UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Generación automática de códigos de corrección de errores en VHDL a partir de su matriz de paridad H

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Autor: Cristina Morilla Rodríguez
Tutor: David de Andrés Martínez
Cotutor: Joaquín Gracia Morán

Curso 2020-2021

Resum

Els sistemes d'informació poden patir danys que modifiquen el missatge que es pretén transmetre, en un moment donat, per causes molt diverses. Com a solució, els sistemes tolerants a fallades, per mitjà dels codis correctors d'errors, són capaços d'assolir que un sistema seguisca funcionant tot i els contratemps; no obstant això, la seua implementació no està exempta de realitzar-se sense fallades. D'aquesta manera, la generació automàtica d'aquests codis es presenta com una ferramenta que persegueix minimitzar la intervenció humana i, en conseqüència, construir sistemes més robustos. El present estudi proposa una solució fonamentada en automatitzar la generació dels circuits encarregats d'implementar aquests codis en VHDL. Tot aquest procés es durà a terme mitjançant el llenguatge Python, recolzant-nos en l'ús d'algunes de les seues llibreries i amb un enfocament basat en la metaprogramació.

Paraules clau: VHDL, Python, codis correctors d'errors, sistemes tolerants a fallades, metaprogramació

Resumen

Los sistemas de información pueden sufrir daños que modifiquen el mensaje que se pretende transmitir, en un momento dado, por causas muy diversas. Como solución, los sistemas tolerantes a fallos, por medio de los códigos detectores de errores, son capaces de conseguir que un sistema siga funcionando a pesar de estos percances; no obstante, su propia implementación no está exenta de realizarse sin errores. De esta forma, la generación automática de dichos códigos se presenta como una herramienta que persigue minimizar la intervención humana y, por consiguiente, construir sistemas más robustos. El presente estudio propone una solución fundamentada en automatizar la generación de los circuitos encargados de implementar estos códigos en VHDL. Todo ello se llevará a cabo a través del lenguaje Python, apoyándonos en el uso de algunas de sus librerías y con un enfoque basado en la metaprogramación.

Palabras clave: VHDL, Python, códigos correctores de errores, sistemas tolerantes a fallos, metaprogramación

Abstract

Information systems can suffer from damage that could modify the message to transmit, in a moment, because of different reasons. As a solution, fault tolerant systems, by means of error correction codes, are capable to achieve that a system keeps going on spite of these incidents; however, this implementation is not free from being made without errors. This way, generating these codes automatically presents itself as a tool to minimize human intervention and, as a result, build more robust systems. The present study suggests a solution based on automatizing the generation of the circuits that implements these codes in VHDL. All of that will be accomplished by means of Python language, relying on the use of some of its libraries and with a metaprogramming approach.

Key words: VHDL, Python, error correction codes, fault tolerant systems, metaprogramming

Índice general

Índice de figuras

Índice de tablas

CAPÍTULO 1
Introducción

Todos estamos familiarizados con el siguiente escenario: un emisor transmite un mensaje que viaja por un canal que, en otro punto, será procesado por un receptor. Esta circunstancia nos recuerda a la diaria comunicación humana, cada vez más acontecida a través de medios digitales. Al igual que es común que la comunicación verbal se «distorsione», provocando malentendidos, la transmisión de la información por medios digitales también es capaz de sufrir modificaciones que comprometan el funcionamiento de un sistema. Para este último caso, existen mecanismos específicos capaces de dotar de robustez a los mismos, permitiendo que estos sigan funcionando a pesar de los contratiempos pues, ocasionalmente, su correcto desempeño puede resultar de una relevancia crítica.

Una de las formas por las que este comportamiento puede alcanzarse es a través del diseño de unos circuitos especiales preparados para soportar los fallos que pudieran darse. Si bien las bases teóricas de los mismos son muy anteriores a la introducción de los sistemas de información en prácticamente casi todos los aspectos de nuestras vidas, las posibilidades tecnológicas desde entonces no han hecho más que crecer, por lo que resulta razonable llevar a cabo nuevas aproximaciones adaptadas a las herramientas actuales. Entre las mismas, se hallan los lenguajes de descripción de hardware, que son especialmente útiles para definir circuitos electrónicos digitales.

Al igual que el desarrollo de soluciones software, la definición de estos circuitos es un proceso largo, costoso, y muy orientado al detalle, por lo que realizar un despliegue íntegramente manual del mismo puede dar pie a que se introduzcan fallos humanos durante esta fase. Así, el presente trabajo pretende, inspirándose en enfoques previos, presentar una solución para automatizar todo lo posible la generación del código que describa a estos circuitos.

1.1 Motivación

Cursando los estudios de Ingeniería Informática, en la asignatura de Diseño de Sistemas Digitales, pudimos aprender, entre otras cosas, a construir sistemas digitales mediante lenguajes de descripción de hardware e implementarlos en dispositivos lógicos reconfigurables. Hacia el final de la asignatura, pudimos vislumbrar la importancia que el diseño basado en FPGAs estaba cobrando en distintos ámbitos, tanto aquellos relacionados con la industria como con la investigación. Una de las áreas que despertó nuestro interés fue la de los Sistemas tolerantes a fallos; si bien durante la carrera hemos podido estudiar mecanismos que operan bajo el mismo principio (el protocolo TCP, por ejemplo) estamos acostumbrados a concebir los fallos como sucesos catastróficos; por ello, el hecho de que un sistema sea capaz de continuar con su correcto funcionamiento a pesar de

los mismos, siempre resulta asombroso. Aunque esta sea un área muy compleja que requiera de mucho más grado de especialización que las que otorga la carrera, nos pareció interesante encontrar un hueco en el que poder aportar mejoras gracias a las habilidades adquiridas durante estos cuatro años de aprendizaje.

1.2 Objetivos

El presente estudio consta de los siguientes objetivos:

- automatizar la creación de circuitos tolerantes a fallos;
- verificar dichos circuitos a través de la simulación funcional;
- reducir costes de desarrollo y ejecución;
- permitir la escalabilidad del programa;
- derivado de lo anterior, disminuir la posibilidad de cometer errores en el desarrollo de dichos circuitos.

1.3 Estructura de la memoria

La presente memoria consta de cinco capítulos principales. En *Análisis del problema*, se presentan conceptos básicos de codificación, la taxonomía relacionada con los tipos de fallos y se introducen los códigos correctores de errores, haciendo especial hincapié en los códigos de Hamming. A continuación, en *Estado de la cuestión*, se hace un recorrido por los últimos avances concernientes a la automatización de código tolerantes a fallos, tanto por parte de la industria como de la academia. En *Arquitectura de la solución*, se hace un repaso del plan que pretendemos llevar a cabo para automatizar la generación de códigos correctores de errores, pasando por sus partes principales. Seguidamente, en *Desarrollo de la solución propuesta*, explicamos más en detalle todo el proceso relacionado con la implementación del programa, justificando posibles cambios y mejoras que pudieran darse con respecto a la idea inicial. Finalmente, en *Resultados*, verificaremos si el programa cumple o no con los objetivos propuestos; al mismo tiempo, también llevaremos a cabo una estimación de los tiempos de ejecución.

CAPÍTULO 2
Análisis del problema

En 1948, Claude Shannon publicó uno de sus artículos más influyentes: *A Mathematical Theory of Communication* (1948), que sentaría las bases de la Teoría de la Información, y en el cual desarrolló conceptos clave como la eficiencia y la confiabilidad en la transmisión de datos a través de canales con ruido. Según afirman Richardson y Urbanke [1], en este artículo, Shannon vislumbraba una posible solución a este problema: la codificación. Más adelante, Richard Hamming desarrolló un código que sería bautizado con su nombre, y que constituye uno de los ejemplos fundacionales de la teoría de la codificación. Aunque ambas se encuentren íntimamente relacionadas (parafraseando a Moser y Chen [2]: «la teoría de la información proporciona los límites performativos que pueden alcanzarse mediante la codificación»), de un tiempo a esta parte han venido desarrollándose de forma paralela. Así pues, en los siguientes apartados nos centraremos en definir aquellos aspectos de la codificación que resultan relevantes para entender el problema al que nos enfrentamos. Cabe resaltar que la codificación es una teoría con una fuerte carga matemática, pero algunas de sus mecánicas se pueden entender, en gran medida, a través de ejemplos. Ya que el presente trabajo no pretende, en modo alguno, profundizar en las bases matemáticas, contextualizaremos los conceptos teóricos a través de los mismos.

2.1 Conceptos básicos

Antes de adentrarnos en los aspectos más detallados de la codificación, resulta apropiado definir una serie de conceptos que representan las amenazas para la confiabilidad de un sistema: hablamos de los *fallos*, *errores* y *averías*.

Según Avizienis [3], una avería es un evento que ocurre cuando el comportamiento de un sistema se desvía del correcto, entendiéndose dicha corrección como la especificación funcional esperada. Cada desvío particular del funcionamiento esperado dentro del estado total del sistema representa, a su vez, un error. Por último, el fallo constituye la causa hipotética que materializa el error. A su vez, los fallos pueden ser internos o externos a un sistema (uno interno puede permitir que otro externo dañe el sistema) y activos o latentes (dependiendo de si causan o no un error).

Ejemplo de la cadena *fallo, error, avería* en el mundo físico: la anomalía del Atlántico Sur [4]
El campo magnético terrestre, entre otras cosas, nos protege de la radiación que viene del espacio, produciéndose así cinturones de partículas ionizadas conocidos como *cinturones de Van Allen*; sin embargo, su disposición no es homogénea en todo el globo. Existe, en Sudamérica, una zona donde el campo magnético terrestre es más débil, lo cual provoca que la radiación incidente sea mayor que en otras zonas.

En consecuencia, esto hace que el cinturón de Van Allen que la atraviesa aparezca a una menor altura, de tal forma que interfiere en la órbita de objetos espaciales como los satélites. Para evitar daños en sus componentes electrónicos, generalmente, estos satélites se apagan cuando atraviesan esta zona. Si quisiéramos identificar en este ejemplo del mundo físico la cadena *fallo*, *error*, *avería*, quedaría como sigue:

- fallo: la partícula ionizada del cinturón impacta en una celda de la memoria RAM del satélite, aumentando su nivel de carga (fallo externo);
- error: el valor lógico que representa la carga en esa celda de memoria ha cambiado de '0' a '1' (fallo activo), con lo que el estado del sistema es diferente (error). Si se leyera esa celda de memoria para operar con el dato almacenado, entonces el error se propagaría por el circuito. Si nunca se accediera a esa celda de memoria, estaría latente. Por último, si se sobrescribiera con un valor correcto antes de leerlo, el error se habría enmascarado;
- avería: al operar con el dato erróneo, la salida que proporciona el satélite no es la correcta (avería). En el caso de que la salida fuese la correcta significaría, también en este caso, que el error se habría enmascarado.

En términos generales, podemos entender la confiabilidad de dos formas: «la habilidad de ofrecer un servicio que pueda ser fiable de forma justificada» o, de forma alternativa, «la habilidad de evitar averías que sean más frecuentes y severas de lo aceptable» [3]. Las siguientes representan las propiedades principales que harían a un sistema confiable:

- disponibilidad: probabilidad de que un sistema proporcione el servicio correctamente en el instante de tiempo t (el tiempo en el que proporciona el servicio esperado dividido por el tiempo total de operación);
- fiabilidad: probabilidad de que un sistema proporcione el servicio correctamente hasta el instante de tiempo t (el sistema proporcionará el servicio correcto desde el instante de tiempo 0 hasta t);
- seguridad: la ausencia de consecuencias catastróficas en los usuarios y el entorno;
- integridad: la ausencia de alteraciones en el sistema;
- mantenibilidad: la habilidad de someterse a modificaciones y reparaciones.

2.1.1. Tipos de fallos

Existe una taxonomía específica para clasificar los fallos, sumando hasta ocho los elementales. Además de estos, la clasificación permite combinaciones de los mismos, siempre y cuando estas sean consistentes. Así, Avizienis *et al.* han identificado hasta 31 combinaciones, las cuales podemos visualizar en la Tabla 2.1.

Fallos	Fase de creación u ocurrencia	Desarrollo: ocurren durante (a) el desarrollo del sistema, (b) mantenimiento durante la fasede uso o producción, y (c) generación de los procedimientos para la operación o el mantenimiento del sistema
		Producción: ocurren durante la fase de uso o producción
	Límites del sistema	Internos: se originan dentro de los límites del sistema
		Externos: se originan fuera de los límites del sistema y los errores se propagan dentro a través de la interacción o interferencia con el mismo
	Causa fenomenológica	Naturales: causados por fenómenos naturales sin interacción humana
		Humanos: resultado de acciones humanas
	Dimensión	Hardware: se originan o afectan al hardware
		Software: afectan al software (datos o programas)
	Objetivo	Maliciosos: introducidos por un humano con la intención de dañar al sistema
		No maliciosos: introducidos sin mala intencionalidad
	Intención	Deliberados: resultados de una mala decisión
		No deliberados: introducidos de forma inconsciente
	Capacidad	Accidentales: introducidos de forma inadvertida
		Por incompetencia: resultados de la falta de profesionalidad o inadecuación de los profesionales del equipo en el proyecto
	Persistencia	Permanentes: de presencia continuada en el tiempo
		Transitorios: de presencia limitada en el tiempo

Tabla 2.1: Fallos elementales

En nuestro caso, no obstante, cabe destacar los fallos físicos. Dentro de ellos encontramos los naturales, que son todos aquellos causados por fenómenos del mundo físico sin intervención humana. De igual forma, podemos clasificar los fallos naturales como sigue:

- en desarrollo: aquellos que se originan durante el desarrollo del sistema;
- en producción: se originan en la fase de uso. En esta fase, los fallos pueden ser:
  - internos: relacionados con el deterioro físico inevitable;
  - externos: relacionados con los procesos naturales que se originan fuera del sistema en cuestión, como puede ser la radiación, ruido, etc.

También resulta de especial interés la clasificación que se establece en el informe Fault Representativeness [5], teniendo en cuenta esta vez de la duración de los fallos:

- permanentes: su durabilidad es larga e indefinida;
- transitorios: normalmente, de corta duración;
- intermitentes: coinciden con los transitorios en su corta perdurabilidad, pero aparecen y desaparecen repetidamente a lo largo del tiempo de forma no periódica.

Como añadido, y para entenderlos mejor, podemos hacer uso de los modelos de fallos, mediante los cuales podemos organizar sus efectos a distintos niveles de abstracción del circuito. Así, si bien podemos identificar el origen de un fallo a nivel de transistor, también podemos establecer en el nivel lógico (que es en el que nosotros trabajamos) sus respectivos modelos.

Estos serían, para los permanentes, los modelos que encontramos [5]:

- *stuck-at*: la salida de la puerta lógica o del biestable se queda fija a un valor lógico determinado (0 o 1);
- *delay*: se incrementa el retardo de salida del componente o línea;
- *short*: se establece una conexión entre dos pistas existentes (por ejemplo: se conectan las salidas de dos componentes, o varias entradas entre sí);
- *open-line*: se rompe una pista y la línea deja de estar conectada con el componente, con lo que tendrá entradas o salidas sin conectar;
- *bridging*: una combinación de *short* y *open-line*; es decir, se ha producido una conexión entre dos pistas que anteriormente se han roto;
- *indetermination*: el valor de voltaje de la salida del componente está entre los límites del nivel alto y bajo, con lo que resulta difícil conocer qué nivel lógico interpretará el siguiente componente que utilice ese valor; de hecho, si fueran varios los que se alimentaran de esta señal, unos podrían interpretarla como un 0 y otros como un 1.

Para los transitorios, los modelos son los que siguen:

- *bit-flip*: se invierte, aunque no de forma permanente, el nivel lógico almacenado en las memorias o biestables. Al ser un fallo transitorio, el valor podría sobrescribirse en el siguiente ciclo de reloj;
- *pulse*: se invierte el nivel lógico de una pista, afectando a la lógica combinacional. Al expirar este efecto, la pista vuelve a su valor natural;
- *short*, *delay* e *indetermination*: análogos a los explicados en los fallos permanentes, pero de duración transitoria.

Por último, para los intermitentes los más utilizados son los *delay*, que sería un incremento del retardo de salida del componente de corta duración y de forma reiterada.

En este aspecto, nos preocupan principalmente los fallos de inversión de bits (*bit-flip*), de duración transitoria y que, como hemos explicado, consisten en la inversión de los valores lógicos de un registro; esto es, de 0 a 1 o de 1 a 0. Este hecho ha sido señalado por Karnik en su artículo *Characterization of Soft Errors Caused by Single Event Upsets in CMOS*

Processes; en él se explica la gran sensibilidad a la radiación que tienen las memorias con respecto a otros componentes de los circuitos electrónicos [6]. Por ello, resulta de especial importancia poder proteger la información almacenada en escenarios como el ejemplificado al inicio de esta sección.

Hemos hablado de las amenazas y la confiabilidad, pero nos falta un último elemento para completar esta cuestión: los medios (*means*) para la confiabilidad, que nos permiten hacer frente a las amenazas una vez se producen. Entre los principales medios, encontramos los que siguen:

- prevención: evitar que ocurran o se introduzcan fallos. Un ejemplo es el *rad-hard*, que consiste en la fabricación de dispositivos capaces de resistir los efectos de la radiación ionizante;
- eliminación: detectar fallos existentes y eliminarlos antes de pasar a la fase de producción. El ejemplo más típico es el del los *tests*;
- predicción: hacer una estimación de la ocurrencia de los fallos que se producen y los que pudieran producirse antes de que se materialicen, además de sus consecuencias, para así poder establecer una hoja de ruta. En este punto podríamos encontrar la inyección de fallos, a partir de la cual podríamos determinar si un fallo desemboca en una avería y enriquecer así el modelo teórico;
- tolerancia: el medio que resulta de que los tres anteriores no hayan sido suficientes para evitar la ocurrencia del fallo. Con ello, la última opción consiste en asumir que el fallo aparece y de dotar al sistema de técnicas que lo toleren para que este no resulte en una avería. Aquí encontraríamos técnicas de redundancia espacial (*Triple Modular Redundancy*), redundancia temporal (*checkpoint* o puntos de control y *rollback*, que es volver a una anterior versión de un programa o producto físico libre de los errores encontrados en la nueva) y redundancia en la información, entre los cuales encontramos los códigos correctores de errores, que introduciremos en la siguiente sección.

2.2 Códigos correctores de errores

Tanto la transmisión como el almacenamiento de información son escenarios susceptibles de sufrir errores que comprometan la integridad de los datos; en este contexto, la codificación se presenta como una técnica capaz de otorgar tolerancia a fallos a un sistema concreto. Además de su uso en sistemas de almacenamiento de información (Reed-Solomon [7], usado en CDs o DVDs) o comunicaciones (Códigos de Redundancia Cíclica o CRC, usado en telefonía), los códigos correctores de errores (en adelante, ECCs, por sus siglas en inglés Error Correction Codes) han visto su uso extendido al ámbito de los procesadores o memorias; un ejemplo es el dispositivo *system-on-chip* GR740 [8] usado por la ESA, pensado para soportar radiaciones altas y que incorpora el anteriormente mencionado Reed-Solomon en sus memorias e interfaces principales. Existen distintos tipos de códigos, y es precisamente esta multiplicidad de ECCs la que nos permite llevar a cabo una mayor adecuación al problema en cuestión, por lo que es de suma importancia conocer las peculiaridades de la aplicación a tratar. Asimismo, es conveniente tener una visión general de las características de los códigos.

