UNIVERSIDAD COMPLUTENSE DE MADRID

Facultad de Informática

TÉCNICAS DE PROTECCIÓN DE CIRCUITOS PARA APLICACIONES AEROESPACIALES SOBRE HARDWARE RECONFIGURABLE

Silvia Alcázar Andrés

Almudena Alonso de la Iglesia

Beatriz Álvarez-Buylla Fernández

Proyecto de Sistemas Informáticos de Ingeniería en Informática

Departamento de Arquitectura de Computadores y Automática

Directores: Hortensia Mecha López y Juan Antonio Clemente

Madrid, 2013

Autorización de difusión y utilización

Los autores de este proyecto autorizan a la Universidad Complutense de Madrid a difundir y utilizar el presente trabajo de investigación, tanto la aplicación como la memoria, únicamente con fines académicos, no comerciales y mencionando expresamente a sus autores. También autorizan a la Biblioteca de la UCM a depositar el trabajo en el Archivo Institucional E-Prints Complutense.

Autores

Silvia Alcázar Andrés

Almudena Alonso de la Iglesia

Beatriz Álvarez-Buylla Fernández

Fecha

21/06/2013

Agradecimientos

En primer lugar queremos agradecer este trabajo a nuestras familias y amigos que nos han apoyado en todos los momentos a lo largo de la carrera y sobre todo en este último año durante la realización de este proyecto.

También, queremos agradecérselo a todos los profesores que nos han acompañado a lo largo de la carrera y que han aportado algo a lo que hoy se ha convertido en nuestro proyecto de final de carrera.

Finalmente, damos las gracias a Felipe Serrano que nos ha facilitado la comprensión de nuestros fallos y nos ha ayudado a salir adelante frente a ellos. Y por supuesto, a nuestros directores de proyecto, Hortensia Mecha y Juan Antonio Clemente, que nos han orientado en el enfoque de nuestro trabajo y en los momentos más duros.

Índice

Índice de figuras

Índice de abreviaturas

- ASIC: Application-Specific Integrated Circuit
- BRAM: Block Random Access Memory
- CLB: Configurable Logic Block
- CMT: Clock Management Tile
- CPLD: Complex Programmable Logic Device
- DCM: Digital Clock Manager
- DDR: Double Data Rate
- DSP: Digital Signal Processor
- EDK: Embedded Development Kit
- FPGA: Field Programmable Gate Array
- IEEE: Institute of Electrical and Electronics Engineers
- IOB: Input/Output Block
- ISE: Integrated Software Environment
- JTAG: Joint Test Action Group
- LMB: Local Memory Bus
- LUT: Look Up Table
- PC: Personal Computer
- PLB: Processor Local Bus
- PLL: Phase Locked Loop
- RAM: Random Access Memory
- RS232: Recommended Standard 232
- SDK: Software Development Kit
- SEFI: Single Event Functional Interrumpt
- SEU: Single Event Upset
- SRAM: Static Random Access Memory
- TMR: Triple Module Redundancy
- UART: Universal Asynchronous Receiver-Transmitter
- USB: Universal Serial Bus
- VHDL: VHSIC hardware description language
- VHSIC: Very High Speed Integrated Circuit
- XUP: Xilinx University Program

Resumen

Las FPGAs surgen en 1984 como una evolución de los CPLDs. Sus principales ventajas son su diseño sencillo, su fiabilidad, el ahorro en los costes que suponen en la producción y, la posibilidad de reconfigurarlas, lo que les proporciona mayor flexibilidad que cualquier otro dispositivo.

Por este motivo se han convertido en indispensables para algunos sectores como son el aeronáutico y aeroespacial, sectores en los cuales la tolerancia ante los fallos resulta primordial. El impacto de una partícula cósmica podría resultar fatal si alterara la configuración de dicho dispositivo. Por ello es necesario estudiar técnicas para proteger los diseños de alteraciones producidas por las mencionadas partículas y sus consecuencias.

En este proyecto se realiza el estudio y la puesta en marcha de una de estas técnicas de protección de circuitos sobre una plataforma de inyección de errores previamente desarrollada (NESSY 5.0) y una FPGA Xilinx™ Virtex 5. Hemos obtenido y comparado resultados reales de la aplicación de dicha técnica sobre diferentes diseños y hemos realizado un estudio de los mismos.

El resultado es una ampliación de la plataforma NESSY que además de permitir emular el impacto de una partícula sobre un circuito, permite incluir las técnicas de protección y comparar los resultados de las inyecciones de errores.

Hemos comprobado que los circuitos protegidos tienen menor vulnerabilidad frente a posibles impactos de partículas cósmicas respecto al circuito sin proteger, así que la técnica usada sería eficiente para ser utilizada en aplicaciones reales.

Palabras clave

FPGA, Virtex 5, Hardware reconfigurable, Bitflip, ICAP, Aplicaciones aeroespaciales, Reconfiguración, Inyección, Bitstream.

Abstract

The FPGAs appear in 1984 as an evolution of CPLDs. Its main advantages are its simple design, reliability, saving in the involved costs in the production and, specially, the ability to reconfigure, giving them more flexibility than any other device.

For this reason they have become indispensable to some sectors like the aviation and the aerospace, in which the tolerance to failures is very important. The cosmic particle impact could be fatal if it alters your device configuration. Therefore is necessary to study techniques for protecting the mentioned designs of alterations produced by particles and their consequences.

In this project is developed the study and the implementation of some of those circuit protection techniques on an injection platform previously developed and a Xilinx™ Virtex 5 FPGA. We have obtained and compared the real results of the application of those techniques on different designs and we have made a study of them.

The result is an extension of the platform which, besides allowing NESSY emulate the impact of a particle on a circuit, can include protection techniques and comparing the results of injections of errors.

We found that the protected circuits have lower vulnerability to potential impacts of cosmic particles relative to unprotected circuit, so the technique used would be efficient to be used in real applications.

Keywords

FPGA, Virtex 5, Reconfigurable hardware, Bitflip, ICAP, Space applications, Reconfiguration, Injection, Bitstream.

Capítulo 1: Introducción

1.1. Motivación

Las FPGAs (*Field Programmable Gate Arrays*) surgen en 1984 como una evolución de los CPLDs (*Complex Programmable Logic Devices*). En ellas aumenta el número de puertas lógicas equivalentes respecto a los CPLDs y cambia la arquitectura pasando de una más rígida en los CPLDs a una arquitectura mucho más flexible en las FPGAs. Sin embargo, la diferencia más importante es que en las FPGAs se pueden encontrar funciones de alto nivel embebidas en la propia matriz de interconexiones, así como bloques de memoria.

Las FPGAs pueden ser configuradas en tiempo de ejecución mediante un mapa de bits que se carga en memoria. Se pueden programar desde funciones sencillas hasta sistemas complejos.

