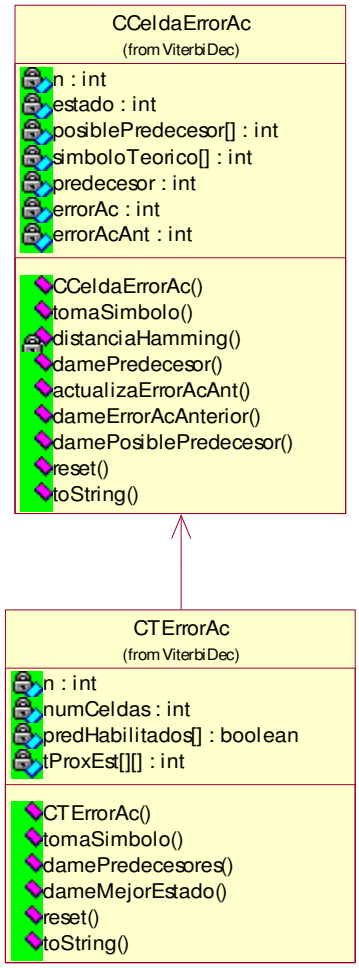


Figura 6.2.3.1.3

- CTHistorial: esta clase se corresponde con el AdC general. Esta clase se encarga de crear los cuatro submódulos (AdC parciales), a las cuales les enviará los estados predecesores supervivientes en el instante anterior.

- CCeldaHistorial: esta clase se corresponde con el AdC parcial. Un objeto de esta clase presenta dos memorias, una para leer los estados de la trama anterior y otra para escribir los estados de la trama actual. Esto es necesario debido a que la lectura se realiza en orden inverso, y por lo tanto hasta que no se haya procesado una trama entera no se podrá comenzar con la lectura de dicha trama.

En la figura 6.2.3.1.4 mostramos el diagrama para estas dos clases:

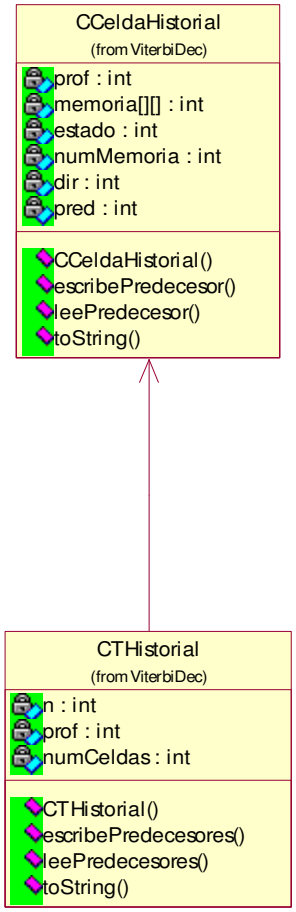


Figura 6.2.3.1.4

- CInversorEstados: extiende de la clase CCeldaHistorial y presenta las mismas características.

- CTEstados: se encarga de recorrer la tabla historial, eligiendo como estado inicial el estado que presente el mínimo error acumulado. A partir de éste, construye el camino de los estados que permitirá obtener correctamente el mensaje enviado.

En la figura 6.2.3.1.5 mostramos el diagrama para estas dos clases:

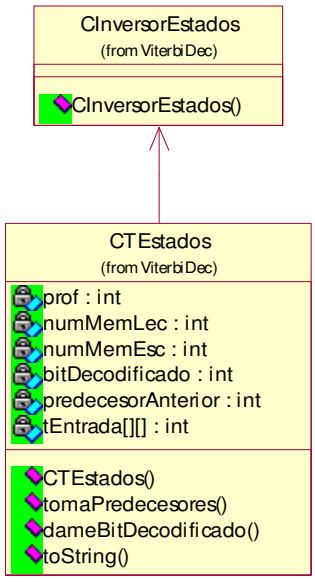


Figura 6.2.3.1.5

- CBufferSalida: A partir de los estados que hemos obtenido a través de CTEstados, construimos el mensaje correcto que fue enviado.

En la figura 6.2.3.1.6 mostramos el diagrama para esta clase:

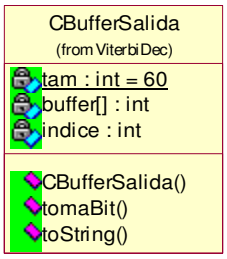


Figura 6.2.3.1.6

- CViterbiDec: El objeto de esta clase rige el comportamiento del resto de los objetos referentes al decodificador, es decir, coordina el correcto funcionamiento del decodificador Viterbi.

En la figura 6.2.3.1.7 mostramos el diagrama para esta clase:

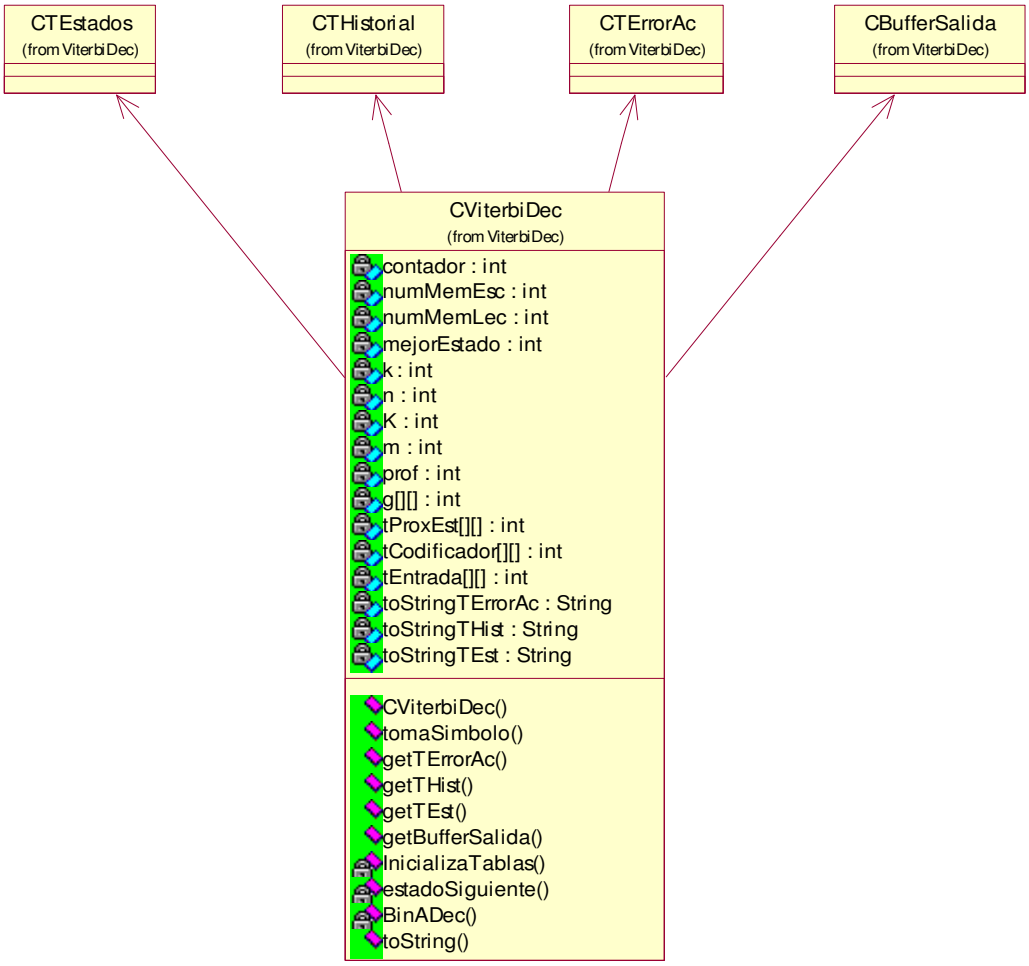


Figura 6.2.3.1.7

- CodecViterbiGUI: es la interfaz del codificador/decodificador que nos permite introducir la cadena de bits que se va a transmitir. Dicha cadena de bits atraviesa el codificador, que genera los símbolos correspondientes, los cuales circulan por el canal de comunicación simulado, produciéndose la alteración en algunos de ellos que finalmente son decodificados, mostrando los bits inicialmente transmitidos.

En la figura 6.2.3.1.8 mostramos el diagrama de clases:

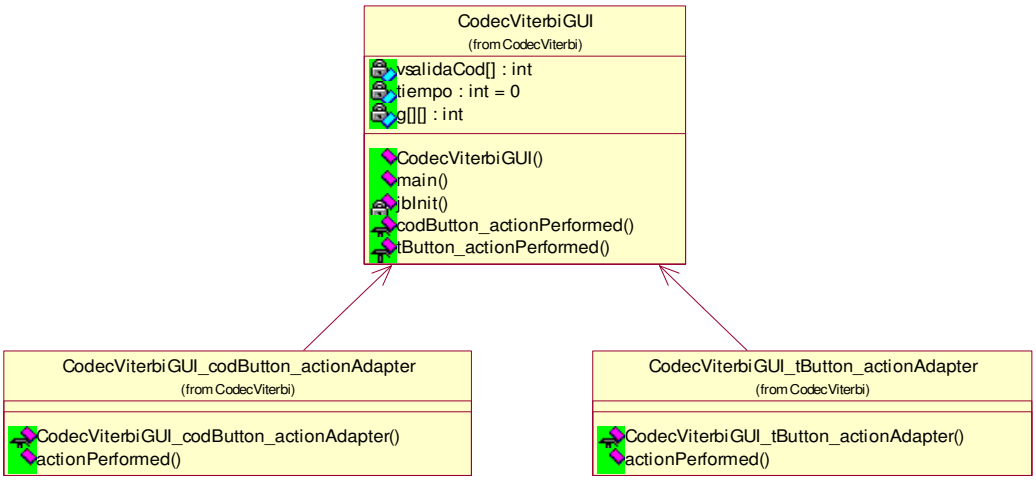


Figura 6.2.3.1.8

6.2.3.2. Simulación

A través de la simulación hemos pretendido hacer visible la segmentación de código que hemos llegado a cabo por medio de la siguiente interfaz (figura 6.2.3.2.1):

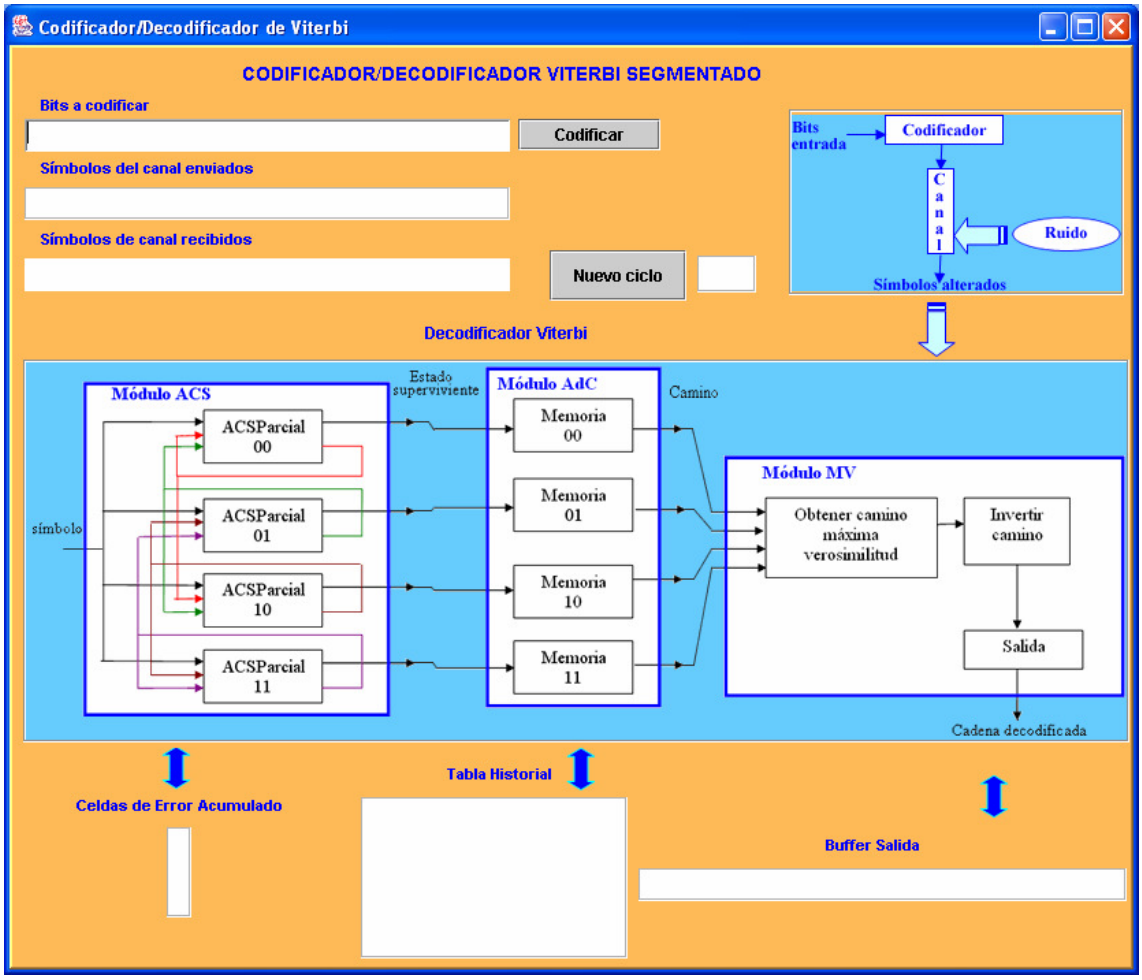


Figura 6.2.3.2.1

En esta interfaz vamos a poder ver los bits que inicialmente queremos transmitir, éstos se encuentran en el campo bits a codificar (figura 6.2.3.2.2):

Figura 6.2.3.2.2

Una vez hayamos pulsado el botón Codificar, veremos qué símbolos se han creado nada más salir del codificador y qué símbolos se han modificado una vez se hallan transmitido por el canal y hayan sufrido el ruido existente en el mismo (figura 6.2.3.2.3).