A grandes rasgos, estos se dividen en dos subtipos: los convolucionales y los de bloque. Por un lado, los convolucionales son aquellos cuya codificación y decodificación dependen tanto del mensaje enviado como de los anteriores, siendo esta la causa por la

cual deben implementarse con circuitos secuenciales. En cambio, para los de bloque, la dependencia es exclusiva del mensaje que se pretende transmitir en un momento dado, por lo que que pueden implementarse mediante circuitos combinacionales. Dentro de los de bloque encontramos, de nuevo, otra división: si la combinación lineal de dos palabras del código es a su vez otra palabra del mismo, hablamos de códigos de bloque lineales; en caso contrario, estamos ante códigos de bloque no lineales. En esta situación, nos interesan especialmente los códigos de bloque lineales, ya que el conjunto de sus características permiten sistematizar su codificación y decodificación, y esto se traduce en la posibilidad de generalizar la implementación de los mismos.

Antes de introducir un ejemplo de este tipo de códigos que nos resulta de gran interés, definiremos algunos conceptos comunes a todos los códigos de bloque lineales.

En primer lugar, un mensaje o palabra es un conjunto de k símbolos que se transmite por un canal. Antes de la codificación, nos solemos referir al mensaje como la palabra de datos u, que es un vector de k bits que representa la información original. Esta palabra será la que se codifique mediante el código de bloque lineal $\mathbb{B}(n, k)$, pasando esta a tener n bits. Según Saiz [9], el código de bloque lineal binario se define por una matriz generadora G, y es el componente que define la propia codificación y añade la redundancia.

Entendemos por redundancia la relación de los bits de código que se añaden (n-k) sobre los bits de datos originales (k). De esta forma, después de la codificación, identificamos a la palabra codificada como un nuevo vector b de n bits, obteniéndolo mediante la regla de codificación $b = u \cdot G$.

Una vez transmitida, la palabra recibida pasa a designarse mediante r, también de n bits. Por último, definimos otro vector e, del mismo tamaño que el anterior, que es el error inducido por el medio. Identificamos si hay o no error en un bit determinado en el caso de que dicho bit tome el valor 1. Podemos comprobar si la palabra recibida es correcta mediante la siguiente expresión: $r = b \oplus e$.

Aunque existen distintos tipos de decodificación, la que mejor se adapta a las características de los códigos de bloque lineales es la decodificación por síndrome. Para llevarla a cabo, hacemos uso de la matriz de paridad H. El síndrome se obtiene a partir de esta matriz y depende, exclusivamente, del vector de error. Si se desea corregir cada vector de error, deberemos tener síndromes distintos. Si queremos corregir y no solo detectar el error, el síndrome asociado deberá figurar en la tabla de búsqueda que relaciona cada síndrome con su error.

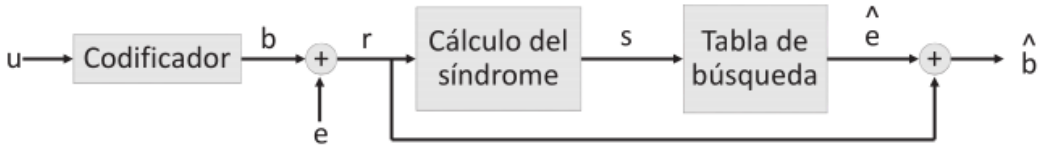

Figura 1. Proceso de codificación, transmisión y decodificación. Fuente [9]

2.2.1. Códigos de Hamming

Los códigos de Hamming, los cuales deben su nombre a su creador, Richard Hamming, se enmarcan dentro de los códigos de bloque lineales. En este apartado, materializaremos los conceptos definidos en la Sección 2.2 mediante ejemplos para afianzarlos y conocer más de cerca el problema.

Para ello, utilizaremos como ejemplo el código Hamming (7,4). En este caso, contamos con cuatro bits de datos originales y siete de código en total. El primer paso consiste en numerar los bits de la palabra, esto es, del uno al siete, y lo haremos en binario. Aquellos bits que solo tengan un 1 en su representación binaria serán bits de código; el resto, serán de datos. A continuación, disponemos en una matriz, que llamamos matriz de paridad H, todos los valores por columnas:

$$H = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix}$$

pudiendo obtener, a partir de H, las fórmulas requeridas para la codificación[1]. La forma más visual de hacerlo es situando la anterior matriz en una tabla, como sigue:

b_0	b_1	b_2	b_3	b_4	b_5	b_6	Fórmulas
		u_0		u_1	u_2	u_3	
1	0	1	0	1	0	1	$b_0 = u_0 \oplus u_2 \oplus u_3$
0	1	1	0	0	1	1	$b_1 = u_0 \oplus u_2 \oplus u_3$
0	0	0	1	1	1	1	$b_3 = u_1 \oplus u_2 \oplus u_3$

Tabla 2.2: codificación Hamming (7, 4)

Como podemos observar, para cada bit de código tenemos una fila en la tabla. Dentro de esa misma fila comprobamos, para cada bit de dato, si este está a 1 y, en caso afirmativo, dicho bit será parte de la fórmula de paridad, que no es más que la concatenación de estos bits aplicando operaciones XOR sucesivas.

Una vez codificada la palabra, esta se envía por un canal que podría producir errores en la misma. Como hemos indicado en la Sección 2.2, definimos la palabra recibida mediante r. La forma que tenemos para comprobar que la palabra recibida no ha sufrido ninguna modificación inducida por el medio es mediante la decodificación. Esta vez, la paridad calculada en cada fila nos servirá para el cálculo del síndrome y este, a su vez, será el elemento que nos indicará si se ha producido o no un error en la transmisión. De forma análoga, disponemos la palabra r en una tabla tal y como hemos hecho con b:

r_0	r_1	r_2	r_3	r_4	r_5	r_6	Fórmulas
		u_0		u_1	u_2	u_3	
1	0	1	0	1	0	1	$s_0 = r_0 \oplus r_2 \oplus r_4 \oplus r_6$
0	1	1	0	0	1	1	$s_1 = r_1 \oplus r_2 \oplus r_5 \oplus r_6$
0	0	0	1	1	1	1	$s_2 = r_3 \oplus r_4 \oplus r_5 \oplus r_6$

Tabla 2.3: decodificación Hamming(7, 4)

En caso de que el síndrome sea igual a 0, podemos considerar como correcta la palabra transmitida. En caso contrario, el valor del síndrome nos indicará en qué columna se ha producido el error, siempre que este valor coincida con alguna de ellas.

Vamos a concretar esta situación mediante ejemplos numéricos. Imaginemos que queremos transmitir una palabra $u = 1101$. Emplazamos dicha palabra en la tabla que hemos construido para ilustrar la codificación y calculamos el resultado de los bits de código:

[1]Cabe destacar que las fórmulas codificación y decodificación pueden obtenerse tanto a partir de G como de H. Dado que es H la matriz a partir de la cual realizaremos este estudio, hemos optado por ilustrar este proceso solo a partir de la misma.

b_0	b_1	b_2	b_3	b_4	b_5	b_6	Fórmulas
		1		1	0	1	
1	0	1	0	1	0	1	$b_0 = 1 \oplus 1 \oplus 1 = 1$
0	1	1	0	0	1	1	$b_1 = 1 \oplus 0 \oplus 1 = 0$
0	0	0	1	1	1	1	$b_3 = 1 \oplus 0 \oplus 1 = 0$

Tabla 2.4: ejemplo de codificación Hamming para una palabra $u = 1101$

De esta forma, la palabra codificada que enviamos será $b = 1010101$. Ahora vamos a suponer que, durante la transmisión, el último bit se ha invertido, siendo $r = 1010100$. Si colocamos a r en una tabla (tal y como hicimos en Tabla 2.2), deberemos obtener un valor del síndrome distinto de 0.

1	0	1	0	1	0	0	Fórmulas
		u_0		u_1	u_2	u_3	
1	0	1	0	1	0	1	$s_0 = 1 \oplus 1 \oplus 1 \oplus 0 = 1$
0	1	1	0	0	1	1	$s_1 = 0 \oplus 1 \oplus 0 \oplus 0 = 1$
0	0	0	1	1	1	1	$s_2 = 0 \oplus 1 \oplus 0 \oplus 0 = 1$

Tabla 2.5: ejemplo de decodificación Hamming para una palabra $u = 1101$

Podemos ver, transformando el valor binario del síndrome a decimal, que su valor es $s = 7$, coincidiendo con la columna que contiene el bit erróneo, y concluimos que ha sido en este bit donde se ha producido una inversión de su valor original.

A continuación, mostramos los circuitos resultantes, tanto del codificador como del decodificador, para este código Hamming concreto:

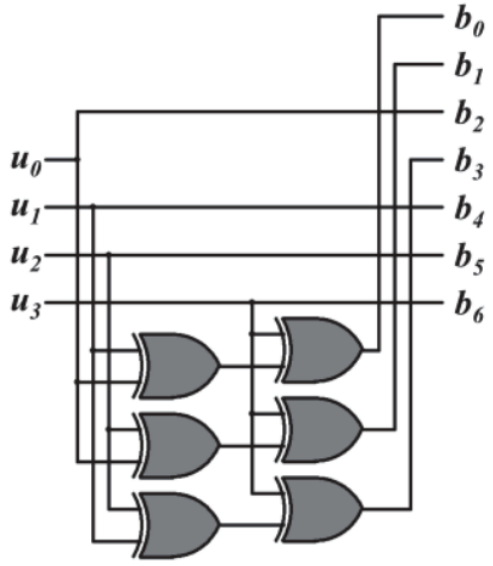

Figura 2. Circuito codificador para el código Hamming(7, 4). Fuente [10]

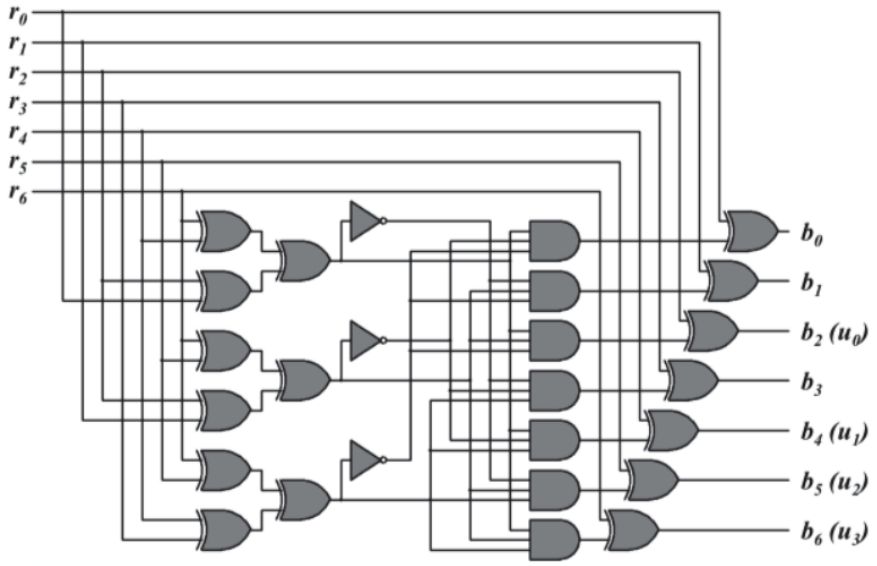

Figura 3. Circuito decodificador para el código Hamming(7, 4). Fuente [10]

2.3 Problemas detectados

Si bien, para nuestro ejemplo, hemos asumido una palabra de cuatro bits, existen otras dimensiones con las que poder definir un código Hamming: por ejemplo, de tamaño (12, 8), (21, 16), hasta poder alcanzar otros mayores, como (71, 64). Por tanto, cuanto más grande sea una palabra, mayor será el tiempo diseño de estos circuitos y, por muy pormenorizado que sea el mismo, la probabilidad de cometer errores en dicho proceso será a su vez más elevada.

El estudio teórico inicial nos ha servido para poder vislumbrar ciertos patrones en la codificación y decodificación que pueden ser generalizados, lo que en nuestro caso se traduce en poder automatizar la generación de los mismos y, de esa forma, ahorrar en tiempo, pero también en asegurarnos de que los circuitos diseñados son más robustos y pueden cumplir con las características que hacen a un sistema confiable y tolerante a fallos. Esto es, con la automatización pretendemos conseguir lo siguiente:

- separación de conceptos (lo que en inglés denominamos *separation of concerns*). Cada experto se dedica exclusivamente al área de la cual es especialista; esto es, el experto en hardware se dedica a desarrollar el módulo hardware y, el de tolerancia a fallos, a los mecanismos capaces de proporcionarla. Esto es así porque, dado el nivel de especialización cada vez mayor que requieren prácticamente todas las áreas del conocimiento, es muy complicado que una sola persona sea capaz de conocer en profundidad todas las partes de un problema. De esta forma, es menos probable que se produzcan errores por desconocimiento de un área concreta.
- reducción de la intervención humana en el proceso. Cuanto más manual el proceso de desarrollo, más probabilidades de que se introduzcan errores en el sistema. Para asegurarnos de que la solución automatizada es confiable, bastaría con añadir pruebas masivas que verifiquen el correcto funcionamiento del sistema en presencia y ausencia de fallos.

CAPÍTULO 3
Estado de la cuestión

Debido a los estrictos tiempos de fabricación que marca el mercado, tanto la academia como la industria han tenido que aunar esfuerzos en definir metodologías y herramientas para la generación y el despliegue automático de códigos tolerantes a fallos en el diseño de hardware. A grandes rasgos, podemos distinguir entre dos enfoques: la que se define a partir de programación orientada a aspectos (en inglés, *aspect-oriented programming*) y la que lo hace mediante la metaprogramación [11].

3.1 Primer enfoque: programación orientada a aspectos

La programación orientada a aspectos es un paradigma que otorga modularidad y, en consecuencia, una mejor separación de conceptos. En el ámbito de los lenguajes de descripción de hardware, esta metodología se encuentra en un estado más incipiente ya que, a diferencia de los lenguajes de alto nivel, la programación orientada a aspectos tan apenas ha sido aplicada en el diseño de los mismos. Con todo, se han hecho avances relacionados, a nivel lógico, con un planificador de recursos basado en SystemC. También se ha trabajado en el prototipo de una extensión de VHDL llamada AspectVHDL [11].

3.2 Segundo enfoque: metaprogramación

La metaprogramación es una técnica mediante la cual un programa puede generar o manipular a otro. En el diseño de hardware, la separación de conceptos mediante la metaprogramación se conseguiría a través del diseño de programas escritos en un lenguaje distinto al HDL (del inglés, *Hardware Description Language*) para conseguir una mayor personalización, reusabilidad y flexibilidad.

Dentro de los mecanismos de tolerancia a fallos, los primeros avances están relacionados con la redundancia modular triple (TMR, por sus siglas en inglés *Triple Modular Redundancy*). En este caso, la salida que dan tres componentes replicados concretos se somete a un sistema de «votación» y la mayoritaria es la que se considera como correcta, pudiéndose enmascarar el fallo que se diera en uno de ellos, así como códigos de detección y corrección de errores en registros [11]. También se ha conseguido esta automatización a niveles más altos de abstracción, como en máquinas de estado finitas, aunque a costa de perder control sobre el hardware generado y el sobrecoste en términos de área, prestaciones y consumo de energía.

3.2.1. Ejemplos en la industria

Los principales trabajos que ha desempeñado la industria son los siguientes:

- Por parte de Xilinx: *TMR Manager* (antes *XTMR*). En el ámbito industrial, fue la primera herramienta diseñada específicamente para ambientes con alta radiación. Permite hacer uso de la TMR de forma directa, aunque solo para FPGAs de Xilinx, con lo que está optimizada para esta marca pero, al mismo tiempo, limitada por ella. Aunque este mecanismo se haya usado en misiones espaciales críticas, la inclusión del mismo en el diseño de los circuitos no siempre resulta fácil. De esta forma, desarrollaron una herramienta adicional: TMRTool que, en palabras del fabricante, «fue diseñada para simplificar el proceso de diseño de un sistema con TMR» y «funciona con cualquier HDL y herramienta de síntesis para automatizar la inclusión de TMR en un sistema» [12]. Gracias a esto, los diseñadores de hardware pueden centrarse en el diseño lógico. También soporta la implementación de máquinas de estado finitas seguras y ECC con SEC (correción de un bit) en su versión más actualizada.

- Por parte de Siemens-EDA (antes Mentor Graphics): *Precision Hi-Rel*. En este caso, la herramienta no está optimizada para una marca concreta, por lo que puede ser usada por distintos fabricantes. A grandes rasgos, consiste en la adición de TMR y la codificación segura de máquinas de estado finitas durante el proceso de síntesis, lo que, en según ellos «permite a los usuarios establecer un compromiso entre seguridad, área y rendimiento» [13]. Al igual que *TMR Manager*, también soporta ECC con SEC.

- Por parte de Synopsys: *Synplify Premier*. Soporta los tres mecanismos de tolerancia a fallos mencionados en los anteriores fabricantes, además de facilitar mecanismos de inyección de fallos que permiten acelerar la implementación de diseños confiables [14].

3.2.2. Ejemplos en la academia

A continuación destacamos algunos de los avances más reseñables en el ámbito académico, todos ellos sacados de [11]:

- *vMAGIC (VHDL Manipulation and Generation Interface)*: desarrollada por la Universidad de Paderborn, es un mecanismo de generación automática de código que hace uso de una librería Java para leer, manipular y escribir código VHDL. La inclusión de esta librería mejora la fiabilidad del código y reduce el tiempo de desarrollo, si bien cubre aspectos genéricos más que específicos de la tolerancia a fallos.

- *BYU EDIF Tools*: con un enfoque similar al anterior, pero se trabaja con Electronic Design Interchange Format (EDIF) en lugar de con VHDL, encontrándose el primero a un nivel más bajo de abstracción que el segundo. De esta forma, el proceso de implementación se trata de forma más óptima, aunque se necesita mucho nivel de experiencia. Este conjunto de herramientas proporcionan aplicaciones TMR automatizadas para desplegar automáticamente la replicación de hardware.

- *CODESH*: en la misma línea que los anteriores, aunque cabe destacar que cuenta con un proceso de compilación abierto. Desde su librería es posible hacer uso de una replicación hardware *N-Modular Redundancy* y, para la información almacenada en registros, un mecanismo de corrección y detección Hamming.

3.2.3. Ejemplos en la comunidad

Dentro del ámbito del código abierto encontramos *MyHDL* [15], una librería que permite describir hardware mediante Python. El núcleo de la misma son los *generadores*, que son funciones análogas a los procesos de VHDL o los bloques de Verilog y que soporta la conversión a los dos lenguajes de descripción de hardware principales.

Así, un componente sería una función que devolvería generadores. El rasgo más significativo es que, al tratarse de un lenguaje de alto nivel, *MyHDL* permite abstraer conceptos de más bajo nivel en clases Python. Por ejemplo, existe una clase específica para definir señales que unan distintos generadores, así como otra para trabajar con operaciones bit a bit y, si bien la conversión a los dos HDLs principales tiene algunas limitaciones, puede resultar muy útil como una conversión intermedia entre ambos.

3.3 Diseño basado en FPGAs

Una FPGA (del inglés *Field-programmable Gate Array*) es un dispositivo formado por bloques lógicos programables en los que se pueden implementar circuitos de todo tipo: desde operaciones básicas con puertas lógicas hasta otros diseños más complejos, como procesadores. Esto se consigue modificando la interconexión de los bloques que lo conforman, y se hace a través de los lenguajes de descripción de hardware (en adelante, HDL, por sus siglas en inglés *Hardware Description Language*). No solo se pueden usar para diseñar prototipos de estos productos y testearlos, sino que también son útiles para llevar a cabo implementaciones finales; la prueba es que actualmente ya se usan en ámbitos muy diversos, desde el sanitario hasta el espacial.

Los principales fabricantes de estos productos son Xilinx [16] (comprada por AMD en 2021 por un valor de 35000 millones de dólares [17]) y Altera [18] (comprada por Intel por un valor de 16700 millones de dólares en 2015 [19]). Además de dedicarse a la fabricación de las FPGAs, también se ocupan de los entornos de desarrollo mediante los cuales se podrán implementar los circuitos. Normalmente, estos programas solo sirven para implementarlos en las FPGAs del fabricante; en el caso de Xilinx, el entorno de desarrollo es *Vivado* [20] y, en el de Altera, *Quartus* [21].