Además, se utilizan en aplicaciones parecidas a los ASICs (*Application-Specific Integrated Circuit, C*ircuito Integrado para Aplicaciones Específicas), aunque son más lentas, tienen un mayor consumo de potencia y no pueden abarcar sistemas tan complejos como ellos.

Sin embargo, las FPGAs tienen las siguientes ventajas lo que las hace muy útiles para determinados sistemas:

- ✓ Diseño sencillo.
- ✓ Alto rendimiento.
- ✓ Tiempo de desarrollo menor que en otros dispositivos.
- ✓ Fiabilidad.
- ✓ Ahorro en costo, tanto de desarrollo como de adquisición.
- ✓ Reprogramación, lo que añade flexibilidad.
- ✓ Seguridad.

Por otra parte, las aplicaciones en las que habitualmente son usadas las FPGAs son:

- ✓ Sistemas aeronáuticos y aeroespaciales.
- ✓ Módulos de comunicación.
- ✓ Aplicaciones nucleares con restricciones de tiempo real muy severas.
- ✓ Formación en el diseño de sistemas hardware.
- ✓ Procesamiento de señales digitales.
- ✓ Simulación y depuración en el diseño de microprocesadores y microcontroladores.
- ✓ Prototipos de ASICs.

En los sectores aeronáutico y aeroespacial, las FPGAs han adquirido especial relevancia en los últimos años, ya que los dispositivos usados en satélites y aviones tienen que procesar mucha información antes de mandarla a la Tierra, debido al reducido ancho de banda en el canal de comunicación que conecta ambos.

En las aplicaciones nucleares, su importancia se debe a que las restricciones son de tiempo real, es decir, es importante que los sistemas produzcan respuestas correctas en un tiempo determinado.

En los últimos 20 años se ha producido un incremento en el uso de las FPGAs basadas en memoria SRAM (*Static Random Access Memory*) para aplicaciones aeroespaciales por ser reprogramables, tener un menor coste de desarrollo y adquisición que otros dispositivos y un tiempo de desarrollo también menor.

Sin embargo, con el aumento de su uso, también se ha producido un incremento de la preocupación por los efectos que las radiaciones pueden causar sobre las mismas. En particular, es importante el efecto de los SEUs *(Single Event Upsets*) que son producidos por el impacto de una partícula cósmica al colisionar con el dispositivo. Estas colisiones pueden provocar la alteración de la lógica estática, de las BRAMs (*Random Access Memory Blocks*) y también del contenido de la SRAM que se utiliza para la configuración de la FPGA.

Una alteración en un bit de la SRAM puede alterar la funcionalidad del diseño y así provocar fallos en la ejecución. De aquí surge la necesidad de hacer FPGAs con

memorias de configuración más robustas, especiales para embarcar, pero que siguen siendo muy vulnerables.

Debido a la importancia de proteger los circuitos programados en FPGAs, se han llevado a cabo multitud de estudios sobre técnicas de protección de circuitos. A continuación vamos a detallar las que han tenido más éxito y/o difusión para así explicar por qué nos decantamos por una en particular para el desarrollo de este proyecto.

1.2. Trabajo relacionado

Distintos grupos se investigación han propuesto diversas técnicas de protección de circuitos digitales para así mostrarnos alternativas con sus ventajas e inconvenientes.
De este modo, estudiándolas podremos decidir cuál es más adecuada en cada momento.

1.2.1. Técnicas de protección

Todas las técnicas que presentaremos a continuación se basan en la redundancia definida como "*el empleo de información, recursos o tiempo adicionales, por encima de los estrictamente necesarios para el correcto funcionamiento de un sistema.*" [JoAd89]

1.2.1.1. Redundancia hardware

Este tipo de redundancia consiste en incorporar hardware adicional, normalmente con el fin de detectar fallos o conseguir una alta tolerancia ante los mismos. Es la técnica más frecuentemente utilizada.

A continuación, detallamos diferentes variantes de la misma:

Redundancia pasiva

Esta técnica explota el concepto de enmascaramiento de los fallos, es decir, se encubren las consecuencias que éstos puedan tener en el correcto funcionamiento de los circuitos. En ellas no es necesario que se realice ninguna acción externa por parte del sistema que controla el funcionamiento del circuito o del usuario final.

Este tipo de técnicas será la que más nos detendremos en detallar, ya que será la que posteriormente utilicemos en nuestro proyecto para proteger circuitos.

El primer tipo de técnicas de redundancia pasiva que mencionaremos es la **Redundancia Triple Modular (TMR**), la cual tiene como concepto básico ejecutar tres copias del mismo circuito de manera que cada una de ellas realice individualmente todos los cálculos y genere una salida. En la figura 1.1, podemos ver un esquema de cómo quedaría el sistema. Como se puede observar, las tres salidas son procesadas por un selector de mayoría (o votador) que genera un único resultado (la salida será la entrada del votador que más se repita). De esta manera, si cualquiera de las tres copias del circuito falla, el error se puede corregir mediante las otras dos. Esta técnica es tolerante a un solo fallo.

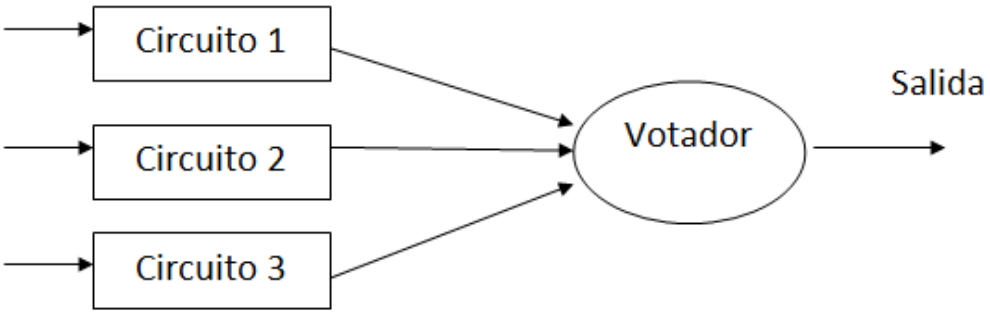


Figura 1.1: Sistema triplicado mediante TMR

Si se desea que esta técnica tenga una mayor tolerancia frente a SEUs, puede ampliarse añadiendo más módulos redundantes. En este caso el sistema se llama **redundancia n-modular (NMR, *n-modular redundancy***) [JoAd89].

Por ejemplo el QMT (*Quadruple Modular Rendundancy*) es igual que el TMR pero en este caso el número de módulos replicados es cuatro en vez de tres. De esta manera se analizan las cuatro salidas del circuito y mediante el voto mayoritario se selecciona la salida correcta, para compensar cualquier fallo que se presente en cualquiera de los cuatro circuitos [Sol91].

Sin embargo, la desventaja de la redundancia n-modular sigue siendo la misma: esta técnica, en cualquiera de sus variantes, genera un único resultado, por lo que al final del proceso debe haber un votador sin redundancia, lo que le hace sensible a fallos [JoAd89].

