Universitat Politècnica de València
Departamento de Informática de Sistemas y Computadores

Biblioteca de soporte para el despliegue automático de mecanismos tolerantes a fallos para mejorar la robustez de circuitos implementados mediante High Level Synthesis

Trabajo Fin de Máster

Máster Universitario en Ingeniería de Computadores y Redes

Autor: Raúl Lozano Torres
Tutores : David de Andrés Martínez y Juan Carlos Ruiz García

Septiembre 2021

Resumen

High Level Synthesis (HLS) aparece como un nuevo paradigma para el diseño de sistemas digitales, en el que la funcionalidad del circuito se describe utilizando lenguajes de alto nivel como C o C++, y las herramientas de análisis obtienen automáticamente una descripción funcionalmente equivalente utilizando lenguajes de descripción de *hardware*, como VHDL o Verilog, para su implementación en dispositivos lógicos configurables de tipo FPGA (*Field-Programmable Gate Arrays*).

A pesar de que los entornos que soportan HLS disponen de un gran número de parámetros de configuración que permiten adaptar la síntesis realizada para que se ajuste a las necesidades funcionales existentes, no existen actualmente parámetros que permitan ajustar esta síntesis para mejorar la robustez del circuito resultante.

Este Trabajo Fin de Máster pretende estudiar la posibilidad de desarrollar en Python bibliotecas de apoyo al desarrollador de circuitos mediante HLS para automatizar la integración y despliegue de mecanismos de tolerancia a fallos basados en redundancia espacial. Con ello, se consigue la separación de preocupaciones (*separation of concerns*), de tal forma que el desarrollador del circuito centra su atención en la definición de la funcionalidad del circuito en lenguaje C, y el conjunto de bibliotecas desarrollado despliega automáticamente y de forma transparente los mecanismos de tolerancia a fallos definidos por el experto en confiabilidad.

Resum

High Level Synthesis (HLS) apareix com un nou paradigma per al disseny de sistemes digitals, en el qual la funcionalitat del circuit es descriu utilitzant llenguatges d'alt nivell com a C o C++, i les ferramentes d'anàlisis obtenen automàticament una descripció funcionalment equivalent utilitzant llenguatges de descripció de *hardware*, com ara VHDL o Verilog, per a la seua implementació en dispositius lògics configurables de tipus FPGA (*Field Programmable Gate Arrays*).

A pesar que els entorns que suporten HLS disposen d'un gran nombre de paràmetres de configuració que permeten adaptar la síntesi realitzada perquè s'ajuste a les necessitats funcionals existents, no existeixen actualment paràmetres que permeten ajustar la síntesi per a millorar la robustesa del circuit resultant.

Aquest Treball Fi de Màster pretén estudiar la possibilitat de desenvolupar en Python biblioteques de suport al desenvolupador de circuits mitjançant HLS per a automatitzar la integració i desplegament de mecanismes de tolerància a fallades basades en redundància espacial. Amb això, s'aconsegueix la separació de preocupacions (*separation of concerns*), de tal forma que el disenyador del circuit centra la seua atenció en la definició de la funcionalitat del circuit en llenguatge C, i el conjunt de biblioteques desenvolupat desplega automàticament i de manera transparent els mecanismes de tolerància a fallades definides per l'expert en confiabilitat.

Abstract

High Level Synthesis (HLS) appears as a new paradigm for the design of digital systems, in which the circuit functionality is described using high-level languages such as C or C++, after this, the analysis tools automatically obtain a functionally equivalent description using hardware description languages, such as VHDL or Verilog. This equivalent description could be implemented in configurable logic devices of the FPGA (Field-Programmable Gate Arrays) type.

Despite the fact that the environments that support HLS have a large number of configuration parameters that allow to adapt the synthesis performed to fit the existing functional needs, there are currently no parameters that allow to adjust this synthesis to improve the robustness of the resulting circuit.

This Master Thesis aims to study the possibility of developing in Python libraries to support the circuit developer using HLS to automate the integration and deployment of fault tolerance mechanisms based on spatial redundancy. In this way, separation of concerns is achieved, so that the circuit developer focuses his attention on the definition of the circuit functionality in C language, and the set of libraries developed automatically and transparently deploys the fault tolerance mechanisms defined by the reliability expert.

Índice general

1. Introducción **5**
1.1. Motivación . . . 5
1.2. Objetivo . . . 6
1.3. Estructura del documento . . . 7

2. Conceptos Previos **8**
2.1. Dispositivos FPGA . . . 8
2.2. *High Level Synthesis* (HLS) . . . 10
2.3. *Single Event Effects* (SEE) . . . 11
2.4. Mecanismos de tolerancia a fallos . . . 12
2.4.1. Redundancia Espacial . . . 12
2.4.2. Redundancia Temporal . . . 13

3. Entorno de trabajo **15**
3.1. *Visual Studio Code* . . . 15
3.2. Git . . . 15
3.3. Vitis HLS . . . 16
3.4. Vivado . . . 17

4. Proceso de desarrollo **18**
4.1. Revisión conceptual HLS . . . 18
4.2. Pruebas protección herramientas HLS . . . 19
4.2.1. Redundancia espacial . . . 19
4.2.2. Redundancia temporal . . . 24
4.3. Desarrollo injección fallos usando TCL . . . 27

5. Vision general de la biblioteca **31**
5.1. Descripción del proceso de protección . . . 31
5.2. NMR_C_Parser . . . 33
5.2.1. Estructura . . . 34
5.3. NMR_Verilog_Parser . . . 37
5.3.1. Funcionalidad . . . 37
5.4. Utilidades . . . 37

6. Experimento **39**
6.1. Multiplicación matriz por constante 39
6.1.1. Generación algoritmo tolerante a fallos 40
6.1.2. Verificación del mecanismo de tolerancia a fallos desplegado 44

7. Conclusiones **48**
7.1. Contratiempos y problemas sufridos 48
7.1.1. Estilos de programación 48
7.1.2. CHStone *benchmark* 49
7.2. Consideraciones y aprendizaje 50
7.3. Trabajo a futuro 51
7.3.1. Experimentos con CHStone 51
7.3.2. Redundancias 51
7.3.3. Integración herramientas Xilinx 51

Índice de figuras

2.1. Ejemplo 3-LUT 9
2.2. *Logic Block* 9
2.3. *Arquitectura interna de una FPGA* 10
2.4. SEUs capturados en la memoria de un satélite. 12
2.5. Diagrama NMR 13
2.6. Redundancia Temporal 13
2.7. Redundancia Temporal - Ejemplo de fallo transitorio 14

3.1. Visual Code 16

4.1. *Scheduler* del código "suma"procesado con HLS 20
4.2. *Scheduler* del código "suma"procesado con HLS con *pragma inline off* añadido 21
4.3. *Scheduler* del código "sumaütilizando *pramgas* y nombres de función diferentes 22
4.4. Lanzamiento Co-Simulación NMR_Suma 23
4.5. Simulación Vivado - Resultado correcto después de inyectar un SEU en la primera réplica. 24
4.6. Módulos después de sintetizado 25
4.7. Módulos después de sintetizado con atributo añadido 25
4.8. Redundancia Temporal *Pragma Dataflow* 26
4.9. Redundancia Temporal *While lock* 27

5.1. Proceso protección código C 32
5.2. Proceso antioptimización Verilog 33

6.1. *Scheduler* 42
6.2. Co-simulación 43
6.3. Síntesis del diseño utilizando el atributo *DONT_TOUCH* 45
6.4. Forzado de bit a 1 46

ASIC *Application Specific Integrated Circuit*

ESA *European Space Agency*

FPGA *Field Programable Gate Array*

HLS *High Level Synthesis*

IDE *Integrated Development Environment*

RTL *Register transfer level*

TCL *Tool Command Language*

LUT *Lookup Tables*

VHDL *VHSIC Hardware Description Language*

VHSIC *Very High Speed Integrated Circuit*

Capítulo 1

Introducción

En el presente trabajo se ha implementado una biblioteca para el despligue automático de mecanismos de tolerancia a fallos mediante el uso de *High Level Syntesis (HLS)*. Esta herramienta ha sido parametrizada para aportar una mayor flexibilidad a la hora de realizar la protección de las funciones deseadas del diseño.

Para el desarrollo de estas bibliotecas se ha optado por el uso de Python debido a la facilidad que otorga este lenguaje para el tratamiento de textos, pudiendo facilitar el desarrollo de la herramienta.

1.1. Motivación

La motivación viene dada por la falta de herramientas existentes para ayudar al diseñador a implementar mecanismos de tolerancia a fallos de manera transparente en HLS. Si bien es cierto que existe la posibilidad de añadir ciertos tipos de redundancia haciendo uso de herramientas de Xilinx, para su uso es necesario conocimientos claros de las metodologías de tolerancia a fallos, de las herramientas del propio vendedor e incluso de conocimientos altos sobre el diseño de sistemas digitales haciendo uso de lenguajes HDL. En muchas ocasiones la persona que realiza la implementación de la lógica de la aplicación no dispone de estos conocimientos, especialmente si está acostumbrada y focalizada en en el desarrollo de ciertos algoritmos científicos y no en sistemas digitales para sistemas embarcados. Además las herramientas de Xilinx XTMRTool no cuentan con una gran variedad de opciones para su configuración.

Dada la problemática indicada anteriormente se presenta como necesario el desarrollo de nuevas técnicas que permitan a los diseñadores separar lo puramente funcional del diseño de las metodología de tolerancia a fallos necesarias como bien se indica en [1].

En la actualidad, si bien es cierto que se tiende a la profesionalización y, por tanto, a la división de tareas dentro de casi cualquier tipo de profesion, también es habitual observar como los equipos de diseñadores funcionales deben hacerse

cargo de tareas para las cuales no son expertos. Hablamos de la tolerancia a fallos, donde se requiere de experiencia adicional en el campo para poder diseñar un sistema que cumpla con los requerimientos que el producto necesita. De hecho, puede revisarse [3] para comprobar de primera mano como en instituciones como la *European Space Agency (ESA)* hacen especial hincapié en este tipo de temática, muy importante para sistemas espaciales, donde disponer de mecanismos de tolerancia fallos es algo indispensable y se precisa por tanto documentación adicional en la que se detalle el porqué del uso de este tipo de técnicas, qué problemas eliminan o reducen entre otra mucha información adicional.

Junto a la necesidad explicada, cabe remarcar que la importancia del *separation of concerns* es cada vez mayor y es aquí donde herramientas que faciliten esta separación pueden ser más utilites, tanto para el profesional funcional, como para el experto en técnicas de tolerancia a fallos.

Por otro lado, actualmente han aparecido nuevas técnicas para el diseño de sistemas digitales que intentan abstraer al diseñador del bajo nivel que proporcionan los lenguajes de descripción de *hardware* más utilizados en la actualidad como son Verilog o *VHSIC Hardware Description Language (VHDL)*. Este proceso, llamado *High-Level Synthesis* permite al diseñador del sistema describirlo usando código C, de esta manera se abre un nuevo abanico de profesionales del sector al mundo del diseño de dispositivos lógicos configurable de tipo *Field-Programmable Gate Array (FPGA)*.

Es por ello que, al aunar estas dos problemáticas, la necesidad de implantar técnicas de tolerancia a fallos y de abstraer al diseñador de los lenguajes de descripción de *hardware*, hace necesaria la creación de nuevas herramientas que permitan al profesional del diseño de sistemas usar HLS apoyado por bibliotecas que implementen soluciones tolerantes a fallos sin la necesidad de que el diseñador cuente con una amplia experiencia en las técnicas anteriomente indicadas.

1.2. Objetivo

El objetivo principal de este trabajo es la generación de unas bibliotecas que ayuden a los diseñadores a centrarse únicamente en la funcionalidad del programa en C desarrollado, dejando en manos del experto en toleracia a fallos toda la parte relacionada con ella. Es decir, que el diseñador del código C simplemente deba preocuparse en conseguir que la aplicación desarrollada haga lo que tiene que hacer sin más. Para lograr ese objetivo se realizará un estudio del estado actual de la tecnología HLS, así como diversos experimentos para concretar cuál sería la opción más viable a la hora de automatizar el proceso descrito y por último, se desarrollarán los códigos *Python*, a los cuales nos referiremos como *parser* a lo largo del todo el documento.

1.3. Estructura del documento

La estructura del documento será la siguiente:

Primero se abordarán ciertos conceptos previos que será necesario revisar para entender de mejor manera los conceptos que se irán mostrando a lo largo del trabajo. Aquí entraremos en conceptos más básicos como podrían ser la definición de los dispositivos FPGA, como ciertos mecanismos de tolerancia a fallos, así como el porqué de su uso.

Más adelante, se detallará el entorno de trabajo usado para el desarrollo del trabajo, herramientas de desarrollo, versionado, lenguajes entre otras herramientas más orientadas a dispositivos comerciales.

Una vez explicados, tanto los conceptos básicos como las herramientas con las que se realizó el trabajo, pasaremos a concretar cómo se realizó el desarrolló del trabajo. Dividiendo esta parte en diversas fases, cada una de ellas fuertemente relacionadas con los tipos de redundancia que se estudiaron. Además se explicará de forma generalista las aproximaciones realizadas para cada uno de los *parser* desarrollados. Por último se añadirá también un pequeño comentario acerca del porqué del uso de *Tool Command Language (TCL)* durante el trabajo.

Explicado el flujo de trabajo realizado a lo largo de este proyecto, se pasará a hablar de la visión general de la biblioteca desarrollada, tanto del *parser* de C como el de Verilog, ambos desarrollado de manera similar. Dentro de cada uno explicaremos cual sería su funcionalidad detallada, así como si se precisaron de bibliotecas adicionales.

Una vez comprendida la funcionalidad de los *parsers*, se expondrán el experimento realizado usando un algoritmo de ejemplo y realizaremos una comparación del comportamiento inyectando fallos, utilizando una versión sin proteger (sin usar el *parser* desarrollado) y otra con la función protegida (usando el *parser*).