3.3.1. Arquitectura interna de las FPGAs

Antes de especificar cómo es el flujo del diseño basado en FPGAs, vamos a ilustrar en qué consiste su estructura interna para comprender mejor cómo funcionan estas fases.

La estructura genérica de una FPGA es la de una matriz de dos dimensiones de bloques lógicos (en adelante, CLB, por sus siglas en inglés *Configurable Logic Block*). Cada bloque lógico está a su vez formado por los siguientes componentes:

- *look-up table* (en adelante, LUT): son generadores de funciones e implementan la lógica combinacional;
- *flip-flop* (en adelante, FF): son elementos de memoria e implementan la lógica secuencial;
- multiplexores, puertas AND y XOR: elementos lógicos y de interconexión.

En la Figura 4 y la Figura 5 podemos analizar, respectivamente, el esquema global de este dispositivo (formado por una matriz de dos dimensiones de CLBs) y otro de la estructura interna de los CLBs (formada por LUTs y FFs).

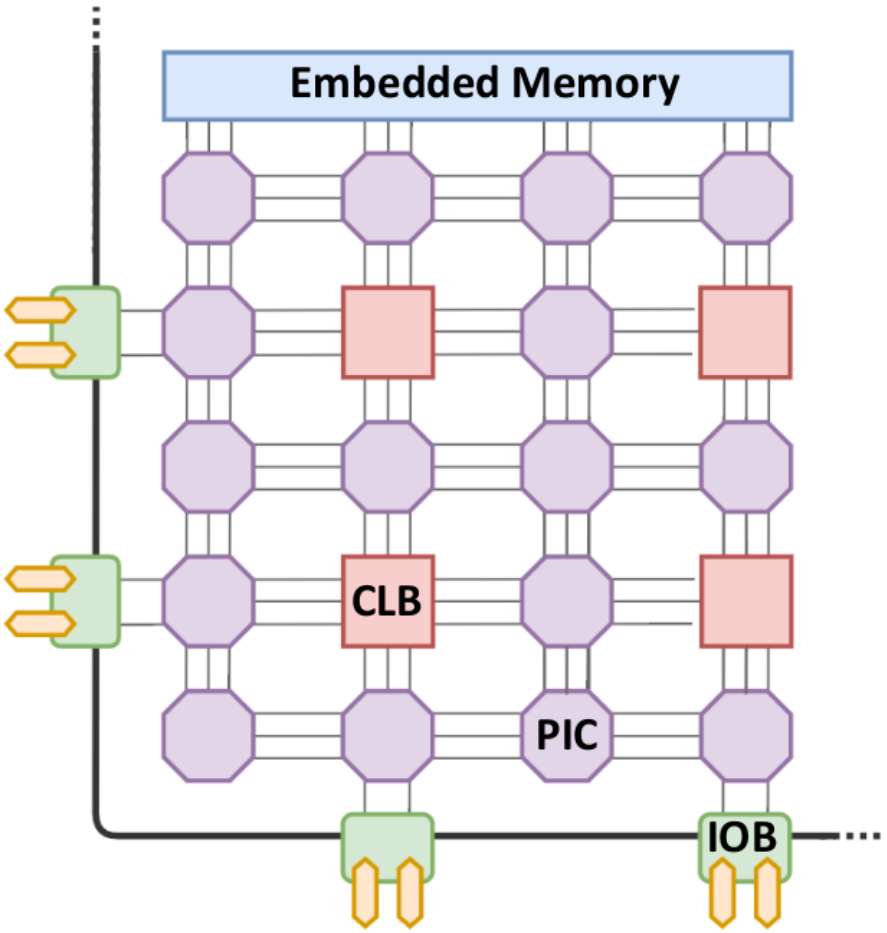

Figura 4. Esquema global de una FPGA. La sigla PIC se corresponde con *Programmable Interconnect* (Interruptor de conexión programable) e IOB con *Input-Output Block* (Bloque de Entrada/Salida). Fuente [22]

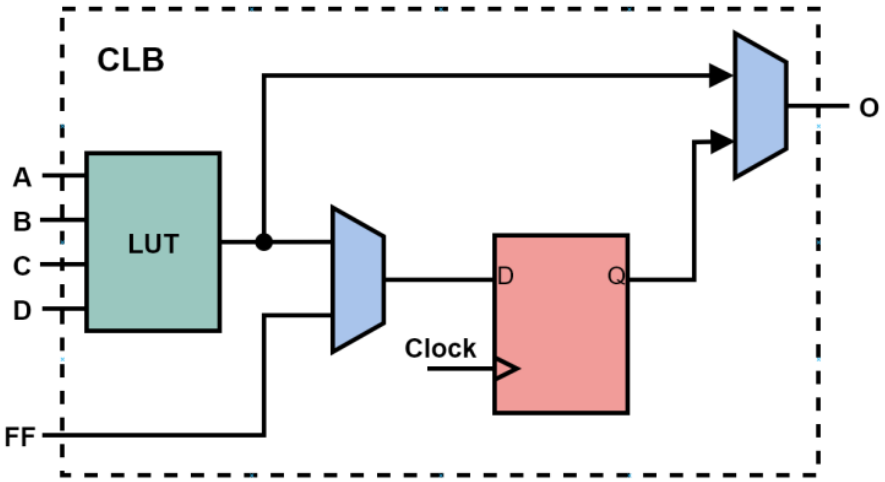

Figura 5. Ejemplo simplificado de un CLB con una LUT de cuatro entradas y un FF. Fuente [22]

Los CLBs se encuentran rodeados por canales de encaminamiento prefabricados, lo que conforma la arquitectura de encaminamiento. Esta cuenta con distintos elementos:

- segmentos de cableados de distintas longitudes;
- bloques de conexión: la parte de los segmentos que conectan dos bloques lógicos entre sí por medio de interruptores programables;
- bloques de encaminamiento: se encuentran en las intersecciones de los canales verticales y horizontales y realizan el encaminamiento, también mediante interruptores programables.

En la Figura 6, podemos observar el aspecto que tiene la arquitectura de encaminamiento en una FPGA.

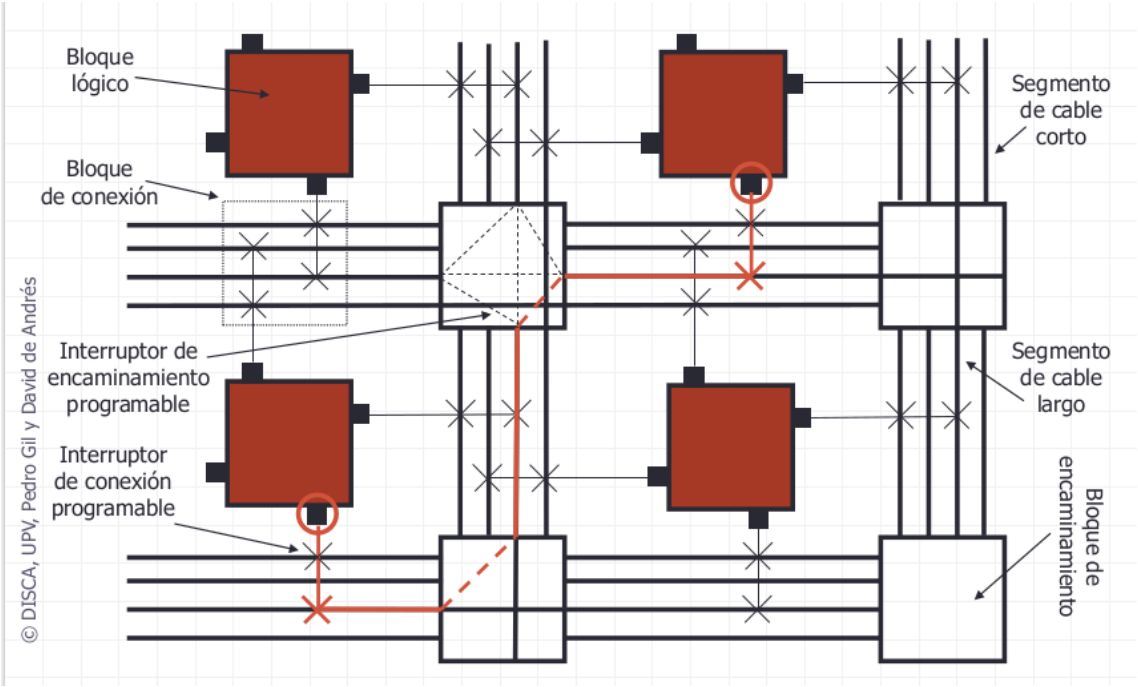

Figura 6. Arquitectura de encaminamiento. Fuente [23]

Además, también podemos encontrar otros elementos, como bloques de memoria, multiplicadores en silicio, microprocesadores o gestores de reloj digital. Un componente crucial es la memoria de configuración, ya que es el encargado de la programación e interconexión de todos los elementos aquí presentados.

3.3.2. Flujo del diseño

Aunque cada programa cuente con sus características concretas, a grandes rasgos, el diseño basado en FPGAs puede abstraerse en las siguientes fases:

- entrada de diseño: fase donde se describe en RTL (Register-transfer Level) el código del circuito. Existen dos HDLs principales:
  - VHDL (Very High-Speed Integrated Circuit HDL) [24]: basado en ADA [25], fue desarrollado durante los años 80 por el departamento de Defensa de Estados Unidos.
  - Verilog [26]: basado en C [27] y desarrollado durante la década de los 90 por Cadence Design Systems. A día de hoy, Verilog está integrado en SystemVerilog, por lo que comparten el mismo estándar IEEE.

- simulación funcional: se comprueba si el circuito diseñado funciona correctamente;
- síntesis: la representación realizada en la entrada de diseño se transforma en una representación a nivel de biestables y puertas lógicas. Concretamente, se realizan las siguientes operaciones:
  - extraer la lógica combinacional del circuito;
  - aplicar optimizaciones independientes de la tecnología minimizando al máximo las funciones lógicas mediante el algoritmo de Quine—McCluskey [28];
  - determinar qué componentes son necesarios para implementar esas funciones dependiendo de la tecnología utilizada. En este caso, al tratarse de una FPGA, el circuito se implementará mediante LUTs. El algoritmo encargado de particionar las funciones lógicas en LUTs para minimizar el retardo final es FlowMap [29];
  - encapsular las LUTs en bloques lógicos configurables;
  - como resultado, se obtiene un *netlist* de bloques lógicos, que esencialmente son elementos básicos conectados entre sí.
- implementación. Consta de dos fases:
  - *placement*: se seleccionan los bloques lógicos de la FPGA que implementarán cada uno de los bloques lógicos del circuito para minimizar el área y el consumo y maximizar la frecuencia de funcionamiento. El algorimo clásico utilizado es el Simulated Annealing [30] que, al ser heurístico, es ejecutado por el entorno de desarrollo varias veces, devolviendo el mejor resultado obtenido;
  - *routing*: consiste en encontrar el camino óptimo entre los pines de los bloques lógicos a conectar. El algoritmo utilizado es el PathFinder [31];
- simulación temporal: verificar el correcto funcionamiento del diseño una vez calculados los tiempos de la implementación realizada en el peor caso de retardo;
- generación del *bitstream*: el *bitstream* (cadena de bits) es el fichero final que contiene el diseño con el que programar la FPGA (de hecho, programar la FPGA se entiende como el momento en que se carga este fichero);
- prueba en el dispositivo real: se comprueba directamente en el dispositivo que el diseño cumple con su función.

Más adelante ilustraremos cómo se sigue este flujo en la herramienta concreta que vamos a utilizar, que en nuestro caso es Vivado.

3.3.3. Inyección de fallos

Con el fin de comprender algunas características relativas a la arquitectura de nuestra solución, vamos a definir qué es la inyección de fallos: esta consiste en la introducción deliberada y controlada de errores en el sistema durante su ejecución para poder observar su comportamiento [32]. Existen varias técnicas para inyectar fallos: las basadas en el hardware (*Hardware-based Fault Injection*), en el software (*Software-based Fault Injection*) y las que lo hacen en la simulación (*Simulation-based Fault Injection*) [33].

En este trabajo haremos uso de la última, ya que el entorno de desarrollo que usaremos cuenta con un simulador que permite utilizarse tanto en la simulación funcional como en la temporal. Además, se puede aplicar en las etapas iniciales del diseño y antes de su implementación, donde es más rápido y menos costoso corregir los errores que pudieran encontrarse.

CAPÍTULO 4
Arquitectura de la solución

El problema que pretendemos resolver es, por lo tanto, conseguir automatizar la generación de código HDL a través de una herramienta genérica y, a partir de ella, implementar automáticamente cualquier tipo de código de corrección de errores. En base a lo visto en la Sección 3.2, para llevarlo a cabo vamos a tomar un enfoque que podría encuadrarse dentro de la metaprogramación. En este apartado nos ocuparemos de describir de forma genérica la idea que queremos llevar a cabo, así como las herramientas y lenguajes elegidos para ello.

4.1 Entorno de desarrollo: *Vivado*

Vivado es el entorno de desarrollo de Xilinx que hemos elegido para llevar a cabo todo el flujo de diseño del circuito. Todas las fases de las que consta el diseño basado en FPGAs, desde el diseño del circuito hasta la generación del *bitstream*, pueden llevarse a cabo desde la interfaz de usuario. Como podemos ver en la Figura 7, podemos incluir el código del diseño directamente en el propio editor integrado.

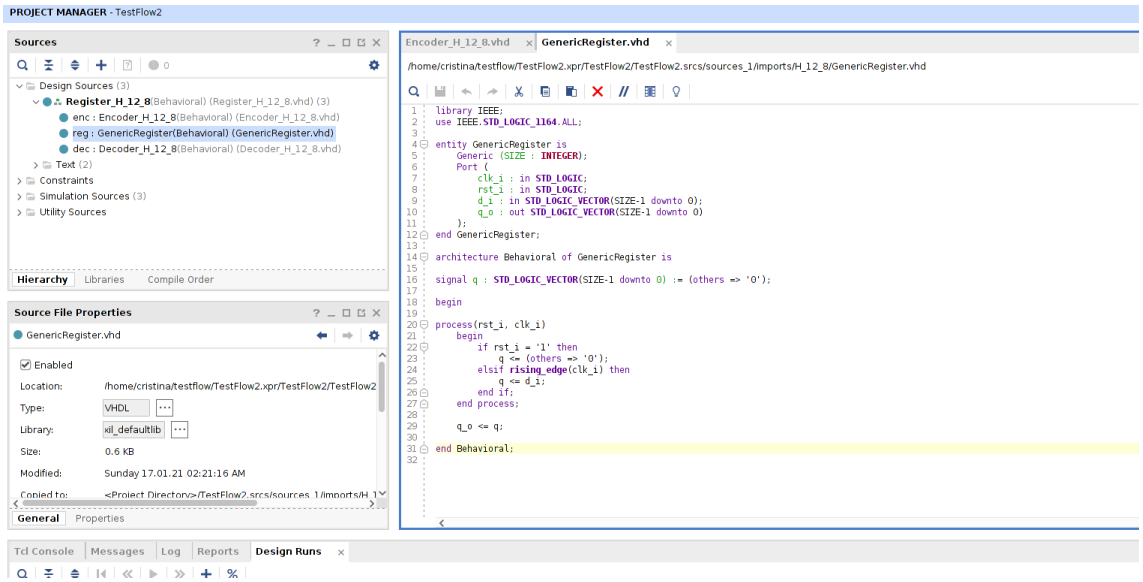

Figura 7. Vista del editor de Vivado

El resto de fases, que corresponden con las identificadas en la sección 3.3, pueden realizarse de forma secuencial desde el menú lateral izquierdo, como muestra la Figura 8:

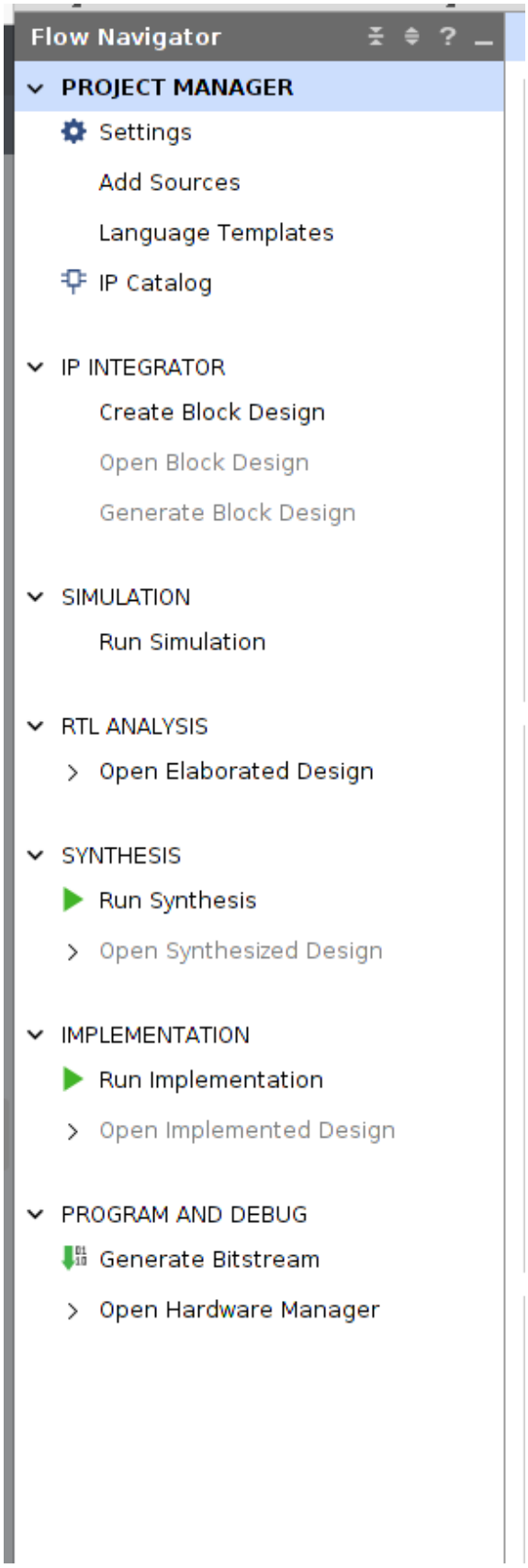

Figura 8. Menú lateral de Vivado

Gracias a las fases de simulación, podemos comprobar si el circuito responde o no a la funcionalidad y el rendimiento esperados antes de implementarlos en una FPGA. Esto

cobra vital importancia conforme la complejidad y tamaño del circuito aumenta, ya que el tiempo que tardará en implementarse también será mayor.

Como añadido, algo que merecemos destacar y que tiene que ver con la implementación del circuito en la FPGA es la definición de un fichero de restricciones `.xdc` (Xilinx Design Constraints), que especifica a qué pines concretos se mapean las señales definidas en el circuito.