Esta técnica de redundancia no podría funcionar con una simple duplicidad de circuitos (sólo dos copias del circuito y el votador) ya que aunque se detectase un error por diferencia en las salidas no sabríamos cuál de las dos es la salida correcta.

En general, la expresión de la fiabilidad de un sistema con redundancia n-modular es:

$$R_{NMR} = \sum_{i=0}^{n} \left(\frac{N}{i}\right) * (1 - R_M)^i * R_M^{(N-i)}$$

Donde, N es el número de módulos redundantes y n es el número de fallos que puede detectar = (N-1)/2.

Por lo tanto, en un sistema TMR, la probabilidad de que el conjunto funcione, suponiendo que no falle el sistema de votación, es la suma de las probabilidades de tres módulos funcionando y dos módulos funcionando (ya que, como hemos dicho antes, sólo es tolerante cuando hay fallo en una de las copias o en ninguna), es decir:

$$R_{TMR} = R_M^3 + 3R_M^2 * (1 - R_M) = 3R_M^2 - 2R_M^3$$

Las ventajas de este método son la recuperación autónoma e inmediata de un circuito afectado por un SEU, alta tolerancia a los SEUs y que la conversión de un sistema no redundante a uno redundante es muy sencilla.

Sin embargo, entre las desventajas podemos encontrar que esta técnica es muy costosa

tanto en área como en energía y que no es tolerante a fallos en el votador. Esto puede corregirse incorporando también redundancia en dicho detector como se muestra en la figura 1.2 [KaKa05].

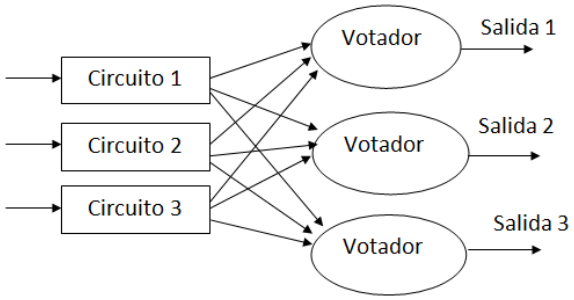


Figura 1.2: TMR con votador triplicado

El segundo tipo de técnicas de redundancia pasiva que mencionaremos es el **TMR selectivo**, el cual es una mejora del TMR para reducir coste. En esta técnica se triplica sólo una parte del circuito.

Mediante esta técnica, el diseñador debe identificar qué regiones del diseño es más conveniente proteger haciendo un estudio previo de las regiones que son más o menos críticas en el sistema, para así proteger en primer lugar las que lo son más [Pra06].

Las ventajas de esta técnica son que hay menos penalización en área, ya que en el TMR se triplica el circuito entero, mientras que con el TMR selectivo hay dos copias reducidas del circuito y una del circuito original, tal y como se indica en la figura 1.3. Esta técnica reduce los costes de protección.

En cuanto a las desventajas, esta técnica puede introducir retardos adicionales [Ka05] y, además, quedan desprotegidas las señales de reloj y de reset, así como la lógica interna del dispositivo afectada por los SEFI (*Single Event Functional Interrupt*) [Ber10].

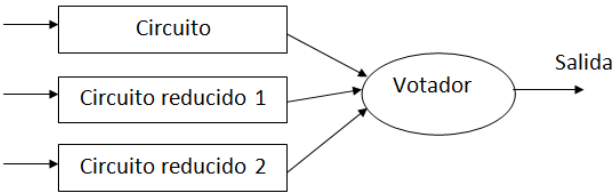


Figura 1.3: TMR selectivo

Como menos significativas tenemos la *Block Triple Modular Redundancy* (BTMR) y la *Distributed Triple Modular Redundancy* (DTMR). La primera necesita retroalimentación para corregir los errores y por lo general no se puede aplicar la corrección interna desde las salidas del votador, así que los errores no se corrigen y por tanto no es una técnica efectiva [Ber10]. La segunda, es igual que el TMR global pero sin replicación de la señal de reloj [Ber10].

Redundancia activa o dinámica

Con otro enfoque, tenemos la redundancia activa o dinámica en la que se consigue la tolerancia a fallos detectando el error y corrigiéndolo, sustituyendo el módulo erróneo por otro. Esta técnica requiere que el sistema se reconfigure para continuar funcionando correctamente. La desventaja, es que la nueva configuración tardará un tiempo en establecerse, por lo que estos sistemas deben permitir un funcionamiento erróneo durante un corto período de tiempo.

Un método de redundancia activa, que es capaz de detectar un fallo pero no de localizarlo, es **la duplicación con comparación.** Otro de los defectos de esta técnica es que no detecta los fallos ocurridos en el votador [JoAd89]. Un esquema de esta técnica se muestra en la figura 1.4.

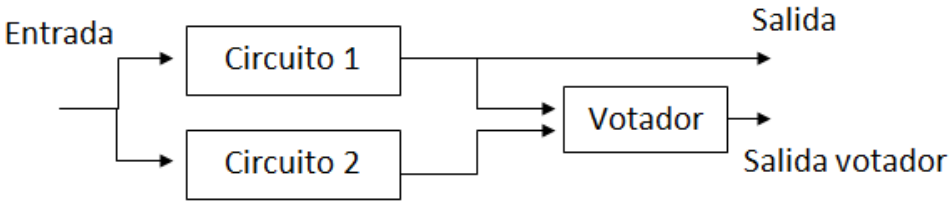


Figura 1.4: Funcionamiento del método de duplicación con comparación

Otro método de redundancia activa son los **repuestos en espera,** técnica que se muestra en la figura 1.5. Esta técnica consiste en tener el módulo crítico replicado, sin embargo, sólo uno está en funcionamiento en cada instante, el resto están inactivos en espera. Cuando un módulo falla, se pone en marcha el repuesto, por lo que un sistema con N módulos tolera N-1 errores. Los detectores de error seleccionan la entrada activa del multiplexor.

Una ventaja de este método es que un sistema con N módulos idénticos puede proporcionar mucha tolerancia a fallos con un número de repuestos bastante inferior a N, ya que no es necesario tener un repuesto por módulo.
Por otra parte, tiene las desventajas de que no es tolerante a fallos en el multiplexor y que es un sistema más complejo y costoso [Ber10].