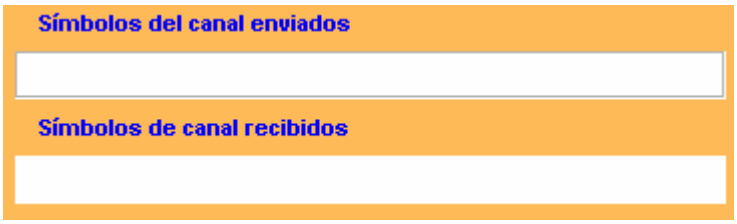


Figura 6.2.3.2.3

Por lo tanto lo explicado anteriormente se corresponde con la figura 6.2.3.2.4.

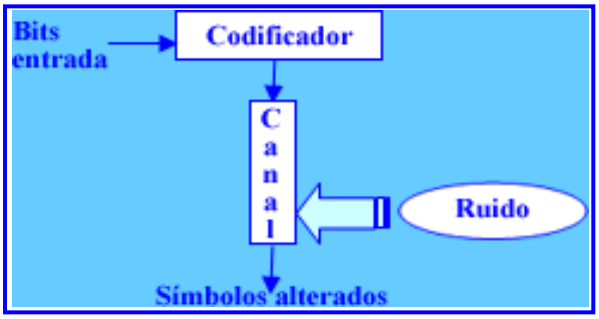


Figura 6.2.3.2.4

La interfaz también nos permite saber que es lo que almacena cada uno de los tres módulos del decodificador (celdas de error acumulado, tabla historial y el buffer de salida) en cada instante de tiempo (figura 6.2.3.2.5).

Figura 6.2.3.2.5

Mediante un ejemplo vamos a poder ver el funcionamiento segmentado del codificador/decodificador Viterbi. Los bits que queremos transmitir los introducimos en el campo correspondiente a bits a codificar (figura 6.2.3.2.6).

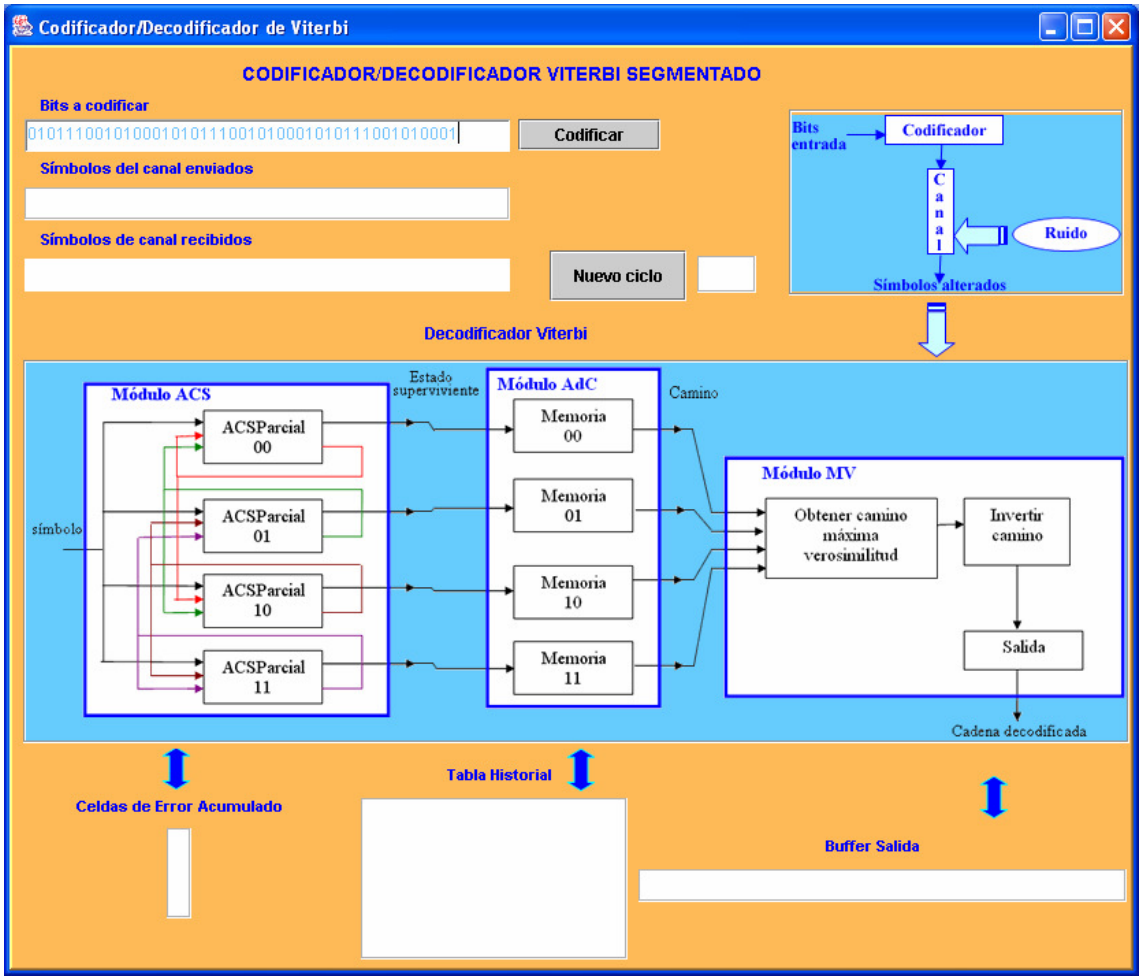


Figura 6.2.3.2.6

Una vez introducidos los bits, queremos codificarlos, por lo tanto pincharemos sobre el botón codificar para obtener los símbolos correspondientes y posteriormente obtendríamos los símbolos resultantes del ruido blanco existente en el canal. De esta forma la interfaz quedaría como se muestra en la figura 6.2.3.2.7.

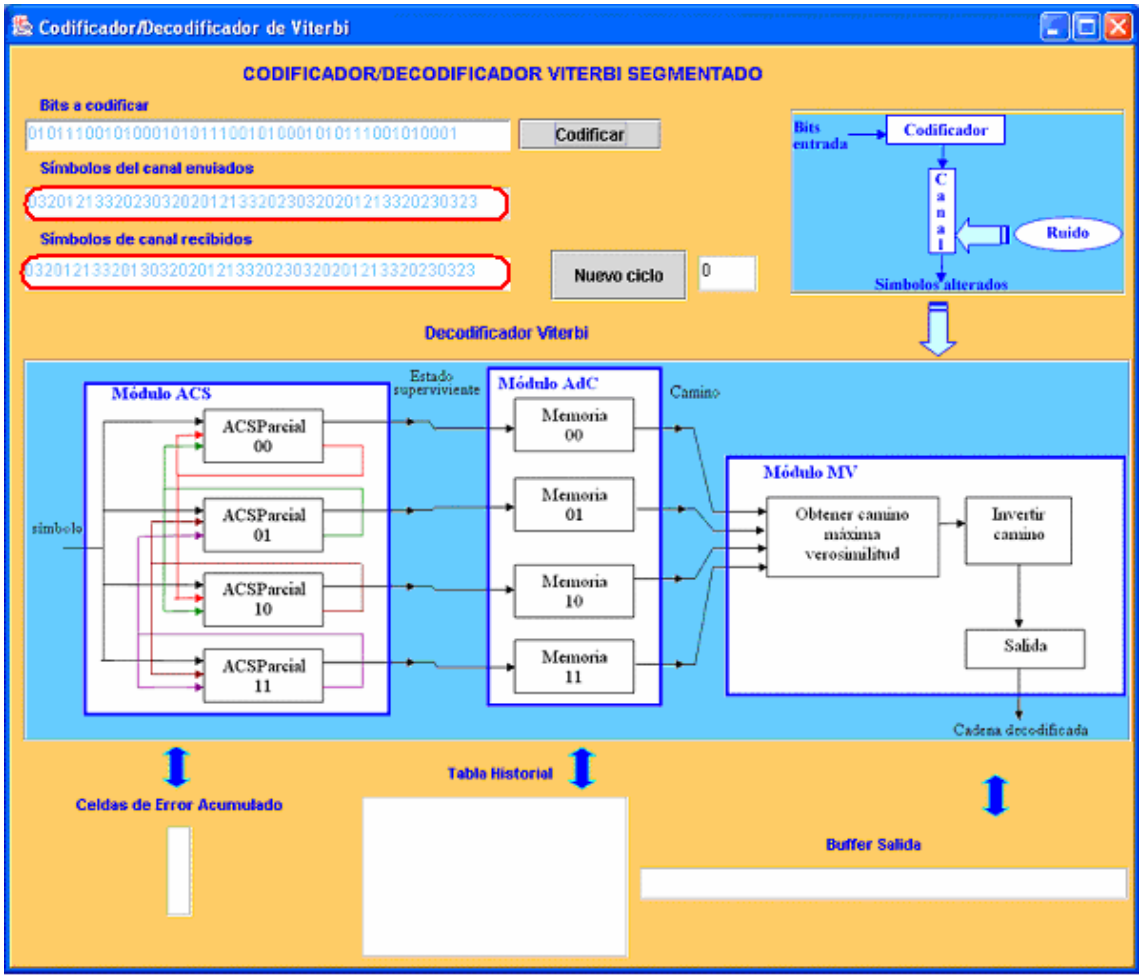


Figura 6.2.3.2.7

En este momento comenzamos a decodificar la cadena de símbolos que proviene del canal. Cuando pulsemos sobre el botón de nuevo ciclo, sabremos en qué instante de ciclo estamos y podremos observar qué es lo que exactamente existe en cada uno de los tres módulos del decodificador.

El módulo ACS se corresponde con las celdas de error acumulado y guarda en cada ciclo el mínimo error acumulado. El módulo AdC se corresponde con la tabla historial y presenta las memorias de lectura y escritura de las distintas tramas, en tabla historial mostraremos las memorias de escritura. Por último, el buffer de Salida se corresponde con el módulo MV y mostrará la cadena de bits resultante de la transmisión (figura 6.2.3.2.8).

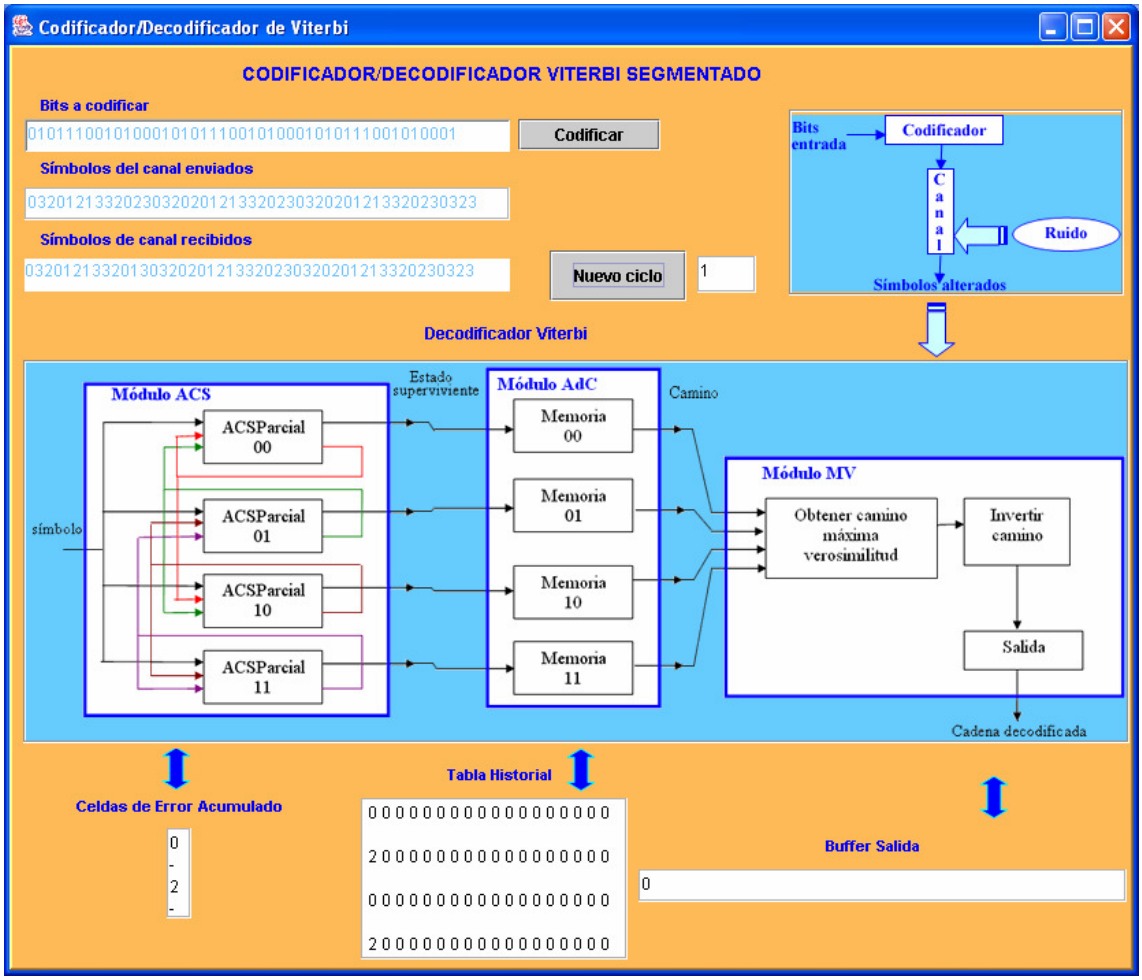


Figura 6.2.3.2.8

En el ciclo 1 observamos que las celdas de error acumulado han sido rellenas con el valor de error acumulado correspondiente, la tabla historial ha escrito en dicho instante los estados predecesores supervivientes correspondientes. En el buffer de salida aparece un cero pues aún no le ha llegado nada para poder sacar el bit decodificado.

Si ejecutáramos un número determinado de ciclos, las celdas de error acumulado cambiarían en cada instante y la tabla historial lo haría igualmente, el buffer de salida hasta que le llegue algo no podrá decodificar. Por ejemplo en el ciclo 18 podemos observar lo que aparece en la figura 6.2.3.2.9.