El lenguaje de descripción de hardware que vamos a emplear va a ser VHDL. Los ficheros fuente y de simulación, por lo tanto, son de tipo `.vhd`. Además de realizar todo el proceso desde la interfaz de usuario, podemos unificarlo mediante un lenguaje de scripting llamado TCL (del acrónimo en inglés *Tool Command Language*) [34]. De hecho, a medida que vamos efectuando todas las operaciones de forma manual, en la consola TCL que aparece en el menú inferior van surgiendo los comandos equivalentes en lenguaje TCL, como muestra más detalladamente la Figura 9. Esto será interesante para poder generar automáticamente el proyecto y unificar el flujo de la solución.

```
start_gui
open_project /home/cristina/testflow/TestFlow2.xpr/TestFlow2/TestFlow2.xpr
open_project /home/cristina/testflow/TestFlow2.xpr/TestFlow2/TestFlow2.xpr
WARNING: [filemgmt 56-3] Default IP Output Path : Could not find the directory '/home/cristina/testflow/TestFlow2.xpr/TestFlow2/TestFlow2.gen/sources_1'.
Scanning sources...
Finished scanning sources
INFO: [IP_Flow 19-234] Refreshing IP repositories
INFO: [IP_Flow 19-1704] No user IP repositories specified
INFO: [IP_Flow 19-2313] Loaded Vivado IP repository '/tools/Xilinx/Vivado/2020.2/data/ip'.
INFO: [IP_Flow 19-3899] Cannot get the environment domain name variable for the component vendor name. Setting the vendor name to 'user.org'.
open_project: Time (s): cpu = 00:00:11 ; elapsed = 00:00:10 . Memory (MB): peak = 7180.812 ; gain = 38.047 ; free physical = 3127 ; free virtual = 9863
update_compile_order -fileset sources_1
launch_simulation
Command: launch_simulation
INFO: [Vivado 12-5682] Launching behavioral simulation in '/home/cristina/testflow/TestFlow2.xpr/TestFlow2/TestFlow2.sim/sim_1/behav/xsim'
INFO: [SIM-utils-51] Simulation object is 'sim_1'
INFO: [SIM-utils-54] Inspecting design source files for 'Register_H_12_8_tb' in fileset 'sim_1'...
INFO: [USF-XSim-97] Finding global include files...
INFO: [USF-XSim-98] Fetching design files from 'sim_1'...
INFO: [USF-XSim-2] XSim::Compile design
INFO: [USF-XSim-61] Executing 'COMPILE and ANALYZE' step in '/home/cristina/testflow/TestFlow2.xpr/TestFlow2/TestFlow2.sim/sim_1/behav/xsim'
```

Figura 9. Consola TCL de Vivado

4.2 Estructura del código VHDL

La estructura general del diseño de un circuito en VHDL suele ser la siguiente:

- los componentes del circuito se definen en ficheros separados. Ocasionalmente, podrían definirse en el mismo, pero no es una buena práctica;
- cada componente dispone de una interfaz, que son los puertos de entrada y salida, y una descripción entre las que se listan a continuación:
  - comportamental (*behavioural*): especifica su funcionamiento;
  - de flujo de datos (*dataflow*): identifica cómo se calculan las salidas a partir de las entradas;
  - estructural (*structural*): detalla qué otros componentes son necesarios para implementarlo y cómo se interconectan.

La descripción de nuestro código corrector de errores Hamming se realizará estructuralmente en un fichero denominado *Top-level*. El resto de componentes necesarios, presentados en la sección 2.2.1 y en la Figura 10, se definirán en ficheros individuales donde se describirá su comportamiento (en el caso del registro) o su flujo de datos (si se trata del codificador o el decodificador). Resumiendo el aspecto del circuito, quedaría como sigue:

- codificador: realiza las operaciones necesarias sobre la palabra original, obteniéndose como resultado los bits de código. Su entrada conecta con la del *Top-level* y su salida con la entrada del registro;

- registro o memoria: es el componente que almacenará la información. En lugar de disponer la información en dos componentes (registro original y registro adicional para los bits de paridad), la depositaremos en un único registro de $N + K$ bits, con lo que su entrada conecta tanto con la salida del codificador como con la entrada del *Top-level* (palabra original), mientras que su salida conecta con la entrada del decodificador;
- decodificador: conecta con la salida del registro y realiza los cálculos necesarios para obtener, por un lado, la palabra original y, por otro, comprobar a través del cálculo por síndrome que la información no ha sido dañada. En el caso de haber sufrido alguna modificación, será este componente el encargado de corregir el fallo. Su salida conecta con la salida del *Top-level*;
- *Top-level*: el elemento que englobará a los anteriores, de un nivel más alto de abstracción. Además de una entrada y una salida, tendrá una señal de reloj y otra de inicialización.

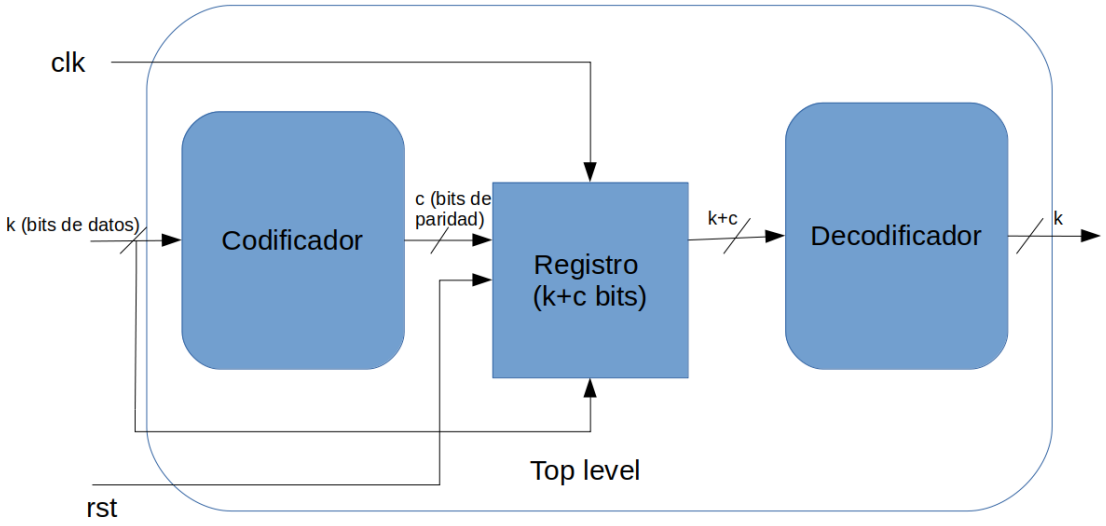

Figura 10. Estructura del circuito en VHDL

4.3 Fichero de entrada

Este es un fichero de texto plano que está formado por los parámetros que el programa que hemos construido necesita para poder generar el ECC correspondiente, así como la validación posterior; es decir, no forma parte de la entrada del diseño en sí. Este debe estar conformado, como mínimo, por los siguientes parámetros, en el mismo orden:

- el número de bits totales (datos y redundancia) y el de la palabra original que se transmiten. Los valores que pueden tomar son los mismos que las dimensiones de un código Hamming determinado, como hemos visto en la Sección 2.3. Por ejemplo: 7 y 4, 12 y 8 o 71 y 64;
- los valores de las columnas, en numeración decimal, de la matriz H. Por ejemplo, para un código Hamming(7,4), los valores serían 1, 2, 4, 3, 5, 6 y 7;

- el tipo de código corrector de errores. En este caso, será conveniente determinar unas constantes para referirnos al tipo de código empleado. Así, definimos `H`, por ser la inicial de Hamming, y podríamos hacer uso de la misma convención si añadiéramos más códigos;
- el listado de todas las hipótesis de fallo del ECC, que se refiere a aquellos tipos de fallos a los que se espera que el sistema pueda enfrentarse durante su ciclo de vida. Al igual que el parámetro anterior, también definiremos unas constantes. En este aspecto, si quiséramos realizar una verificación funcional, tomaría el valor `GOLDEN_RUN` (ya que es el término que usamos para referirnos a una ejecución o simulación en ausencia de fallos). Si trataramos de inyectar el fallo en un bit (*single bit-flip*), que es el número máximo de fallos que tolera un código Hamming simple, lo definiríamos como `1_BIT`; y así de forma análoga para el resto de hipótesis. En cualquier caso, esta lista deberá contener algún valor, y siempre se ejecutará la `GOLDEN_RUN` primero;
- el modelo de FPGA de Xilinx con la que se quiere hacer la simulación. Deberá estar instalada en el equipo.

Un ejemplo del fichero de entrada sería como el que podemos ver en la Figura 11:

Figura 11. Ejemplo de fichero de entrada para H(7,4)

```
7,4
1,2,4,3,5,6,7
H
GOLDEN_RUN
xc7vx485tffg1157-1
```

En un futuro podríamos añadir otros valores para referirnos a más tipos de códigos. Es decir, si bien en este trabajo nos vamos a ceñir a efectuar las pruebas con un código Hamming con tolerancia a un fallo en un bit, el tener presente estos parámetros en la generación de los scripts nos puede servir para, más adelante, poder añadir más tipos de código de una forma más sencilla y sin que ello suponga un perjuicio para el resto del programa.

4.4 Scripts de generación de código: ficheros fuente y de simulación

De forma análoga al diseño *semicustom* de FPGAs explicado en la Sección 3.3, para construir un mecanismo con tolerancia a fallos podemos a seguir el siguiente flujo:

Figura 12. Flujo del diseño de un mecanismo con tolerancia a fallos

- diseño: consiste en programar, de forma modular, los *scripts* de generación de código VHDL; es decir, haremos un *script* por cada componente del circuito, a excepción

del registro o memoria ya que, como veremos, en VHDL es posible una descripción genérica de los mismos que hace innecesaria su parametrización de forma externa;

- verificación funcional: constatar que el circuito generado funciona correctamente, lo que equivale al lanzamiento de la hipótesis `GOLDEN_RUN`;
- inyección de fallos y verificación del circuito corrector de errores: realizar una primera prueba en Vivado con los *scripts* generados. Esta consistirá en ejecutar la simulación del circuito inyectándolos de forma deliberada. Esta «inyección manual» se puede conseguir invirtiendo algún bit de la señal que representa al estado del registro que almacena el valor codificado, y comprobando que la salida del circuito es correcta; es decir, hemos forzado un error que el circuito es capaz de corregir.

La inyección de fallos puede realizarse en cualquiera de las fases de las que consta el diseño *semicustom*. En nuestro caso, vamos a hacerla durante la verificación funcional. Si, por ejemplo, implementáramos el diseño en una FPGA, podríamos realizar la inyección sobre el propio dispositivo y comprobar *in situ* si cumple con los objetivos (a esta técnica se la conoce como *FPGA-based fault injection* [35]).

Llegados a este punto, deberemos plantearnos también la generación automática de los ficheros de simulación. Esto es así porque, si pretendemos probar y asegurarnos de que nuestra solución funciona correctamente, tendremos que realizar pruebas mucho más exhaustivas; esto es, realizar todas las combinaciones con todos los posibles fallos en todos los bits de la palabra. Por un lado, automatizaremos la generación del fichero de simulación VHDL que solemos llamar por convención *testbench* y, por otro, haremos lo mismo con el fichero que introducirá los fallos de forma masiva.

4.5 Generación automática del proyecto y unificación del flujo

Además de hacerlo desde la interfaz gráfica de Vivado, es posible efectuar todo el proceso de simulación mediante comandos TCL, como hemos explicado en la Sección 4.1. Como nuestro propósito es agilizar al máximo el proceso, resultaría muy conveniente poder ejecutar y automatizar todos estos pasos desde la propia consola, en lugar de tener desacoplados el proceso de generación de código y la posterior creación y simulación del proyecto.

Cabe destacar que sigue siendo necesario disponer de una instalación de Vivado en el equipo donde vayan a realizarse las pruebas pero, gracias a esta utilidad, evitamos tener que entrar al programa desde la interfaz gráfica con los sobrecostes en tiempo que conlleva esta acción, minimizamos la probabilidad de cometer errores y hacemos innecesario que el usuario esté familiarizado con el entorno de desarrollo. De este modo, tras unificar el flujo, el usuario podrá obtener los resultados finales de la simulación y comprobar si el circuito cumple con el fin para el que fue diseñado.

4.6 Lenguaje de programación: Python

Una vez trazado el plan de cómo queremos que sea nuestra aplicación, falta por determinar qué lenguaje de programación vamos a utilizar. En nuestro caso, nos hemos decantado por Python por los siguientes motivos:

- es un lenguaje con una sintaxis muy clara y sencilla: su filosofía radica en hacer lo más legible posible el código;

- es multiparadigma: soporta la programación orientada a objetos, la imperativa y también la funcional;
- cuenta con una gran cantidad de módulos y librerías de ámbitos muy diversos, entre los cuales podemos encontrar la computación científica.

Este último punto ha condicionado especialmente nuestra elección ya que, como veremos en el siguiente capítulo, para resolver nuestro problema necesitaremos manipular matrices numéricas; concretamente, haremos uso de una librería llamada *NumPy*, que cuenta con una gran cantidad de rutinas a través de las cuales poder realizar estas operaciones.

CAPÍTULO 5
Desarrollo de la solución propuesta

En este apartado vamos a hacer un recorrido por el proceso de implementación de la solución que hemos ideado en el Capítulo 4, centrándonos en resolver el problema para un código Hamming con tolerancia a fallos en un bit. Este desarrollo se ha ido modificando paulatinamente con respecto al diseño inicial; no obstante, podemos diferenciar todo este proceso en dos fases. En la primera, nos hemos encargado de conseguir la funcionalidad básica que resolviera el problema. En la segunda, hemos mejorado el flujo general para que nuestro programa fuese más amigable de cara al usuario, así como refactorizado el código del programa.

5.1 Primera fase: enfoque imperativo

En esta fase, el objetivo principal es generar los componentes del circuito en VHDL por separado para integrarlos manualmente en Vivado y comprobar que el sistema funciona correctamente desde el entorno de desarrollo. Lo hacemos así porque, en caso de que existan errores (esto es, que falle algún punto de la implementación), es mucho más sencillo identificarlos, ya que podemos configurar la simulación de Vivado con las señales del circuito que consideremos oportunas y realizar una traza de la misma. De este modo y a grandes rasgos, todos los scripts contarán con la misma estructura genérica, que es como sigue:

- manejo de errores: en todo caso, el usuario debe especificar el fichero de entrada tal y como se ha explicado en la Sección 4.3. En caso contrario, salta un error;
- procesamiento del fichero de entrada: los parámetros de los que está formado este fichero se procesan y se guardan en variables, que son las que permitirán parametrizar adecuadamente cada código. Estas son las mismas especificadas en la Sección 4.3;
- en algunos componentes, como el codificador y el decodificador, generaremos los cálculos asociados a las fórmulas en funciones auxiliares, no solo para mejorar la legibilidad del código, sino para que sea más sencilla la integración de otros ECCs en un futuro;
- generación del código VHDL: esta parte consiste en transformar los cálculos realizados anteriormente al código VHDL equivalente del componente en cuestión. Todo este código es guardado como una cadena de caracteres en una variable;

- escritura del código en un fichero de salida: el código VHDL generado se escribe en un nuevo fichero con la extensión que proceda. En el caso de los componentes principales, será `vhd`.

Cabe destacar que el componente de almacenamiento de la información, que en nuestro caso es el registro, no necesita ser generado a partir del fichero de entrada, a diferencia del resto. Esto se explica porque el único componente variable es su tamaño y dicho parámetro puede dejarse indicado de forma genérica, por lo que será independiente del código que queramos probar.

La automatización relacionada con la simulación del circuito resulta más sencilla, ya que solo algunos parámetros variarán dependiendo del código. Al igual que los componentes del circuito, estos también requerirán del fichero de entrada para su generación.

- *testbench*: la única parte variable es el nombre del componente que, por convención, tomaremos del nombre del fichero de entrada, además del número de bits de datos. En un primer momento querremos realizar pruebas exhaustivas, lo que significa verificar el funcionamiento con cada posible valor para la entrada del circuito. Por ejemplo: si el tamaño de la palabra es de cuatro bits, los valores que podrá tomar la entrada del circuito (y, en consecuencia, el número de experimentos realizados) será de $2^4 = 16$. Sin embargo, para palabras de tamaño más grande (en nuestro caso, a partir de 16 bits) se realizarán dos experimentos para dos tipos de patrones: `0101...01` y `1010...10`[1];

- fichero de inyección de fallos TCL: en este caso, solo tendremos en cuenta el nombre del fichero de entrada. Este fichero se encargará de inyectar errores en un bit (esto es, invertirá su valor con respecto a la señal original) para cada posible entrada del circuito. Si el tamaño de la palabra es más grande y realizamos los experimentos con dos patrones, habrá que cambiar los tiempos de ejecución de este programa para que esté sincronizado con el *testbench*;

- fichero de creación del proyecto y simulación: esta vez, además del nombre del fichero de entrada, deberemos indicar la ruta donde se creará y ejecutará el proyecto, así como su nombre. Esto es la secuencia de comandos TCL que hemos podido ver en la Figura 9. A grandes rasgos, se empieza por la creación del proyecto y configuración del lenguaje HDL, se añaden todos los ficheros (tanto los fuente como los de la simulación), se lanza la simulación durante un tiempo dependiente del código y el número de experimentos realizados y, por último se cierra la simulación.

5.1.1. NumPy

Como hemos adelantado en la Sección 4.6, la parte crucial del desarrollo del código del codificador y del decodificador es aquella donde calculamos sus fórmulas, ya que de ellas depende que todas las señales se encuentren mapeadas de forma correcta. Para ello, el enfoque que utilizaremos será el siguiente: generaremos matrices de dos dimensiones, a partir de los parámetros del fichero de entrada, que contengan de forma compacta la información relativa a las fórmulas. De esta forma, esta estructura podrá contener toda la

[1]Se consideró esta opción teniendo en cuenta que, a palabras mayores, mayor número de experimentos e inyecciones, lo que supone un incremento nada desdeñable del tiempo de ejecución del programa; por ello, para palabras de mayor tamaño que 8 bits, hemos optado por realizar este tipo de simulación. Como en el presente trabajo únicamente realizaremos inyecciones de fallos en un bit, sigue siendo plausible usar esta estrategia pues, aunque no se cubran todas las posibles rutas internas, se siguen comprobando todos los fallos necesarios y, como ventaja, se sale ganando en tiempo de ejecución.

información necesaria para que, una vez procesada, pueda generar y traducir las fórmulas en señales VHDL.

En el caso del codificador, la función encargada de los cálculos inicializará una matriz 2D, cuyas dimensiones aumentarán en proporción al tamaño de la palabra; esto es así porque las columnas de la matriz son los números del fichero de entrada convertidos a binario. Una vez generada, podremos recorrer esta matriz y determinar si un bit de la señal debe o no tenerse en cuenta para la fórmula dependiendo de si hay un 1 o un 0 en su posición. Para el decodificador, el enfoque es el mismo; únicamente hay que generar otra matriz auxiliar que contenga la información relacionada con las fórmulas del síndrome. De forma análoga, cada bit de la señal del síndrome estará o no negada en función de su valor binario.

Para alcanzar nuestro fin vamos a tener que realizar algunas operaciones matriciales que, si bien son bastante básicas, programarlas directamente en Python puede resultar tedioso y proclive a errores. NumPy [36] es una librería dedicada a la computación científica para Python, y es muy potente para la creación y manipulación de estructuras matriciales. Cuenta con rutinas que hacen muy transparente el proceso de trabajar con este tipo de datos, con lo que nos evitamos «tener que reinventar la rueda» gracias a las mismas. Concretamente, las rutinas que han sido de especial uso han sido la inicialización y trasposición de matrices.

5.2 Segunda fase: enfoque orientado a objetos

La generación individual de los componentes tiene una serie de ventajas, entre las cuales destacan el que sea más sencillo identificar errores de programación y verificar modularmente el circuito. Además, esta metodología de trabajo se adapta satisfactoriamente al enfoque ágil del proyecto, que consiste en realizar entregables individuales y funcionales de forma periódica. No obstante, esta arquitectura tiene también sus desventajas, las cuales fueron identificadas cuando nos propusimos realizar todo el proceso desde la línea de comandos, como hemos explicado en la Sección 4.5.

La principal, y de la que nos percatamos en primer lugar una vez realizadas las primeras verificaciones modulares, es que debemos ejecutar tantos programas como componentes tiene el circuito y el proyecto Vivado, lo cual entra en conflicto con el flujo unificado que pretendemos darle al proceso. Además, este enfoque hace que el crear y simular el circuito sea muy confuso de cara al usuario final, ya que al crear el proyecto se necesita, entre otras cosas, tener muy claras las rutas en las que se encuentran los componentes. Por todo esto, resulta necesario añadir transparencia a todo el proceso, lo que se traduce en unificar el flujo lo máximo posible, y ello solo lo podemos conseguir llevando a cabo una refactorización del código.