Una vez expuestos los resultados, mostraremos las conclusiones así como un apartado con los contratiempos y problemas que surgieron a lo largo del desarrollo.

Capítulo 2

Conceptos Previos

2.1. Dispositivos FPGA

Las FPGA son dispositivos que intentan aunar dos mundos diferentes, el *hardware* y el *software* utilizando características de los dos contextos. Por el lado del *hardware* ganaríamos esa mayor potencia de cómputo gracias a no depender de instrucciones *software* como nos pasa en un procesador de cómputo generalista. Adicionalmente esa no dependencia ayuda a no tener a todo el sistema ocupado con la ejecución de esa única instrucción, es decir, podemos crear diseños específicos donde el sistema desarrollado actúe de manera completamente paralela al no depender de este sistema de instrucciones por ciclo. Por el lado del *software* ganaríamos esa reconfigurabilidad donde, en términos de FPGA sería una reconfiguración de sus bloques lógicos, como los que pueden observarse en la figura 2.2, para que el sistema actúe de una manera diferente.

Es por ello que el uso de FPGAs cada vez es más abundante, debido a las características comentadas anteriormente. De hecho, en la actualidad son muy comunes en contextos de prototipado de sistemas digitales, procesado de señales, *machine learning* como en contextos de control simple de comunicaciones en el sector aeroespacial, gracias a su arquitectura masivamente paralela con la cual el sistema puede actuar en el mismo instante de tiempo para dos tareas completamente diferentes, cosa que no es posible hacer un procesador de cómputo generalista.

En cuanto a la arquitectura interna, cabe recordar que cualquiér cómputo puede representarse como una ecuación booleana. Además estas ecuaciones pueden a su vez representarse como una tabla de verdad, por lo tanto, es posible usar tablas de verdad para expresar los cómputos a realizar. Las FPGAs explotan este concepto y usan estas tablas de verdad comentadas anteriormente representándolas como *lookup tables (LUT)* las cuales se implementan con un multiplexor N:1 con una memoria de N-bits.

En la figura 2.1 puede observarse como sería la representación de una LUT diseñada, donde podríamos usando los 3 bits de entrada elegir cualquiera de las

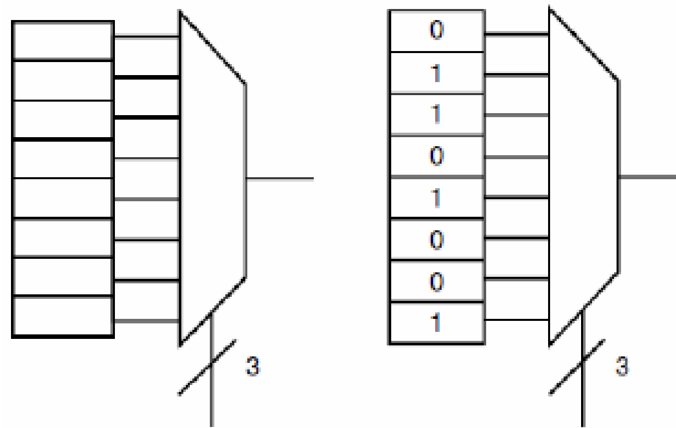

Figura 2.1: Ejemplo 3-LUT

filas de entrada al multiplexor. Las cuales no serían más que el resultado de una tabla de verdad.

Cabe añadir que las dimensiones de estas LUTs no son fijas y cada fabricante de FPGAs puede usar dimensiones diferentes, dado que un aumento/disminución del tamaño de estas LUTs puede afectar al uso de estos recursos.

Los LUTs comentados anteriormente no tienen memoria por lo que se precisa de algún elemento adicional para mantener su estado y por tanto tener esa pequeña persistencia que necesita el sistema. Por esta razón se añade al LUT un flip-flop de tipo D, estos dos elementos ya unidos formarían lo que se llama un bloque lógico de la FPGA, el núcleo de cómputo del dispositivo. Así pues, la lógica configurable del dispositivo está formada por una matriz 2D de bloques lógicos.

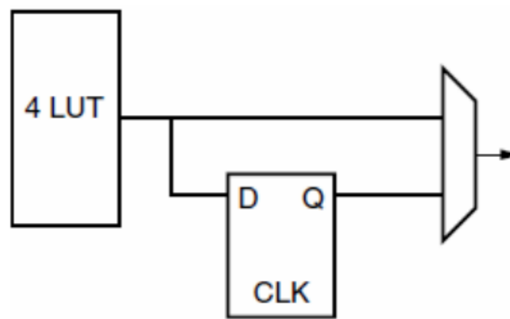

Figura 2.2: *Logic Block*

Por último, cabe destacar que estos bloques lógicos necesitan conectarse unos a otros, y éstos se encuentran rodeados por canales de encaminamiento

prefabricados, lo que conforma la arquitectura de encaminamiento. Esta cuenta con distintos elementos que pueden observarse en la figura 2.3:

- Segmentos de cableados de distintas longitudes que recorren el dispositivo vertical y horizontalmente.
- Bloques de conexión: Permiten conectar los pines de E/S de los bloques lógicos con los segmentos de conexión verticales/horizontales.
- Bloques de encaminamiento: se encuentran en las intersecciones de los canales vertifcales y horizontales y realizan el encamientamiento entre ellos (permiten cambiar de dirección).

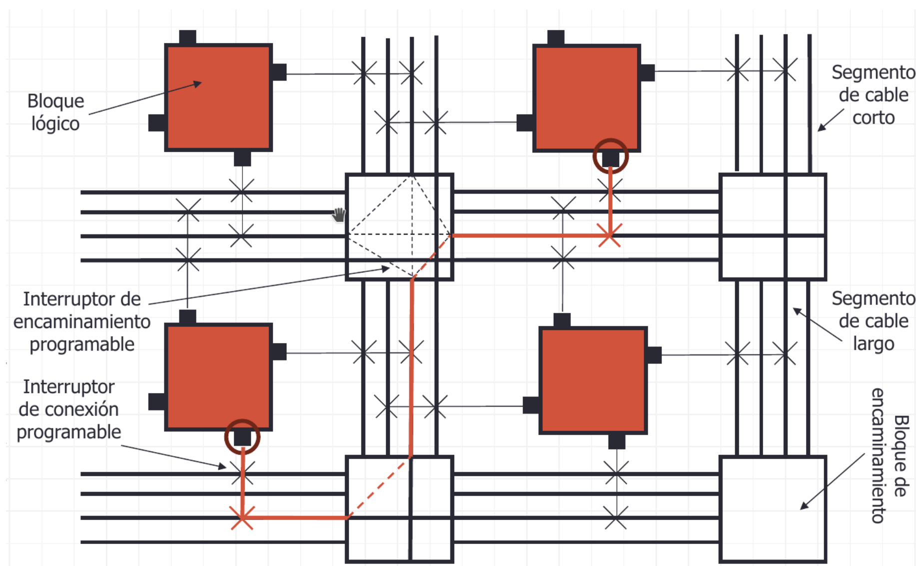

Figura 2.3: *Arquitectura interna de una FPGA*

Adicionalmente, también podemos encontrar otros elementos, como bloques de memoria, gestores de reloj digital e incluso núcleos de procesadores (típicamente ARM).

2.2. ***High Level Synthesis*** (HLS)

Es el proceso por el cual podemos obtener el diseño digital basado en el comportamiento descrito utilizando un lenguaje con un nivel de abstracción superior a Verilog o VHDL, en este caso C. HLS ayuda por tanto al diseñador a especificar el comportamiento que deberá tener el sistema que pretende implementar a nivel de registros (RTL) sin necesidad de tener que cambiar el lenguaje para

la especificación de su circuito, intentando eliminar la barrera que en numerosas ocasiones tiene el diseño de este tipo de sistemas, el cambio a lenguajes de descripción de *hardware* (HDL).

En el trabajo se usó el HLS de Xilinx, donde el fabricante además de proporcionar un entorno donde sintetizar nuestros diseños desde C hasta Verilog/VHDL, podremos añadir diversos *pragmas* con los cuales alteraremos el comportamiento de esa síntesis ayudando al programador a que tener más control sobre el proceso de síntesis, especialmente en las estructuras de control como pueden ser los bucles *for*, elementos muy importantes en cualquier diseño de este tipo.

2.3. *Single Event Effects* (SEE)

En la actualidad el uso de dispositivos FPGA basados en tecnologías SRAM es cada vez más habitual. Es por ello, que este tipo de dispositivos están expuestos a diversas problemáticas, especialmente a los *Sigle Event Effects*, que pueden ocurrir cuando usamos estos dispositivos en el ámbito aeroespacial, causados por el impacto de una o más partículas con energía en alguna de las celdas de la memoria de configuración del dispositivo. Al impactar y cargar la celda pueden ocasionar la variación del valor digital leído de la celda y especificamente en contextos FPGA, hacer que el diseño implementado en nuestro dispositivo actúe de manera diferente a la esperada.

En este trabajo se centrará el esfuerzo en la mitigación de los *Single Event Upsets (SEU)*, que ocasionan el cambio de estado de componentes tanto digitales, como analógicos u opticos. Centrándonos en el ámbito del trabajo, implicaría cambiar el valor de un bit de manera que este no reflejara el estado correcto que debería tener sino uno diferente pudiendo afectar de esta manera al comportamiento general del sistema. Los SEUs se consideran *soft* errors y pueden llegar a solucionarse realizando un *reset* del sistema o simplemente reescribiendo el bit que ha sido afectado por el evento. Cabe destacar que la reprogramación de FPGAs no es un proceso simple y rápido y por tanto la reprogramación del dispositivo una vez está en operación no es una actividad que se realice habitualmente, debido a los problemas que podrían derivar de este proceso.

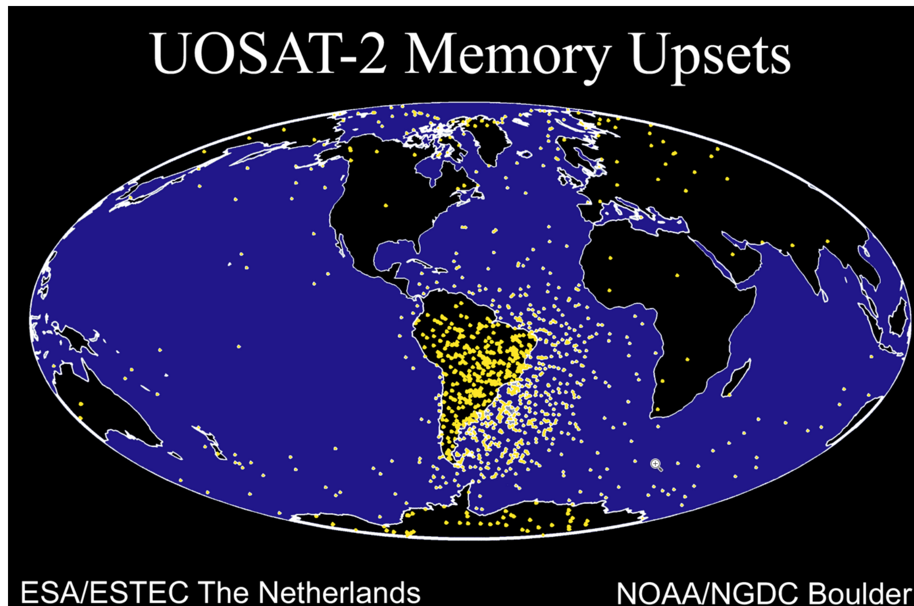

Figura 2.4: SEUs capturados en la memoria de un satélite.

Como puede observarse en la figura 2.4, en la que se indican los SEUs que tuvo un satélite mientras orbitaba, vemos que estos eventos no pueden considerarse despreciables, especialmente en la zona de la Anomalía del Atlántico Sur, y es por ello que se precisa implementar mecanismos que nos protegan ante este tipo de problemática. En el caso del trabajo desarrollado usaremos mecanismos de tolerancia a fallos que enmascaren los posibles fallos que podrían ocurrir y por tanto que no deriven en un error del sistema.

Para más información acerca de los SEE puede revisarse [4], donde en el apartado 3.2.3 hay información más detallada acerca de este tipo de eventos.

Además, en [6], pueden encontrarse datos relacionados con anomalías que ciertos satélites han ido detectando a lo largo de su vida útil.

2.4. Mecanismos de tolerancia a fallos

2.4.1. Redundancia Espacial

Este tipo de redundancia *hardware* es pasiva, tipo de redundancia que enmascara los fallos y de esta manera no necesita de ninguna acción externa por parte del usuario y/o administrador del sistema. Aplicándola al diseño de cualquier sistema digital realizaría una replicación del componente y/o función que queremos proteger ante fallos, además de generar un un votador al final del bloque protegido para que el valor resultante más votado sea tomado como correcto. De esta manera, si cualquiera de los módulos replicados fallara, no propagaría

el fallo y por tanto no ocasionaría un error en el sistema.

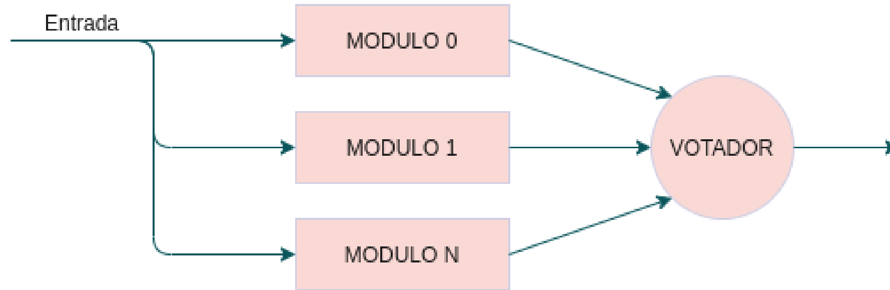

Figura 2.5: Diagrama NMR

En la figura 2.5 se puede observar cual sería la implementación básica del sistema comentado, donde varios módulos en paralelo realizarán la misma acción y su resultado pasaría posteriormente a un votador donde se decidiría cual es la opción más votada. Cabría remarcar que el mínimo número de réplicas necesarias para que la tolerancia a fallos sea efectiva serían 3 réplicas debido a que el uso únicamente dos obligaría al sistema a tener que escoger una sin poder determinal cuál es la correcta.