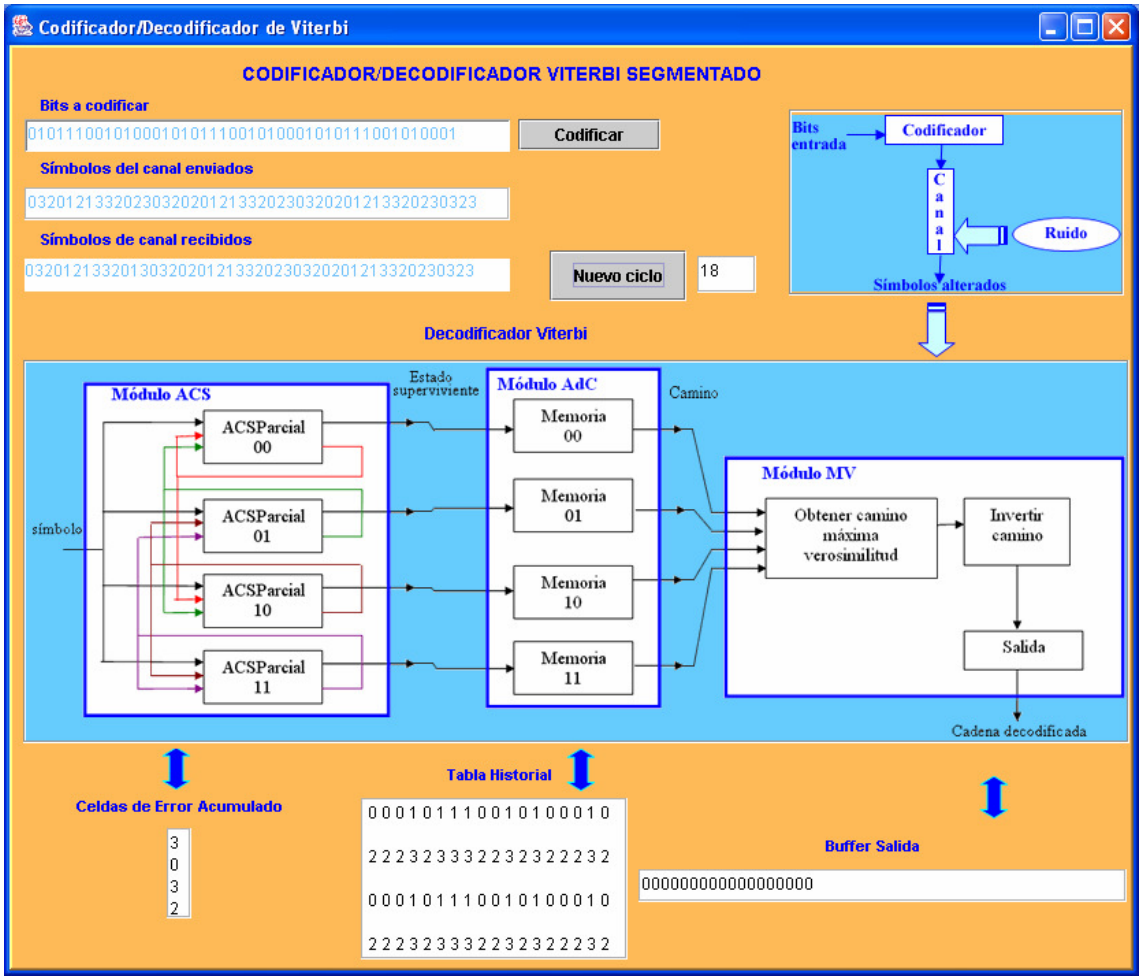


Figura 6.2.3.2.9

En el ciclo 18 hemos llegado al fin de la primera trama y el buffer de salida sigue sin poder mostrar los bits decodificados.

En el ciclo 19 comenzaría la segunda trama, por lo tanto podemos observar lo que aparece en la figura 6.2.3.2.10.

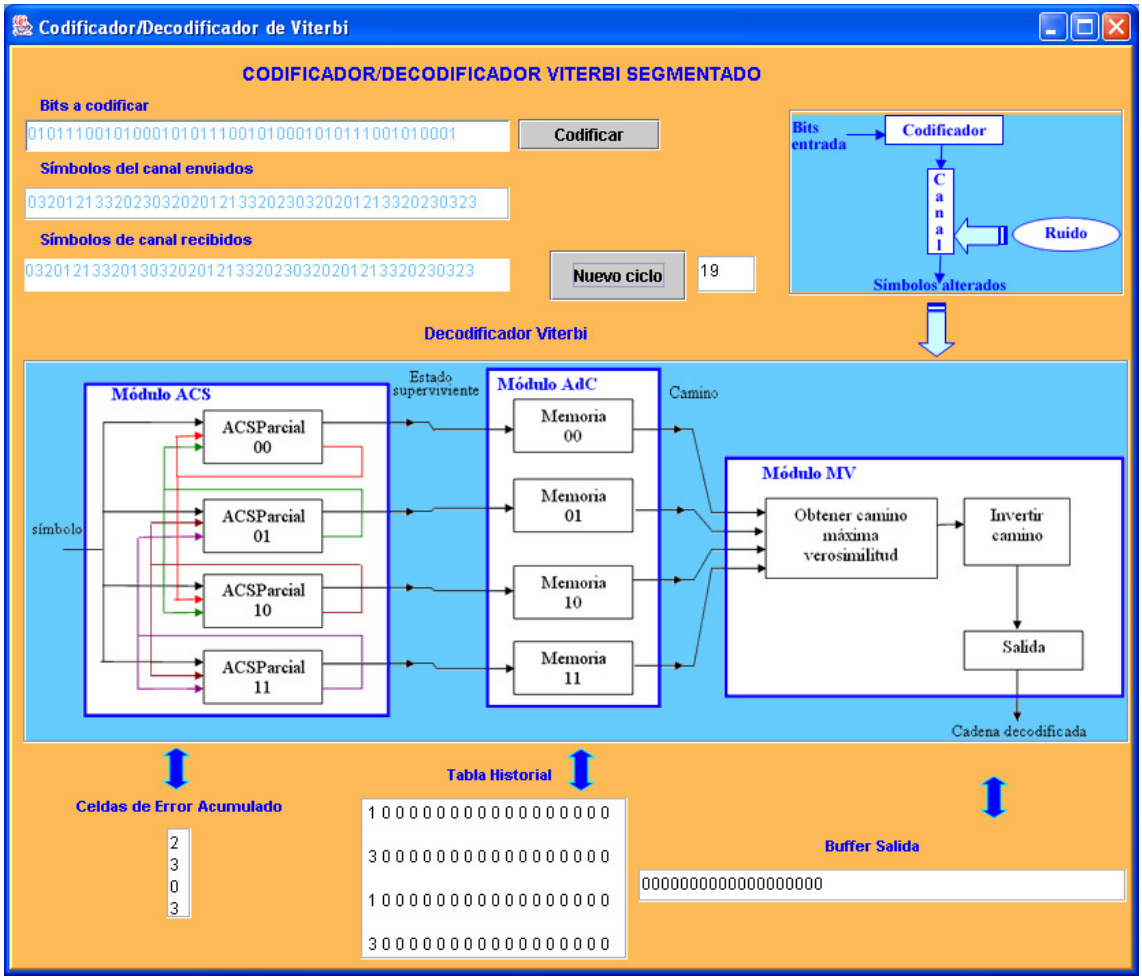


Figura 6.2.3.2.10

Hemos alternado las memorias de escritura y por ello aparecen ceros en vez de los números que aparecían antes, excepto la primera posición que se corresponde con la segunda trama.

En el ciclo 37 podemos ver que comienza a decodificarse los bits codificados al principio, por lo tanto en el ciclo 38 podemos observar lo que aparece en la figura 6.2.3.2.11.

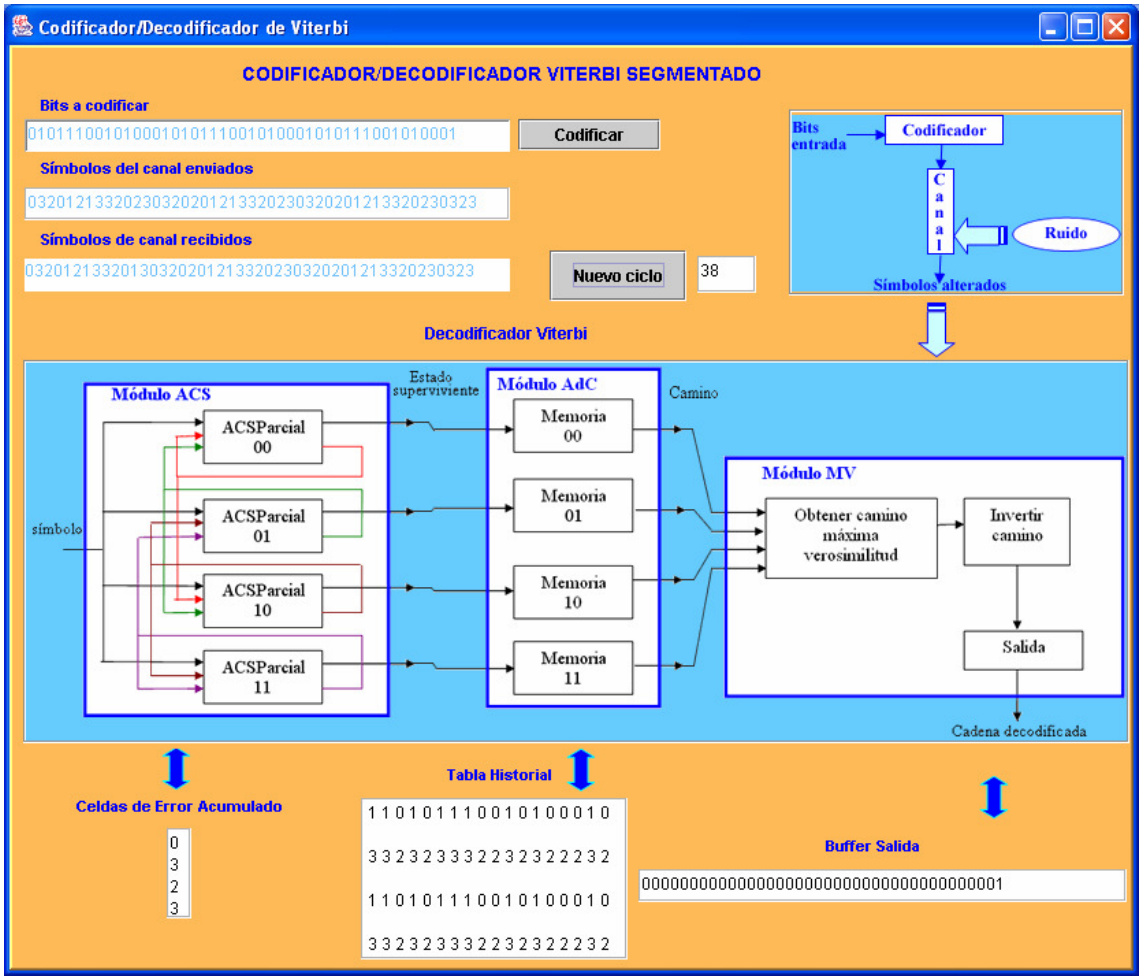


Figura 6.2.3.2.11

Finalmente en el ciclo 49 podemos ver la evolución de la decodificación de las tramas correctamente, a pesar de los errores introducidos por el ruido del canal (figura 6.2.3.2.12).

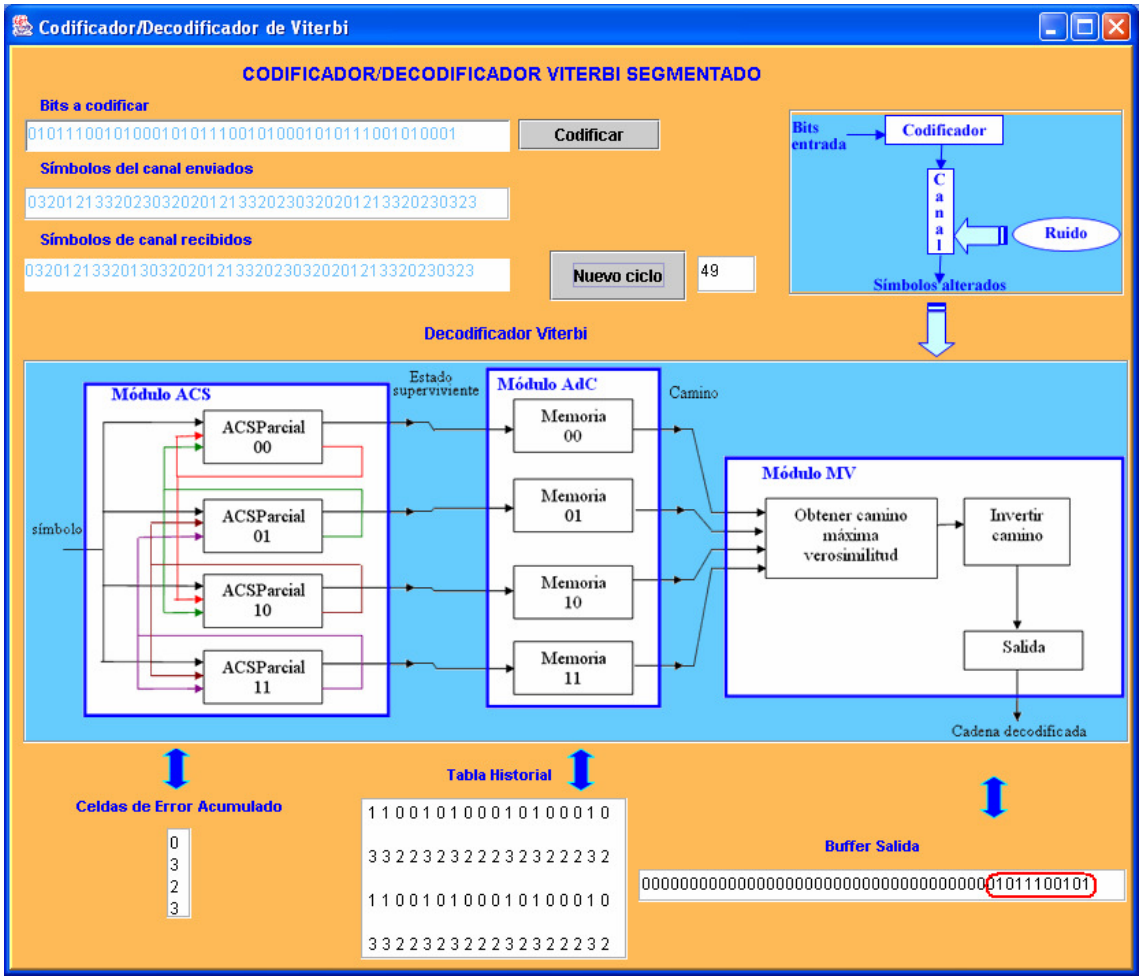


Figura 6.2.3.2.12

6.2.4. VHDL

6.2.4.1. Implementación

VHDL es un lenguaje de descripción de hardware, por tanto, lo vamos a utilizar para describir el comportamiento del decodificador Viterbi como un circuito de tal modo que en un futuro pueda ser implementado físicamente.