5.2.1. Refactorización

La forma más directa de unificarlo es disponiendo toda la lógica de los componentes en un mismo archivo. Para ello, y con el fin de otorgar más legibilidad y organización al código, va a resultar muy conveniente cambiar del enfoque imperativo inicial al orientado a objetos. De esta forma, cada componente del circuito formará una clase individual. También habrá una clase para el *testbench*, el programa TCL que inyecta los fallos y para el que crea y simula el proyecto. Todos los componentes estarán preparados para una posible escalabilidad futura en caso de querer añadir otros códigos o hipótesis de fallos. Además de todo el código desarrollado en la primera fase, cada objeto estará formado

por un constructor que será inicializado con una serie de atributos que se corresponden con los parámetros del fichero de entrada. El nuevo flujo quedará, pues, como sigue:

- manejo de errores: será obligatorio pasarle al programa el fichero de entrada; de lo contrario, saltará un error;
- procesamiento del fichero de entrada: los parámetros se guardan en variables globales;
- inicialización y creación de los componentes (objetos): se usan las variables globales para inicializar los objetos. Acto seguido, para cada uno de ellos, se llama a las funciones encargadas de la generación del código VHDL;
- creación y simulación del proyecto: se le pide interactivamente al usuario nombrar el proyecto y se inicia el proceso de simulación. Cabe resaltar que, en todo caso e independientemente de que se indiquen otras hipótesis de fallo, siempre se ejecutará en primer lugar la simulación encargada de llevar a cabo la verificación funcional;
- muestra de los resultados obtenidos por pantalla.

5.2.2. Funcionalidad adicional

Como hemos especificado en la Sección 5.2, durante la creación del proyecto se necesitan importar todos los archivos necesarios, tanto los ficheros fuente como los de simulación. Para poder manejar y cohesionar este proceso, se ha optado por generar una estructura de directorios como sigue:

- directorio padre: se generará en el directorio actual donde se ejecute el programa. En este caso, tomará como nombre `HammingECC_<nombre del fichero de entrada>`. Este, a su vez, contendrá los siguientes subdirectorios y archivos:
  - `sim`: donde se guardará el archivo *testbench*;
  - `src`: donde se guardarán los ficheros fuente del circuito;
  - dentro del directorio padre, también se almacenarán los archivos TCL encargados de la creación y simulación del proyecto, uno para cada hipótesis de fallo, así como el propio proyecto generado;
  - por último, también se generará una carpeta llamada `results`, responsable de almacenar los resultados obtenidos en las simulaciones. Para cada hipótesis de fallo se generará una carpeta con su archivo CSV correspondiente y, una vez finalizadas todas, se generará otra final que contendrá todos los resultados en un único archivo. La razón por la que hay que seguir esta estrategia es que, cada vez que se reinicia una simulación y se ejecuta otra, los ficheros de salida anteriores se sobrescriben con los nuevos resultados, con lo que hay que guardarlos antes.

A la hora de simular el proyecto, es necesario indicar dónde se encuentra el ejecutable de Vivado. Como este dependerá del equipo en el que se ejecute el programa, hemos decidido que la solución óptima es que el usuario guarde en una variable de entorno (`VIVADO_PATH`), a priori, la ruta a este. De esta forma, cuando se inicia el programa se comprueba que esta variable existe y, en caso contrario, se aborta el programa.

Respecto a la compatibilidad multiplataforma, debemos mencionar que todo el proceso de desarrollo se ha llevado a cabo desde un sistema GNU/Linux (concretamente,

Ubuntu 20.04). No obstante, al realizar pruebas desde otros dispositivos, nos percatamos de que el programa no era compatible con sistemas Windows, principalmente porque el sistema de archivos es distinto a los sistemas Unix, con lo que tuvimos que adaptar aquellas partes del programa en las que se manejara este tipo de estructuras.

5.2.3. Pandas

En un primer momento, los resultados se generaron en un fichero plano. Posteriormente, decidimos cambiar el formato a CSV con el fin de poder manipularlos y mostrarlos mejor por pantalla, para lo cual el uso de la librería Pandas ha resultado de gran utilidad. Pandas [37] es una librería para Python que está destinada al análisis y manejo de datos de forma optimizada y sencilla. De esta forma, una vez obtenidos todos los resultados, podemos hacer uso de sus utilidades para procesarlos masivamente, en forma de tabla y, en consecuencia, mejorar la legibilidad de los mismos.

CAPÍTULO 6
Resultados

En el presente apartado, mostraremos los resultados obtenidos, tanto lo que respecta a los ficheros generados y su estructura e implementación internas, como a la ejecución de cada fase tratada en el Capítulo 5 a través de la simulación de distintos códigos, con el fin de verificar su correcto funcionamiento.

6.1 Preparación del entorno y ejecución del programa final

Antes de mostrar los resultados, listaremos aquellas herramientas que son necesarias tener instaladas en el equipo de prueba para que el programa funcione:

- Vivado: la versión que hemos utilizado ha sido la 2020.2;
- Python3: es recomendable que la versión sea igual o mayor a la 3.6;
- NumPy: en el momento de realizar este trabajo, la 1.20.0;
- Pandas: en este caso, la 1.2.4;
- tener preparada, como se ha especificado en la Subsección 5.2.2, la variable de entorno del ejecutable de Vivado para el equipo en el que se realizan las pruebas.

Una vez instaladas, la forma de ejecutar el programa es como se muestra en la Figura 13:

Figura 13. Ejemplo de ejecución del programa

```
1    python3 <nombre_del_script_final>.py H_<bits_totales>_<bits_datos>.txt
```

Más adelante, una vez se creen todos los ficheros necesarios, nos preguntará por el nombre que le queremos poner al proyecto Vivado.

A continuación, se procederá a crear y simular el proyecto, generando los resultados y mostrándolos por pantalla, como veremos en la Sección 6.4.

6.2 Código resultante e implementación

En primer lugar, mostramos en el Archivo 6.1 cuál es el fichero de entrada, en este caso, para H(7,4):

```
7,4
1,2,4,3,5,6,7
H
1_BIT
xc7vx485tffg1157-1
```

Archivo 6.1: Fichero de entrada para H(7,4)

A continuación, y con el fin de demostrar que al final obtenemos una implementación Hardware, presentamos un ejemplo del código resultante que describe a cada componente junto con su diseño e implementación, en este caso, para H(7,4). Cabe destacar que cada fichero es creado dinámicamente, y su estructura interna dependerá del tipo de código que se quiera generar (tamaño de la palabra, hipótesis de fallo, etc), a excepción del registro, con el que es posible realizar una descripción genérica gracias al uso de la palabra reservada en VHDL `SIZE`.

6.2.1. Registro

En el Archivo 6.2, desde la línea 4 hasta la 12 (es decir, toda aquella parte del código englobada por `entity`) se realiza la declaración de su interfaz; la estructura interna se lleva a cabo desde la línea 14 hasta la 31 (todo aquello dentro de `arquitecture`). Esta disposición es común a todos los componentes descritos mediante VHDL. El registro se describe en estilo comportamental, y su conducta esperada es que almacene correctamente las palabras codificadas; esto es, que los valores de la entrada y la salida sean iguales.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity GenericRegister is
5      Generic (SIZE : INTEGER);
6      Port (
7          clk_i : in STD_LOGIC;
8          rst_i : in STD_LOGIC;
9          d_i : in STD_LOGIC_VECTOR(SIZE-1 downto 0);
10         q_o : out STD_LOGIC_VECTOR(SIZE-1 downto 0)
11     );
12 end GenericRegister;
13
14 architecture Behavioral of GenericRegister is
15
16 signal q : STD_LOGIC_VECTOR(SIZE-1 downto 0) := (others => '0');
17
18 begin
19
20 process(rst_i, clk_i)
21     begin
22         if rst_i = '1' then
23             q <= (others => '0');
24         elsif rising_edge(clk_i) then
25             q <= d_i;
26         end if;
27     end process;
28
29     q_o <= q;
30
31 end Behavioral;
```

Archivo 6.2: Registro generado para H(7,4)

En su estructura RTL, que aparece en la Figura 14, podemos apreciar que la señal de salida (`q[SIZE-1] downto 0`) se transforma en `q_reg[6:0]`, lo que significa que se genera un registro de siete bits para esa señal. Por otro lado, la señal de inicialización (`rst`) se conecta con la entrada `CLR` del registro, inicializándolo a 0 de forma asíncrona.

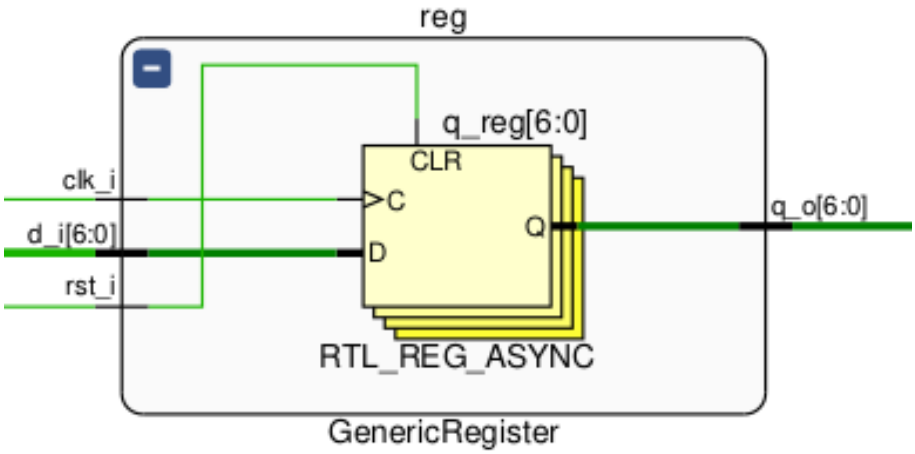

Figura 14. Esquema RTL para el registro

Respecto a su esquema *netlist*, se puede apreciar, en la Figura 15, que el registro se implementa como siete componentes de tipo FDCE (desde `q_reg[0]` hasta `q_reg[6]`), formados cada uno por Flip-Flops de tipo D con una señal *clear* y otra de *enable* que en este caso estará conectada a VCC, por lo que siempre se encontrarán activos.

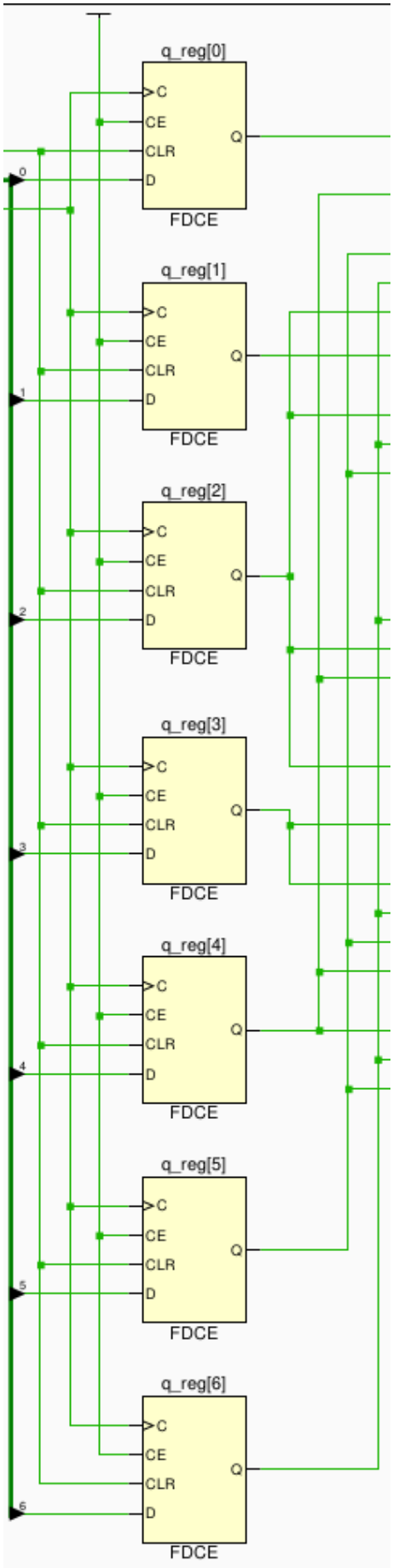

Figura 15. Esquema *netlist* (síntesis) para el registro

6.2.2. Codificador

En el caso del codificador, descrito en el Archivo 6.3, podemos observar que la entrada `u_i` se corresponde con el tamaño de la palabra (en este caso, de 4 bits), siendo un vector que va del bit 3 al 0 (describiéndose en VHDL como `3 downto 0`). En su arquitectura interna se llevan a cabo los cálculos de las fórmulas de codificación (desde la línea 11 hasta la 16). Este componente se describe en estilo *dataflow* o flujo de datos, que indica cómo se calculan las salidas a partir de las entradas.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity Encoder_H_7_4 is
5      Port (
6          u_i : in STD_LOGIC_VECTOR(3 downto 0);
7          b_o : out STD_LOGIC_VECTOR(2 downto 0)
8      );
9  end Encoder_H_7_4;
10
11 architecture Behavioral of Encoder_H_7_4 is
12 begin
13     b_o(0) <= u_i(0) xor u_i(1) xor u_i(3);
14     b_o(1) <= u_i(0) xor u_i(2) xor u_i(3);
15     b_o(2) <= u_i(1) xor u_i(2) xor u_i(3);
16 end Behavioral;
```

Archivo 6.3: Codificador generado para H(7,4)

En cuanto a su esquema RTL, en la Figura 16 se pueden observar las seis puertas XOR necesarias para calcular los tres bits de salida de la señal `b_o`:

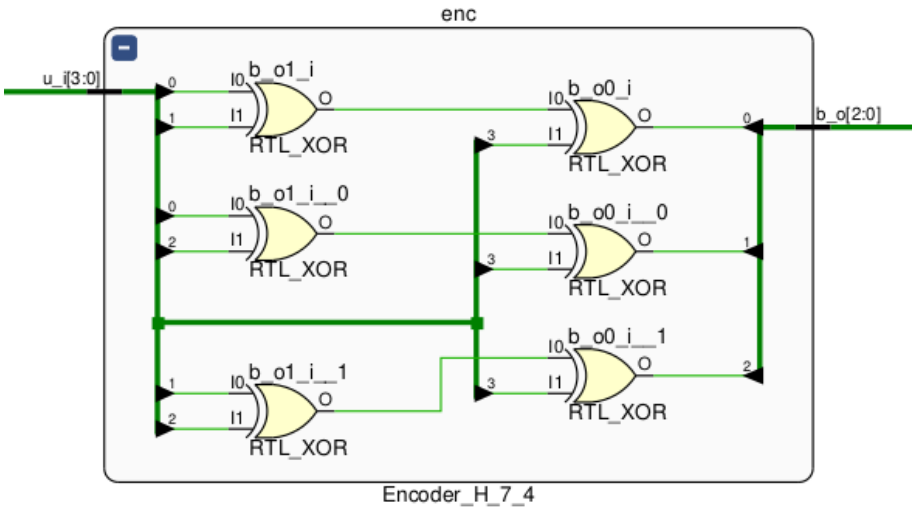

Figura 16. Esquema RTL para el codificador

En la Figura 17, podemos ver que aparecen tres LUTs de tres entradas, ya que implementan funciones de tres variables (LUT3), coincidiendo con el hecho de que, en una FPGA, la lógica combinacional se implementa por medio de estos componentes.