2.4.2. Redundancia Temporal

La redundancia tempora es una técnica que puede ayudarnos tanto a diferenciar entre fallos permanentes y transitorios así como enmascarar los transitorios. Consiste en relanzamiento de un proceso, programa o similar varias veces. Puede observarse el diagrama básico de un sistema implementado este tipo de redundancia en la figura 2.6.

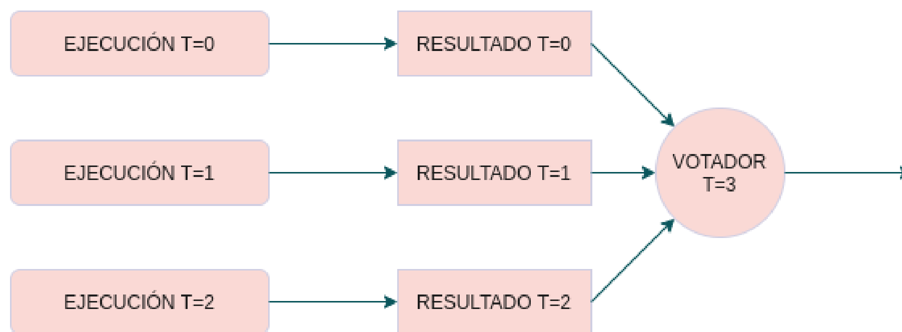

Figura 2.6: Redundancia Temporal

Al realizarse varias ejecuciones separadas en el tiempo podemos enmascarar posibles fallos transitorios dado que el sistema ejecutará multiples veces el

mismo proceso guardándose el resultado y votando posteriormente, De esta manera el resultado que tiene más indice de aparición será tomado como correcto. En la figura 2.7 puede observarse como reaccionaria el sistema si en la primera de tres ejecuciones el sistema obtuviera un resultado erróneo debido a un fallo transitorio. Como se ha comentado anteriormente gracias a la redundancia temporal ese fallo quedaría registrado, pero como posteriormente dos ejecuciones obtendrían un resultado correcto, el votador elegiría esta pareja de resultados como los resultados finales y por tanto el fallo en t=0 no se propagaría por el sistema final.

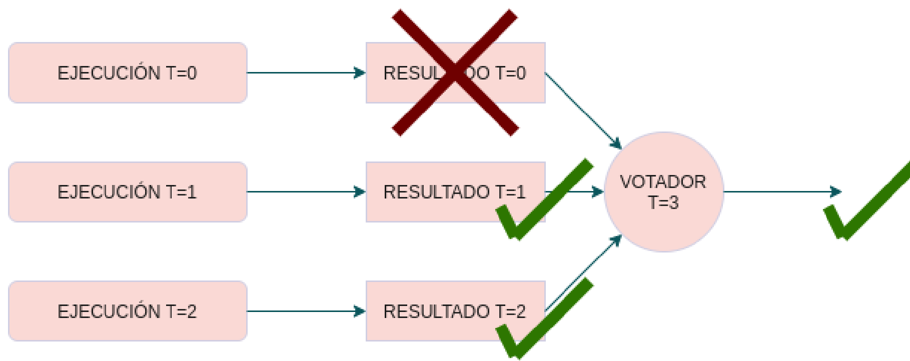

Figura 2.7: Redundancia Temporal - Ejemplo de fallo transitorio

Capítulo 3

Entorno de trabajo

Se presentarán en esta sección las herramientas utilizadas para la realización de este trabajo.

3.1. *Visual Studio Code*

Para la realización de todo el desarrollo de código se ha utilizado Visual Studio Code, herramienta que desde su lanzamiento ha ido ganando popularidad entre los desarrolladores. Se escogió este editor al inicio del trabajo debido al gran abanico de sintaxis de lenguajes que abarca, centralizando de esta manera la gran mayoría de tareas en un único programa.

Además gracias a su licencia abierta, gran cantidad de desarrolladores generan nuevas extensiones que pueden ser usadas, en su gran mayoría sin coste adicional. Durante el proceso de generación del código necesario, se utilizaron las siguientes extensiones para Visual Studio:

- Python: ms-python.python.
- C: ms-vscode.cpptools.
- Verilog: teros-technology.teroshdl
- TCL: sleutho.tcl

Además de una interfaz sencilla, eliminando los númerosos botones y/o menús de muchos de los IDEs habituales ayudando al programador a poder disponer de mayor porcentaje de código en pantalla. Puede observarse parte de su interfaz en la figura 3.1

3.2. Git

En cuanto al control de versiones, se utilizó la heramienta Git. Este *software* posibilita la gestión sencilla del versionado del código, pudiendo gestionarlo

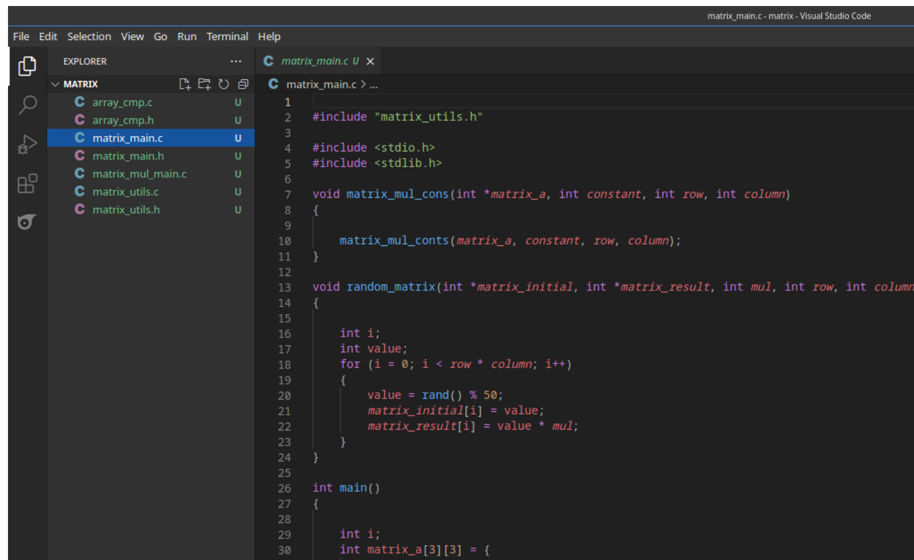

Figura 3.1: Visual Code

mediante ramas, etiquetas entre otras múltiples opciones. Cabe remarcar que como servidor de esta herramienta se utilizó github donde existe un repositorio, privado durante el desarrollo del trabajo, con el código de los dos *parsers* desarrollados.

3.3. Vitis HLS

Para poder hacer uso de la herramienta HLS, necesitaremos utilizar el IDE proporcionado por Xilinx.

Este entorno adaptado desde el famoso IDE Eclipse proporciona al desarrollador un entorno familiar donde realizar todas las tareas necesarias, lanzando incluso cuando sea preciso otras herramientas del propio fabricante con el objetivo de seguir el flujo de trabajo, desde la programación en C, hasta la programación de la FPGA con el diseño hardware generado, pasando a su vez por etapas de simulación y síntesis del diseño.

El flujo de trabajo principal sería el siguiente:

1. Desarrollo código C.

2. Simulación del código C desarrollado.

3. Síntesis del código C, generando código VHDL y Verilog que contendrá la lógica necesaria.

4. Co-simulación del *software* donde podremos comprobar en una primera aproximación utilizando el *testbench* C ya creado, que el código HDL sintetizado funciona según lo previsto. En esta última parte se hará uso de otra herramienta de Xilinx, llamada Xilinx Vivado.

5. Exportar el diseño como IP Core para poderlo utilizar en Vivado para la implementación de otros componentes/sistemas en FPGAs del fabricante.

Para más información acerca de esta herramienta, en la página web del fabricante [7] existen multitud de guías donde se exponen sus características y usos comunes.

3.4. Vivado

Siguiendo con la misma temática que para Vitis HLS, Vivado es el IDE desarrollado por Xilinx que ayudará al diseñador de la lógica a realizar todo el proceso, desde el diseño de código HDL hasta la propia programación de la FPGA. En su guía de uso [8], podremos encontrar información más detallada acerca de su uso.

En cuanto al flujo de trabajo de la aplicación, vendría a resumirse de la siguiente manera:

1. Desarrollo del código HDL.

2. Simulación del código HDL desarrollado.

3. Síntesis del diseño HDL.

4. Simulación del diseño sintetizado.

5. Implementación del diseño generado.

6. Simulación del diseño implementado.

7. Programación de la FPGA.

Capítulo 4

Proceso de desarrollo

El proceso de desarrollo de las herramientas precisó de varias fases. Se necesitó de esta división en varios pasos para acotar de manera plausible los tiempos respecto al desarrolló del trabajo.

De manera genérica agruparemos las fases del desarrollo en las siguientes:

1. Estudio HLS, características y limitaciones.
2. Pruebas conceptuales
3. Desarrollo *parser*
4. Experimento algoritmo

4.1. Revisión conceptual HLS

De cara a poder proteger las funciones utilizando las librerías desarrolladas, necesitamos primero entender los tipos de protecciones que podemos ofrecer al usuario mediante las herramientas que nos proporciona el propio vendedor, en este caso Xilinx.

Disponíamos de varias rutas a la hora de realizar la protección de diseños basados en HLS, directivas de compilación utilizando el compilador de Vitis HLS o bien mediante el uso de *pragmas* añadidos al código del propio usuario. Dada la mayor facilidad de uso y sobretodo de cara a poder tratar el código HLS proporcionado por el usuario, se optó por la vía de la adición de *pragmas*, debido a que el uso de directivas de compilación eran más generalistas en cuanto a la afección de todo el proyecto y dado que nuestro objetivo identificar una función en concreto y proteger únicamente la que el usuario nos indique, optamos por la opción de los *pragmas*, gracias a que uso está más enfocado a aplicarse a ciertas partes concretas del código.

Durente el estudio de las características también se revisaron las limitaciones que Xilinx indica referentes al uso de memoria dinámica, dado que el código C

acabará teniendo que ser estático debido a la naturaleza de la FPGA entre otro tipo de peculiaridades.

4.2. Pruebas protección herramientas HLS

Para realizar las protecciones sobre las funciones que se nos indique haremos uso de pragmas como se indicó anteriormente, pero deberemos hacer uso de pragmas diferentes según el tipo de redundancia que queramos implementar.

4.2.1. Redundancia espacial

Para lograr la redundancia espacial y siguiendo con lo comentado en el apartado 2.4, en un primer momento se optó por realizar pruebas con la opción más directa, es decir, realizar una llamada tres veces a la función suma, almacenar su resultado y posteriormente realizar la votación, utilizando una función votador. El código base para las pruebas se muestra en el *listing* 4.1.

```
#include "suma.h"
#include "voter.h"

int NMR_suma(int a, int b){
    int replica0;
    int replica1;
    int replica2;

    int voted_result;

    replica0 = suma(a,b);
    replica1 = suma(a,b);
    replica2 = suma(a,b);

    voted_result = voter(replica0,replica1,replica2);
    return voted_result;
}
```

Listing 4.1: Código básico redundancia espacial

Una vez preparado el diseño utilizamos la simulación en C para comprobar que comportamiento de la función era el esperado, para ello diseñamos un test sencillo:

```
int main(){

    int suma;

    suma = 0;
    suma = NMR_suma(100,2);

    printf("VOTED_RESULT: %d \n", suma);

    if(suma==102){
        printf("VOTED_RESULT: OK \n");
        return 0;
    } else {
```

```
14          printf("VOTED_RESULT: FAILED \n");
15          return 1;
16      }
17  }
```

Listing 4.2: Código básico test

Y como puede observarse en la salida obtenida en el *listing* 4.2, el codigo C funciona según lo previsto, así pues pasaremos a comprobar que el sintetizado del mismo acaba implementado varias instancias del mismo módulo, además de su votador y por tanto generando sistema tolerante a fallos.

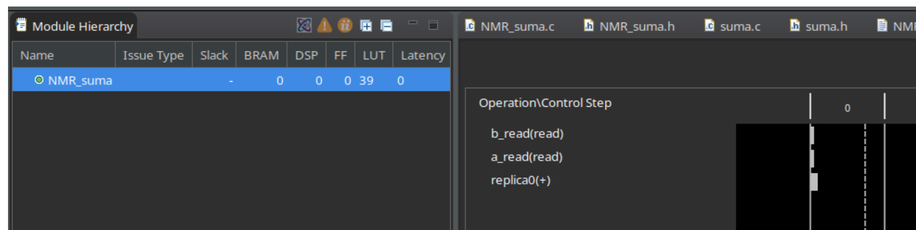

Figura 4.1: *Scheduler* del código "suma"procesado con HLS

En la parte izquierda de la figura 4.1 podemos observar como después del procesado con HLS sólo se ha obtenido un módulo, es decir, HLS ha considerado que el módulo era optimizable en cuanto a área y ha eliminado nuestras funciones replicadas y el votador, uniendo todo el código en un único módulo HDL.

Es por ello que para lograr la redundancia espacial necesaria eliminando el problema comentado anteriormente, se procedió a utilizar el *pragma: #pragma HLS inline off*. Al añadir esta sentencia al código, HLS separa toda la lógica de la función a la que acompaña, generando de esta manera códigos sintetizables separados del resto de la lógica del diseño. Es decir, forzando a que cada función tenga su propia equivalencia en Verilog/VHDL.

En el *listing* 4.3 podemos observar el código de ejemplo usado:

```
1  int suma (int a, int b){
2  #pragma HLS inline off
3      return a+b;
4  }
```

Listing 4.3: Adición *pragma* en función a proteger

Una vez añadido este nuevo *pragma* volvimos a comprobar con el test en C que el resultado de la función era también correcta y además relanzamos el proceso de HLS, para posteriormente revisar en el *scheduler* si el sistema había generado las tres réplicas que buscamos.