Por tanto, vamos a detallar el comportamiento a nivel de circuito de cada uno de los tres módulos de los que se compone la arquitectura.

Módulo ACS:

Como hemos dicho anteriormente, al escoger un decodificador de cuatro estados, este módulo consta de cuatro submódulos iguales.

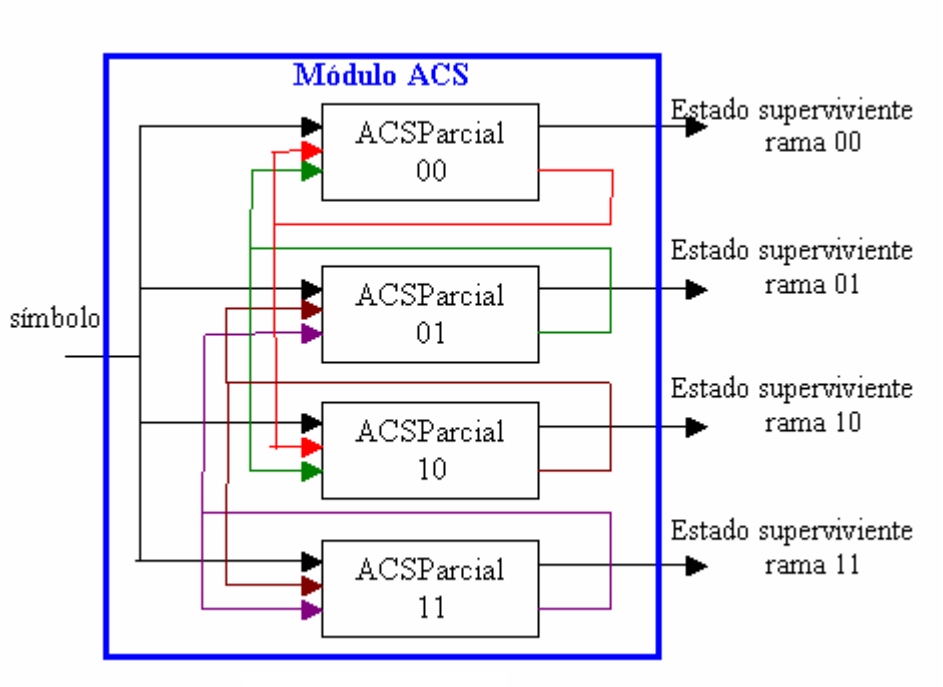


Figura 6.2.4.1.1

Como los cuatro submódulos son iguales, pasamos a detallar uno de ellos, por ejemplo, el correspondiente al estado 00.

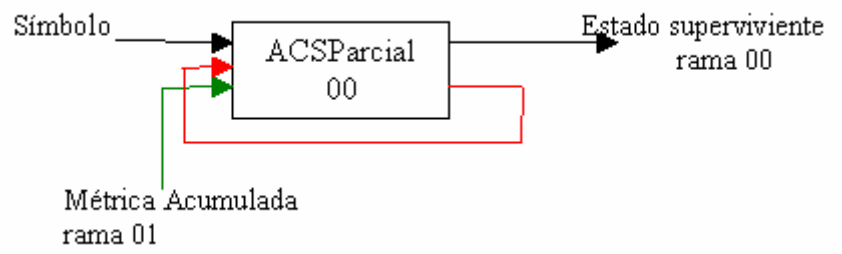


Figura 6.2.4.1.2

Este submódulo tiene como funciones: Acumular la métrica en el estado correspondiente y seleccionar aquel estado que converge en dicho estado y que tenga la menor métrica.

Para ello, primero calculamos la métrica para cada una de las ramas que convergen en el estado 00, nos quedamos con la menor de ellas y pasamos al módulo siguiente el estado superviviente.

Por tanto, contamos con varias unidades básicas: Una que calcule cada una de las métricas, otra que las sume con la métrica acumulada en cada rama y un comparador.

El esquema de este submódulo ACSParcial sería:

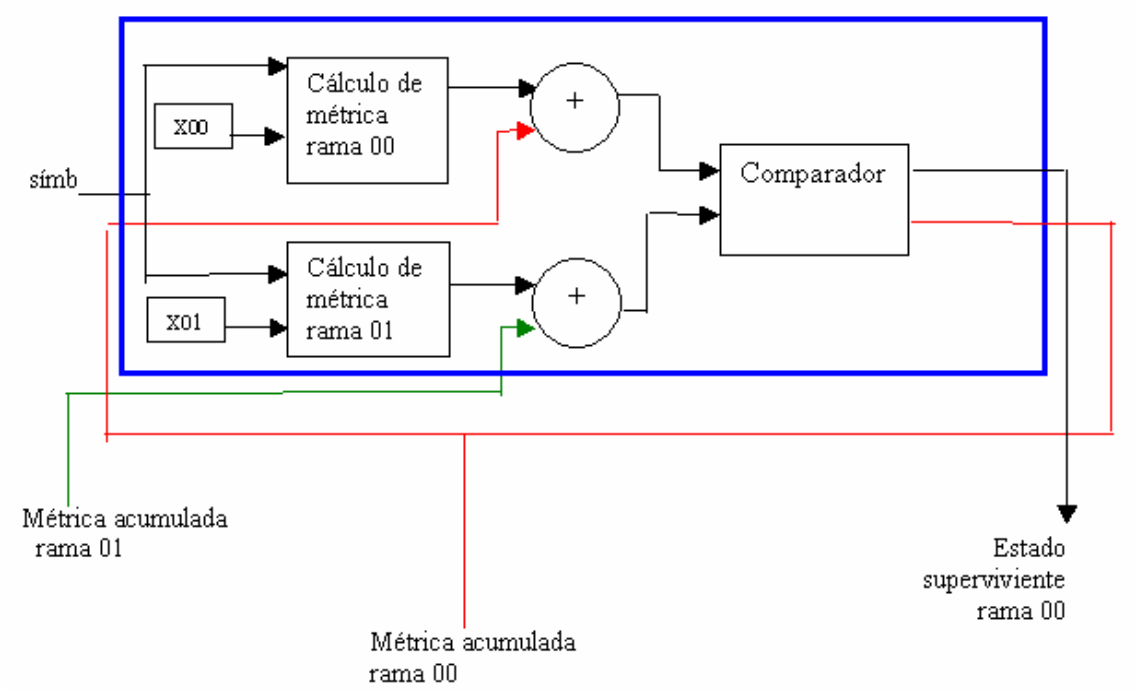


Figura 6.2.4.1.3

x00 y x01 son los símbolos teóricos que deberían haber entrado en el módulo para pasar del estado 00 al estado 00 y para pasar del estado 01 al estado 00, respectivamente.

El cálculo de la métrica se hace calculando la distancia de Hamming entre el símbolo que entra en el módulo y los símbolos teóricos. Calculamos la distancia de Hamming entre el símbolo que llega al decodificador y el símbolo teórico de la rama 00, x00. Mostramos el circuito que lo implementa:

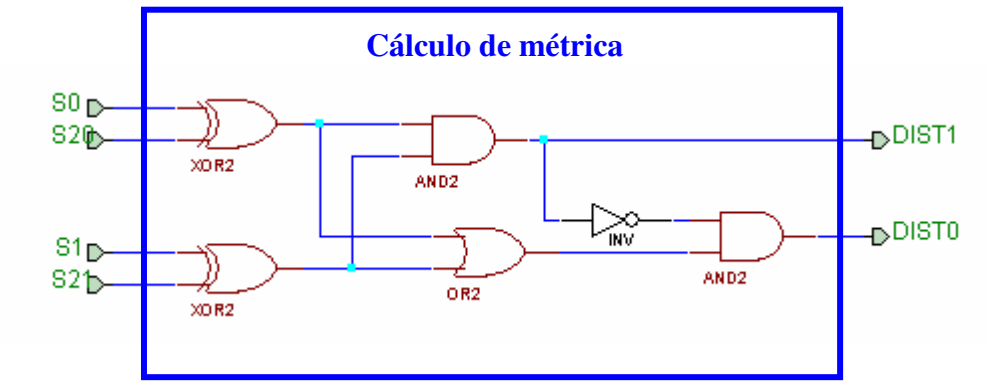


Figura 6.2.4.1.4

A cada una de las métricas hay que sumarle la métrica acumulada de los respectivos estados anteriores (en el instante anterior). Esto se hace con un sumador binario de 2 bits:

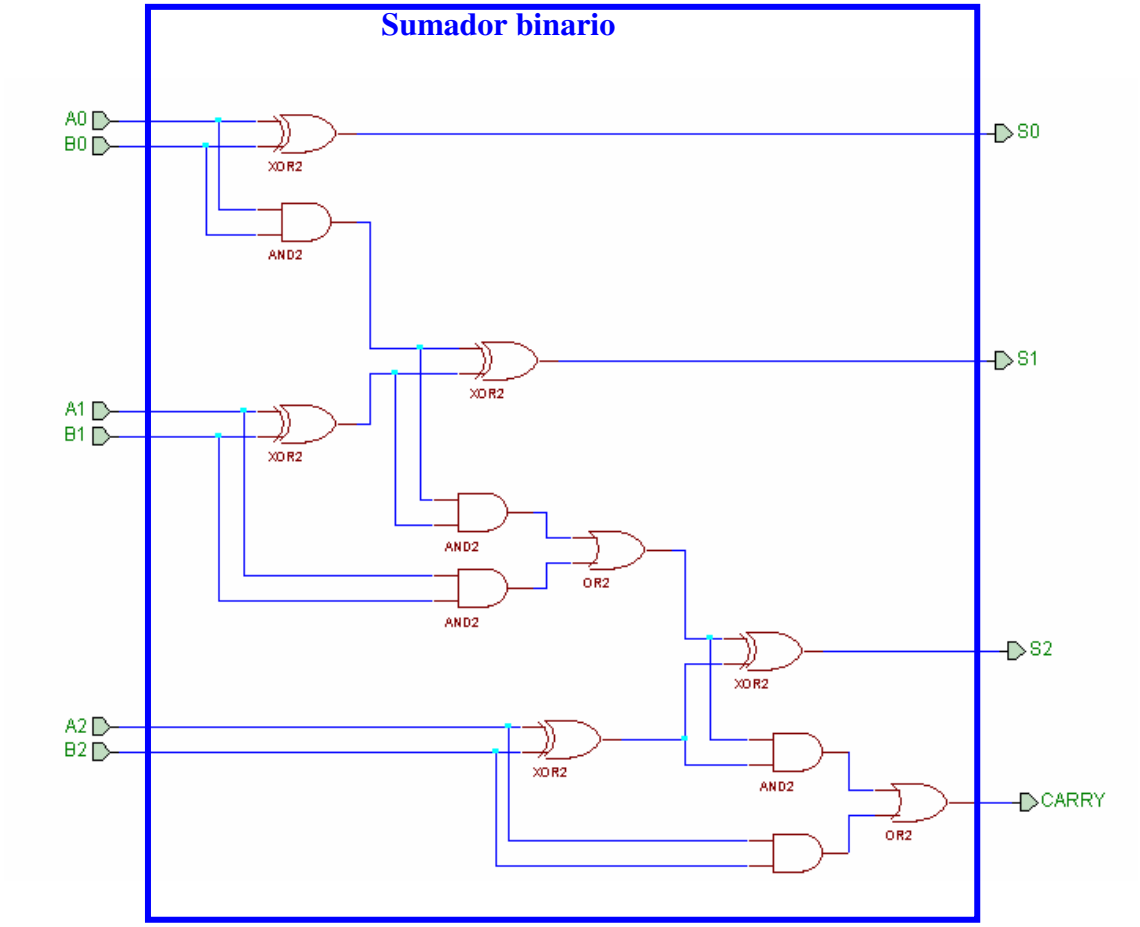


Figura 6.2.4.1.5

Finalmente, el comparador selecciona la rama con la menor métrica y, por tanto, el estado superviviente en el instante de tiempo actual.

Si ($M_{00} \leq M_{01}$) entonces
Métrica acumulada rama 00 = M_{00}
Estado superviviente rama 00 = 00

Si (M_{00} > M_{01}) entonces
Métrica acumulada rama 00 = M_{01}
Estado superviviente rama 00 = 01

Debemos conservar la métrica acumulada en cada instante de tiempo. Para ello, utilizamos registros a la salida del comparador. La salida de cada registro permitirá entregar la métrica acumulada a los estados que la precisen.