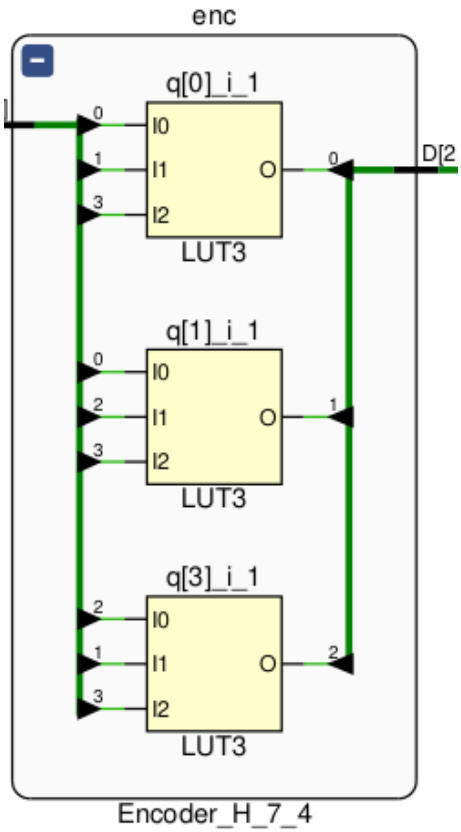

Figura 17. Esquema *netlist* (síntesis) para el codificador

6.2.3. Decodificador

La forma del decodificador, en el Archivo 6.4, es análoga a la del codificador: su entrada `r_i` consta de 7 bits (`6 downto 0`), correspondiéndose con los bits totales del código. A partir de la línea 11 se llevan a cabo los cálculos del síndrome y la corrección de errores, si los hubiese. Al igual que el codificador, también se describe en estilo *dataflow*.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity Decoder_H_7_4 is
5      Port (
6          r_i : in STD_LOGIC_VECTOR(6 downto 0);
7          u_o : out STD_LOGIC_VECTOR(3 downto 0)
8      );
9  end Decoder_H_7_4;
10
11 architecture Behavioral of Decoder_H_7_4 is
12 signal syn : STD_LOGIC_VECTOR(2 downto 0);
13 begin
14     syn(0) <= r_i(0) xor r_i(2) xor r_i(4) xor r_i(6);
15     syn(1) <= r_i(1) xor r_i(2) xor r_i(5) xor r_i(6);
16     syn(2) <= r_i(3) xor r_i(4) xor r_i(5) xor r_i(6);
17     u_o(0) <= r_i(2) xor (syn(0) and syn(1) and not syn(2));
18     u_o(1) <= r_i(4) xor (syn(0) and not syn(1) and syn(2));
19     u_o(2) <= r_i(5) xor (not syn(0) and syn(1) and syn(2));
20     u_o(3) <= r_i(6) xor (syn(0) and syn(1) and syn(2));
21 end Behavioral;
```

Archivo 6.4: Decodificador generado para H(7,4)

En la Figura 18, apreciamos el árbol de puertas XOR que calcula el síndrome, así como el conjunto de puertas XOR y AND que permiten corregir el dato en caso necesario.

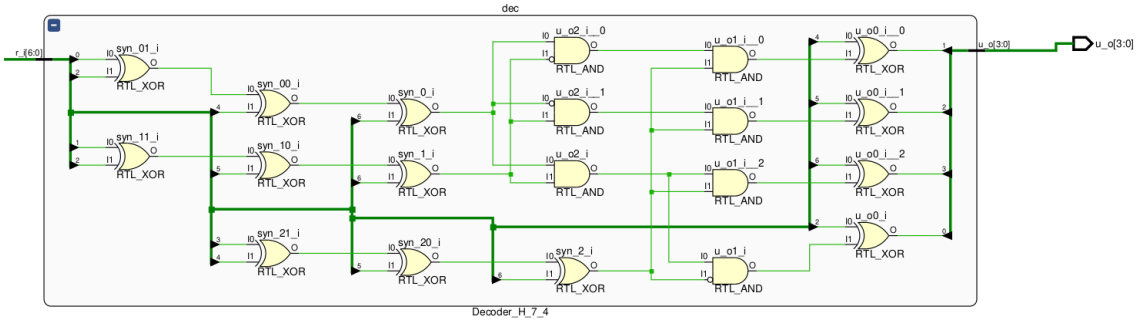

Figura 18. Esquema RTL para el decodificador

Por su parte, en la Figura 19, podemos ver la implementación de la lógica combinacional, que se realiza mediante LUTs de diferente tamaño (LUT6, LUT5 y LUT4), dependiendo de la complejidad de las funciones resultantes de fraccionar las ecuaciones logicas necesarias.

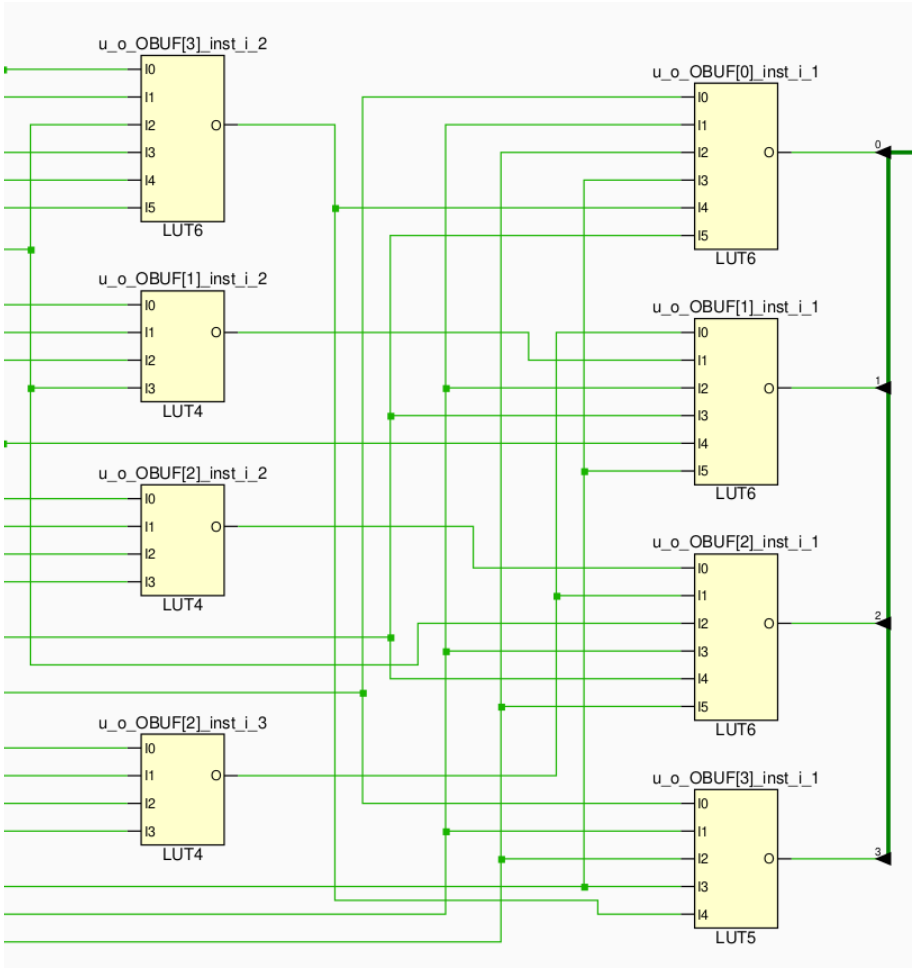

Figura 19. Esquema *netlist* (síntesis) para el decodificador

6.2.4. Diseño estructural

Para el diseño estructural, o *Top-level*, como podemos ver en el Archivo 6.5, su entrada `u_i` y su salida `u_o` son del mismo tamaño de la palabra (cuatro bits). También cuenta con una señal de reloj e inicialización. Enlazando con lo comentado en Sección 4.2, y tal y como puede visualizarse en la Figura 10, a partir de la línea 11 se lleva a cabo el mapeo de sus señales internas con las señales correspondientes de los otros componentes presentados (registro, codificador y decodificador). El diseño se describe en estilo estructural, que consiste en describir los componentes necesarios y cómo se interconectan.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Register_H_7_4 is
    Port ( clk_i : in STD_LOGIC;
           rst_i : in STD_LOGIC;
           u_i   : in STD_LOGIC_VECTOR(3 downto 0);
           u_o   : out STD_LOGIC_VECTOR(3 downto 0));
end Register_H_7_4;

architecture Behavioral of Register_H_7_4 is
    component Encoder_H_7_4 is
        Port ( u_i : in STD_LOGIC_VECTOR(3 downto 0);
               b_o : out STD_LOGIC_VECTOR(2 downto 0));
    end component;
    component GenericRegister is
        Generic (SIZE : INTEGER);
        Port (
            clk_i : in STD_LOGIC;
            rst_i : in STD_LOGIC;
            d_i : in STD_LOGIC_VECTOR(SIZE-1 downto 0);
            q_o : out STD_LOGIC_VECTOR(SIZE-1 downto 0)
        );
    end component;
    component Decoder_H_7_4 is
        Port (
            r_i : in STD_LOGIC_VECTOR(6 downto 0);
            u_o : out STD_LOGIC_VECTOR(3 downto 0)
        );
    end component;

signal code: STD_LOGIC_VECTOR(2 downto 0);
signal reg_o : STD_LOGIC_VECTOR(6 downto 0);

begin
enc : Encoder_H_7_4 port map (
    u_i => u_i,
    b_o => code
);

reg : GenericRegister
generic map (SIZE => 7)
port map (
    clk_i => clk_i,
    rst_i => rst_i,
    d_i(6) => u_i(3),
    d_i(5) => u_i(2),
    d_i(4) => u_i(1),
    d_i(3) => code(2),
    d_i(2) => u_i(0),
    d_i(1) => code(1),
    d_i(0) => code(0),
    q_o => reg_o
);

dec : Decoder_H_7_4 port map (
    r_i => reg_o,
    u_o => u_o
);
end Behavioral;
```

Archivo 6.5: Diseño estructural (*Top-level*) para H(7,4)

En la Figura 20, podemos apreciar los diferentes componentes definidos y su interconexión.

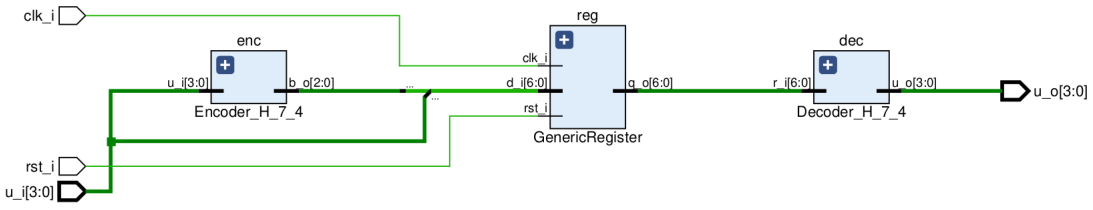

Figura 20. Esquema RTL global

La implementación de cada componente individual, en la Figura 21, se corresponde con la mostrada en la Sección 4.2, con la peculiaridad de que el registro y el decodificador se han integrado en un único componente (`reg`). Adicionalmente, se observan los buffers de entrada y salida necesarios para conectar la lógica que implementa el diseño con los pines de entrada y salida del dispositivo lógico reconfigurable.

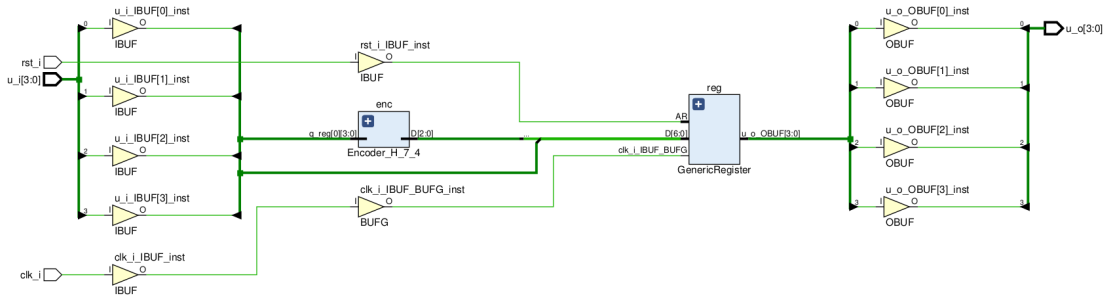

Figura 21. Esquema *netlist* (síntesis) global

En la Figura 22 se observan diferentes rectángulos de colores etiquetados (de la forma `X[0...1] y Y[0...6]`), que corresponden a las diferentes regiones de reloj definidas en el dispositivo lógico configurable para el que se ha implementado el diseño; en nuestro caso, `xc7vx485tffg1157-1`: Xilinx Series-7, Virtex-7, con 485000 celdas lógicas, encapsulado *flip-chip* BGA de 1157 pines, y velocidad -1. El diseño se ha ubicado completamente en la región de reloj X0Y1, contenido en el recuadro rojo y azul, estando en este último la señal de reloj `clk_i`.

En la Figura 23 pueden apreciarse aquellos componentes internos seleccionados para su implementación: aparecen nueve pines (PAD, en rojo), siete biestables (en verde) y nueve LUTs (en azul). El pin PAD que falta se encuentra un poco más arriba de esta región, como hemos indicado, en el recuadro azul de la Figura 22.

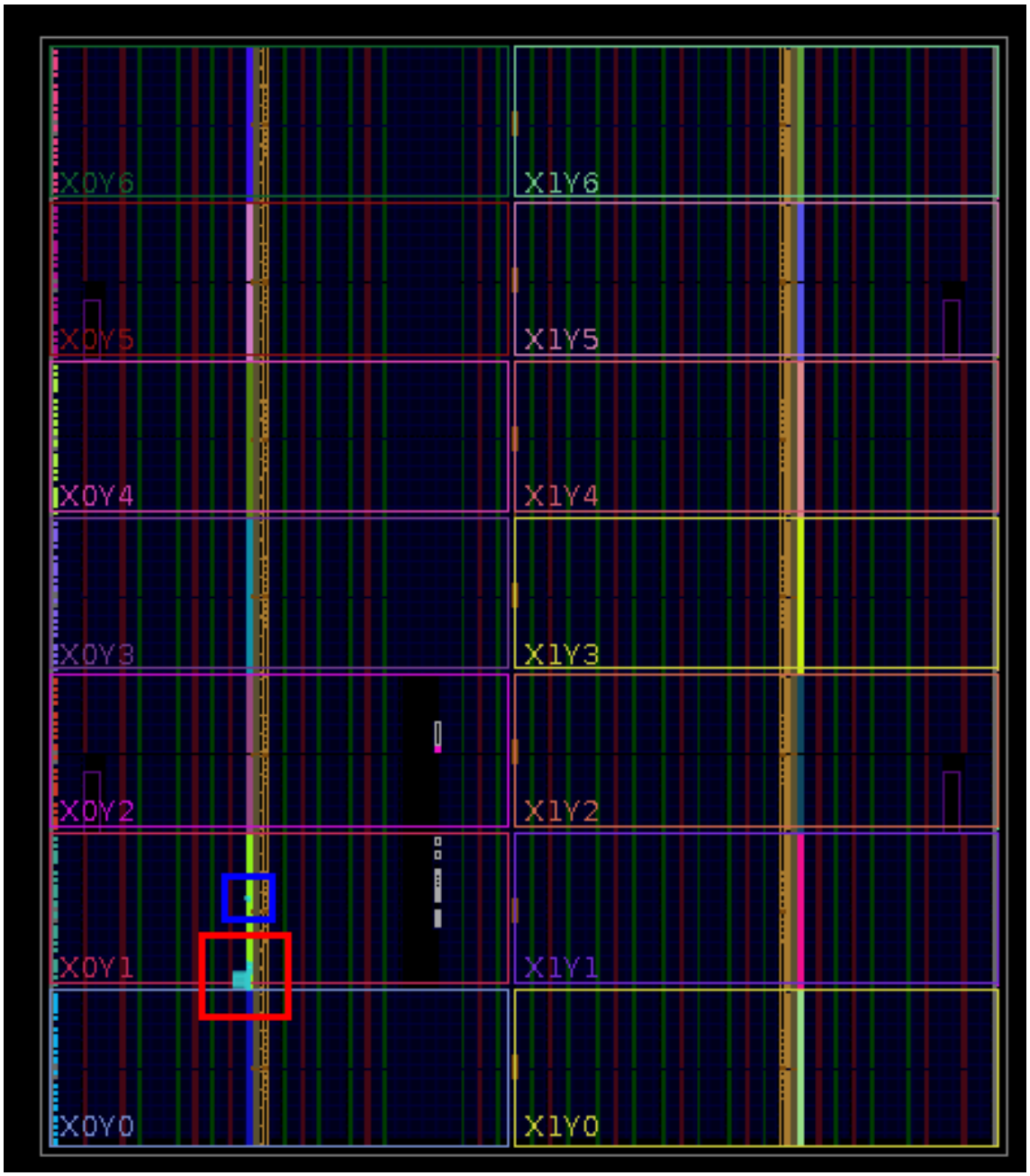

Figura 22. Vista general de la implementación del circuito

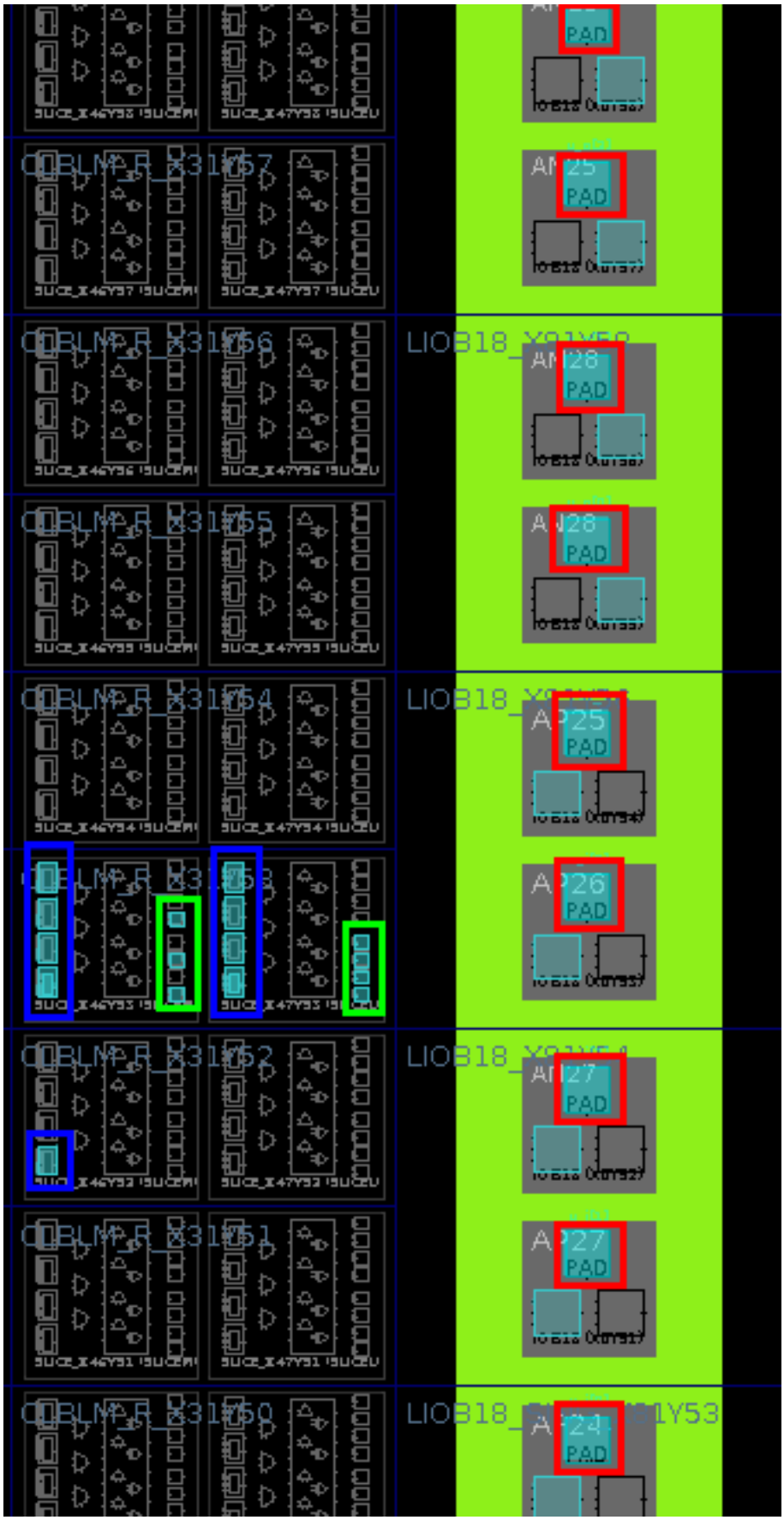

Figura 23. Vista ampliada de la implementación del circuito

Vivado también nos permite conocer cómo se han interconectado todos estos componentes; en la Figura 24, mostramos una vista ampliada de la interconexión de los pines PAD, encuadrados en rojo, y en la Figura 25 presentamos una vista ampliada de la matriz de encaminamiento, que es la zona por donde pasan todas las señales del circuito. Por otro lado, en la Figura 26, encontramos los biestables (en amarillo) y las LUTs (en azul). Por último, en la Figura 27, mostramos una vista global de las rutas de encaminamiento.