Si bien es cierto que para esta prueba, puede comprobarse en la figura 4.2, HLS ha separado la función suma y el votador, al cual también le hemos añadido el *pragma* mencionado, no ha implementado réplicas de la misma. Para HLS las funciones con igual nombre a las cuales le pasas los mismos *inputs* son funciones claramente optimizables, ¿para qué realizar tres veces el mismo proceso que

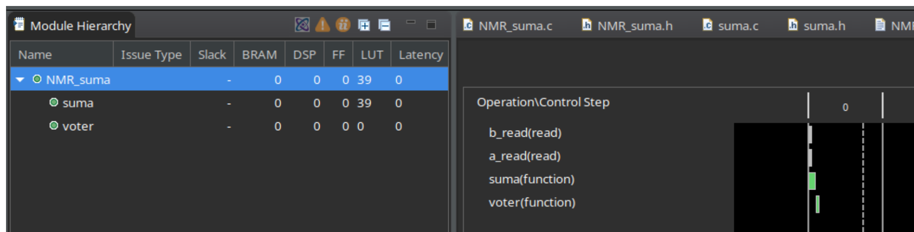

Figura 4.2: *Scheduler* del código "suma"procesado con HLS con *pragma inline off* añadido

además va a obtener el mismo resultado? Llegados a este punto es conveniente indicar que para lograr el objetivo que perseguimos es necesario ir en contra de las directrices que el propio HLS tiene.

Dicho esto, y adicional al *pragma* añadido, se separó la función suma en tres funciones *wrapper* que con nombre diferente acabarán llamando a la función suma inicial, de esta manera, intentamos 'engañar' a HLS para que piense que a pesar de tener todas las réplicas los mismos parámetros de función el resultado puede ser distinto ya que a efectos de nombre, son funciones claramente diferentes. En resumen, dispondremos ahora de tres réplicas con sus respectivos *pragma inline off* que tendrán una llamada a la función suma original. Las nuevas funciones suma que harán a su vez de *wrapper* de la función suma original tendrán la forma expuesta en el *listing* 4.4.

```
1  #include "suma.h"
2  int suma_0 (int a, int b){
3  #pragma HLS inline off
4      return suma(a,b);
5  }
6
7  #include "suma.h"
8  int suma_1 (int a, int b){
9  #pragma HLS inline off
10     return suma(a,b);
11 }
12
13 #include "suma.h"
14 int suma_2 (int a, int b){
15 #pragma HLS inline off
16     return suma(a,b);
17 }
```

Listing 4.4: Separación de función a proteger en diversas funciones con nombres diferenciados

Además también deberemos adaptar la función NMR_suma para que utilice estas nuevas funciones como se muestra en el *listing* 4.5.

```
1 #include "suma_0.h"
2 #include "suma_1.h"
3 #include "suma_2.h"
```

```
4 #include "voter.h"
5
6 int NMR_suma(int a, int b){
7     int replica0;
8     int replica1;
9     int replica2;
10
11     int voted_result;
12
13     replica0 = suma_0(a,b);
14     replica1 = suma_1(a,b);
15     replica2 = suma_2(a,b);
16
17     voted_result = voter(replica0,replica1,replica2);
18     return voted_result;
19 }
```

Listing 4.5: Adición *pragma* en función a proteger

Como en las anteriores pruebas, se volvió a realizar la comprobación del correcto funcionamiento de la función lanzando primero una simulación en C. La cual, como cabe esperar finalizó correctamente obteniendo un resultado correcto. Posteriormente se procedió a lanzar el proceso de HLS y el resultado fue el que refleja la figura 4.3.

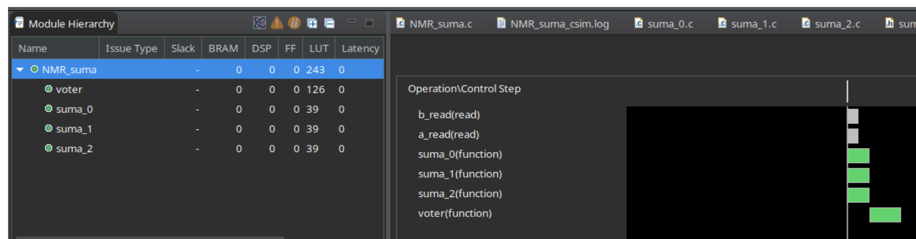

Figura 4.3: *Scheduler* del código ”sumaütilizando *pramgas* y nombres de función diferentes

Podemos observar en la parte izquierda de la imagen como HLS a creado un votador y varias réplicas de la función a proteger, además, se puede comprobar en las barras de la parte izquierda de la pantalla, como esas tres réplicas se ejecutan de manera paralela para después, una vez obtenido su resultado ejecutar la votación. Por tanto es esta la técnica correcta para lograr implementar rendundancia espacila en códigos C usando HLS, recordamos, creación de funciones *wrapper* con nombres diferentes y además cada una de estas nuevas funciones debe contener el pragma HLS inline off.

Este método será por tanto el que se usará como base a la hora de la creación de las bibliotecas necesarias que automaticen el proceso mencionado y eviten la necesidad de realizar estos cambios manualmente. La automatización de este proceso estará contenido en un fichero Python llamado NMR_C_Parser.py.

Como prueba adicional se lanzó la co-simulación desde Vitis HLS para comprobar ya a nivel de Verilog que el sistema obtenido tiene el comportamiento

deseado. En la figura 4.4 podremos observar dos cosas, la primera, revisando la parte izquierda de la imagen, como HLS ha generado un módulo Verilog por réplica, además del votador y por otro lado, en la forma de onda veremos que el resultado de la operación es 102, es decir, el resultado correcto.

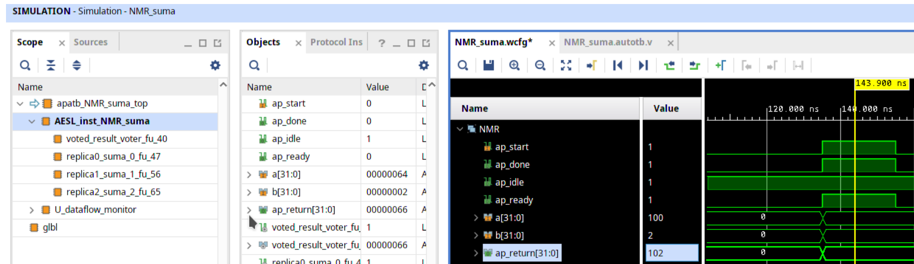

Figura 4.4: Lanzamiento Co-Simulación NMR_Suma

Una vez comprobado que las réplicas se mantienen después del proceso de HLS, se verificó que al injectar errores en alguna de las mismas el resultado sigue siendo correcto, dado que el votador seguirá eligiendo el resultado mayoritario como resultado correcto.

Se puede observar en la figura 4.5 como a pesar de que el resultado de la primera réplica (result0[31:0]) es incorrecto, las dos réplicas restantes coinciden en su resultado y por tanto se vota ese resultado como correcto. Es decir, podemos confirmar que los módulos Verilog generados a través de HLS utilizando las técnicas indicadas anteriormente funcionan como deberían.

Para finalizar el proceso de *parseo* nos faltaría comprobar que Vivado no elimina las réplicas una vez realizamos la síntesis del diseño. Pero como era previsible, si no realizamos modificaciones en el código generado desde HLS, de igual manera que Vitis intentaba realizar optimizaciones, Vivado elimina las réplicas en aras de la optimización en área del diseño. Se puede observar este comportamiento en la figura 4.6

Para evitar el comportamiento descrito, de entre todos los atributos que es posible añadir al código en Verilog se buscó uno que tuviera un comportamiento similar al *pragma inline off* utilizado en C. Así pues se tomó como candidato a *(DONT TOUCH = yes)* el cual se añadió tanto a las réplicas como al votador, tal y como se hizo en el código C.

Este atributo debe añadirse antes de la definición del módulo para que el sintetizador no realice optimizaciones en el código del mismo como puede verse en el *listing* 4.6.

```
1 (* DONT_TOUCH = "yes" *)
2 module NMR_suma_suma_1 (
3         ap_ready,
4         a,
5         b,
6         ap_return);
```

Listing 4.6: Adición *pragma* en función a proteger

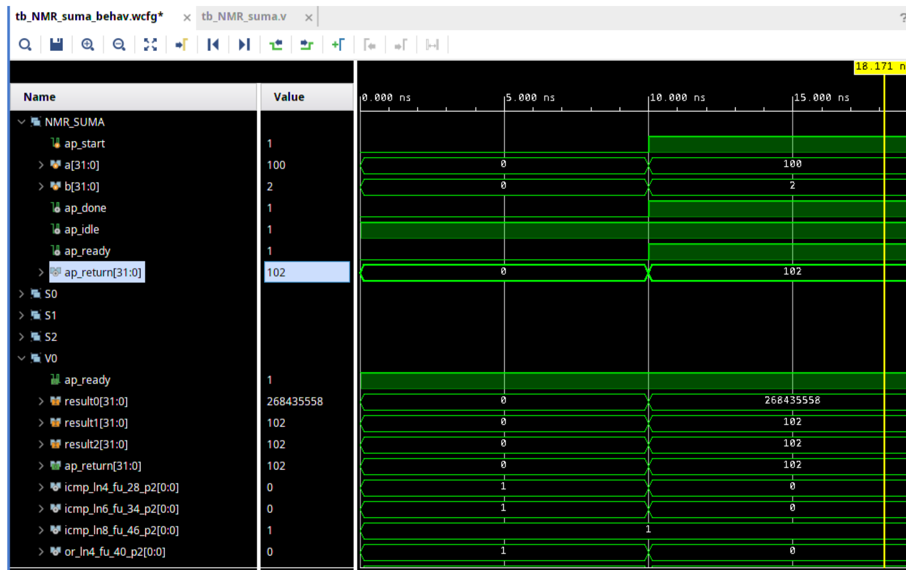

Figura 4.5: Simulación Vivado - Resultado correcto después de inyectar un SEU en la primera réplica.

Posteriormente a la adición de este atributo en todas las réplicas se volvió a lanzar el proceso de sintetizado obteniendo un resultado satisfactorio. Puede observarse en 4.7 como ahora aparecen tanto las tres réplicas como el votador como módulos independientes y no optimizados.

Después de todo el proceso indicado podemos afirmar que existe una manera de añadir redundancia espacial a los diseños en C procesados con HLS. En base a este conocimiento adquirido se desarrollarón dos *parsers* que automatizan parte de este proceso, los cuales serán explicados en posteriores capítulos de este documento.

4.2.2. Redundancia temporal

De cara a hacer una primera aproximación de los *pragmas* que podrían usarse para implementar mecanismos en HLS que proporcionen las herramientas necesarias para proteger ciertas funciones utilizando redundancia temporal, se realizaron diversas pruebas, basándonos en la experiencia obtenida durante el desarrollo de los mecanismos de redundancia espacial.

Las pruebas realizadas fueron las siguientes:

1. *Pragma Dataflow*
2. *While lock*

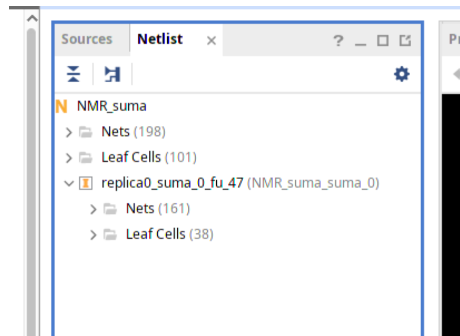

Figura 4.6: Módulos después de sintetizado

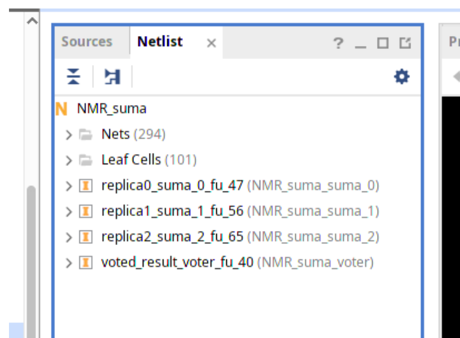

Figura 4.7: Módulos después de sintetizado con atributo añadido

Pragma Dataflow

Haciendo uso del *pragma dataflow* intentamos que el módulo a proteger fuera reusado varias veces, es decir, se hiciera el cómputo del resultado de manera iterativa comprobando que los resultados son idénticos, es decir, se pretende que siga la metodología de redundancia temporal. Por ello se hicieron pruebas con el *pragma dataflow* el cual genera una cadena de modulos encadenados los cuales le pasan datos de uno a otro como el propio nombre en inglés sugiere.

La idea del uso de este *pragma* era forzar a HLS a crear un flujo de datos sobre el módulo en cuestión y que éste volviera a realizar su proceso tantas veces como quisiéramos.

En la figura 4.8 se puede observar que el compilador de HLS, intenta optimizar, en cuanto a ejecución paralela se refiere, ejecutando simultáneamente dos de las tres réplicas, por lo que no obtuvimos un resultado positivo.

While lock

Otra de las aproximaciones que se barajaron a la hora de implementar la redundancia temporal fue intentar replicar comportamientos puramente *software*

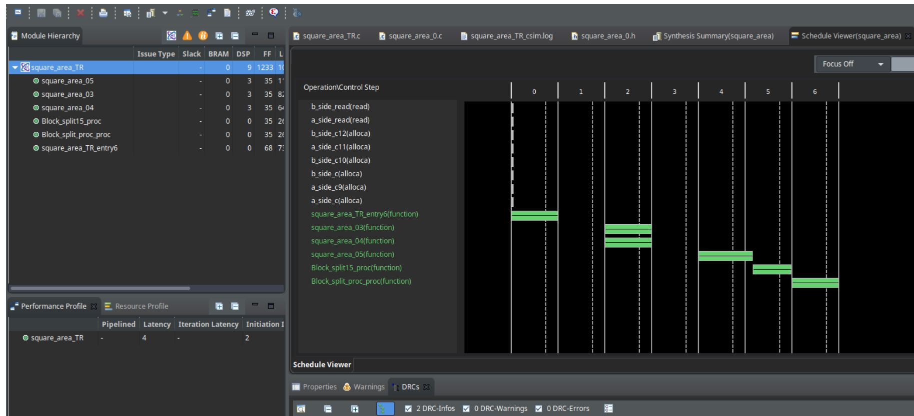

Figura 4.8: Redundancia Temporal *Pragma Dataflow*

en la FPGA. Utilizando para este caso una espera activa, intentando de esta manera solventar el problema indicado en la sección 4.2.2, el objetivo era forzar a HLS a no paralelizar las ejecuciones.