Al final, el diagrama de este submódulo quedaría del siguiente modo:

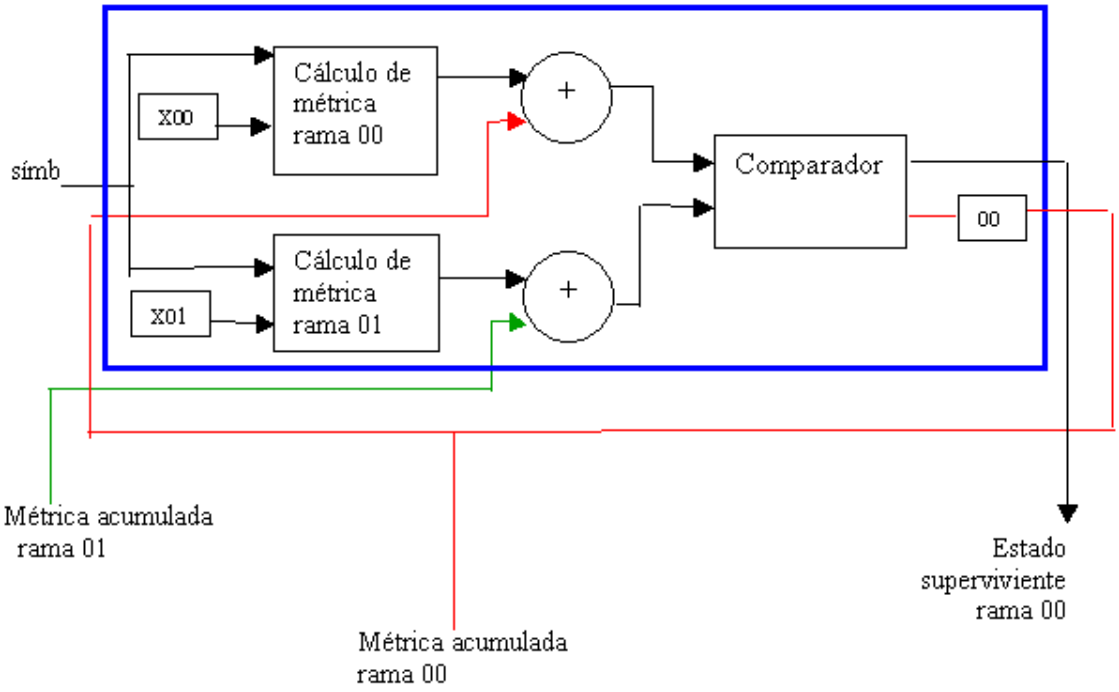


Figura 6.2.4.1.6

Si nos fijamos en el inicio del d ɔs damos cuenta que cuando llegan los dos primeros símbolos (entre t = 0 y t = 1, y entre t = 1 y t = 2), no se producen todas las transiciones, es decir, empezando en el estado 00 sólo hay dos posibles estados a los que se puede ir, el 00 y el 10. Esto se debe a que el codificador ha sido inicializado en el estado todo-ceros y, por tanto, dado cualquier bit de entrada, 0 o 1, sólo se puede hacer una transición a los estados 00 y 10.

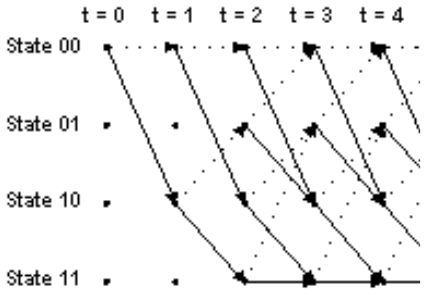


Figura 6.2.4.1.7

Debido a esta cuestión, tuvimos que plantearnos la posibilidad de anular los valores de las métricas de los estados que no teníamos que tener en cuenta, que la salida de los registros sea nula. Si no realizábamos esta anulación, llegabamos a tener valores erróneos por las conexiones de los módulos ACSParcial.

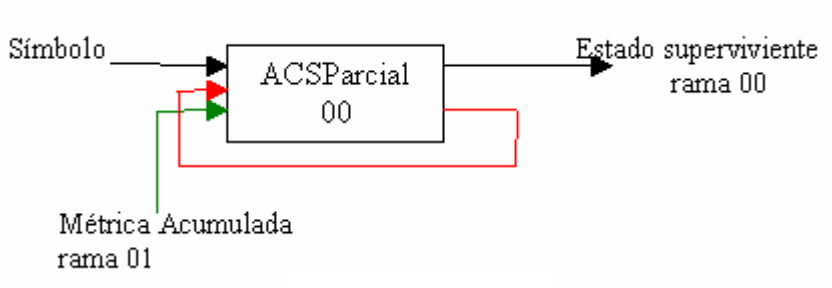


Figura 6.2.4.1.8

Para calcular la métrica acumulada de la rama 00 en t = 1 (ha llegado el primer símbolo) debemos tener en cuenta que la métrica acumulada en la rama 01 tiene que ser 0. Esta parte no habría problemas porque se inicializan todos los registros a 0 y el resultado sería el correcto. Sin embargo, como se calculan todas las métricas a la vez, habrá que controlar que las métricas que se almacenan en los registros sean los correctos para que, en el ciclo siguiente, las entradas a los distintos ACSParcial sean las correctas.

Para llevar esto a cabo, lo que controlamos es la entrada al comparador.

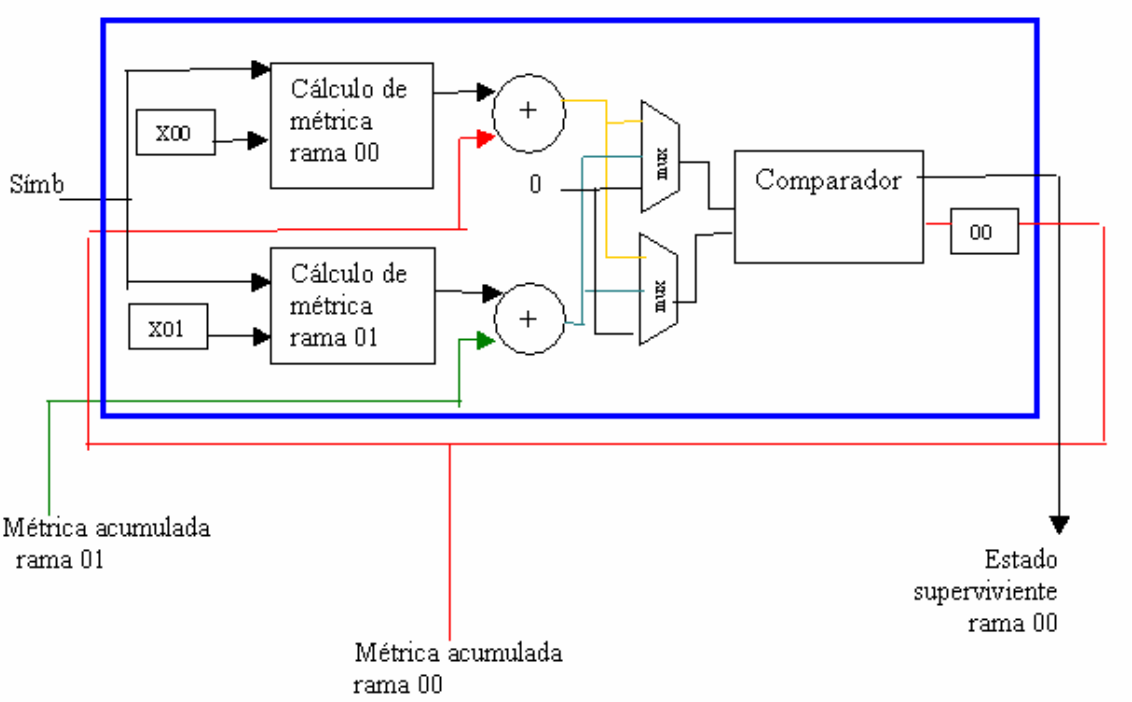


Figura 6.2.4.1.9

Los multiplexores tienen como entrada de selección una señal *índice* que se encarga de contar cuántos símbolos han entrado en el decodificador. Esta señal nos servirá para controlar cuándo se produce un cambio de trama, es decir, cuándo han llegado todos los símbolos correspondientes a una misma trama.

Vemos el funcionamiento de estos multiplexores y el valor de las entradas del comparador (Entrada1 y Entrada2) según la señal índice.

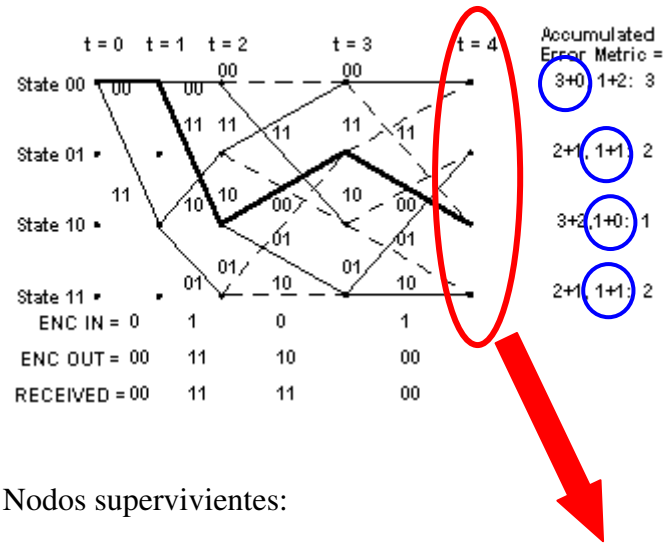
Cuando índice = 0:
- En los módulos ACSParcial 01 o ACS Parcial 11:
  - Entrada1 = 0
  - Entrada2 = 0
- En los módulos ACSParcial 00 o ACS Parcial 10:
  - Entrada1 es la salida del sumador superior.
  - Entrada2 es la salida del sumador superior.

Cuando índice = 1:
- En los módulos ACSParcial 01 o ACS Parcial 11:
  - Entrada1 es la salida del sumador superior.
  - Entrada2 es la salida del sumador superior.
- En los módulos ACSParcial 00 o ACS Parcial 10:
  - Entrada1 es la salida del sumador superior.
  - Entrada2 es la salida del sumador superior.

Cuando índice > 1:
- Entrada1 es la salida del sumador superior.
- Entrada2 es la salida del sumador inferior.

Por último, debemos resetear los valores de las métricas acumuladas de los estados cuando se llegue al último símbolo de la trama para que comience correctamente el procesamiento de la segunda trama.

Veamos un ejemplo de ejecución de este módulo.



Nodos supervivientes:

	índice = 0	**índice = 1**	**índice = 2**	**índice = 3**	**índice = 4**
00	0	0	1	0	
01	0	2	2	3	
10	0	0	0	1	
11	0	2	2	3	

Figura 6.2.4.1.10

Módulo AdC:

El módulo AdC recibe los estados supervivientes en el instante anterior y tiene que almacenar todos estos estados a medida que avanza el diagrama de Trellis para luego poder recorrerlos empezando por el final, tal y como indica el algoritmo de Viterbi. Por tanto, tendrá que disponer de cuatro unidades de memoria, una por cada nodo.

Sin embargo, tal y como explicamos anteriormente, estas unidades de memoria son dobles, es decir necesitamos dos memorias por nodo, una para escribir los símbolos de una trama i que entran en el módulo y otra para leer los símbolos de la trama i-1. De este modo, se pueden procesar dos tramas sin necesidad de parar el pipeline.

El esquema de cada submódulo sería:

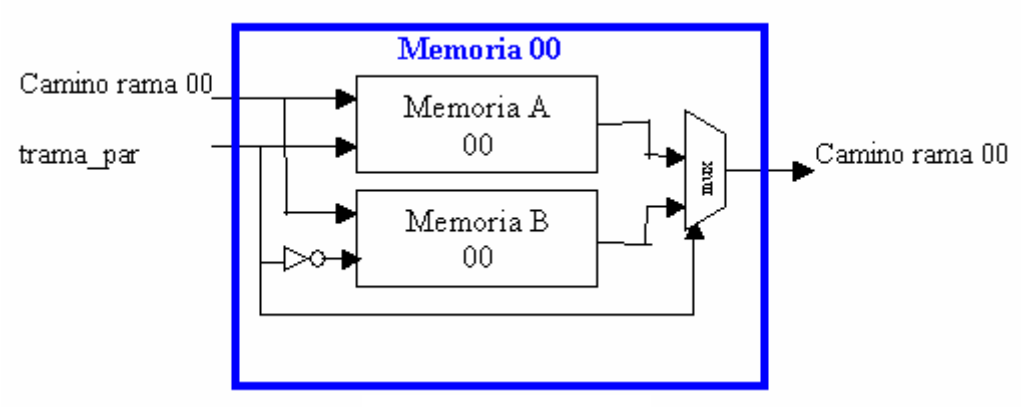


Figura 6.2.4.1.11

La entrada *control* determina en qué memoria se escribe el dato que se recibe y de qué memoria hay que leer el dato que tiene que recibir el módulo MV.

Módulo MV:

El módulo MV selecciona la secuencia de estados con la menor métrica a partir de los cuatro caminos que le llegan del módulo AdC. Para ello, realiza un proceso de lectura de la memoria correspondiente del módulo anterior en sentido inverso.