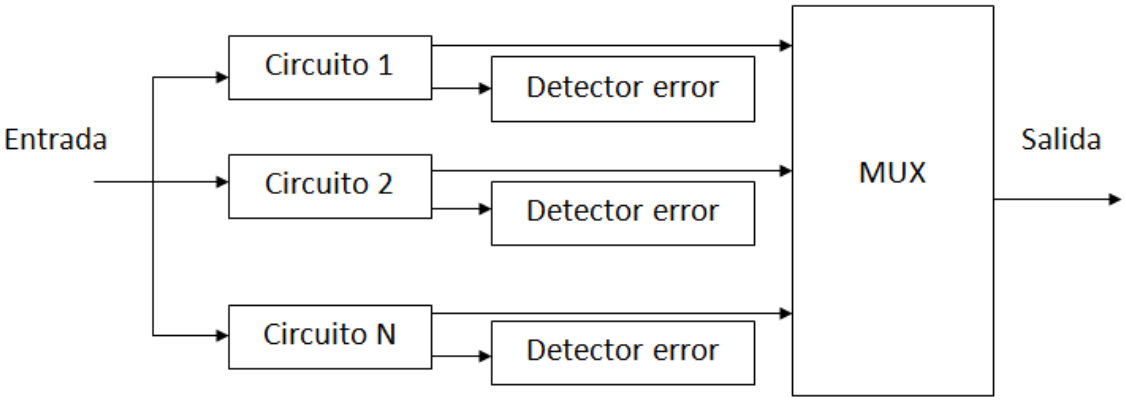


Figura 1.5: Método de los repuestos en espera

Esta técnica puede aplicarse de dos formas:

- ✓ En **frio**: los módulos de reserva no están operativos hasta que no son seleccionados por producirse algún fallo en el módulo activo, lo que nos lleva a que durante un tiempo el sistema no está disponible a cambio de minimizar el consumo eléctrico y conservar mejor los módulos en reserva (técnica usada en satélites).
- ✓ En **caliente:** los módulos en reserva están continuamente en funcionamiento para así minimizar el tiempo de recuperación del error.

Métodos híbridos

Finalmente tenemos los métodos híbridos. En ellos se emplea el enmascaramiento de fallos, pero, el sistema también puede reconfigurarse en caso de fallo.
Diferentes técnicas de estos métodos son la redundancia n-modular con repuestos, la redundancia con autoeliminación, o la arquitectura triple-dúplex [Ber10].

La **redundancia n-modular** con repuestos, ilustrada en la figura 1.6, consiste en reforzar la redundancia n-modular con módulos de repuesto adicionales. De este modo, se compara la salida de cada módulo con la salida del votador y, si no coinciden, se deduce que el módulo está fallando y se sustituye por un repuesto. La red de conmutación debe dar la posibilidad de reemplazar cada uno de los módulos por uno de los repuestos.

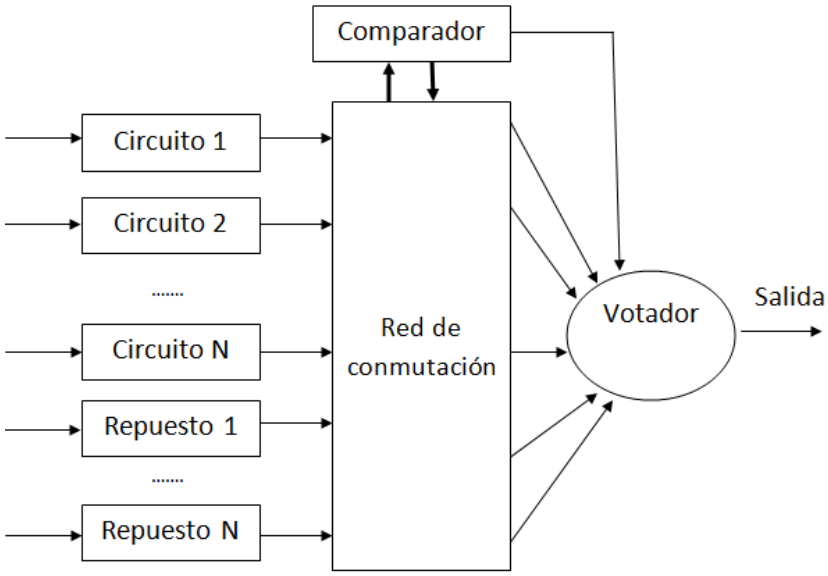


Figura 1.6: Esquema de principio de la redundancia n-modular con repuestos

En cuanto a la **redundancia con autoeliminación**, consiste en modificar la redundancia n-modular para comparar la salida de cada módulo con la salida del detector de mayoría, como podemos ver en la figura 1.7. En caso de que difieran, ese módulo se invalidará por lo que su salida no entrará en el votador.

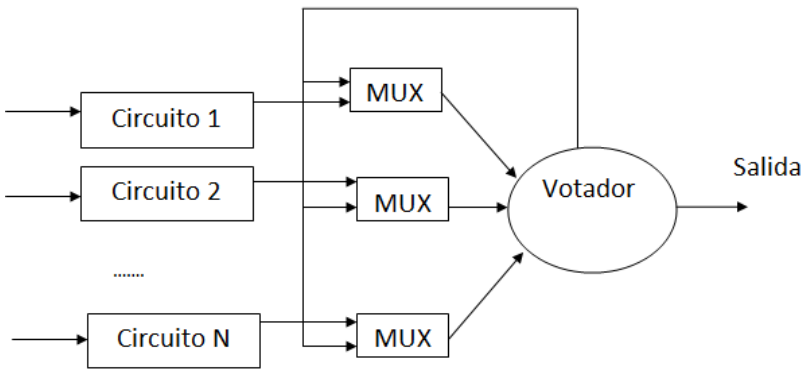


Figura 1.7: Esquema de principio de la redundancia con autoeliminación

Por último, **la arquitectura triple-dúplex**, mostrada en la figura 1.8, consiste en emplear el método de duplicación con comparación para detectar fallos en los módulos y, cuando hay diferencia en el resultado, se excluye la salida del comparador del votador de la redundancia triple modular.

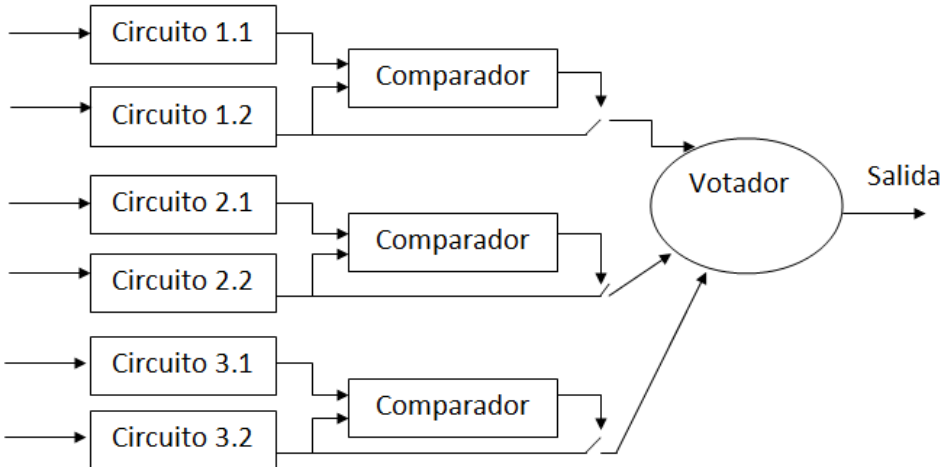


Figura 1.8: Redundancia híbrida con arquitectura triple-dúplex

1.2.1.2. Redundancia software

Este tipo de redundancia consiste en añadir en los programas líneas de código adicionales para evitar errores, detectándolos o corrigiéndolos. Este código suplementario puede tener diferentes funciones, tales como evitar que los datos se salgan de rango, prevenir errores aritméticos… [Ber10].