Haciendo que el núcleo del programa, tuviera la estructura mostrada en el *listing* 4.7:

```
int square_area_TR(int a_side, int b_side) {

  int result0;
  int result1;
  int result2;
  int final_result;

  int lock0, lock1, lock2, lock3;

  lock0 = 0;
  lock1 = 1;
  lock2 = 1;
  lock3 = 1;

      square_area_0(a_side, b_side, &result0, &lock0, &lock1);
      square_area_0(a_side, b_side, &result1, &lock1, &lock2);
      square_area_0(a_side, b_side, &result2, &lock2, &lock3);
      voter_TR(&result0, &result1, &result2, &final_result, &lock3)
    ;

  return final_result;

}
```

Listing 4.7: Salida *parser* NMR_C_Parser

Por desgracia como puede comprobar en la figura 4.9 HLS en aras de optimizar el consumo en ciclos del sistema, optimiza nuestro código dado que entiende que la operación resultante será la misma realicemos una ejecución o miles, evitando de esta manera poder desplegar mecanismos de redundancia temporal

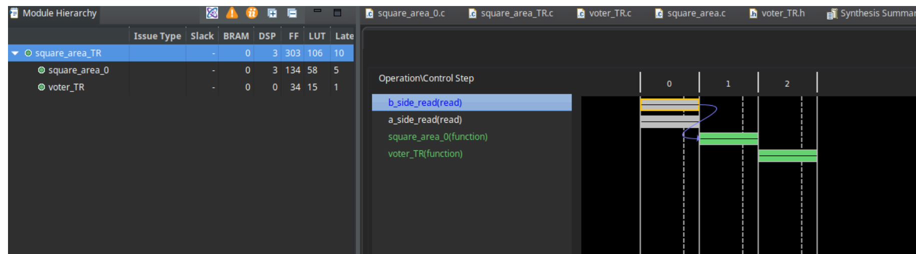

Figura 4.9: Redundancia Temporal *While lock*

utilizando la técnica de la espera activa.

——————————————-

4.3. Desarrollo injección fallos usando TCL

Cabe añadir que, para comprobar la correcta implementación, además del habitual *testbench* en Verilog, se crearon pequeños *scripts* para cada caso de prueba, para que, sin alterar el diseño previamente compilado desde Vitis_HLS y sintetizado desde Vivado, podamos simular el impacto de partículas que generen SEEs, de manera similar a lo que ocurriría en un entorno espacial real. Cada *script* TCL dispondrá de una estructura básica que irá replicándose y adaptándose en cada una de las pruebas realizadas. Para ello, primero tendremos una lista con las señales que queremos alterar, simulando ese impacto de partículas con carga, y después varios bucles donde según unos parámetros definidos, harán que aleatoriamente cambien bits de las señales elegidas.

Un ejemplo básico sería el mostrado en el *listing* 4.8, donde alteramos el valor de una señal después de un tiempo determinado para así poder observar en la forma de onda como se comporta el sistema en base a esta alteración.

```
1 run 10 ns
2 add_force {/tb_NMR_suma/suma/replica0_suma_0_fu_47/ap_return[28]}
      -radix hex {1 0ns}
3 run 5 ns
4 run 5 ns
5 removes_force {/tb_NMR_suma/suma/replica0_suma_0_fu_47/ap_return
      [28]}
```

Listing 4.8: Salida *parser* NMR_C_Parser

El anterior *script* está pensado para la función de ejemplo suma expuesta en capítulos anteriores del documento, es decir, estaríamos hablando de una inyección sencilla a modo de ejemplificación. En diseños más elaborados es posible utilizar un mayor abanico de las funcionalidades que nos ofrece TCL y así generar inyecciones de errores más complejas para análisis más avanzados de los diseños generados. Puede observarse a continuación un *script* TCL (*listing* 4.9) en el cual podemos definir un porcentaje de bits expuestos a inyección de errores y además definir qué probabilidad existe de que se inyecte un error dentro de

los bits expuestos, de esta manera podríamos realizar un estudio más minucioso del comportamiento ante posibles errores de nuestro diseño.

```
restart

puts "Running simulation..."
#Replicas are computing at 285ns

########################
## METRICS
########################
set failures 0
set experiments 0
set total_experiments 1000

# create a list with all the target signals

# Not used -> TO DO: Should be used... :)
set lst {
/tb_NMR_matrix_mul_conts_3_3/DUT/grp_matrix_mul_conts_3_3_0_fu_146/
    matrix_a_d0
/tb_NMR_matrix_mul_conts_3_3/DUT/grp_matrix_mul_conts_3_3_1_fu_152/
    matrix_a_d0
/tb_NMR_matrix_mul_conts_3_3/DUT/grp_matrix_mul_conts_3_3_2_fu_158/
    matrix_a_d0
}

########################
## FAILURE PARAMETERS
########################

#% of the total register area which could be damaged
set area_allowed 0.2
#% of failure
set prob_failure [expr 1+0.1]
#Total bits allowed to fail...
set total_bits 96
#Bits allowed to fail...
set allowed_fail_bits [expr $total_bits*$area_allowed]


puts "Running simulation..."

########################
## SIMULATION
########################

# force each target signal
for {set experiments 0} {$experiments < $total_experiments} {incr
    experiments 1} {

                puts "Experiment/Total_Experiments: $experiments/
    $total_experiments"
                run 285ns
                set time [current_time]
                puts "Current time: $time"

                set length_signal_0 [llength [get_objects /
```

```
    tb_NMR_matrix_mul_conts_3_3/DUT/
    grp_matrix_mul_conts_3_3_0_fu_146/matrix_a_d0[*]]]
51              set length_signal_1 [llength [get_objects /
    tb_NMR_matrix_mul_conts_3_3/DUT/
    grp_matrix_mul_conts_3_3_1_fu_152/matrix_a_d0[*]]]
52              set length_signal_2 [llength [get_objects /
    tb_NMR_matrix_mul_conts_3_3/DUT/
    grp_matrix_mul_conts_3_3_2_fu_158/matrix_a_d0[*]]]
53
54              set failed_bits 0
55              # These 'fors' generate random failures in the
    result register of each replica. Check the failure parameters
    to more details
56
57              ##Forcing or not, each bit from the two replicas
58              for {set bit_signal_0 0} {($bit_signal_0 <
    $length_signal_0) && ($failed_bits<$allowed_fail_bits)} {incr
    bit_signal_0 1} {
59                  set force_bit_0 [expr { int(rand()
    *$prob_failure)}]
60                  if {$force_bit_0 == 1} {
61                      add_force /tb_NMR_matrix_mul_conts_3_3/DUT/
    grp_matrix_mul_conts_3_3_0_fu_146/matrix_a_d0[$bit_signal_0]
    -radix hex $force_bit_0 0ns
62                      incr failed_bits 1
63                  }
64              }
65              for {set bit_signal_1 0} {($bit_signal_1 <
    $length_signal_1) && ($failed_bits<$allowed_fail_bits)} {incr
    bit_signal_1 1} {
66                  set force_bit_1 [expr { int(rand()
    *$prob_failure)}]
67                  if {$force_bit_1 == 1} {
68                      add_force /tb_NMR_matrix_mul_conts_3_3/DUT/
    grp_matrix_mul_conts_3_3_1_fu_152/matrix_a_d0[$bit_signal_1]
    -radix hex $force_bit_1 0ns
69                      incr failed_bits 1
70                  }
71              }
72              for {set bit_signal_2 0} {($bit_signal_2 <
    $length_signal_2) && ($failed_bits<$allowed_fail_bits)} {incr
    bit_signal_2 1} {
73                  set force_bit_2 [expr { int(rand()
    *$prob_failure)}]
74                  if {$force_bit_2 == 1} {
75                      add_force /tb_NMR_matrix_mul_conts_3_3/DUT/
    grp_matrix_mul_conts_3_3_2_fu_158/matrix_a_d0[$bit_signal_2]
    -radix hex $force_bit_2 0ns
76                      incr failed_bits 1
77                  }
78              }
79
80              run 10 ns
81
82              #PRINT WORD 0
83              set failure_word_0 [get_value /
    tb_NMR_matrix_mul_conts_3_3/DUT/
```

```
    grp_matrix_mul_conts_3_3_0_fu_146/matrix_a_d0]
84                 puts "Word_0: $failure_word_0"
85
86                 #PRINT WORD 1
87                 set failure_word_1 [get_value /
    tb_NMR_matrix_mul_conts_3_3/DUT/
    grp_matrix_mul_conts_3_3_1_fu_152/matrix_a_d0]
88                 puts "Word_1: $failure_word_1"
89
90                 #PRINT WORD 1
91                 set failure_word_2 [get_value /
    tb_NMR_matrix_mul_conts_3_3/DUT/
    grp_matrix_mul_conts_3_3_2_fu_158/matrix_a_d0]
92                 puts "Word_2: $failure_word_1"
93
94
95                 #for remove_forces de todos los bits
96                 for {set bit_signal_0 0} {$bit_signal_0 <
    $length_signal_0} {incr bit_signal_0 1} {
97                     remove_forces /tb_NMR_matrix_mul_conts_3_3/DUT/
    grp_matrix_mul_conts_3_3_0_fu_146/matrix_a_d0[$bit_signal_0]
98                 }
99                 for {set bit_signal_1 0} {$bit_signal_1 <
    $length_signal_1} {incr bit_signal_1 1} {
100                    remove_forces /tb_NMR_matrix_mul_conts_3_3/DUT/
    grp_matrix_mul_conts_3_3_1_fu_152/matrix_a_d0[$bit_signal_1]
101                }
102                for {set bit_signal_2 0} {$bit_signal_2 <
    $length_signal_2} {incr bit_signal_2 1} {
103                    remove_forces /tb_NMR_matrix_mul_conts_3_3/DUT/
    grp_matrix_mul_conts_3_3_2_fu_158/matrix_a_d0[$bit_signal_2]
104                }
105
106                #waiting to testbench result
107                run 500ns
108
109                set time [current_time]
110                puts "Current time: $time"
111
112                set test_res [get_value /
    tb_NMR_matrix_mul_conts_3_3/test_res]
113                puts "Experiment result: $test_res"
114                if {$test_res == 1} {
115                    incr failures 1
116                }
117
118                incr experiments 1
119                restart
120
121 }
122
123 puts "$area_allowed AREA EXPOSED TO ERROR INJECTION"
124 puts "RESULTS: failures/experiments"
125 puts "RESULTS: $failures/$experiments"
```

Listing 4.9: Salida *parser* NMR_C_Parser

Capítulo 5

Vision general de la biblioteca

La biblioteca generada permite la protección de funciones concretas que posteriormente se implementaran en un dispositivo FPGA haciendo uso de la tecnología HLS. Este conjunto de herramientas cuenta con varios ficheros, programas y funciones que trabajan conjuntamente para lograr el objetivo de añadir tolerancia a fallos a un sistema digital definido en C.

Los ficheros en cuestión son los siguientes:

- NMR_C_Parser.py: Contiene todo lo relativo al proceso de *parseado* del programa C inicial.
- NMR_Verilog_Parser.py: Dispone del código necesario para proteger el código Verilog resultante del proceso HLS.
- parser_util.py: Biblioteca con utilidades especilamente focalidadas en el análisis de texto plano usadas en ambos *parsers*.

En esta sección se detallará el proceso que será necesario seguir para partiendo de un código C, protegerlo utilizando técnicas de tolerancia a fallos y tansformarlo en un diseño Verilog que cuente tanto con toda la funcionalidad descrita en el código C así como de las técnicas introducidas por la herramienta.

Una vez detallado el proceso a seguir, se explicará el contenido de los dos *scripts* Python que conforman el núcleo de la biblioteca comentados anteriormente.

5.1. Descripción del proceso de protección

El proceso de transformación tiene dos partes claramente diferenciadas, donde en cada una de ellas tendremos un lenguaje con un sistema descrito el cual analizaremos con los *parsers* desarrollados y modificaremos para añadir las técnicas de tolerancia a fallos indicadas a lo largo de todo el documento.

Para ello cabe remarcar que los dos procesos vendrán determinados por el lenguaje utilizado en su fase inicial y por tanto su división será la siguiente:

- Código inicial C, donde utilizaremos NMR_C_Parser.py para realizar la protección del diseño.
- Código inicial Verilog, donde utilizaremos NMR_Verilog_Parser para realizar la protección del diseño generado automáticamente.

El primer proceso, relacionado con el *parser* de C es el encargado de proteger una función de un programa en C dado. Para este proceso usaremos el primer *script* generado, el cual, dándole como *input* el programa C, analizará y modificará dicho código para dar como resultado un código dónde la función elegida está N-plicada. Además, se añadirán ciertas funciones adicionales como puede ser un votador y una función que haga de controlador de la redundancia espacial añadida. Este código seguirá las reglas y usará *pragmas* aceptados por el proceso HLS. El detalle del comportamiento de las funciones añadidas se explicará en la sección 5.2.1.

Para una visión más ejemplificadora del proceso mencionado, la figura 5.1 expone el proceso como un conjunto de pasos que deben seguirse para llegar a obtener el código Verilog. Donde se puede observar que después de utilizar el *NMR_C_Parser* será necesario procesar el código con HLS para que como resultado un código Verilog que describa el comportamiento del código C inicial con las protecciones añadidas. Cabe remarcar que este nuevo código Verilog, a pesar de que ante cualquier simulación funcionaría según lo previsto, es decir, tiene los mecanismos de tolerancia a fallos añadidos, necesitaría de un segundo paso para proteger de optimizaciones de cara a la implementación en FPGA, esta problemática es la que genera la necesidad de un segundo proceso.