Este módulo consta de tres partes:

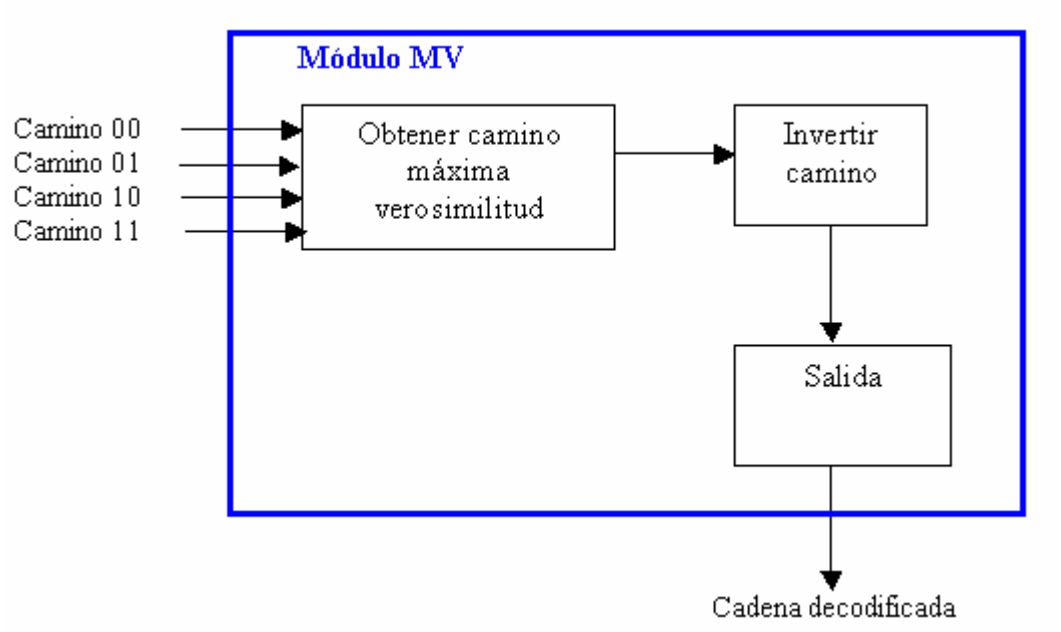


Figura 6.2.4.1.12

El submódulo *Mejor Camino* evalúa y selecciona el camino con la menor métrica. Para saber cuál es el camino con la menor métrica, se debe haber almacenado, en el primer módulo (Módulo ACS) cuál es el estado con la menor métrica (mejor) en el momento que han llegado ya todos los símbolos de una trama. A partir de este estado, se recorren los caminos que llegan del módulo AdC seleccionando los predecesores.

Este submódulo lo hemos implementado de la siguiente forma:

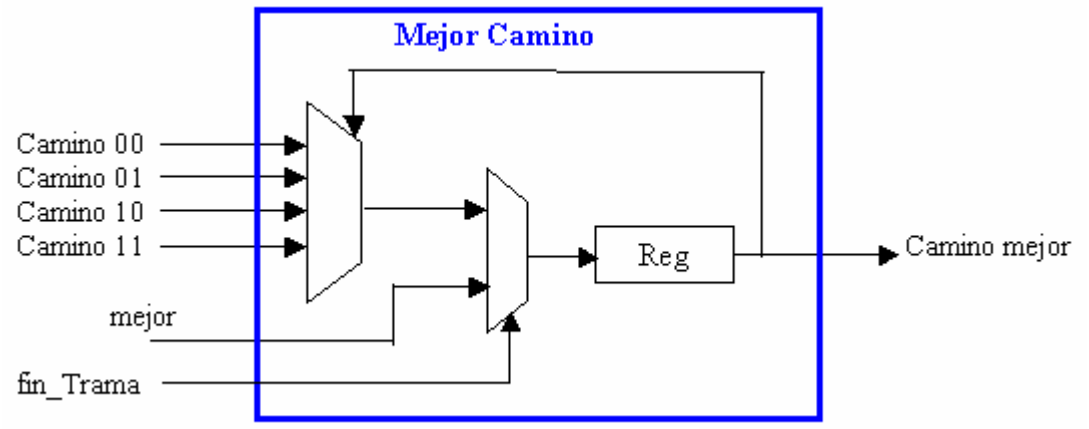


Figura 6.2.4.1.13

La señal *fin_Trama* es una señal de control que nos indica cuándo han llegado todos los símbolos de una trama y, por tanto, tenemos el valor de la señal *mejor* correcta para poder empezar a seleccionar el camino mejor.

Hay que tener en cuenta que el camino mejor se ha obtenido a partir del recorrido inverso de la memoria de AdC, por lo tanto, para seguir con el procesamiento de esta trama, debemos volver a invertir el camino. De no hacerlo, obtendríamos la cadena de bits decodificada en sentido inverso.

Para esto tenemos el submódulo *Inversión*, que en sí, tiene la misma funcionalidad que el módulo AdC que hemos comentado con anterioridad. En este caso, como sólo es uno el camino que hay que invertir, sólo necesitaremos un bloque de memorias:

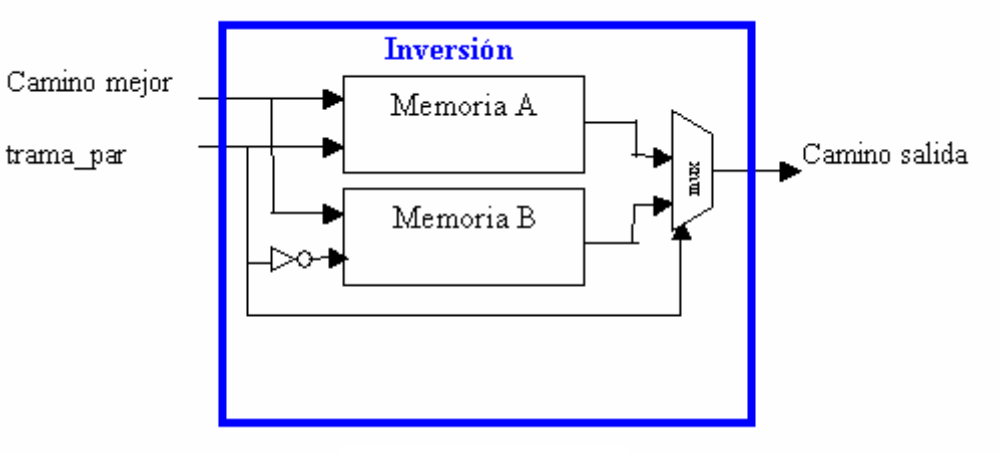


Figura 6.2.4.1.14

La señal de *control* tiene el mismo significado que en el módulo AdC.

Una vez que ya tenemos el camino invertido con la menor métrica, únicamente tenemos que, con la ayuda de la tabla de salida, decodificar los estados para obtener los bits de salida.

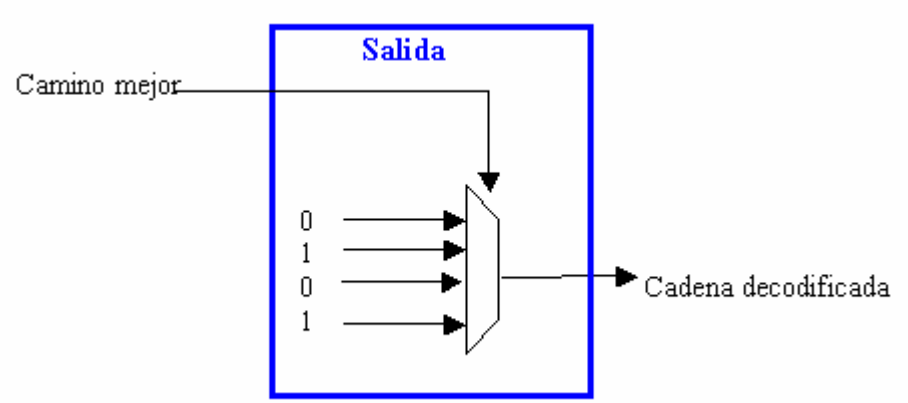


Figura 6.2.4.1.15

Controlador:

Como hemos visto en la descripción de los módulos que componen el decodificador Viterbi, hay unas señales que deben ser generadas por un módulo externo, al que llamaremos *Controlador*.

Una de las señales que debe controlar es el contador de símbolos que han entrado en el decodificador, la señal *índice*. Cada vez que entra un símbolo se incrementa en 1 este contador hasta llegar a 17 (por usar un decodificador de 4 estados), momento en que vuelve a 0.

La señal que controla en qué memoria tanto del módulo AdC como del submódulo Inversión (dentro del módulo MV) también tiene que ser generada por el Controlador. Esta señal, *trama_par*, tiene que cambiar su valor (de 0 a 1 o de 1 a 0) cuando comience el procesamiento de una nueva trama. Así, por ejemplo, si trama_par es 0, la memoria A estará en modo escritura, mientras la memoria B adoptará el modo lectura.

Otra señal que tiene que generar el Controlador es la señal de fin de trama, *fin_Trama*. Esta señal controla uno de los multiplexores del submódulo Mejor Camino para que se comience a seleccionar el camino de menor métrica a partir del valor de la señal mejor.

Todas estas señales las genera únicamente a partir de la señal de reloj.
El esquema del controlador sería:

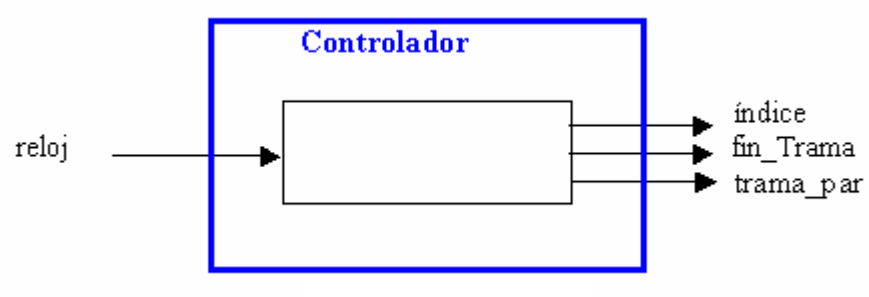


Figura 6.2.4.1.16

6.2.4.2 Simulación

6.2.4.2.1 V-System

A continuación mostramos los resultados de un ejemplo simulado en V-SYSTEM.

Los símbolos que hemos introducido para realizar la simulación son los equivalentes a la transmisión del siguiente mensaje que se corresponde con el ejemplo que hemos estado describiendo durante todo el proceso:

- el mensaje de entrada es:

  0 1 0 1 1 1 0 0 1 0 1 0 0 0 1

  que codificado en símbolos se corresponde con

  00 11 10 00 01 10 01 11 11 10 00 10 11 00 11 10 11

  y que con la alteración de bits debida al ruido AWGN se traduce a

  00 11 11 00 01 10 01 11 11 10 00 00 11 00 11 10 11

La cadena recuadrada se corresponden por tanto con la entrada a nuestro decodificador, y la salida del mismo como vemos a continuación es el mensaje de entrada.

Para la simulación en V-system usamos un archivo .do con las siguientes entradas forzadas:

```
force simb 00,11 20,11 40,00 60,01 80,10 100,01 120,11 140,11 160,10
           180,00 200
force simb 00 220,11 240,00 260,11 280,10 300,11 320
force relojBuffer 1,0 5 -repeat 10
```

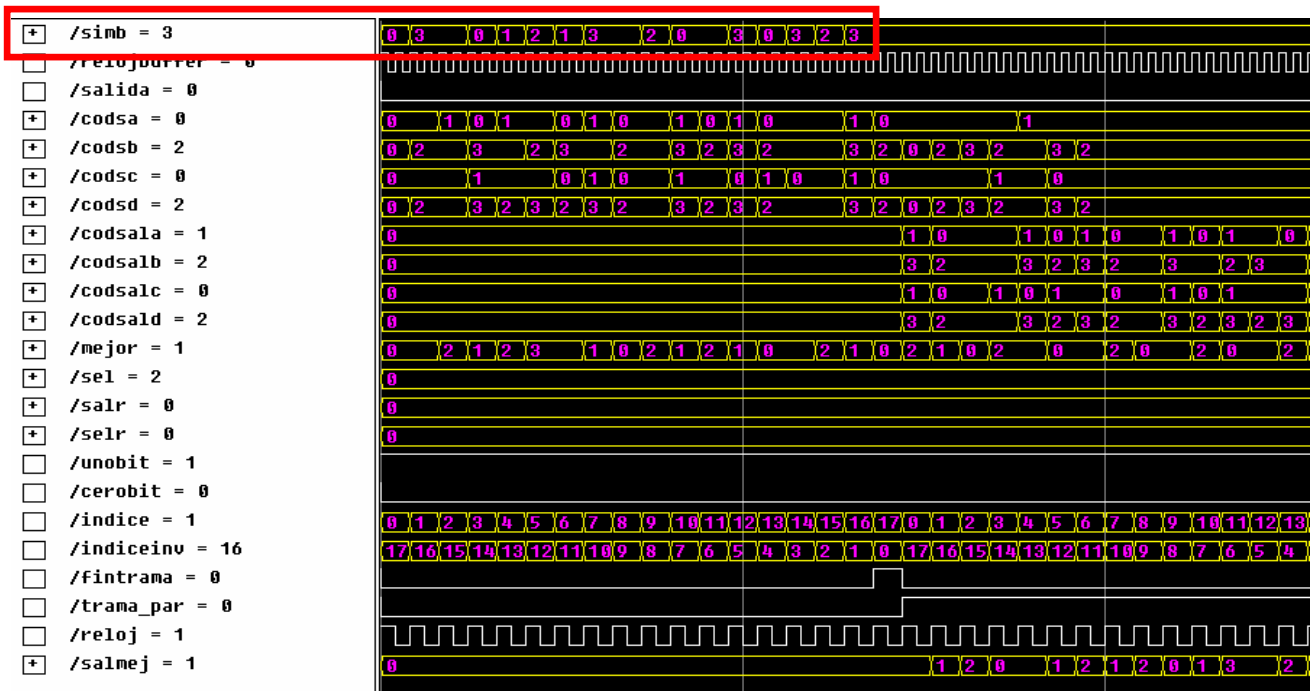


Figura 6.2.4.2.1.1

En la simulación de la figura 6.2.4.2.1.1 podemos observar como la entrada "simb" se corresponde con los símbolos mencionados anteriormente para la trama.

En la figura 6.2.4.2.1.2 observamos la salida del decodificador para dicha tramas.