Las variantes de redundancia software son:

- **Pruebas de consistencia,** que se basan en comprobar si una magnitud sale o no fuera de unas cotas previamente establecidas, de manera que si se sale se puede deducir la existencia de un error. Un ejemplo de magnitud es el rendimiento. Si midiéndolo por software se observa que baja de forma injustificada, será un indicio de fallo.
- **Pruebas de hardware,** que consisten en habilitar procesos que se dediquen a verificar el hardware.

- **La programación con N versiones,** que consiste en diseñar y codificar los módulos software N veces por N equipos de programadores, independientes unos de otros y aplicar métodos similares a los de redundancia hardware pasiva explicada anteriormente.

1.2.1.3. Redundancia informacional

Este tipo de redundancia consiste en añadir información adicional a los datos para permitir la detección o corrección de errores. Aquí podríamos incluir métodos como los **códigos de paridad** o los **checksums** [Ber10].

1.2.1.4. Redundancia temporal

Esta técnica consiste en emplear un pequeño software que repita los cálculos varias veces para comparar los resultados y ver si existe alguna diferencia [Ber10].

Este método tiene una importante ventaja sobre los anteriores: no precisa ningún hardware o software adicional por lo que en coste es menor. Sin embargo, a cambio se necesita un tiempo extra, por lo que se suele utilizar en circuitos cuya ejecución no sea larga [JoAd89]

1.2.2. Plataformas de inyección de errores existentes en la literatura

Es necesario validar el comportamiento de nuestras técnicas de protección de circuitos para ver si son efectivas. Para ello, sería necesario mandar los circuitos protegidos con las técnicas diseñadas al espacio y ver cómo se comportan, sin embargo, esto es muy costoso en tiempo y dinero.

Por ello, son necesarias plataformas que emulen el comportamiento de la incidencia de un SEU sobre un circuito diseñado en la FPGA. De esta manera, podemos probar la validez de las técnicas de protección diseñadas de una manera mucho más rápida y menos costosa.

A continuación se presentan cuatro plataformas de inyección de errores: FLIPPER [Ald07], desarrollada por el Instituto Nacional de Astrofísica (INAF-ISAF) de Milán, FT-UNSHADES [St08] y los dos sistemas presentados por el grupo CAD, en la Universidad Politécnica de Túrin [StVi07], [BaStVi08].

1.2.2.1. FLIPPER

Es una plataforma desarrollada para emular SEUs escribiendo en la memoria de configuración de una FPGA basada en SRAM, en este caso una Virtex-II Pro. Se utilizan dos FPGAs, una para colocar el dispositivo bajo prueba y la otra para controlar el experimento inyectando errores en la memoria de la FPGA que contiene el diseño a testear.

Sin embargo, esta plataforma tiene como desventajas un alto coste de implementación, debido a que son necesarias dos FPGAs y un diseño ad-hoc de la plataforma, y es difícil de portar a otros tipos de FPGAs debido al mapeado fijo de elementos de E/S. Además, es una plataforma intrusiva, ya que inyectan *bitflips* en todo el área de la FPGA, como si el diseño ocupara todo en vez de sólo la zona que realmente ocupa.

1.2.2.2. FT-UNSHADES

Esta plataforma utiliza 2 FPGAs y un PC. Inicialmente se desarrolló como un emulador de hardware para inyectar y analizar el efecto de SEUs en diseños VLSI *(Very Large Scale Integration*) [St08].
Posteriormente, se presentó una extensión de la plataforma llamada FT-UNSHADES-C, que emula SEUs en la memoria de configuración de una FPGA Virtex-II Pro. En esta plataforma es necesario implementar dos copias del diseño: una con el circuito sin errores y otra en la que se emulan los SEUs para el test.
Esta plataforma tiene como ventaja que alcanza un alto rendimiento, sobre todo en grandes inyecciones.

Por otra parte, tiene como desventajas el consumo de muchos recursos lógicos, poca portabilidad a otros dispositivos debido a su diseño ad-hoc para la plataforma Virtex-II

Pro y que sólo inyecta bitflips en los bits de configuración que modifican el contenido de los elementos secuenciales del circuito bajo test, por lo que no se puede considerar una prueba completa ya que no inyecta en todos los bits de la memoria de configuración de la FPGA.

Además, también es un sistema muy intrusivo ya que se colocan dos copias del circuito en la misma FPGA, sin ninguna restricción, por lo tanto el diseño probado puede diferir mucho del real.

1.2.2.3. SISTEMAS PRESENTADOS POR EL GRUPO CAD

Este grupo presenta dos plataformas de inyección de errores para FPGAs Xilinx™ Virtex-II.

Por una parte, la plataforma presentada en [BaStVi08] es una arquitectura basada en un microprocesador similar a NESSY. Este sistema cuenta con un microprocesador Power PC, una memoria empotrada, un controlador para gestionar el puerto de configuración para las inyecciones en el circuito y el propio circuito a testear implementado como un periférico conectado al bus del sistema.

La ventaja de esta plataforma es que logra un alto rendimiento ya que con la inyección de un solo *bitflip* el retardo introducido por los procedimientos que manejan el puerto de configuración es mínimo.

Sin embargo, la desventaja es que es muy intrusivo, ya que se asignan en la FPGA el circuito bajo test y el hardware necesario para llevar a cabo el experimento sin ninguna restricción en la colocación y rutado del circuito a testear. Por tanto, la colocación final del circuito a testear depende en gran medida de la localización del resto del sistema.

Por otro lado, la plataforma presentada en [StVi07] es una mejora de [6], ya que garantiza la no intrusividad. Para ello, se aísla el circuito bajo test en una FPGA mientras que un PC controla la inyección de SEUs a través del puerto JTAG. Como consecuencia de ello, este sistema no introduce ninguna modificación en el circuito bajo test, ni en la asignación de pines, ni en la aplicación software que se ejecuta en ella.

Sin embargo, la desventaja de esta plataforma es una gran pérdida de rendimiento debida al método de inyección, lo que limita su aplicación para circuitos grandes.

Como ninguna de las plataformas anteriores cumple los requisitos de eficiencia y no intrusividad, decidimos utilizar la plataforma NESSY desarrollada dentro del equipo de investigación GHADIR, y que presentaremos en el capítulo 3.

1.3. Objetivo de este proyecto

El objetivo de este proyecto es el desarrollo de técnicas para la protección de circuitos basadas en TMR. Para comprobar su eficiencia se realizará una inyección de errores sobre los mismos, utilizando una plataforma previamente desarrollada llamada NESSY.