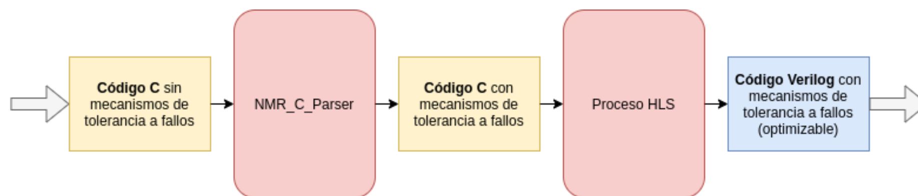

Figura 5.1: Proceso protección código C

Siguiendo con lo comentado en el párrafo anterior, si bien es cierto que el diseño generado por HLS contiene las funciones que implementan la redundancia espacial, si intentáramos implementar este diseño en una FPGA, el sintetizador de Vivado se daría cuenta de que estamos realizando tres veces la misma función en paralelo y optimizaría en área del diseño, es decir, eliminaría toda la redundancia que se ha añadido en el proceso anterior. Para ello, y siguiendo el proceso que se muestra en la figura 5.2, se procesará ese código Verilog generado desde Vitis HLS para añadirle abritutos que le indican al sintetizador de Vivado que no optimice los módulos redundantes creados.

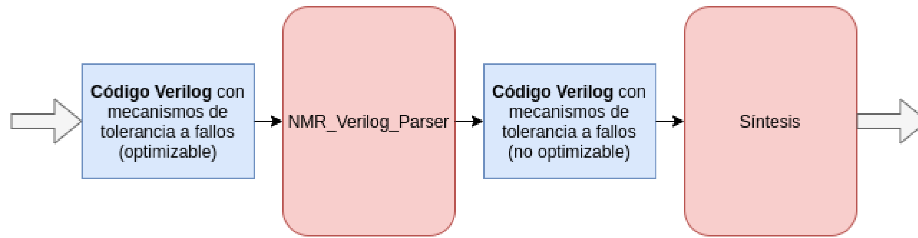

Figura 5.2: Proceso antioptimización Verilog

5.2. NMR_C_Parser

Como se ha detallado a lo largo de la sección 4.2.1 después de diversas pruebas se concluyo que sí es posible utilizar HLS para generar diseños desde código C aplicándoles redundancia espacial.

Por ello, una vez comprobado que conocemos de los métodos necesarios para conseguir esa redundancia, se pasó a intentar automatizar lo más posible este proceso y para ello, se creo un *parser* que realizará automaticamente las acciones necesarias para introducir en una función dada en código C, redundancia espacial. Código que después de ser procesado con HLS daránlugar a un código Verilog tolerante a fallos. En este caso, estaríamos hablando de generar varias réplicas de una función que a su vez contengan el *pragma inline off* además de una pequeña variación en su propio nombre para que HLS entienda que son funciones claramente diferenciadas y lo las optimice como ya se comentó en secciones anteriores.

En cuanto al desarrollo de este *parser*, cabe destacar el uso de *Python*, la elección de este lenguaje y no cualquier otro viene dada por su facilidad en el tratamiento de textos, especialmente con el uso de expresiones regulares. Además se debe remarcar también la pequeña curva de aprendizaje que necesita para poder utilizarse, valor positivo que ayuda a tener herramientas desarrolladas habitualmente más rápido que con cualquier otro lenguaje.

Es conveniente indicar que para el uso de este *parser* hay que tener presentes las restricciones que añade HLS a la programación en C, especialmente al uso de memoria dinámica. No será posible usar funciones *malloc* o similares, además de ser obligatorio definir las dimensiones de cualquier puntero a memoria para que HLS sea capaz de inferir el tamaño del registro que deberá crear cuando genere el código Verilog/VHDL. Para más información acerca de estas restricciones revísese [9];

A continuación podremos ver un ejemplo de uso:

```
1 python3 NMR_C_parser.py encrypt aes_enc.c aes_enc.h aes.c 3 1 int
    statemt 32
```

Listing 5.1: Ejemplo llamada *Parser*

En la sentencia anterior podemos ver los argumentos que es necesario pasar al *parser*, estos son: el nombre de la función que queremos proteger dentro del

programa (*encrypt*), su fichero .c asociado (*aes_enc.c*) así como su .h (*aes_enc.h*) donde se define la función, además también indicaremos el nombre del archivo que contiene el main (*aes.c*), la cantidad de réplicas que queremos que tenga el diseño resultante, así como varios argumentos adicionales referentes al paso de parámetros. Estos cuatro últimos parámetros harán referencia a si queremos usar la metodología simple del paso de parámetros de función, es decir, los parámetros serán meros *inputs* de la función y será a través del *return* de la misma de donde obtendremos el resultado. En caso de precisarse el uso de punteros deberán cambiarse estos últimos argumentos de la llamada a NMR_C_parser.py. y le indicaremos por tanto, el tipo de los elementos a los que apunta el puntero (int), así como el nombre de la variable puntero (statemnt) y el tamaño de la variable en memoria (32).

5.2.1. Estructura

La estructura del *script* Python relativo al proceso de *parseo* del código C desarrollado consta de dos flujos claramente diferenciados, esta división viene dada por cómo se realiza el paso de parámetros. Podremos elegir entre el uso habitual de paso de parámetros de C, es decir, el paso de parámetros sin punteros y por tanto, podríamos considerar esos parámetros de la función como un *input* de un módulo HDL y el *return* de la función un *output* o bien, el paso utilizando punteros, donde ese puntero puede llegar a ser considerado una variable tanto de entrada como de salida. Esta diferenciación es debida al uso habitual del paso de punteros como parámetros de una función en C.

Además de esta diferencia respecto a la gestión de los parámetros de la función a proteger cabe remarcar que el proceso de protección de la función tiene varias etapas, todas ellas implementadas en ambos flujos, con y sin punteros.

1. Lectura de argumentos.
2. Generación de archivos '.h' de las réplicas.
3. Generación funciones '.c' réplicas de la protegida.
4. Generación de *main* encargado de llamar a las réplicas y al votador.
5. Generación '.h' de la función *main*, la cual debe tener la misma definición que la función a proteger, haciendo trasparente el cambio en el programa origen del diseñador.
6. Generación votador.
7. Generación fichero *header* del votador.

Remarcar, como se ha indicado anteriormente que según el tipo de paso de parámetros elegido, las fases serán identicas cambiando pequeñas porciones de código referentes a la gestión de los parámetros con o sin uso de punteros.

Lectura de argumentos

En un primer paso del *script* se analizarán los argumentos indicados en la llamada, se configurarán ciertos parámetros y se inicializarán ciertas variables que se usarán posteriormente, especialmente el nombre de los ficheros que contienen las definiciones de las funciones en C.

Generación réplicas

Una vez se ha llevado a cabo la inicialización de las variables se comenzará en análisis de los ficheros para extraer cierta información necesaria para la construcción de las réplicas. Esta información sería el nombre de los argumentos de la función a proteger así como su tipo. Es aquí donde el código ya empezará a tener ciertas diferencias respecto a si hemos elegido una función con punteros como parámetros o simplemente variables sencillas como parámetros de función. Una vez obtenidos la información necesaria acerca de los parámetros se generarán tanto los ficheros '.h' así como los '.c' asociados. Recordar, como se ha venido comentado, especialmente en 4.2.1 que las réplicas actuarán como meros *wrapper* de la función a proteger, y por tanto su tamaño en cuanto a código será pequeña.

Podemos ver un ejemplo de lo que sería una réplica en el siguiente fragmento de código:

```
1 #include "matrix_utils.h"
2  void matrix_mul_conts_3_3_0 (int matrix_a[9], int constant){
3 #pragma HLS inline off
4  return matrix_mul_conts_3_3(matrix_a, constant);
5 }
```

Listing 5.2: Réplica de una función de multiplicación de matriz por constante

Generación NMR

Una vez el programa ha generado la totalidad de las réplicas deseadas pasará a construir lo que será la función que sustituirá a la función protegida en el programa principal. Esta función, la cual su nombre seguirá el patrón NMR_nombre-función será la encargada de pasarle las entradas a todas las réplicas y de aunar sus resultados en un votador.

Puede verse el ejemplo del código relativo al NMR generado en un programa que hace uso de punteros en el paso de parámetros.

```
1 #include "matrix_utils_0.h"
2 #include "matrix_utils_1.h"
3 #include "matrix_utils_2.h"
4 #include "voter.h"
5
6  void NMR_matrix_mul_conts_3_3 (int matrix_a[9], int constant){
7 // Variables
8 int matrix_a_0[9];
9 int matrix_a_1[9];
10 int matrix_a_2[9];
```

```
11
12 int i;
13
14 // Computing results
15 for(i=0;i<9;i++){
16 matrix_a_0[i] = matrix_a[i];
17 matrix_a_1[i] = matrix_a[i];
18 matrix_a_2[i] = matrix_a[i];
19 }
20 matrix_mul_conts_3_3_0(matrix_a_0, constant);
21 matrix_mul_conts_3_3_1(matrix_a_1, constant);
22 matrix_mul_conts_3_3_2(matrix_a_2, constant);
23
24 voter(matrix_a_0,matrix_a_1,matrix_a_2,matrix_a);
25
26 }
```

Listing 5.3: NMR de la función multiplicación de matriz por constante

Generación votador

Una vez finalizadas el resto de tareas, se generará el votador, esta función difiere según la opción de paso de parámetros elegida.

Se mostará a continuación cual será la estructura de un votador que usa punteros para el paso de parámetros:

```
1 #include "array_cmp.h"
2 void voter(int result_0[9],int result_1[9],int result_2[9],int
      voted_result[9])
3 {
4 #pragma HLS inline off
5 int i;
6 if(array_cmp_9(result_0, result_1) == 0){
7     for (i = 0; i < 9; i++)
8 { voted_result[i] = result_0[i];
9  }
10 } else {
11     if(array_cmp_9(result_0, result_2) == 0){
12     for (i = 0; i < 9; i++)
13 { voted_result[i] = result_0[i];
14  }
15     } else {
16         if(array_cmp_9(result_1, result_2) == 0){
17     for (i = 0; i < 9; i++)
18 { voted_result[i] = result_1[i];
19  }
20         } else {
21     for (i = 0; i < 9; i++)
22 { voted_result[i] = -1;
23  }
24         }
25     }
26 }
27 }
```

Listing 5.4: Votador con punteros

Dada la complejidad que da lugar el uso de punteros a lo largo de las funciones se han creado diversas funciones de comparación para aliviar la carga en cuanto a generación de ficheros durante el proceso del NMR_C_Parser.

5.3. NMR_Verilog_Parser

5.3.1. Funcionalidad

Como se indicó en capítulos anteriores, una vez realizado el procesado del código C con el *NMR_C_Parser* y su posterior tratamiento con HLS, dispondremos de un código tanto Verilog como VHDL que representará el diseño en C inicial, una vez en este punto precisaremos de más protecciones ante las nuevas optimizaciones que Vivado intentará realizar. Para ello se desarrolló un *parser* adicional que intenta hacer transparente este proceso al diseñador. Gracias a que el propio HLS sigue patrones concretos para la nomenclatura de sus modulos derivados de los diseños en C originales, podremos saber qué modulos HDL hacen referencia tanto a las funciones creadas como a la función protegida y de esta manera aplicar nuevos *pragmas* que obliguen al sintetizador de Vivado a no optimizar los módulos relativos a las funciones creadas anteriormente.

El desarrollo del *Verilog parser* cuenta con una componente de desarrollo más sencilla, dado que simplemente deberemos indicarle al sintetizador de Vivado que no realice optimizaciones de espacio en los módulos creados previamente desde Vitis_HLS y por tanto sólo deberemos fijarnos en los nombres que ha elegido Vitis_HLS para los módulos de NMR creados y así protegerlos utilizando este nuevo *parser*.

5.4. Utilidades

Durante el desarrollo de las bibliotecas fue necesaria la creación de diversas utilidades que apoyaran al programa principal a leer y/o modificar textos, especialmente para el análisis de la función a proteger y la extracción de los argumentos, sus tipos y otro tipo de variables relacionadas. Remarcar especialmente el uso de diversas expresiones regulares que ayudaron a la hora de realizar la selección de partes del código en concreto.

El tipo de funciones generadas a lo largo del trabajo para realizar el análisis son similares a la siguiente, la cual busca por el fichero en nombre de una función en concreto y devuelve el los parámetros de la misma en un *array* de *strings*.

```
1  # ========================================
2  # GET FUNCTION PARAMETERS ARRAY
3  # ========================================
4  # This function returns the parameters of
5  # the function in a string array.
6
7  def get_function_call_arguments(source_file_name, func_name):
8
9      source_file = open(source_file_name,"r+")
```

```
10      arguments = ""
11
12      for line in source_file:
13          if func_name in line:
14              txt = line
15
16      # This regex removes the argument types from the arguments
        structure
17      regex = "\w+ "
18      p = re.split(regex, txt)
19
20      # This regex removes the array size. Example: value[32] ->
        value
21      regex = "\[.*\]"
22      for y in p:
23          arguments = arguments + re.sub(regex,"",y)
24
25      arguments = arguments.replace("\n", "")
26
27      source_file.close()
28
29      return arguments
```

Listing 5.5: *Get function parameters*

Capítulo 6

Experimento

Para comprobar que la biblioteca desarrollada cumple con los objetivos fijados, es decir, protege contra errores a las funciones indicadas, se realizaron pruebas con y sin la función objetivo protegida.

En esta sección se mostrá un ejemplo del proceso completo del experimento realizado, este no es otro que proteger una función dentro de un programa C dado, testear su correcto funcionamiento en cada una de las etapas y para finalizar realizar una inyección de errores.