Figura 24. Vista detallada de las conexiones internas entre los pines PAD

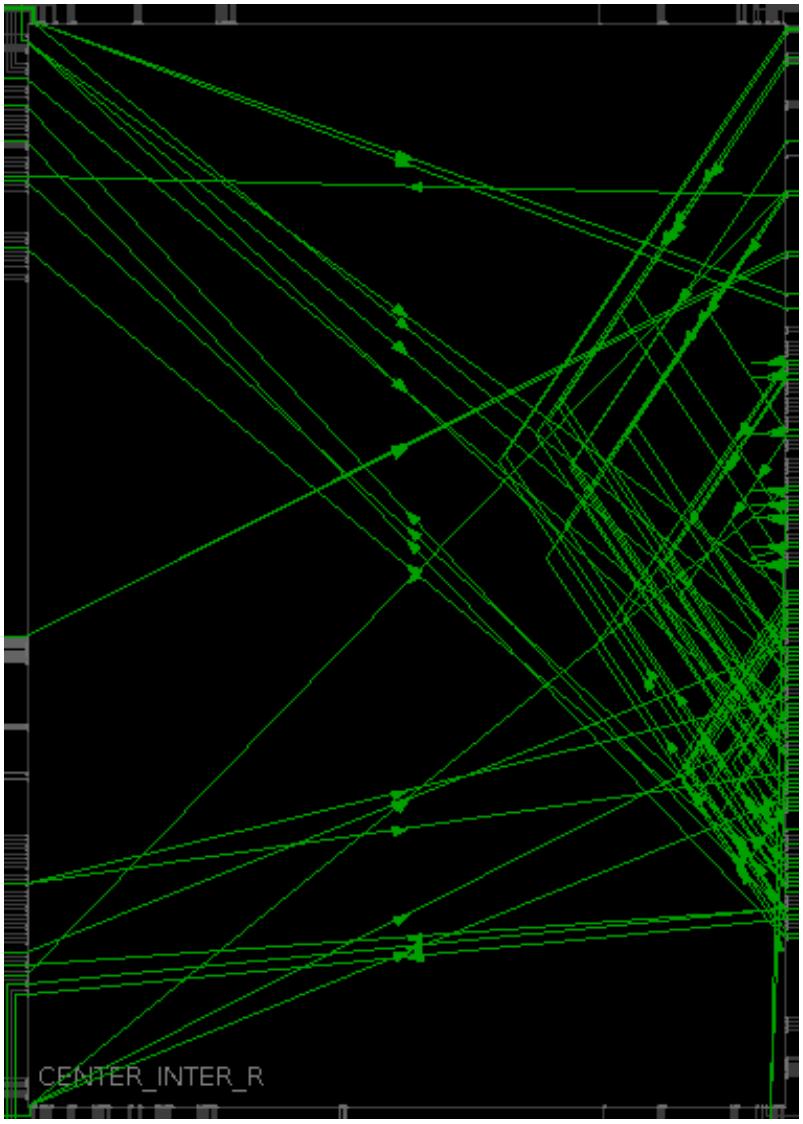

Figura 25. Vista ampliada de la matriz de encaminamiento

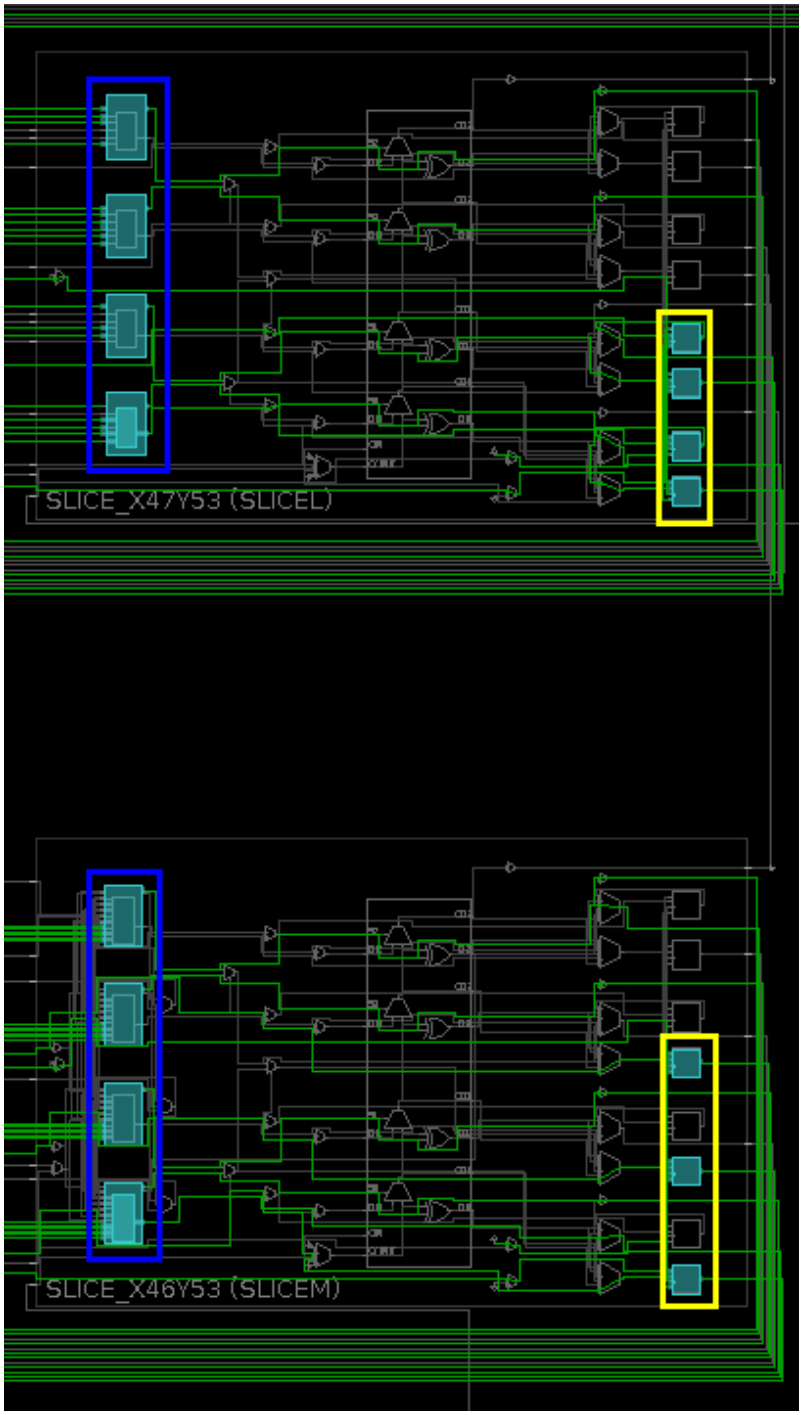

Figura 26. Vista detallada de las conexiones internas entre *flip-flops* y LUTs

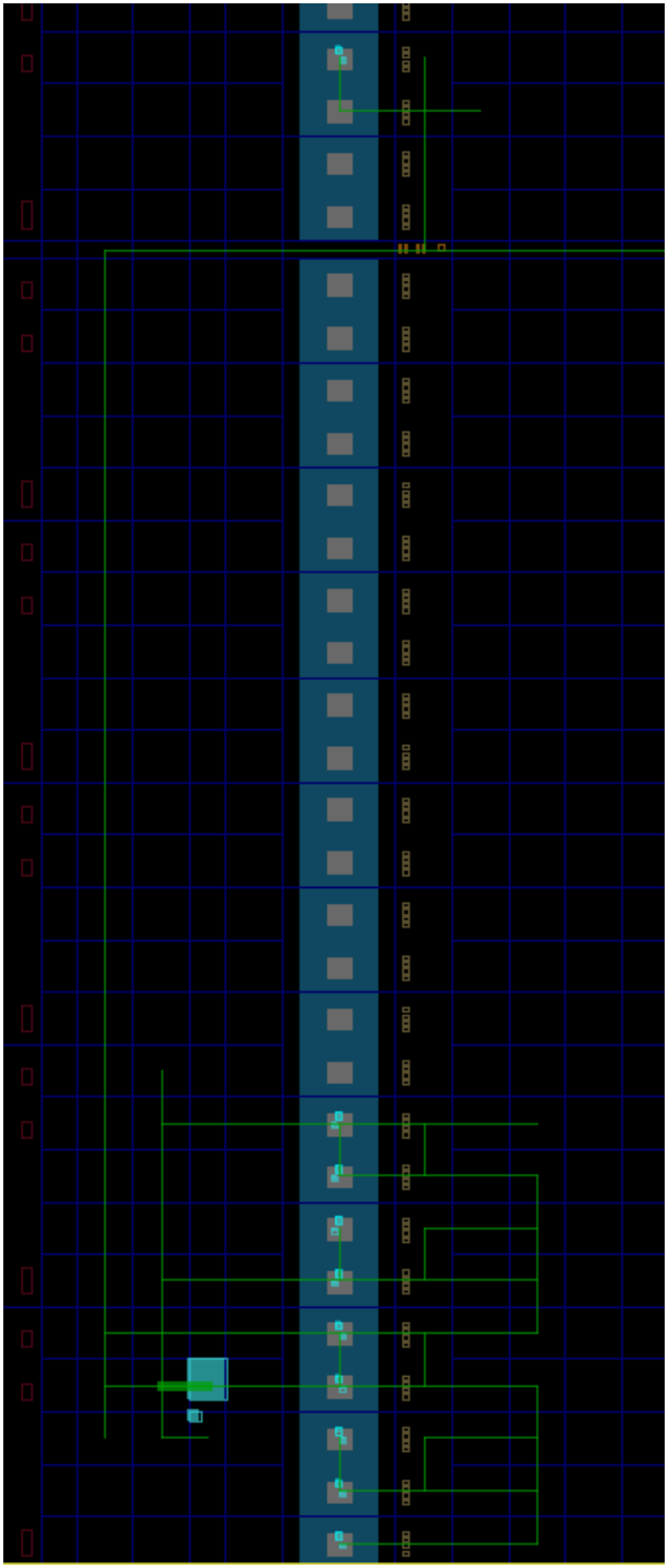

Figura 27. Vista global de las rutas de encaminamiento

6.2.5. *Test-bench*

En lo que respecta al *Test-bench*, que figura en el Archivo 6.6, además de inicializar el componente y definir las señales auxiliares que se usan para poder configurar la simulación, se llevan a cabo los experimentos, en este caso para todas las posibles entradas (desde la línea 81 hasta la 95). Asimismo, se lleva un registro de los fallos acontecidos en cada uno y se registran los resultados en un fichero CSV (línea 68 y líneas desde la 98 hasta la 103).

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  -- Uncomment the following library declaration if using
5  -- arithmetic functions with Signed or Unsigned values
6  use IEEE.NUMERIC_STD.ALL;
7
8  library STD;
9  use STD.TEXTIO.ALL;
10 use IEEE.STD_LOGIC_TEXTIO.ALL;
11
12 -- Uncomment the following library declaration if instantiating
13 -- any Xilinx leaf cells in this code.
14 --library UNISIM;
15 --use UNISIM.VComponents.all;
16
17 entity Register_H_7_4_tb is
18 --  Port ( );
19 end Register_H_7_4_tb;
20
21 architecture Behavioral of Register_H_7_4_tb is
22
23     component Register_H_7_4 is
24         Port ( clk_i : in STD_LOGIC;
25                rst_i : in STD_LOGIC;
26                u_i   : in STD_LOGIC_VECTOR(3 downto 0);
27                u_o   : out STD_LOGIC_VECTOR(3 downto 0));
28     end component;
29
30     -- Inputs
31     signal clk_i : STD_LOGIC := '0';
32     signal rst_i : STD_LOGIC := '0';
33     signal u_i : STD_LOGIC_VECTOR(3 downto 0) := (others => '0');
34     -- Ouputs
35     signal u_o : STD_LOGIC_VECTOR(3 downto 0);
36
37     -- Constants
38     constant CLK_PERIOD : time := 10 ns;
39
40     -- Output file
41     file file_handler : text;
42
43 begin
44
45     -- Unit Under Test
46     uut : Register_H_7_4
47     port map (
48         clk_i => clk_i,
49         rst_i => rst_i,
50         u_i => u_i,
51         u_o => u_o
52     );
53
54     -- Clock generation
```

```
55      clk_i <= not clk_i after CLK_PERIOD/2;
56
57      -- Stimuli
58      process
59          variable log : line;
60          variable experiments : integer := 0;
61          variable failures : integer := 0;
62
63          variable checkValue : STD_LOGIC_VECTOR(3 downto 0) := (others => '0');
64          variable nextValue : STD_LOGIC_VECTOR(3 downto 0) := (others => '0');
65      begin
66
67          -- open file in append mode
68          file_open( file_handler, "fault_injection.csv", append_mode);
69
70          -- Reset
71          rst_i <= '1';
72          wait for CLK_PERIOD * 10; -- Could be any time length
73
74          -- Deactivate reset
75          rst_i <= '0';
76
77          u_i <= (others => '0');
78          wait for CLK_PERIOD / 2;
79
80          -- Provide all possible inputs during 1 clock period each
81          for i in 1 to 2**u_i'length loop
82              wait for CLK_PERIOD / 2;
83
84              nextValue := u_i;
85              u_i <= std_logic_vector(to_unsigned(i, u_i'length));
86
87              wait for CLK_PERIOD / 2;
88              -- Check the output is correct
89              if u_o /= nextValue then
90                  failures := failures + 1; -- Log 1 failure
91              end if;
92
93              experiments := experiments + 1; -- Log 1 experiment
94
95          end loop;
96
97          -- Log number of experiments, failures and robustness
98          write(log, experiments);
99          write(log, string'(","));
100         write(log, failures);
101         write(log, string'(","));
102         write(log, 1.0-(real(failures)/real(experiments)));
103         writeline(file_handler, log);
104
105         report "Done!";
106         -- close file
107         file_close(file_handler);
108
109         wait;
110     end process;
111
112 end Behavioral;
```

Archivo 6.6: *Test-bench* para H(7,4)

6.2.6. Inyección de fallos

Para inyectar los fallos, se deben definir algunas variables como, por ejemplo, el periodo de la señal de reloj o la duración de la inyección (líneas 10 a la 13, en el Archivo 6.7). Seguidamente, se recorre cada bit del registro y se fuerza a su valor invertido. Este fichero se ejecutará al mismo tiempo que el *Test-bench*, con lo que sus parámetros temporales deberán estar sincronizados con el mismo.

```
1  #clear the waveform
2  restart
3  remove_wave -of [current_wave_config] [get_waves -of [current_wave_config] *]
4
5  # create a list with all the target signals
6  set lst { /Register_H_7_4_tb/uut/reg/q }
7  set input /Register_H_7_4_tb/u_i
8
9  # constants
10 set clkPeriod 10
11 set resetDone [expr {$clkPeriod * 10}]
12 set injectionTime [expr {$clkPeriod / 2}]
13 set injectionDuration [expr {$clkPeriod - $injectionTime}]
14
15 #output file
16 set channel [open fault_injection_tcl.csv a+]
17
18 # for each signal
19 foreach item $lst {
20     set length [llength [get_objects $item[*]]]
21     set inputLength [llength [get_objects $input[*]]]
22     set simulationLength [expr {(pow(2, $inputLength) * $clkPeriod) +
       $resetDone + $clkPeriod}]
23
24     # for each bit of that signal
25     for {set bit 0} {$bit < $length} {incr bit} {
26         # get access to that bit - signalName[bit]
27         set target $item[$bit]
28         puts $channel "$target"
29         # run until reset done
30         run $resetDone ns
31         # run until the first injection time
32         run $clkPeriod ns
33         set simTime [expr {$resetDone + $clkPeriod}]
34         # for each clock cycle
35         while {$simTime < $simulationLength} {
36             if {[get_value $target] == 0} {
37                 set fault 1
38             } else {
39                 set fault 0
40             }
41             # force that bit to the faulty value and cancel after the fault
                   duration
42             #puts "force $target to $fault from $simTime to [expr {$simTime +
                   $injectionDuration}]"
43             add_force $target -radix hex "$fault 0ns" -cancel_after
                   $injectionDuration
44
45             # run the simulation for one clock period (until the next injection
                    time)
46             run $clkPeriod ns
47
48             incr simTime $clkPeriod
49         }
50         run $clkPeriod
```

```
51
52          # restart the simulation
53          restart
54      }
55
56  }
57
58  flush $channel
59  close $channel
```

Archivo 6.7: Inyección de fallos en un bit para H(7,4)

6.2.7. Generación y simulación del proyecto

A la hora de generar el proyecto y lanzar la simulación funcional, se ejecutan una serie de comandos que se corresponden con los pasos que se deberían seguir si se realizara este proceso desde la interfaz de usuario de Vivado. Entre ellos, como podemos ver en el Archivo 6.8, destacamos la creación del propio proyecto (`create_project`), la adición de los ficheros fuente y simulación (`add_files`) y la ejecución de la simulación (`launch_simulation`).

```
1  create_project -force Hamming_7_4 /home/cristina/pruebas/HammingECC_H_7_4/
       Hamming_7_4 -part xc7vx485tffg1157-1
2  set_property target_language VHDL [current_project]
3  set_property simulator_language VHDL [current_project]
4  add_files -norecurse { /home/cristina/pruebas/HammingECC_H_7_4/src/
       Decoder_H_7_4.vhd /home/cristina/pruebas/HammingECC_H_7_4/src/
       GenericRegister.vhd /home/cristina/pruebas/HammingECC_H_7_4/src/
       Register_H_7_4.vhd /home/cristina/pruebas/HammingECC_H_7_4/src/
       Encoder_H_7_4.vhd }
5  set_property SOURCE_SET sources_1 [get_filesets sim_1]
6  add_files -fileset sim_1 -norecurse { /home/cristina/pruebas/HammingECC_H_7_4/
       sim/Register_H_7_4_tb.vhd }
7  update_compile_order -fileset sim_1
8  import_files -force -norecurse
9  update_compile_order -fileset sim_1
10 set_property -name {xsim.simulate.runtime} -value {2700ns} -objects [
       get_filesets sim_1]
11 launch_simulation
12 close_sim -force
13 close_project
14 q
```

Archivo 6.8: Generador del proyecto y simulación funcional (*Golden-run*) para H(7,4)

En el caso de la simulación con inyección de un fallo en un bit, mostrado en el Archivo 6.9, se trata del mismo proceso, con la diferencia de que también se debe importar y habilitar el fichero de inyección de fallos explicado en el Archivo 6.7.