Para ello, hemos ampliado la plataforma de inyección de errores NESSY desarrollada sobre FPGAs del tipo Virtex 5, hacia un nuevo sistema que sea capaz de triplicar automáticamente el circuito introducido y quedarse con la salida correcta. Además, la reconfiguración del circuito erróneo será automática. Los cambios introducidos en NESSY serán detallados a lo largo de la memoria.

Con estas modificaciones hemos obtenido unos resultados que serán expuestos al final de este trabajo con los que queremos demostrar que con una protección mediante TMR, la cual no supone demasiado coste, la disminución de errores de un circuito es importante respecto a la inyección sin protección alguna.

Capítulo 2: Entorno de desarrollo

2.1. Entorno hardware

Todos los avances desarrollados en este proyecto están diseñados para funcionar sobre un dispositivo dinámicamente reconfigurable, concretamente sobre una FPGA XC5VLX110T de la familia Virtex 5, la cual está empotrada en una placa de prototipado XUPV505-LX110T (figura 2.1).

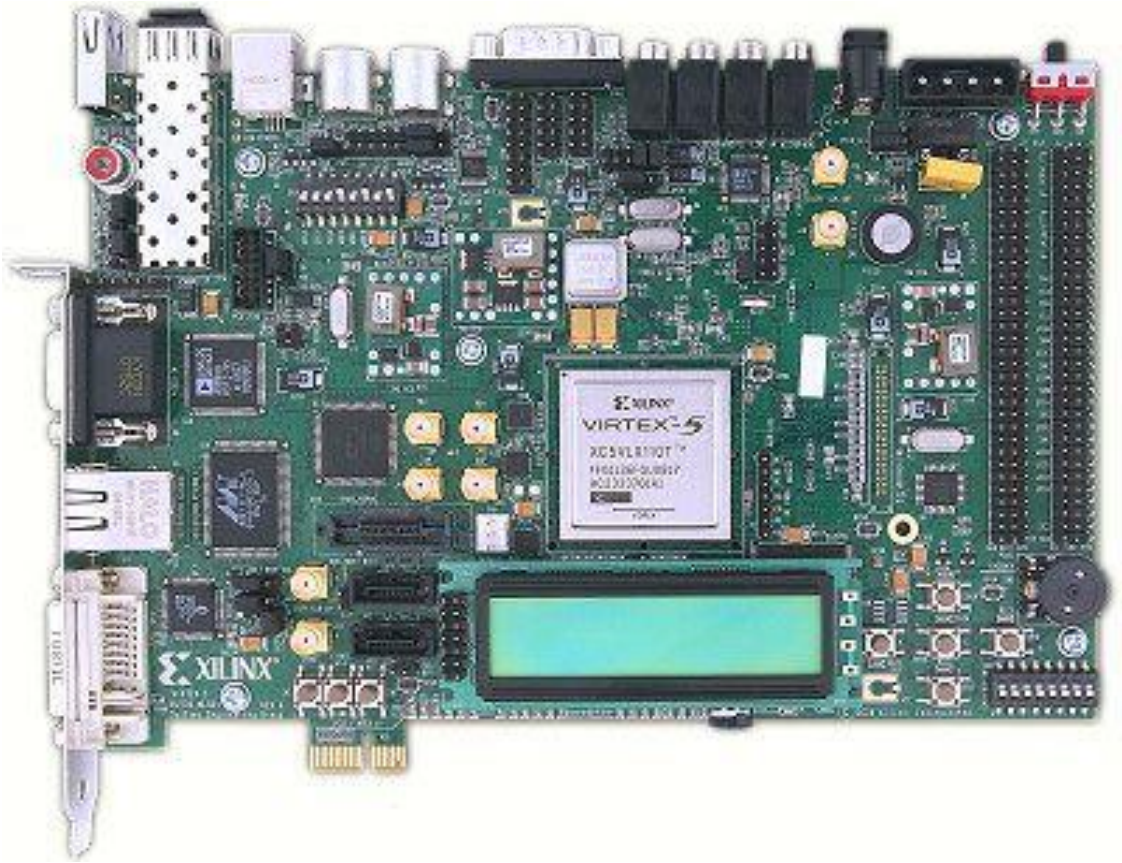


Figura 2.1: FPGA Virtex 5 empotrada en placa de prototipado XUPV505-LX110T

2.1.1. Placa de desarrollo XUPV505-LX110T

Este tipo de placas de desarrollo aportan componentes periféricos a la FPGA. En esta ocasión sólo hemos utilizado los necesarios para la comunicación entre la misma y el PC; es decir, el puerto JTAG y el puerto serie de la UART usando el protocolo RS232 además de la memoria DDR que utilizaremos para almacenar datos.

En la figura 2.2 podemos ver las diferentes conexiones de la mencionada placa de desarrollo con la Virtex 5.