6.1. Multiplicación matriz por constante

En cuanto al experimento realizado, utilizaremos un algoritmo básico creado para la comprobación de las bibliotecas, realizará una multiplicación de una matriz por una constante, de esta manera podremos comprobar de una manera sencilla si las bibliotecas desarrolladas cumplen con la función deseada. Puede revisarse el código base del algoritmo en el **??**.

```
1
2  matrix_mul_3_3 (int matrix_a[9], int matrix_b[9]){
3
4      int a_p;
5      int res_row;
6      int res_column;
7
8      int result[9][9];
9      int *rp;
10
11     int row;
12     int column;
13
14     rp = result[0];
15
16     row = 3;
17     column = 3;
18
19     for (res_column = 0; res_column < column; res_column++)
```

```
20    {
21        for (res_row = 0; res_row < row * column; res_row = res_row
       + 3)
22        {
23            for (a_p = 0; a_p < column; a_p++)
24            {
25                rp[res_column + res_row] += matrix_a[res_row + a_p]
       * matrix_b[(a_p * row) + res_column];
26            }
27        }
28    }
29
30 }
```

Listing 6.1: Código multiplicación de matriz 3x3

En cuanto al entorno de trabajo, seguiremos el flujo de desarrollo propio del vendedor y por tanto, haremos uso de todas las herramientas de desarrollo indicadas en el capítulo 3.

El procedimiento para la prueba será el siguiente, realizaremos todo el proceso de protección utilizando el *NMR_C_Parser* y comprobaremos que después de la realización del proceso el resultado del cómputo utilizando un test en C, es el mismo que antes de realizar la protección. Una vez comprobemos que la prueba tiene un resultado satisfactorio, pasaremos a realizar una co-simulación con el mismo test C anterior, para posteriormente si sigue siendo el resultado correcto, pasar a hacer la traducción de C a Verilog haciendo uso de HLS. Es en esta última étapa donde realizaremos la inyección de errores donde comprobaremos por completo el correcto comportamiento del diseño una vez aplicada la metodología de tolerancia a fallos elegida.

6.1.1. Generación algoritmo tolerante a fallos

Lo primero expondremos las diversas fases del desarrollo, dado que para realizar el despligue la redundancia espacial en el diseño C elegido, tendremos que pasar por diferentes fases.

El flujo será el siguiente:

1. Protección función
2. Test en C.
3. Test en co-simulación. Se utilizará el test en C y el código del sistema en Verilog.
4. Test en Verilog. Se utilizará Verilog tanto para el test como para el diseño del sistema (código del sistema obtenido mediante HLS).

Una vez protegida la función, se habrán generado los archivos adicionales necesarios para dotar de la característica comentada. Estos serán:

- Réplicas de la función a proteger. El número dependerá del número elegido durante el lanzamiento del *parser*.

- Votador.
- NMR_función encargado de relacionar las réplicas con el votador.

Usando la siguiente sentencia invocamos al *parser* generando la protección en el programa deseado:

python3 NMR_C_parser.py matrix_mul_conts_3_3 matrix_utils.c matrix_utils.h matrix_main.c 3 2 int matrix_a 9

El resultado del mismo se muestra en el *listing* 6.2.

```
# ================================== #
# SUMMARY REPORT
# ================================== #
### MAIN PARAMETERS ###
Function name (to proctect):  matrix_mul_conts_3_3
Function C file:  matrix_utils.c
Function H file:  matrix_utils.h
Main program:  matrix_main.c
NMR replicas number:  3
### EXTRA PARAMETERS ###
Function return type:  2
Function type:  int
Function return parameter name:  matrix_a
# ================================== #
# REPLICAS
# ================================== #
File matrix_utils_0.c generated
File matrix_utils_1.c generated
File matrix_utils_2.c generated
File matrix_utils_0.h generated
File matrix_utils_1.h generated
File matrix_utils_2.h generated
# ================================== #
# NMR WRAPPER
# ================================== #
File NMR_matrix_mul_conts_3_3.h generated
File NMR_matrix_mul_conts_3_3.c generated
File voter.h generated
File voter.c generated
# ================================== #
File matrix_main.c modified
# ================================== #
matrix_mul_conts_3_3() PROTECTED
# ================================== #
```

Listing 6.2: Salida *parser* NMR_C_Parser

Ahora se lanzará el test que lanzará una batería de multiplicaciones por una constante aleatoria para comprobar de esta manera que el algoritmo en C es correcto. En el *listing* 6.3fragmento de código, relativo a la última iteración del test, se puede observar como existen 0 cómputos erróneos. El test ejecutado realiza la multiplicación usando la función C protegida y la comparada contra un resultado esperado precalculado.

```
###############
MAIN MATRIX A
```

```
0 20 1
31 20 3
5 9 38
CONSTANT: 999
EXPECTED MATRIX
0 19980 999
30969 19980 2997
4995 8991 37962
COMPUTED MATRIX
0 19980 999
30969 19980 2997
4995 8991 37962
ITERATION: 1000
FAILED: 0
###############
```

Listing 6.3: Resultado test C función protegida

Una vez comprobado que el test retorna con código 0, es decir, finaliza sin errores, pasaremos a comprobar utilizando la herramienta *scheduler* de Vitis_HLS que los procesos inferidos, es decir, los módulos HDL que han resultado después del procesamiento con HLS son los esperados.

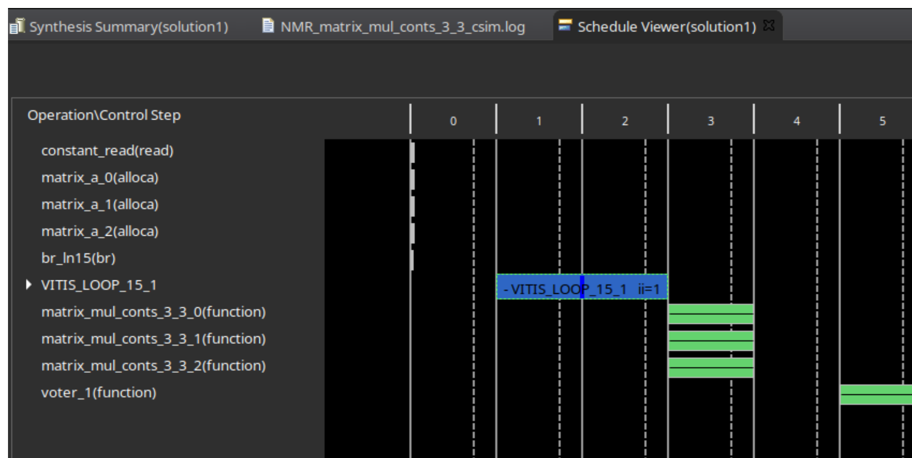

Figura 6.1: *Scheduler*

En la imagen 6.1 podemos ver como se infieren tres procesos (matrix_mul_conts_3_3_0, matrix_mul_conts_3_3_1 y matrix_mul_conts_3_3_2) que hacen alusión a las tres réplicas generadas, además de un votador (voter_1), el cual ejecutará su función una vez las tres réplicas hayan realizado su correspondiente cálculo.

Una vez realizada la prueba anterior, se realizará la prueba de co-simulación, la cual utilizará el código de test en C lanzado en las anteriores pruebas, pero esta vez no usará el diseño C como diseño en test, sino el propio diseño en HDL. Además podremos volver a comprobar que las tareas esperadas son las correctas, deberán aparecer tres réplicas funcionando en paralelo y una vez finalizado su

cómputo el votador elegirá el resultado más votado como correcto, siguiendo de esta manera la tónica del *scheduler*.

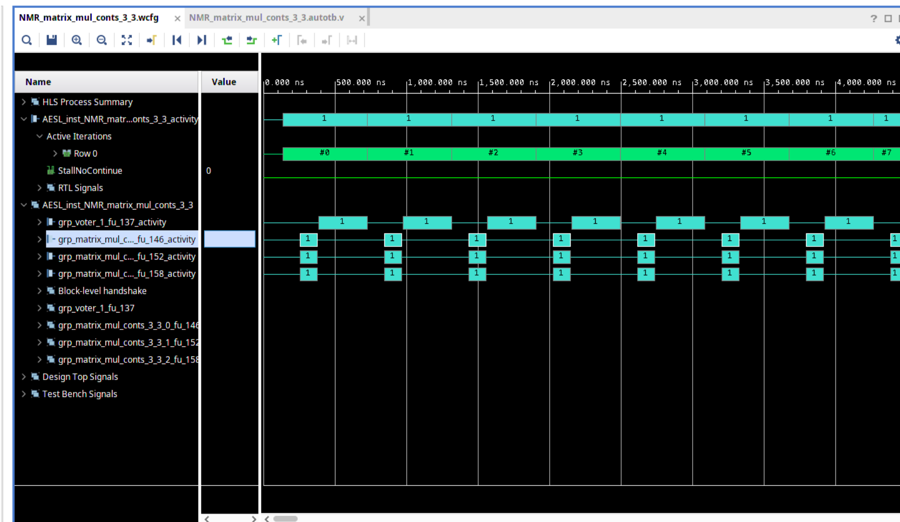

Figura 6.2: Co-simulación

En la 6.2 puede observarse como se generaron los tres procesos para las réplicas así como un proceso votador adicional de la misma manera que cuando se revisó el resultado del *scheduler*, eso sí, esta vez, veremos como estos procesos se repiten debido simplemente a la ejecución de cada una de las iteraciones del test.

Una vez comprobado que se han generado las réplicas en co-simulación, se aplicó el segundo *parser* desarrollado, el cual añadirá a los archivos Verilog que contienen las réplicas el atributo *DONT_TOUCH* del cual hablamos en secciones anteriores y que le indica al sintetizador de Vivado que no optimice los módulos que contienen este atributo.

Puede revisarse en el *listing* 6.4 fragmentos de código de cada una de las réplicas en las cuales se ha añadido el atributo *DONT_TOUCH*.

```
1  (* DONT_TOUCH = "yes" *)
2  module NMR_matrix_mul_conts_3_3_matrix_mul_conts_3_3_0 (
3          ap_clk,
4          ap_rst,
5          ap_start,
6          ap_done,
7          ap_idle,
8          ap_ready,
9          matrix_a_address0,
10         matrix_a_ce0,
11         matrix_a_we0,
12         matrix_a_d0,
13         matrix_a_address1,
14         matrix_a_ce1,
```

```
        matrix_a_q1,
        constant_r
);
(* DONT_TOUCH = "yes" *)
module NMR_matrix_mul_conts_3_3_matrix_mul_conts_3_3_1 (
        ap_clk,
        ap_rst,
        ap_start,
        ap_done,
        ap_idle,
        ap_ready,
        matrix_a_address0,
        matrix_a_ce0,
        matrix_a_we0,
        matrix_a_d0,
        matrix_a_address1,
        matrix_a_ce1,
        matrix_a_q1,
        constant_r
);
(* DONT_TOUCH = "yes" *)
module NMR_matrix_mul_conts_3_3_matrix_mul_conts_3_3_2 (
        ap_clk,
        ap_rst,
        ap_start,
        ap_done,
        ap_idle,
        ap_ready,
        matrix_a_address0,
        matrix_a_ce0,
        matrix_a_we0,
        matrix_a_d0,
        matrix_a_address1,
        matrix_a_ce1,
        matrix_a_q1,
        constant_r
);
```

Listing 6.4: Ejemplo llamada *Parser*

A continuación, y para realizar la verificación de que el atributo ha surtido el efecto deseado, se lanzó el sintetizado y se observó el número de módulos que el sistema detecta después del proceso.

Como puede observarse en la figura 6.3 después del sintetizado las réplicas han quedado inalteradas. Es decir, Vivado no las ha eliminado, por lo que podemos concluir que el proceso ha acabado de manera satisfactoria.

6.1.2. Verificación del mecanismo de tolerancia a fallos desplegado

Para la comprobación final de que el sistema funciona acorde a lo indicado, se inyectó un error en uno de los bits de una réplica generando por tanto que una de las réplicas no devuelva el resultado correcto (simulando un SEU) y por tanto obligando al votador a elegir cuál de los resultados es correcto, donde debería

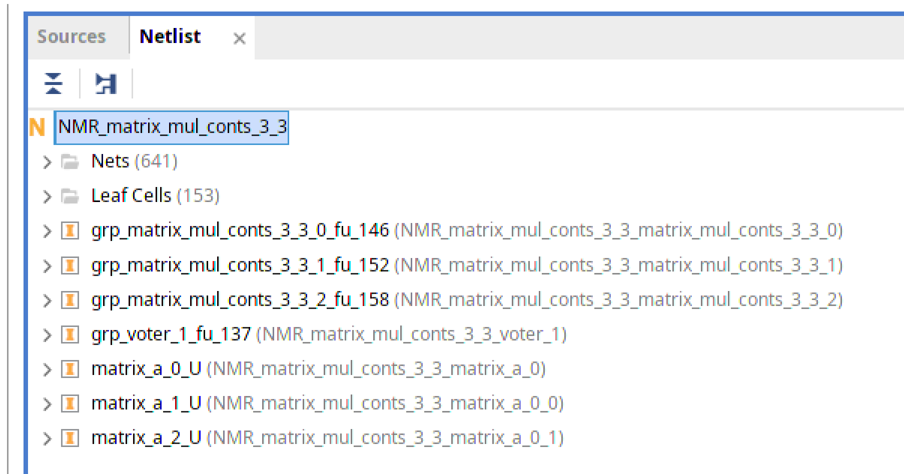

Figura 6.3: Síntesis del diseño utilizando el atributo *DONT_TOUCH*

escojer el resultado de las otras dos réplicas restantes que no están averiadas.