```
create_project -force Hamming_7_4 /home/cristina/pruebas/HammingECC_H_7_4/
    Hamming_7_4 -part xc7vx485tffg1157-1
set_property target_language VHDL [current_project]
set_property simulator_language VHDL [current_project]
add_files -norecurse { /home/cristina/pruebas/HammingECC_H_7_4/src/
    Decoder_H_7_4.vhd /home/cristina/pruebas/HammingECC_H_7_4/src/
    GenericRegister.vhd /home/cristina/pruebas/HammingECC_H_7_4/src/
    Register_H_7_4.vhd /home/cristina/pruebas/HammingECC_H_7_4/src/
    Encoder_H_7_4.vhd }
set_property SOURCE_SET sources_1 [get_filesets sim_1]
add_files -fileset sim_1 -norecurse { /home/cristina/pruebas/HammingECC_H_7_4/
    sim/Register_H_7_4_BitFlip.tcl /home/cristina/pruebas/HammingECC_H_7_4/sim/
    Register_H_7_4_tb.vhd }
update_compile_order -fileset sim_1
set_property is_enabled true [get_files /home/cristina/pruebas/HammingECC_H_7_4
    /sim/Register_H_7_4_BitFlip.tcl]
import_files -force -norecurse
update_compile_order -fileset sim_1
set_property -name {xsim.simulate.runtime} -value {2700ns} -objects [
    get_filesets sim_1]
launch_simulation
close_sim -force
close_project
q
```

Archivo 6.9: Generador del proyecto y simulación inyectando un fallo en un bit (*1-bit injection*) para H(7,4)

6.2.8. Fichero de resultados

En concordancia a lo comentado en la Subsección 5.2.3, el fichero de resultados generado tendrá una estructura como la de la Figura 28:

GOLDEN RUN			
Number of experiments	Number of failures	Robustness	
16	0	1.000000e+00	
1 BIT INJECTION			
Number of experiments	Number of failures	Robustness	Bit injected
16	0	1.0	/Register_H_7_4_tb/uut/reg/q[0]
16	0	1.0	/Register_H_7_4_tb/uut/reg/q[1]
16	0	1.0	/Register_H_7_4_tb/uut/reg/q[2]
16	0	1.0	/Register_H_7_4_tb/uut/reg/q[3]
16	0	1.0	/Register_H_7_4_tb/uut/reg/q[4]
16	0	1.0	/Register_H_7_4_tb/uut/reg/q[5]
16	0	1.0	/Register_H_7_4_tb/uut/reg/q[6]

Figura 28. Hoja de cálculo con los resultados globales

6.3 Primera fase

Para realizar las primeras verificaciones, nos apoyaremos en el uso del simulador de Vivado. A través de esta herramienta, podemos comprobar, de forma muy visual, el valor que toman las distintas señales, tanto cuando realizamos la verificación funcional

(en ausencia de fallos) como cuando inyectamos un fallo (en este caso, forzaremos el valor de un bit concreto).

6.3.1. Verificación funcional

Vamos a ejemplificar este proceso a través de un código Hamming H(12,8). En este caso, lanzaremos la simulación a través del menú de Vivado (como mostramos en la Figura 8). En la Figura 29, podemos ver la señal `u_i` (entrada), la `u_o` (salida) y la `q`, que se corresponde con el valor almacenado del registro, resaltado en azul. Como podemos ver, la entrada y la salida coinciden, con lo que concluimos que el sistema funciona correctamente.

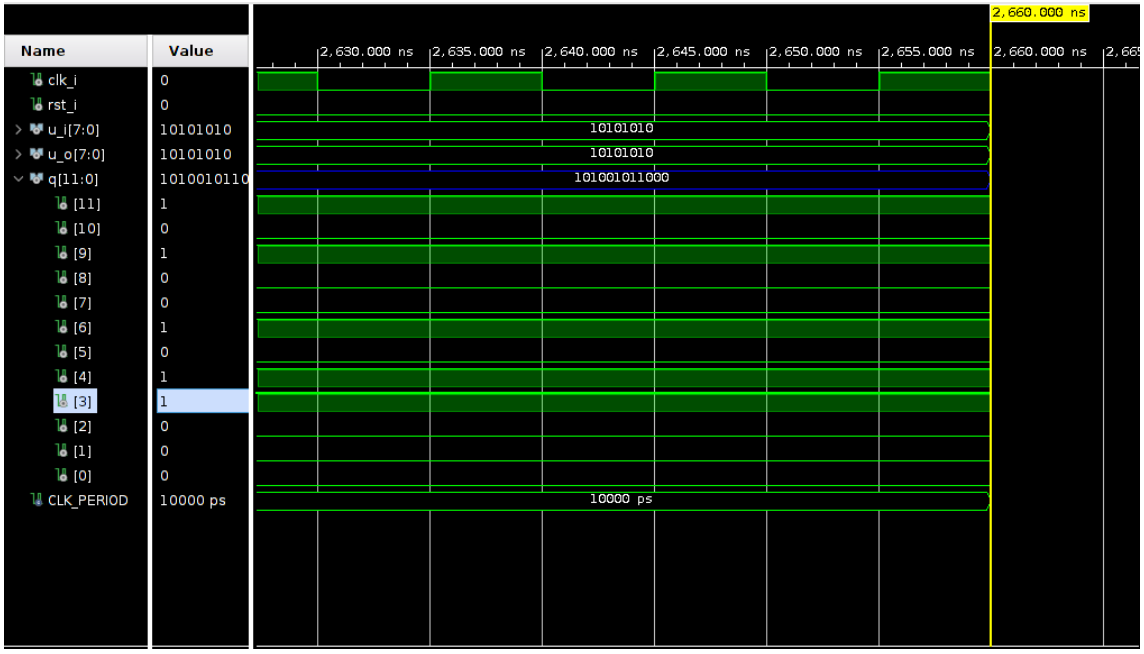

Figura 29. Verificación funcional de la primera fase

6.3.2. Verificación con fallos

Para comprobar que el sistema es tolerante a un fallo en un bit, tendremos que forzar algún bit de la señal de registro al valor contrario con el que fue almacenado. En nuestro caso, hemos optado por el bit 3 (resaltado en rojo en la Figura 30). La forma de ver que el circuito funciona correctamente es que, a pesar de esto, la salida no se ve modificada; es decir, el circuito es capaz de corregir el fallo, porque la entrada y la salida del sistema coinciden.

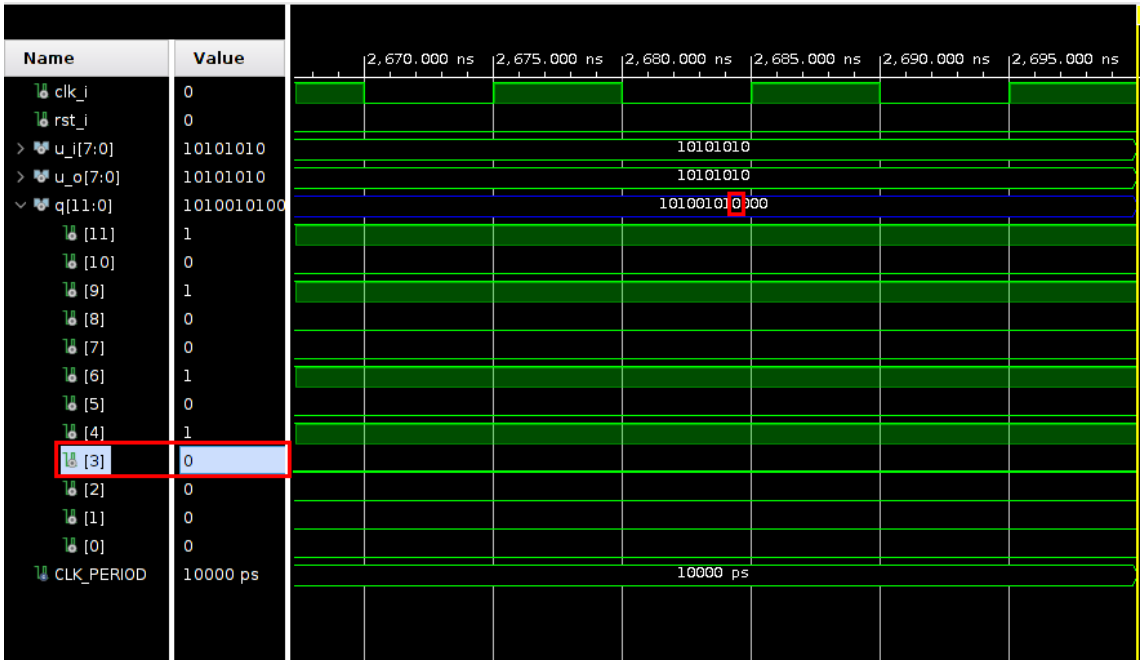

Figura 30. Verificación con fallos de la primera fase

6.4 Segunda fase

En esta fase, procederemos a realizar pruebas de forma más exhaustiva gracias a la automatización de todo el proceso. Al final del mismo, nos aparecerán los resultados por pantalla, aunque también podremos comprobarlos accediendo a la carpeta de resultados.

6.4.1. Verificación funcional

Partimos de un código H(7,4) como ejemplo. Tras la ejecución del programa, obtenemos los resultados que se presentan en la Tabla 6.1:

Golden Run		
Número de experimentos	Número de fallos	Robustez
16	0	1.0

Tabla 6.1: resultados obtenidos en la verificación funcional de H(7,4)

El número de experimentos realizados depende del tamaño de la palabra y de los fallos inyectados. Podemos ver que el número de experimentos es $2^n = 2^4 = 16$, siendo n el tamaño de la palabra. El número de fallos es 0 y la robustez (coeficiente que mide la capacidad del circuito de corregir los fallos) es 1.0 tal y como se esperaba, ya que en este punto todavía no hemos inyectado fallos.

Para códigos H(12,8) y H(38,32), los resultados son análogos y se presentan, respectivamente, en la Tabla 6.2 y Tabla 6.3. En el caso de este último, el número de experimentos es de 2, ya que, recordando lo explicado en la Sección 5.1, a partir de palabras mayores de 8 bits, optamos por realizar experimentos con dos patrones de entrada.

Golden Run		
Número de experimentos	Número de fallos	Robustez
256	0	1.0

Tabla 6.2: resultados obtenidos en la verificación funcional de H(12,8)

Golden Run		
Número de experimentos	Número de fallos	Robustez
2	0	1.0

Tabla 6.3: resultados obtenidos en la verificación funcional de H(38,32)

6.4.2. Verificación con fallos

En este caso, se inyecta un fallo en cada bit del registro para todas las posibles entradas. Esto es: cada fila indica que el bit inyectado N del registro se invierte para todos y cada uno de los posibles valores que puede tomar la palabra. Como podemos observar en la Tabla 6.4, el número de fallos sigue siendo 0 y la robustez 1.0. Por otro lado, para los códigos H(12,8) (Tabla 6.5) y H(38,32) (Tabla 6.6), podemos comprobar que tampoco se dan fallos y, por tanto, la robustez es la óptima.

Inyección en un bit			
Número de experimentos	Número de fallos	Robustez	Bit inyectado
16	0	1.0	0
16	0	1.0	1
16	0	1.0	2
16	0	1.0	3
16	0	1.0	4
16	0	1.0	5
16	0	1.0	6

Tabla 6.4: resultados obtenidos en la verificación con inyección de un fallo en un bit de H(7,4)

Inyección en un bit			
Número de experimentos	Número de fallos	Robustez	Bit inyectado
256	0	1.0	0
256	0	1.0	1
256	0	1.0	2
256	0	1.0	3
256	0	1.0	4
256	0	1.0	5
256	0	1.0	6
256	0	1.0	7
256	0	1.0	8
256	0	1.0	9
256	0	1.0	10
256	0	1.0	11

Tabla 6.5: resultados obtenidos en la verificación con inyección de un fallo en un bit de H(12,8)

Inyección en un bit			
Número de experimentos	Número de fallos	Robustez	Bit inyectado
2	0	1.0	0
2	0	1.0	1
2	0	1.0	2
2	0	1.0	3
2	0	1.0	4
2	0	1.0	5
...	...	...	...
2	0	1.0	32
2	0	1.0	33
2	0	1.0	34
2	0	1.0	35
2	0	1.0	36
2	0	1.0	37

Tabla 6.6: resultados obtenidos en la verificación con inyección de un fallo en un bit de H(38,32)

6.5 Estimación de tiempos

Durante la realización de las pruebas, hemos considerado interesante medir los tiempos de ejecución; en concreto, el tiempo de generación del código y el de verificación, concluyendo que:

- el tiempo de creación del código y proyecto Vivado escala bien independientemente del tamaño de los datos; es decir, no se aprecian diferencias significativas en el tiempo medio de generación del código (que es de 12.8 segundos) aunque el tamaño de la palabra aumente de forma considerable;
- por contra, el tiempo de verificación, al realizarse primero con todos los posibles datos de entrada, siempre es el peor caso posible, aumentando exponencialmente. Esto es debido a que el número de experimentos toma la forma de 2^n, siendo n el

número de bits de la palabra, para cada bit inyectado. Por ejemplo: para H(7,4), el tiempo de verificación es de 3.41 segundos. En el caso de H(12,8), el tiempo sobrepasa el minuto (63.76 segundos). Por esta razón, concluimos que es una buena opción considerar la verificación solo con dos patrones para palabras mayores de 8 bits;

- teniendo esto en cuenta, para palabras de mayor tamaño, hemos observado que, a partir de palabras de 16 bits, el tiempo medio por inyección es de 0.15 segundos. Como la inyección se realiza para dos posibles valores en cada bit del registro de almacenamiento, el tiempo aumenta, pero de forma lineal, ya que el número de experimentos se mantiene constante, siendo este dos. Por ejemplo: para H(21,16), el tiempo de verificación total es de 6.74 segundos; para H(38,32) de 11.26 segundos; y para H(71,64) de 20.99 segundos, siendo este un incremento mucho más admisible que el anterior.

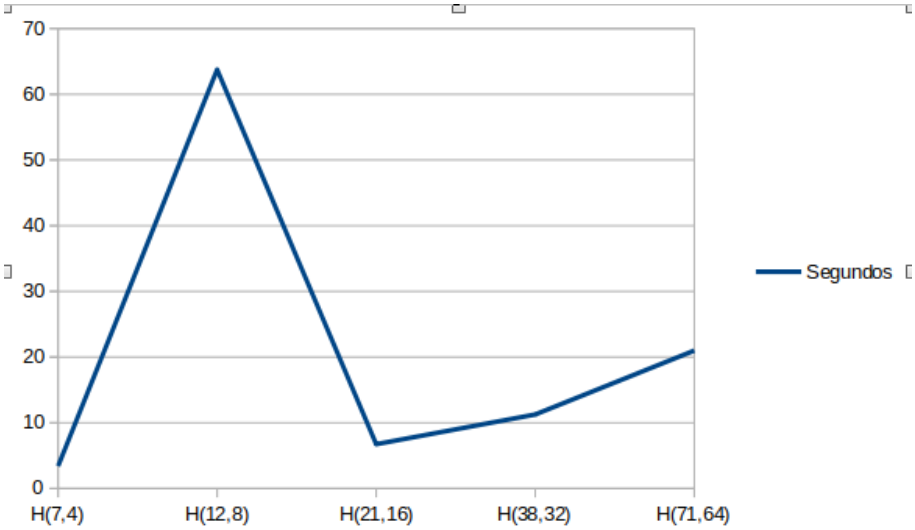

Figura 31. Variación de los tiempos de ejecución para distintos códigos

CAPÍTULO 7
Conclusiones

El presente estudio nos ha servido para aprender conceptos clave de la codificación, su implicación en diversas áreas del mundo actual y cómo podemos aprovechar este conocimiento para construir circuitos tolerantes a fallos. Del mismo modo, hemos podido comprobar empíricamente las ventajas de automatizar el proceso de generación de dichos circuitos, teniendo como base la separación de conceptos y la reducción de la intervención humana en el proceso. También nos ha servido para implementar una solución basada en la metaprogramación, aunando y complementando así las utilidades de un lenguaje de bajo nivel (VHDL) con otro de alto nivel (Python) consiguiendo, no solo instruirnos en el uso de librerías externas, sino también profundizar en conceptos relacionados con el diseño de sistemas digitales, fundamentalmente en lo que respecta a la verificación funcional. De esta forma, a través del desarrollo del programa, hemos podido cubrir los objetivos expuestos en la Sección 1.2. Adicionalmente, nos ha resultado especialmente beneficioso para mejorar en la resolución de problemas que, hasta la fecha de realización del presente análisis, nos eran novedosos.

Como posibles anexos futuros a la investigación realizada, podríamos plantear la adición de más tipos de ECCs, la implementación del circuito en un dispositivo final e, incluso, dotar al programa de una interfaz de usuario. No obstante, cabe recalcar que esta primera aproximación nos ha servido para derivar que, gracias a la automatización conseguida por medio de este proyecto, hemos podido transformar un proceso manual largo, costoso y muy proclive a equivocaciones en otro más eficiente, transparente, robusto, con mínima intervención humana y, por lo tanto, menos propenso a errores.

> *El problema fundamental de la comunicación es el de reproducir, en un punto, otro mensaje que acontece en otro punto, de forma exacta o aproximada.*
>
> — Claude Shannon

Bibliografía

[1] T. Richardson y R. Urbanke. *Modern Coding Theory*. 1ª. Cambridge University Press, 2008.

[2] S. Moser y P. Chen. *A student's guide to coding information*. 1ª. Cambridge University Press, 2012.

[3] A. Avizienis, J.C. Laprie, B. Randell y col. "Basic concepts and taxonomy of dependable and secure computing". En: *IEEE Transactions on dependable and secure computing* 1.1 (2004), págs. 11-33.

[4] *South Atlantic Anomaly*. Wikipedia. Accedida en 29-01-2021. 2021. URL: https://en.wikipedia.org/wiki/South_Atlantic_Anomaly.

[5] P. Gil, J. Arlat, H. Madeira y col. "Dependability Benchmarking". En: *Fault representativeness* 1.1 (2002), págs. 14-15.

[6] T. Karnik, P. Hazucha y J. Patel. "IEEE Transactions on dependable and secure computing". En: *Characterization of Soft Errors Caused by Single Event Upsets in CMOS Processes* 1.2 (2004), págs. 128-129.

[7] G. Solomon e I. Reed. "Polynomial Codes over Certain Finite Fields". En: *Journal of the Society for Industrial and Applied Mathematics* 2.8 (1960), págs. 300-304.

[8] Cobham Advanced Electronic Solutions. *GR740*. GR740. Accedida en 29-01-2021. 2018. URL: https://www.gaisler.com/index.php/products/components/gr740.

[9] L.J. Saiz. "Fallos intermitentes: análisis de causas y efectos, nuevos modelos de fallos y técnicas de mitigación". Tesis doct. Universitat Politècnica de València, 2015.

[10] J. Gracia, L.J. Saiz, P.J. Gil y col. "Protection of Processor Registers by using Very Fast Single Error Correction Codes". En: *WIICT2015: Proceedings of the Workshop on Innovation on Information and Communication Technologies* (2015), pág. 179.

[11] L. Soares, I. Rafiq, M. Rossi y col. *Handbook of Research on Embedded System Design*. 1ª. IGI Global, 2014.

[12] Xilinx. *Xilinx TMRTool User Guide*. Hamburg VHDL Archive. Accedida en 05-02-2021. 2017. URL: https://tams-www.informatik.uni-hamburg.de/vhdl.

[13] Siemens. *Precision Hi-Rel*. Siemens Digital Industries Software. Accedida en 05-02-2021. 2017. URL: https://eda.sw.siemens.com/en-US/ic/precision/hi-rel.

[14] *High Reliability Design: No Room for Error*. Synopsys. Accedida en 05-02-2021. 2021. URL: https://www.synopsys.com/implementation-and-signoff/fpga-based-design/high-reliability.html.

[15] *MyHDL*. MyHDL. Accedida en 05-02-2021. 2021. URL: http://www.myhdl.org/.

[16] *Xilinx*. Xilinx. Accedida en 11-05-2021. 2021. URL: https://www.xilinx.com/.

[17] *AMD Agrees to Buy Xilinx for $35 Billion in Stock*. The New York Times. Accedida en 11-05-2021. 2020. URL: https://www.nytimes.com/2020/10/27/technology/amd-xilinx-35-billion-stock-deal.html.

[18] *Intel FPGAs and Programmable Devices*. Intel FPGA. Accedida en 11-05-2021. 2021. URL: https://www.intel.com/content/www/us/en/products/programmable.html.

[19] *Official At Last: Intel Completes $16.7 Billion Buy of Altera*. Fortune. Accedida en 11-05-2021. 2015. URL: https://fortune.com/2015/12/28/intel-completes-altera-acquisition/.

[20] *Vivado Design Suite User Guide*. Vivado Design Suite. Accedida en 11-05-2021. 2021. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_2/ug1118-vivado-creating-packaging-custom-ip.pdf.

[21] *Intel Quartus Prime Standard Edition User Guide*. Intel Quartus User Guide. Accedida en 11-05-2021. 2021. URL: https://www.intel.com/content/www/us/en/programmable/documentation/yoq1529444104707.html.

[22] A. Ruede. "A Scientist's Guide to FPGAs". En: *Inverted CERN School of Computing (07-03-2019)*. 2019, págs. 7-8.

[23] V. Betz, J. Rose y A. Marquardt. *Architecture and CAD for Deep-Submicron FPGAs*. USA: Kluwer Academic Publishers, 1999. ISBN: 0-7923-8460-1.

[24] *IEEE 1076-2019 - IEEE Standard for VHDL Language Reference Manual*. IEEE Standards Association. Accedida en 15-05-2021. 2021. URL: https://standards.ieee.org/standard/1076-2019.html.

[25] *Ada 2012 Language Reference Manual*. Ada 2012 Language Reference Manual. Accedida en 15-05-2021. 2021. URL: http://www.ada-auth.org/standards/ada12.html.

[26] *IEEE 1800-2017 - IEEE Standard for SystemVerilog*. IEEE Standards Association. Accedida en 15-05-2021. 2021. URL: https://standards.ieee.org/standard/1800-2017.html.

[27] *The GNU C Reference Manual*. The GNU C Reference Manual. Accedida en 15-05-2021. 2021. URL: https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html.

[28] *Algoritmo Quine—McCluskey*. Wikipedia. Accedida en 15-05-2021. 2021. URL: https://es.wikipedia.org/wiki/Algoritmo%5C_Quine---McCluskey.

[29] J. Cong e Y. Ding. "FlowMap: An Optimal Technology Mapping Algorithm for Delay Optimization in Lookup-Table Based FPGA Designs". En: *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems* 13.1 (1994), pág. 1.

[30] *Simulated Annealing*. Wikipedia. Accedida en 15-05-2021. 2021. URL: https://es.wikipedia.org/wiki/Algoritmo%5C_de%5C_recocido%5C_simulado.

[31] *Pathfinding*. Wikipedia. Accedida en 15-05-2021. 2021. URL: https://en.wikipedia.org/wiki/Pathfinding.

[32] J. Arlat, A. Costes, Y. Crouzet y col. "Fault injection and dependability evaluation of fault-tolerant systems". En: *IEEE Transactions on Computers* 42.8 (1993), págs. 913-923. DOI: 10.1109/12.238482.

[33] A. Benso y P. Prinetto. *Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation*. Frontiers in Electronic Testing, 23. Boston, MA: Springer US, 2003. ISBN: 0-306-48711-X.

[34] *TCL/Tk Documentation*. Tcl Developer Site. Accedida en 17-05-2021. 2021. URL: https://www.tcl.tk/doc/.

[35] Abbas Mohammadi, Mojtaba Ebrahimi, Alireza Ejlali y col. "SCFIT: A FPGA-based fault injection technique for SEU fault model". En: *2012 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2012, págs. 586-589. DOI: 10.1109/DATE.2012.6176538.

[36] *What is NumPy*. NumPy v1.20 Manual. Accedida en 17-05-2021. 2021. URL: https://numpy.org/doc/stable/user/whatisnumpy.html.

[37] *User Guide*. pandas 1.2.4 documentation. Accedida en 22-05-2021. 2021. URL: https://pandas.pydata.org/docs/user_guide/index.html#user-guide.