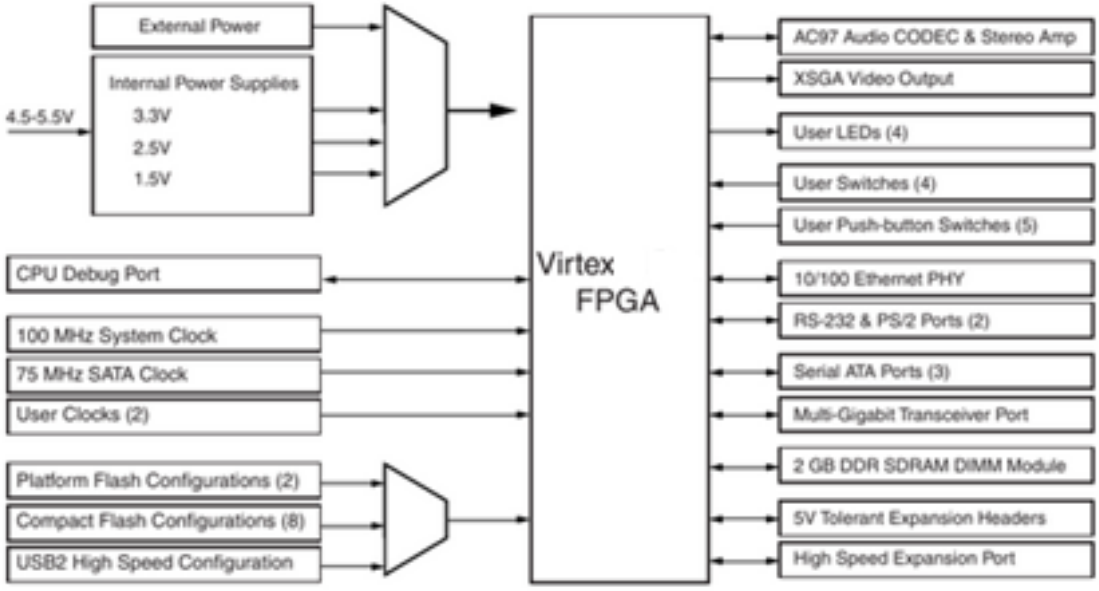


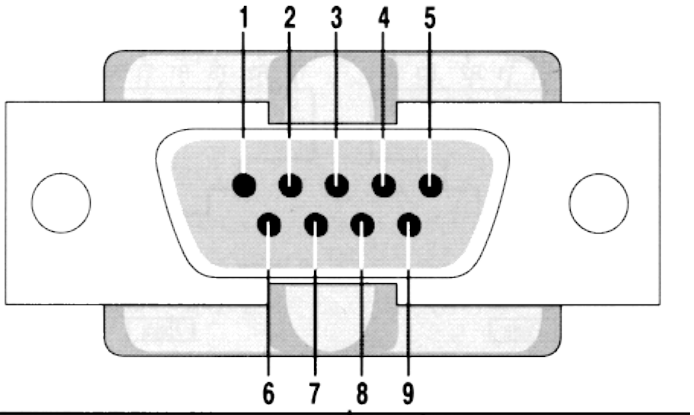
Figura 2.2: Conexiones XUP con Virtex 5

2.1.1.1. UART

Se trata de un protocolo de comunicación cuyas siglas se refieren a *Universal Asynchronous Receiver-Transmitter*. Es un protocolo muy usado para controlar los puertos y dispositivos serie. Sus funciones principales son manejar las interrupciones de los dispositivos conectados al puerto serie y convertir los datos que se encuentran en formato paralelo, transmitidos al bus de sistema, a datos en formato serie, para que puedan ser transmitidos a través de los puertos y viceversa.

Una UART toma bytes de datos y transmite los bits individuales de forma secuencial. Este tipo de transmisión requiere un menor coste de implementación con respecto a la transmisión en paralelo ya que utiliza un único pin de datos para envío y otro para recepción. La conversión de los datos de un formato a otro se realiza mediante un registro de desplazamiento.

El estándar para señalización por voltaje que usaremos es el RS-232 (Recommended Standard 232) , una interfaz que designa una norma para el intercambio de una serie de datos en formato binario. Los pines correspondientes a dicho estándar se muestran en la figura 2.3



Pin	Signal	Pin	Signal
1	Data Carrier Detect	6	Data Set Ready
2	Received Data	7	Request to Send
3	Transmitted Data	8	Clear to Send
4	Data Terminal Ready	9	Ring Indicator
5	Signal Ground		

Figura 2.3: Pines de la interfaz RS232

2.1.1.2. JTAG

Una interfaz JTAG *(Joint Test Action Group)* es una interfaz especial de cuatro o cinco pines agregadas a un chip, diseñada de tal manera que varios chips en una tarjeta puedan tener sus líneas JTAG conectadas en daisy chain [xtp029].

Una sonda JTAG necesita conectarse a un solo "puerto JTAG" para acceder a todos los chips en un circuito impreso. Es un protocolo muy usado para configuración y test de circuitos.

2.1.2. FPGA

Las FPGAs (*Field Programmable Gate Arrays*) son circuitos electrónicos reconfigurables que contienen bloques de lógica cuya funcionalidad e interconexión puede ser configurada mediante una memoria.

De esta manera, cuando un diseño programado en una FPGA deja de ser útil puede ser borrado y reemplazado por otro nuevo, incluso a distancia, lo que lo hace especialmente interesante en aplicaciones espaciales.

En la figura 2.4 podemos ver las partes que constituyen su arquitectura básica

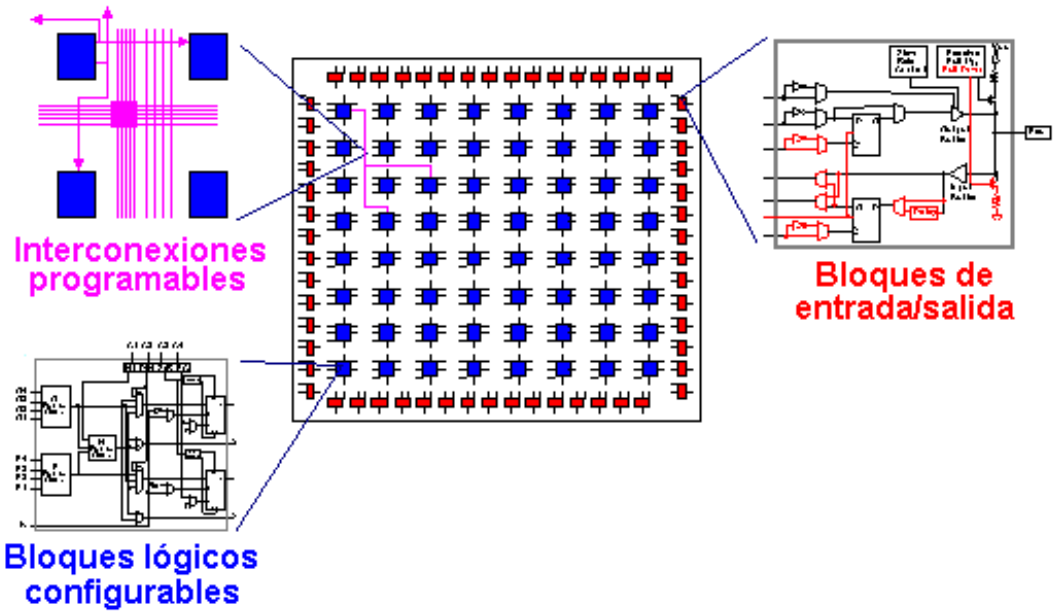


Figura 2.4: Arquitectura básica de una FPGA

2.1.2.1. Arquitectura Xilinx™ Virtex 5

Su arquitectura básica consiste en una matriz de elementos programables (CLBs, IOBs), canales de comunicación, gestores de reloj (CMTs) y bloques de memoria (BRAM) a cuya descripción procedemos a continuación:

- CLB (*Configurable Logic Block*)

  Son los principales elementos configurables de las FPGAs. Constan de una parte combinacional, que permite implementar funciones lógicas booleanas, más una parte secuencial, que permite sincronizar la salida con una señal de reloj externa e implementar registros.

El modelo de FPGA que usamos contiene dos slices en cada CLB. Podemos ver la muestra de uno de ellos en la figura 2.5. Como se puede observar cada uno de los slices está formado, entre otros componentes, por 4 LUTs (*look up tables*). Una LUT es un componente de celdas de memoria SRAM que almacena una tabla de verdad. Las direcciones de las celdas son las entradas de la función lógica que se quiere implementar, y en cada celda de memoria se guarda el resultado para cada una de las combinaciones de las entradas.