Puede observarse mediante el log del *testbench* realizado en Verilog como comprobación de que incluso inyectando un fallo, el resultado es correcto.

```
source /home/raullozano/Escritorio/TFM/FT_Library/Pruebas\ Espacial
    /Matrix_Mul_CTE/error_injection_1.tcl
# restart
INFO: [Simtcl 6-17] Simulation restarted
# add_force {/tb_NMR_matrix_mul_conts_3_3/DUT/
    grp_matrix_mul_conts_3_3_0_fu_146/matrix_a_d0[27]} -radix hex
    {1 300.800ns} -cancel_after 340.000ns
# run all
TB: MATRIX TRANSMISSION STARTS
Sending nothing
Sending matrix element          1, value 00000001
Sending matrix element          2, value 00000001
Sending matrix element          3, value 00000001
Sending matrix element          4, value 00000001
Sending matrix element          5, value 00000002
Sending matrix element          6, value 00000001
Sending matrix element          7, value 00000001
Sending matrix element          8, value 00000001
Sending matrix element          x, value 00000001
TB: MATRIX TRANSMISSION FINISHED
Receiving matrix element 4294967295, value 00000005
Receiving matrix element          0, value 00000005
Receiving matrix element          1, value 00000005
Receiving matrix element          2, value 00000005
Receiving matrix element          3, value 0000000a
Receiving matrix element          4, value 00000005
Receiving matrix element          5, value 00000005
Receiving matrix element          6, value 00000005
Receiving matrix element          7, value 00000005
Receiving matrix element          x, value xxxxxxxx
```

```
TB: MATRIX MULTIPLICATION OK
$finish called at time : 835 ns :
File "/home/raullozano/Escritorio/TFM/FT_Library/Repositorio/
    Verilog/tb_NMR_matrix_mul_conts_3_3.v" Line 70
```

Además se puede observar como se forzó un bit de las réplicas para comprobar que incluso inyectando un error, el sistema sigue dando como resultado un valor correcto. En la figura 6.4 se muestran tres bits del resultado de cada una de las réplicas en las cuales sólo en una se observa un '1' lógico forzado durante unos instantes antes de volver al valor '0' lógico original. Simulando la aparición de un SEU.

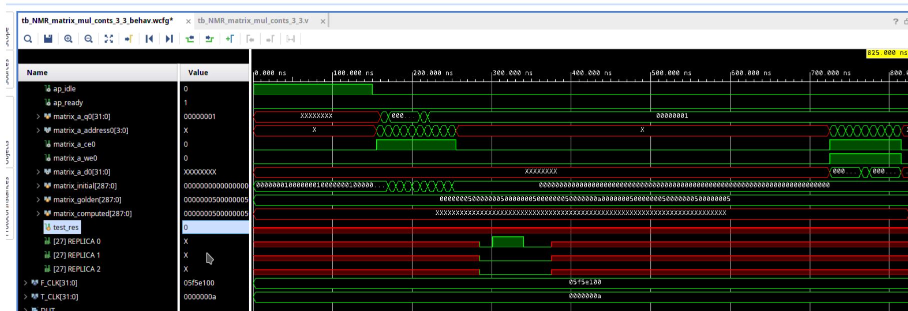

Figura 6.4: Forzado de bit a 1

Por otro lado, y de cara a verificar el comportamiento de la técnica de redundancia añadida. Se realizó otra inyección de errores, haciendo esta vez que dos de las tres réplicas disponibles fallen, ante esta casuística el sistema debe fallar y devolver un resultado erróneo. Podemos ver en **??** que así es. Tal como se ha diseñado el votador, si no hay resultado mayoritario, fijará todo los bits del resultado a 1.

```
TB: MATRIX TRANSMISSION STARTS
Sending nothing
Sending matrix element          1, value 00000001
Sending matrix element          2, value 00000001
Sending matrix element          3, value 00000001
Sending matrix element          4, value 00000001
Sending matrix element          5, value 00000002
Sending matrix element          6, value 00000001
Sending matrix element          7, value 00000001
Sending matrix element          8, value 00000001
Sending matrix element          x, value 00000001
TB: MATRIX TRANSMISSION FINISHED
TB: GOLDEN MATRIX:
00000005 00000005 00000005
00000005 0000000a 00000005
00000005 00000005 00000005
TB: COMPUTED MATRIX:
ffffffff ffffffff ffffffff
ffffffff ffffffff ffffffff
ffffffff ffffffff ffffffff
```

```
51 TB: MATRIX MULTIPLICATION FAILED
```

Capítulo 7

Conclusiones

7.1. Contratiempos y problemas sufridos

A lo largo del desarrollo sufrimos varios problemas, la gran mayoría relativos a la adaptación de nuestro *parser* a la gran variedad de programas contra los que puede usarse. Se mostrarán a continuación varios de los problemas y contratiempos sufridos a lo largo del proceso de elaboración de la biblioteca.

7.1.1. Estilos de programación

A lo largo del desarrollo del trabajo, fuimos alimentando el *parser* con algoritmos básicos, como con algoritmos más complejos para poder realizar un *testeo* más exahustivo. Durante estas pruebas pudimos evidenciar la problemática ocasionada por los diferentes estilos de programación. Entre varios de los problemas, podríamos centrarnos en los dos más evidentes.

Para el *parser*, un salto de linea es un carácter que complica cualquier análisis y posterior modificación del código.

```
1 int suma (int a, int b){
2     return a+b;
3 }
```

Listing 7.1: Suma estilo 0

```
1 int
2 suma (int a, int b)
3 {
4     return a+b;
5 }
```

Listing 7.2: Suma estilo 1

Puede observarse en los *listings* 7.1 y 7.2 fragmentos de código con dos estilos de programación en C diferentes. El *parser* desarrollado deberá interpretar

ambas implementaciones de igual manera, a pesar de no estar escritos igual. Esta problemática generó bastante impacto a la hora de ir abordando algoritmos más complejos.

Otro de los problemas sería el uso o no de uso de punteros a memoria en los parámetros de las funciones.

```
1 void
2 foo (int[32] a, int b)
3 {
4     //Codigo
5 }
```

Listing 7.3: Parámetros estilo 0

```
1 void
2 foo (int *a, int b)
3 {
4     //Codigo
5 }
```

Listing 7.4: Parámetros estilo 1

```
1 void
2 foo (int *a, int[32] b)
3 {
4     //Codigo
5 }
```

Listing 7.5: Parámetros estilo 2

En los *listings* 7.3,7.4 y 7.5, puede observarse como existen multitud de opciones posibles para definir una función y por tanto, se añade una complejidad para nada despreciable a la hora de abordar el análisis de los parámetros por el *parser*. Se deberá actuar de manera diferente si el resultado figura como el primer párametro pasado a la función o si figura como el segundo, el tercero, etc.

7.1.2. CHStone *benchmark*

Para las pruebas relativas al *parser*, se decidió en un principio usar una colección de pruebas ya estipuladas para HLS, en concreto se escogió el *benchmark* de CHStone. El problema de usar esta compilación de pruebas ya estandarizadas es que pueden acaban usando librerías del sistema en muchas ocasiones, a las cuales no tenemos acceso de manera sencilla. Esta problemática nos impidió usar esta batería de pruebas como programas ejemplo contra los que *testear* la biblioteca.

Operaciones Float

En relación también a lo comentado en el apartado anterior, pudimos comprobar como la jerarquía de llamadas puede acabar complicando de manera desmesurada la lógica del NMR_C_Parser. Nos percatamos de esta problemática

durante la ejecución de una prueba utilizando uno de los *benchmark* de CHStone, el cual realiza cómputos costosos en coma flotante. Durante la ejecución del mismo, se genera una jerarquía de llamadas a función que aumentan la complejidad lógica necesaria para analizar el propio fichero, de hecho en ocasiones es pila de llamadas acababan en *includes* propios del sistema operativo a los cuales no se tiene acceso de manera sencilla.

7.2. Consideraciones y aprendizaje

Consideraría lo más importante de este trabajo el fuerte componente I+D que conllevaba, es decir, partíamos de ciertas hipótesis, las cuales había que estudiar y comprobar si era posible realizar, en el caso que nos atañe, utilizando las herramientas HLS de Xilinx, por lo que para ello se tuvo que realizar un proceso de estudio previo partiendo de las bases del conocimiento adquiridas a lo largo del máster para después ampliarlas investigando acerca de lo que Vitis_HLS nos ofrecía. Como todo proyecto I+D, el trabajo no consistía en hacer un uso meramente práctico de lo aprendido, si no que necesitábamos estructurar todo el proceso de una manera diferente a lo que sería un desarrollo de *software* más habitual, donde en base a unas especificaciones, generamos un producto. Se precisó por tanto, un estudio previo de las herramientas disponibles, creación de baterías de pruebas básadas en hipótesis previamente defindas, desarrollo de programa que aplica el conocimiento básado en las hipótesis que fueron válidas, y por último la obtención de resultados utilizando las herramientas desarrolladas.

En resumen, este tipo de trabajos ayudan al alumnado a combinar tanto tareas más académicas y/o de investigación con un mundo más práctico y de obtención de resultados, como puede ser la generación de las herramientas comentadas a lo largo de todo el trabajo.

Además, por la naturaleza del propio proyecto, se tuvieron que utilizar diversos lenguajes de programación, como se ha comentado a lo largo del documento, donde cada uno de ellos aportan visiones diferentes del ámbito de la informática en general y de la ingeniería de computadores en particular.

Los cuales recordamos, fueron los siguientes:

- Python.
- C.
- Verilog.
- TCL.

Gracias a Python pudimos realizar análisis y modificaciones en ficheros de una manera sencilla. Aportando de esta manera una visión más de alto nivel.

También se trabajó con C, tanto para la creación de algoritmos de pruebas para alimentar al *parser* así como para realizar modificaciones en otros programas con implementaciones ya realizadas, cosa que propició la adquisición de práctica acerca de la gestión de memoria que realiza C, conocimiento necesario

para poder realizar la correcta implentación y/o modificación de programas que pasar al *parser* siguiendo las restricciones de uso propias de HLS.

También se trabajó con C, tanto creando algoritmos, como modificando programas ya creados los cuales serían utilizados para comprobar que el *parser* creado funcionaba según lo esperado. Además, gracias al bajo nivel que aporta C en cuanto a manejo de la memoria del sistema, se pudo profundizar en aspectos más especificios del uso de punteros.

Por otro lado, también se perfilaron los conocimientos de Verilog del estudiante, dado que fue necesario la revisión y modificación de programas sintezados con HLS para comprobar que los abributos utilizandos durante el proyecto protegían el diseño de las optimizaciones que el sintetizador de Vivado intenta aplicar.

Cabe destacar también el uso de TCL, lenguaje muy habitualmente utilizado en ambientes de trabajo con FPGAs, se usó precisamente para comprobar que incluso inyectando errores nuestro sistemas, el diseño seguía funcionando sin problemas.

En resumen, destacar que el fuerte contenido I+D así como la necesidad de uso de diferentes lenguajes según el punto del diseño en el que estuviéramos obligó tanto a que el trabajo tuviera un componente que podría denorminarse más académico como más práctico.

7.3. Trabajo a futuro

En cuanto al trabajo a futuro, se vislumbrarían tres grandes lineas de desarrollo para ampliar y/o mejorar las capacidades del *parser* implementado.

7.3.1. Experimentos con CHStone

La primera posible linea podría ser la utlización de la batería de test de CHStone para mejorar el *parser*. Utilizando sus tests como programas ejemplo que ayuden a hacer que el NMR_C_Parser sea más genérico y que poco a poco vaya teniéndo un enfoque más amplio, en cuanto a estilos de programación se refiere.

7.3.2. Redundancias

También sería otra posible linea de trabajo el aumento del número de *parsers* de la biblioteca, generando por ejemplo un nuevo archivo Python para dotar a los diseños en C de redundancia temporal así como de la información. Cabría añadir, que podría comenzarse con la redundancia temporal, basándonos en las pruebas ya realizadas y comentadas en la sección 4.2.2.

7.3.3. Integración herramientas Xilinx

La última linea correspondería con dotar a la biblioteca de integración con las herramientas de Xilinx, de tal manera que el usuario tuviera que interaccionar

lo más mínimo con los dos IDEs comentados, usando comandos TCL desde Python para desplegar los mecanismos de tolerancia a fallos y realizar llamadas a ambos IDEs para utilizar las herramientas de las que disponen, facilitando de esta manera las tareas del diseñador.

Bibliografía

[1] Juan-Carlos Ruiz, David de Andrés, Sara Blanc, Pedro Gil *Generic Design and Automatic Deployment of MR Strategies on HW Cores.* The 14th IEEE Pacific Rim International Symposium on Dependable Computing PRDC 2008.
December 2008 Taipei, Taiwan

[2] Ganghee Lee, Dimitris Agiakatsikas, Tong Wu, Ediz Cetin , and Oliver Diessel. *TLegUp: A TMR Code Generation Tool for SRAM-Based FPGA Applications Using HLS.* Department of Engineering, Macquarie University Australia

[3] ECSS Secretariat ESA-ESTEC *ECSS-Q-ST-30-02C - Space product assurance Failure modes, effects (and criticality) analysis (FMEA/FMECA).* ECSS Secretariat ESA-ESTEC Requirements Standards Division Noordwijk, The Netherlands

[4] ECSS Secretariat ESA-ESTEC *ECSS-E-HB-10-12A - Space engineering Calculation of radiation and it seffects and margin policy handbook.* ECSS Secretariat ESA-ESTEC Requirements Standards Division Noordwijk, The Netherlands

[5] ECSS Secretariat ESA-ESTEC *ECSS-Q-ST-60-15C - Space product assurance Radiation hardness assurance - EEE components.* ECSS Secretariat ESA-ESTEC Requirements Standards Division Noordwijk, The Netherlands

[6] *Satellite Anomalies*, *https://www.ngdc.noaa.gov/stp/satellite/anomaly/satelliteanomaly.html*

[7] *Vitis High-Level Synthesis User Guide*, *https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_2/ug1399-vitis-hls.pdf*

[8] *Vivado Design Suite User Guide - Using the vivado IDE*, *https://www.xilinx.com/support/documentation/sw_manuals/xilinx2021_1/ug893-vivado-ide.pdf*

[9] *Vivado Design Suite User Guide - High-Level Synthesis*, https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug902-vivado-high-level-synthesis.pdf