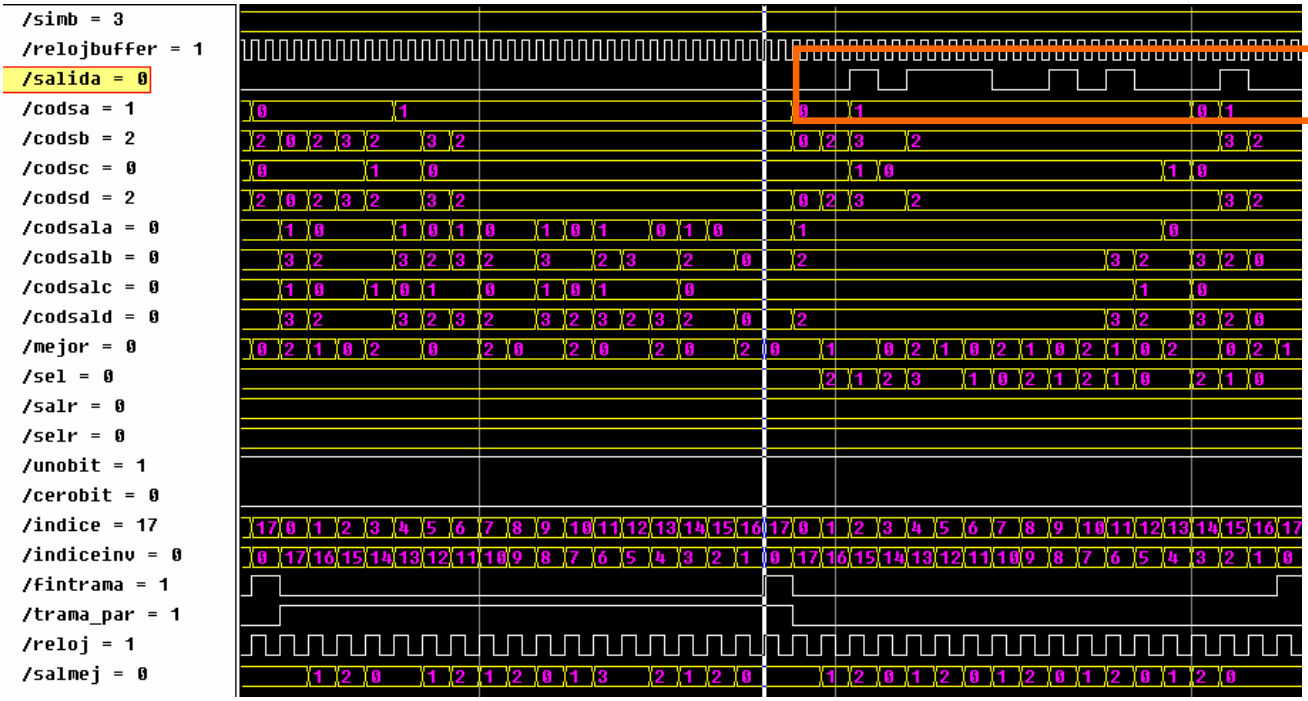


Figura 6.2.4.2.1.2

El recuadro naranja muestra la salida para la trama de símbolos descrita. Podemos observar que el resultado es:

0 0 1 0 1 1 1 0 0 1 0 1 0 0 0 1 0 0

Los bits marcados en rojo deben ser eliminados ya que corresponden a la salida de los flushing bits. El mensaje original se corresponde con los símbolos en negro.

6.2.4.2.2 Xilinx ISE

Otra de las herramientas que hemos utilizado para comprobar el correcto funcionamiento de nuestro sistema es Xilinx ISE. Con esta herramienta se puede ver con mayor detalle una futura implementación en una FPGA.

SINTESIS:

Realizando el proceso de síntesis, se obtiene un circuito, según se ha descrito con VHDL.

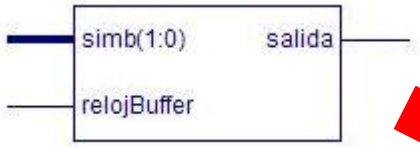


Figura 6.2.4.2.2.1

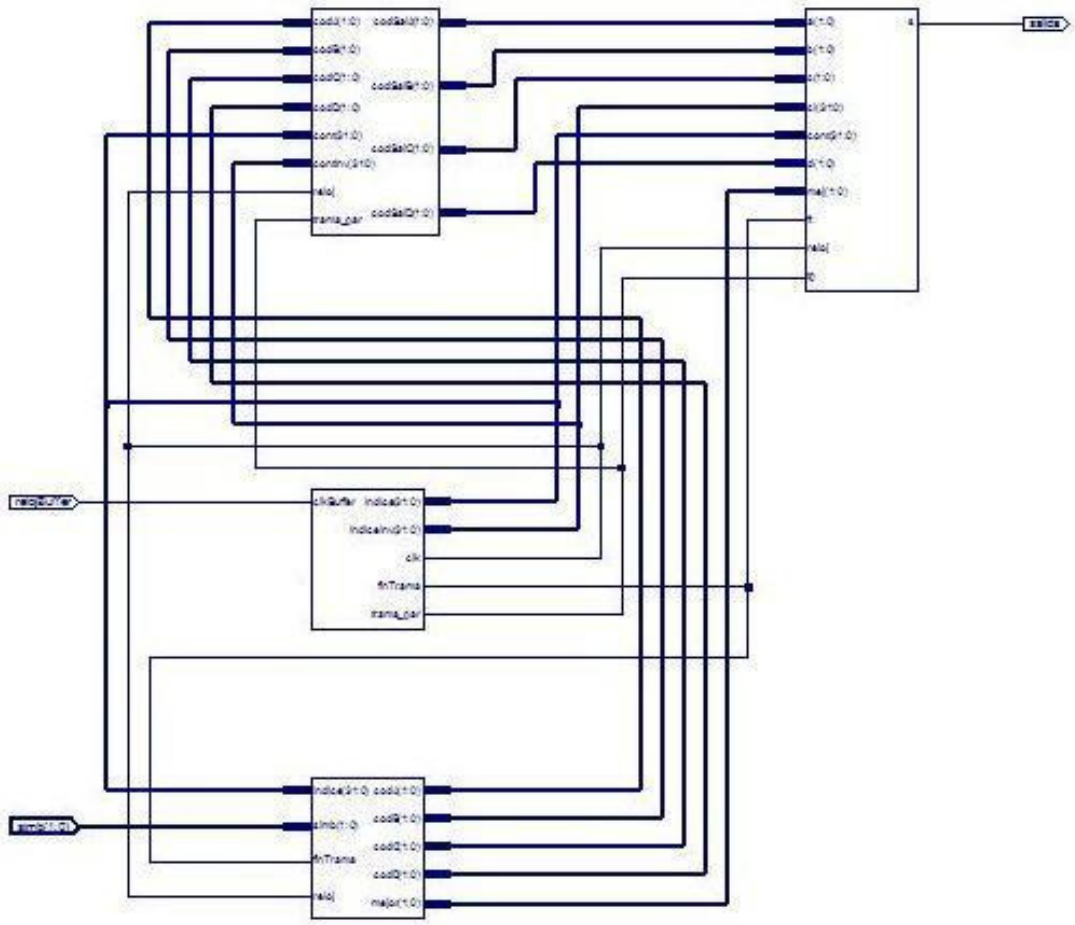


Figura 6.2.4.2.2.2

Éste sería el aspecto del decodificador que se ha implementado. Así, también se podrían ver todos los módulos y submódulos de los que está compuesto.

Ponemos como ejemplo, el módulo MV (Figura 3) consta de los submódulos Mejor Camino (Figura 6.2.4.2.2.4), Inversión (Figura 6.2.4.2.2.5) y Código Salida (Figura 6.2.4.2.2.6).

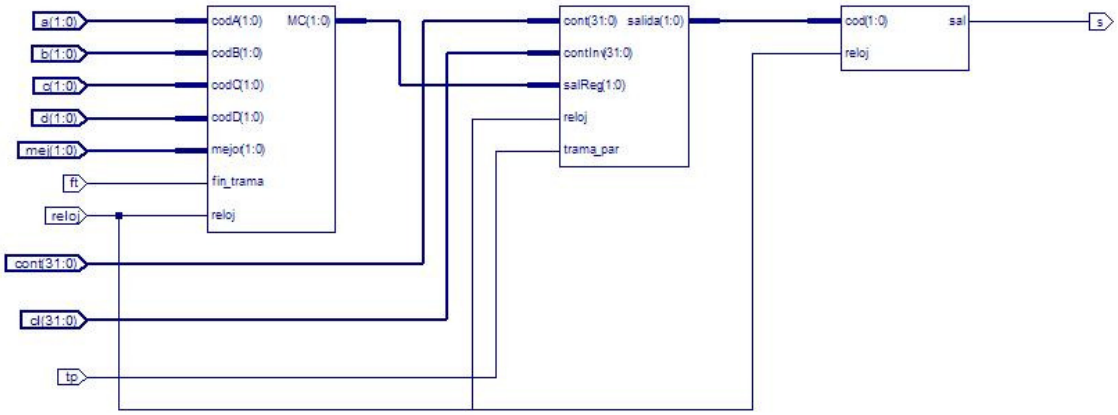


Figura 6.2.4.2.2.3

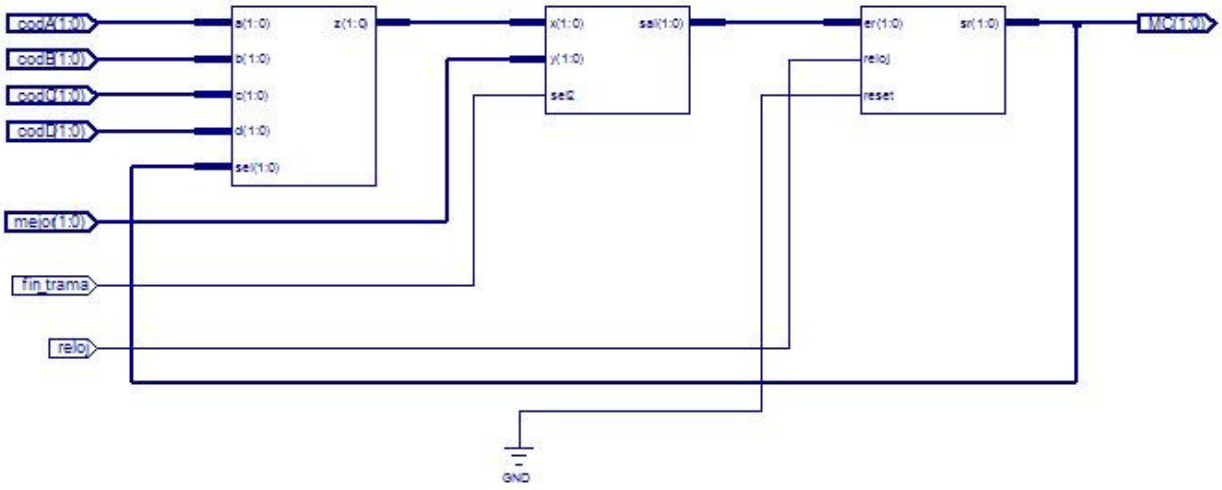


Figura 6.2.4.2.2.4

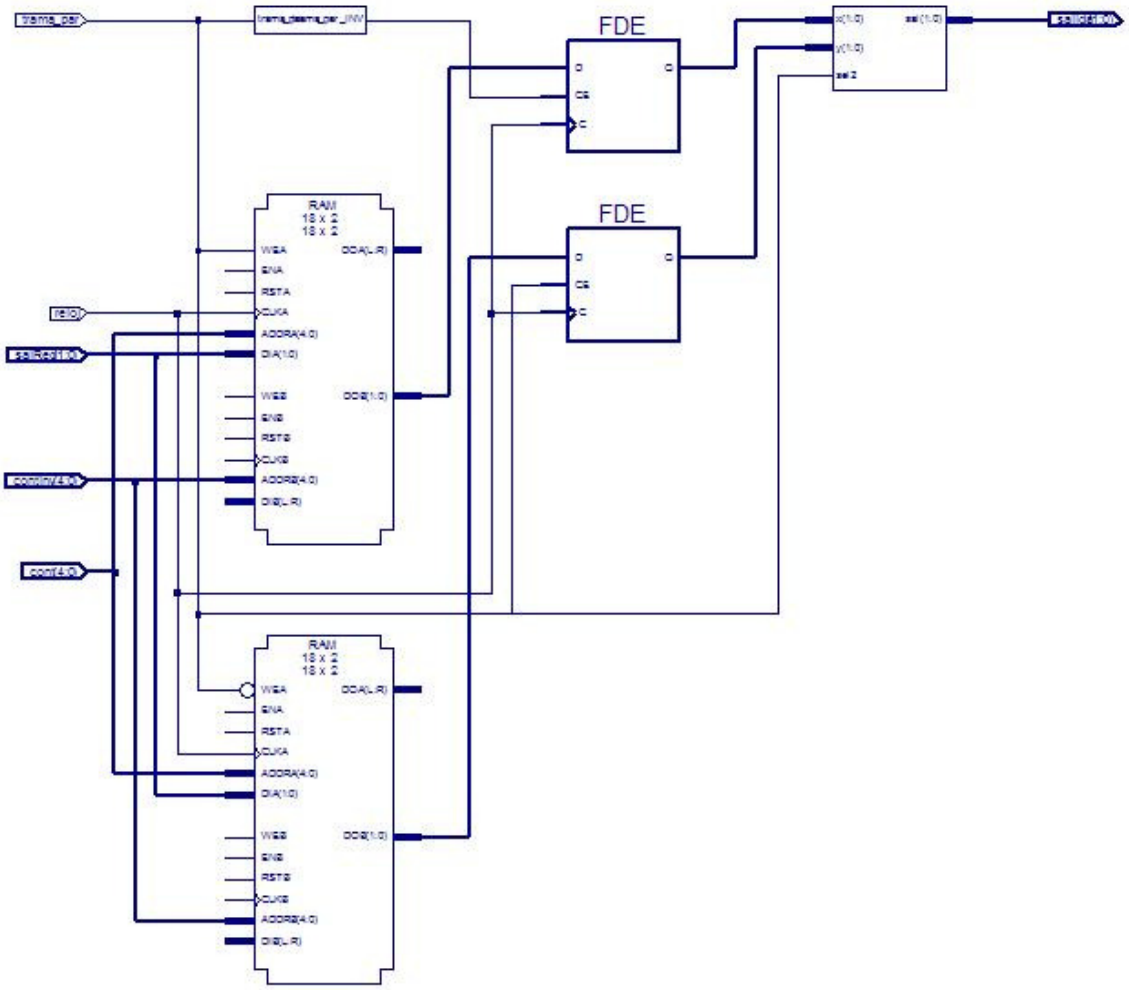


Figura 6.2.4.2.2.5

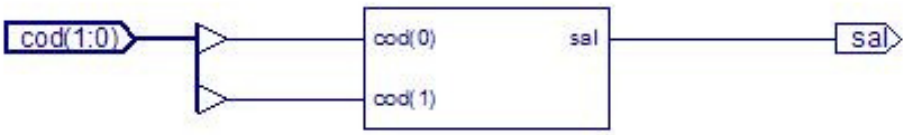


Figura 6.2.4.2.2.6

También se obtienen los siguientes informes acerca de qué elementos de la FPGA se han utilizado.

HDL Synthesis Report

Macro Statistics

# LUT RAMs	: 2
18x2-bit dual-port distributed RAM	: 2
# Adders/Subtractors	: 1
32-bit adder	: 1
# Counters	: 1
32-bit up counter	: 1
# Registers	: 16
1-bit register	: 9
2-bit register	: 1
3-bit register	: 4
32-bit register	: 2
# Latches	: 152
2-bit latch	: 152
# Comparators	: 9
3-bit comparator greater	: 7
32-bit comparator greatequal	: 1
32-bit comparator less	: 1
# Multiplexers	: 17
2-bit 18-to-1 multiplexer	: 8
2-bit 4-to-1 multiplexer	: 5
3-bit 4-to-1 multiplexer	: 4
# Xors	: 56
1-bit xor2	: 56

Advanced HDL Synthesis Report

Macro Statistics

# LUT RAMs	: 2
18x2-bit dual-port distributed RAM	: 2
# Adders/Subtractors	: 1
32-bit adder	: 1
# Counters	: 1
32-bit up counter	: 1
# Registers	: 77
Flip-Flops	: 77
# Latches	: 152
2-bit latch	: 152
# Comparators	: 9
3-bit comparator greater	: 7
32-bit comparator greatequal	: 1
32-bit comparator less	: 1

Multiplexers : 17
2-bit 18-to-1 multiplexer : 8
2-bit 4-to-1 multiplexer : 5
3-bit 4-to-1 multiplexer : 4
Xors : 56
1-bit xor2 : 56

Final Report

Design Statistics
IOs : 4

Cell Usage :
BELS : 726
BUF : 2
GND : 1
INV : 8
LUT1 : 35
LUT2 : 24
LUT2_L : 1
LUT3 : 131
LUT3_D : 6
LUT3_L : 8
LUT4 : 281
LUT4_D : 18
LUT4_L : 19
MUXCY : 68
MUXF5 : 65
MUXF6 : 24
VCC : 1
XORCY : 34
FlipFlops/Latches : 306
FDC : 14
FDE : 43
FDR : 1
FDRE : 32
LDE : 216
RAMS : 4
RAM16X1D : 4
Clock Buffers : 3
BUFG : 2
BUFGP : 1
IO Buffers : 3
IBUF : 2
OBUF : 1

Utilización de dispositivos:

Device utilization summary:

Selected Device : xa2s50etq144-6

Number of Slices:	370	out of	768	48%
Number of Slice Flip Flops:	306	out of	1536	19%
Number of 4 input LUTs:	531	out of	1536	34%
Number used as logic: 523				
Number used as RAMs: 8				
Number of bonded IOBs:	4	out of	102	3%
Number of GCLKs:	3	out of	4	75%

IMPLEMENTACIÓN

Tras el proceso de implementación, se puede ver cómo quedaría el circuito del decodificador qué partes de la FPGA estarían ocupadas, si lo implementaramos sobre una de ellas.

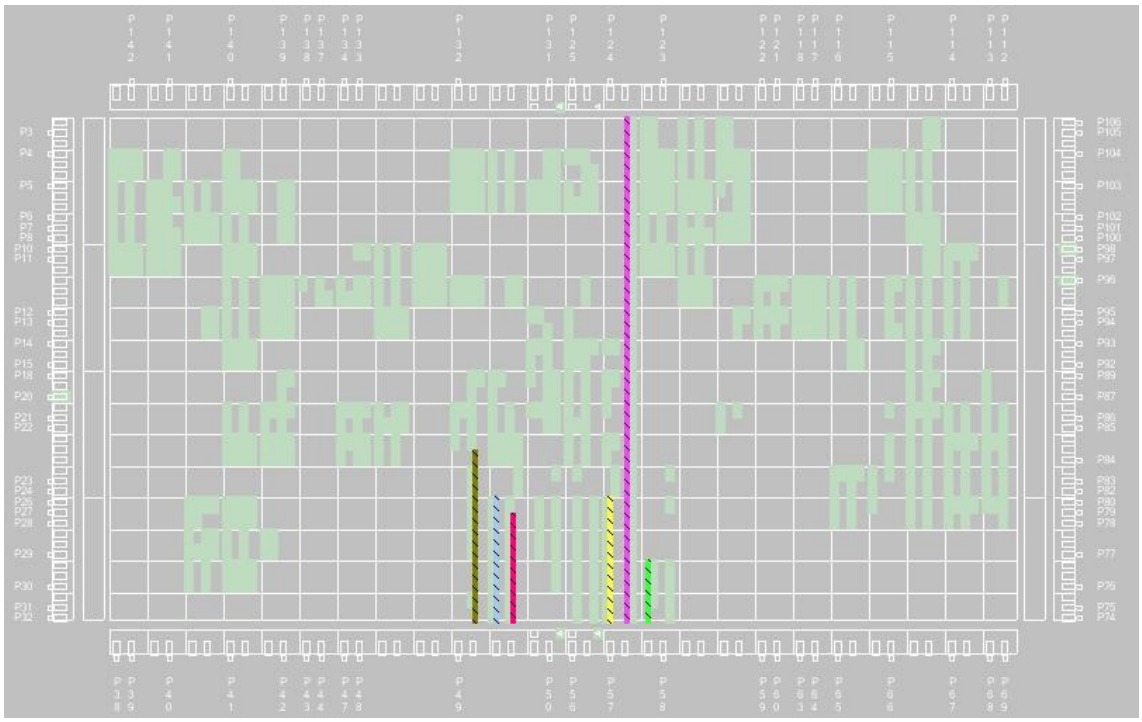


Figura 6.2.4.2.2.7

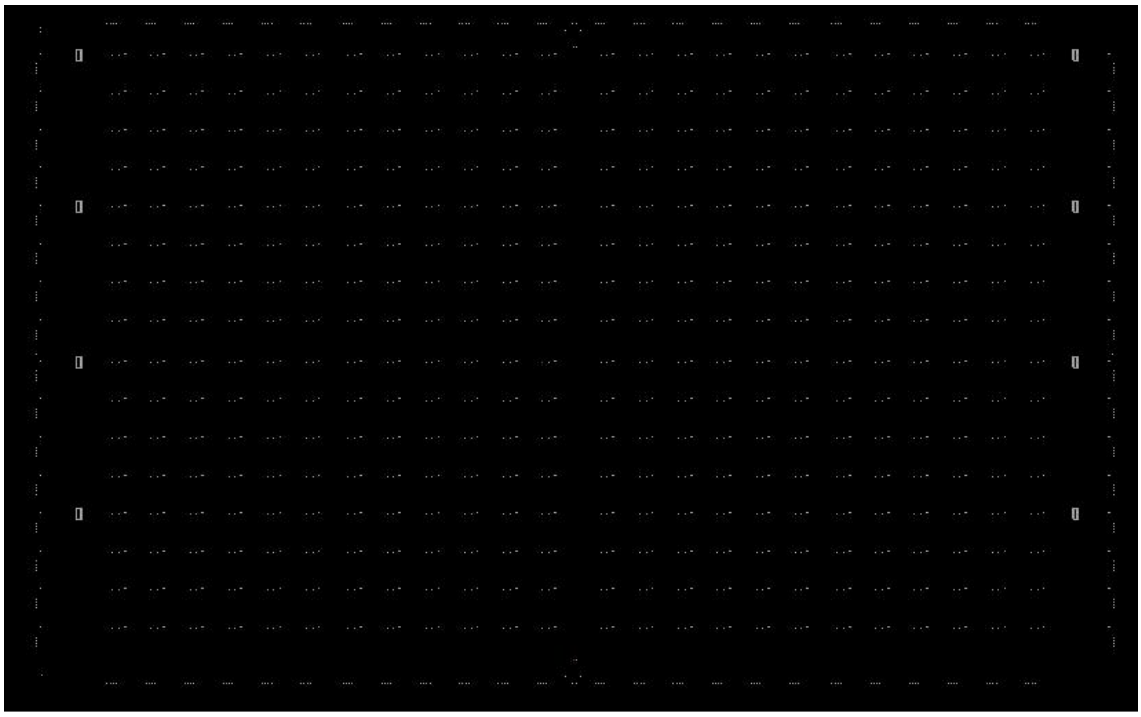


Figura 6.2.4.2.2.8

También se puede ver cuáles son las conexiones con mayor retardo:

```
The 20 worst nets by delay are:
+-----------------+-----------+
| Max Delay       | Netname   |
+-----------------+-----------+
   6.337          acs/ACS_B/comp/_n0000
   6.141          contr/trama_par_1
   5.939          contr/indice<0>
   5.333          contr/trama_par
   5.114          acs/Ker0_wg_cy7
   4.988          acs/Ker0_wg_cy_rn_6_2
   4.969          acs/ACS_A/comp/_n0000
   4.700          contr/indiceInv<1>
   4.627          acs/Ker0_wg_cy_rn_6_1
   4.565          contr/indiceInv<2>
   4.549          contr/indiceInv<3>
   4.417          acs/ACS_C/comp/_n0000
   4.046          contr/indice<1>
   3.966          adc/_n0437
   3.907          adc/_n0440
   3.778          contr/cont
   3.741          contr/indiceInv<4>
   3.700          contr/indiceInv<0>
   3.696          adc/_n0427
   3.666          adc/_n0441
---------------------------------
```

7. Conclusiones

Para la implementación del codificador/decodificador Viterbi fue necesario un estudio previo de dicho sistema con un trabajo de documentación asociado. Esto nos permitió conocer el funcionamiento del sistema para poder comenzar con el diseño del mismo.

Inicialmente diseñamos el codificador/decodificador Viterbi de forma secuencial y lo implementamos en un lenguaje de alto nivel para comprender el funcionamiento y la labor que realiza. Elegimos Java como lenguaje de implementación debido a la familiaridad que mantenemos con el lenguaje, sin embargo, se podría haber elegido cualquier otro lenguaje para habituarse al sistema.

Una vez implementado el proyecto en java, decidimos mejorar el diseño. Para ello optamos por segmentar el decodificador Viterbi, de esta forma conseguiríamos disminuir el tiempo en el cual se decodificara una trama completa al mismo tiempo que aprovecharíamos todos los recursos del decodificador. El codificador no lo segmentamos pues concluimos que no aportaba ningún beneficio, esto se debe a que el codificador es muy simple, por lo tanto no tenía ningún sentido segmentarlo pues no habríamos obtenido ninguna mejora.

El diseño de este codificador/decodificador Viterbi segmentado evolucionó sufriendo modificaciones que dieron lugar a las distintas versiones hasta obtener una versión definitiva que consideramos la mejor de todas ellas.

Esta última versión se implementó inicialmente en Java, aunque simultáneamente se empezó a implementar en un lenguaje de bajo nivel (Vhdl). Acordamos implementarlo en Java como un lenguaje abstracto que nos permitiera comprender su actividad, y al mismo tiempo mostrar el funcionamiento del codificador/decodificador Viterbi mediante una simulación. De esta forma, se podría observar fácilmente qué es lo que estaría realizando nuestro sistema en cada ciclo de reloj, así quedaría patente la segmentación, pues en un ciclo se podrían percibir las acciones realizadas por el decodificador.

La implementación en un lenguaje de bajo nivel era necesaria en nuestro proyecto ya que consistía en un diseño hardware, por lo tanto, la visualización del proyecto en Vhdl nos permitiría aproximarnos a la arquitectura real del sistema. Gracias a la simulación en Vhdl pudimos comprobar el correcto funcionamiento del sistema.

Concluyendo, hemos diseñado la arquitectura de un codificador/decodificador Viterbi secuencial, que tras varias modificaciones hemos conseguido el diseño del codificador/decodificador Viterbi segmentado.

8. Bibliografía

- **Digital video Broadcasting (DVB);framming structure,channel coding and modulation for digital terrestrial television**. ETSI EN 300 744

- **Coarse Grain Reconfigurable Architectures**
  por Reiner Hartenstein (embedded tutorial) CS Dept. (Informatik), University of Kaiserslautern.

- **MaRS: A Macro-pipelined Reconfigurable System**

- **Networks on Chips:A New SoCParadigm**
  Por *Luca Benini* University of Bologna*, Giovanni De Micheli* Stanford University

- **Tutorial on Convolutional Coding with Viterbi decoding**
  http://home.netcom.com/%7Echip.f/viterbi/algrthms.html

- **Arquitectura de un decodificador convolucional a partir del algoritmo de *Viterbi*** *por* Juli Ordeix1, Pere Martí1, Moisès Serra (Dept. d'Electrònica i Telecomunicacions, Universitat de Vic.), Jordi Carabina(Dept. Informàtica, ETSE, Universitat Autònoma de Barcelona.)

  http://www.uvic.cat/eps/recerca/codisseny/docs/publicacions/arquitectura-%20viterbi-04.pdf

- **Utilización de codificación convolucional y decodificación de Viterbi**
  por Álvaro Campuzano Zubeldia, Iker Varela Cuadrado.

  http://www.tecnun.es/Asignaturas/transdat/ficheros%5Cviterbi.pdf

Nota: Aquellos artículos sin referencia Web fueron suministrados por el profesor.

Los abajo firmantes autorizan a la Universidad Complutense de Madrid a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la propia memoria como el código, la documentación y/o el prototipo desarrollado.

Fdo: Laura Díaz Escalona	**Fdo: MªConcepción Fernández Sánchez**	**Fdo: Inés González de Miguel**