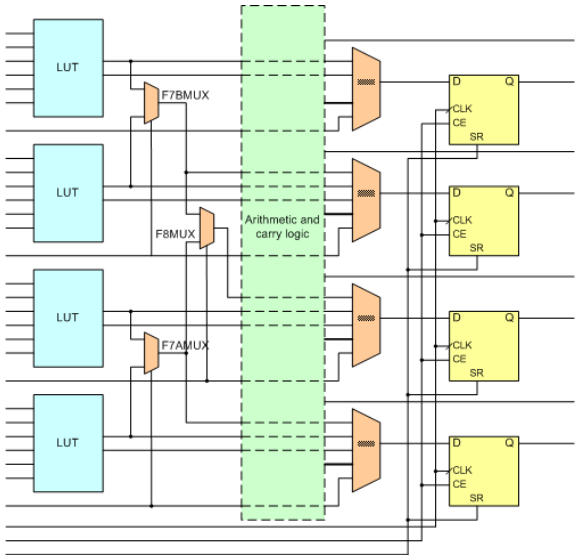


Figura 2.5: Estructura de los slices de las Xilinx™ Virtex 5

En la Virtex 5, las LUT pueden configurarse como funciones de 6 entradas o como dos funciones de 5 entradas (figura 2.6).

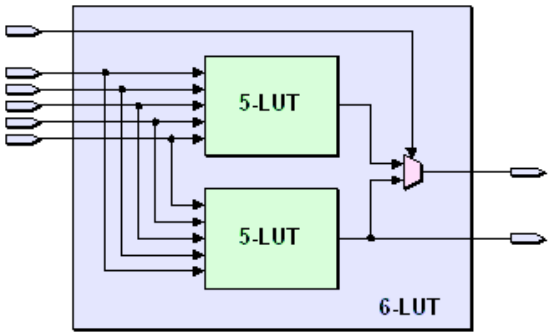


Figura 2.6: Estructura de las LUTs en las Xilinx™ Virtex 5

En concreto, las FPGAs Xilinx™ Virtex 5 poseen una matriz de 160x54 CLBs con un total de 17280 slices y un máximo de 1120 KB de memoria RAM entre todos los CLBs del dispositivo.

- Bloques BRAM (*Random-access Memory block*)

  Son bloques de memoria RAM empotrados en columnas a lo largo de la FPGA, con una capacidad de 36KB cada uno.

  En esta FPGA encontramos 148 módulos de este tipo, pudiendo ser utilizados como una sola memoria de 36KB o como dos memorias independientes de 18KB. Por tanto, contamos en total con 5328KB de memoria RAM distribuida entre todos los bloques.

- IOB (*Input/Output Block*)

  Comunican el dispositivo con el exterior y están situados rodeando la matriz de CLBs. Se pueden programar para que se comporten como puertos de entrada, de salida o entrada-salida según convenga.

- DSP *(Digital Signal Processor*)

  Se trata de módulos específicos para la realización de operaciones aritmético-lógicas. De este modo, la FPGA ahorra recursos a la hora de implementar circuitos que realicen este tipo de operaciones. Cada DSP está compuesto por un multiplicador de 25*18 bits, un sumador de 48 bits y un acumulador según podemos ver en la figura 2.7.

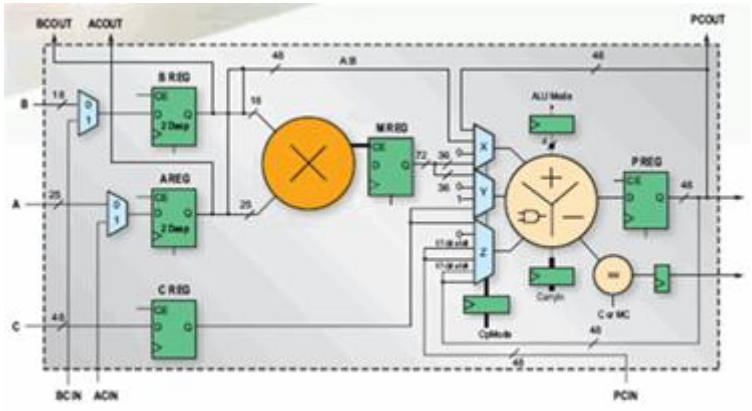


Figura 2.7: DSP48E en las Xilinx™ Virtex 5

En este modelo encontramos una sola columna de DSP48E con un total de 64 módulos.

- CMT (*Clock Management Tile*)
  Se trata de un módulo compuesto por 2 DCMs (*digital clock manager*) y un PLL (*phase-locked-loop*).

  Los DCMs son módulos que manipulan la señal de reloj, pudiendo multiplicar o dividir su frecuencia, recondicionar el *duty cycle* de la señal de reloj y eliminar los *clock skew*. Los componentes PLL son módulos analógicos con realimentación cuyo objetivo es que la señal de reloj que se genera sea lo más perfecta posible. En este caso en concreto contamos con 6 CMTs.

2.1.2.2. Puerto de configuración ICAP (*Internal Configuration Access Port*)

Se trata de un puerto interno que permite a la FPGA auto-reconfigurarse, o lo que es lo mismo, reconfigurarse a sí misma. De esta manera este puerto permite al procesador que la FPGA tenga mapeado leer y escribir la memoria de configuración de la FPGA en tiempo de ejecución. Por ello se trata de un puerto muy rápido.

Para entender bien la utilidad de este puerto debemos saber además lo que es la reconfiguración parcial; se trata de una característica que permite cambiar la configuración de una parte de la FPGA sin necesidad de resetear o reconfigurar el dispositivo al completo. La auto-reconfiguración sólo tiene sentido realizando una reconfiguración parcial.

2.2. Entorno software

En cuanto a los programas utilizados, la parte relacionada con la implementación sobre la FPGA ha sido desarrollada principalmente con la versión 12.1 de Xilinx™ ISE, EDK, SDK, iMPACT y PlanAhead mientras que para la programación de la herramienta en el PC hemos contado con NetBeans. En las siguientes subsecciones se describen en detalle cada una de estas herramientas.

2.2.1. Xilinx™ ISE (*Integrated Software Environment*) 12.1

Se trata de una herramienta para el diseño de sistemas digitales con la cual se pueden crear, verificar, simular y sintetizar diseños basados en FPGA. Estos diseños pueden generarse en lenguaje VHDL (que será el que utilizaremos), Verilog, o mediante esquemáticos o diagramas de estado. Tras generar el diseño, la herramienta lo sintetiza, es decir, transforma el código inicial a una *netlist* descrita en un lenguaje interno para un hardware concreto. Además, se definen restricciones temporales, asignación de pines de entrada/salida y optimizaciones de cada uno de los componentes a partir de la descripción del comportamiento del sistema.

El siguiente paso es la implementación dividido en tres fases: translate, map y place&route. Lo que hace este proceso es enlazar los módulos con las condiciones indicadas en el fichero de restricciones y reducirla a primitivas de Xilinx. Tras esto se adapta al dispositivo en el cual se va a volcar el diseño. Los procesos de los que disponemos en Xilinx™ ISE podemos verlos en la figura 2.8

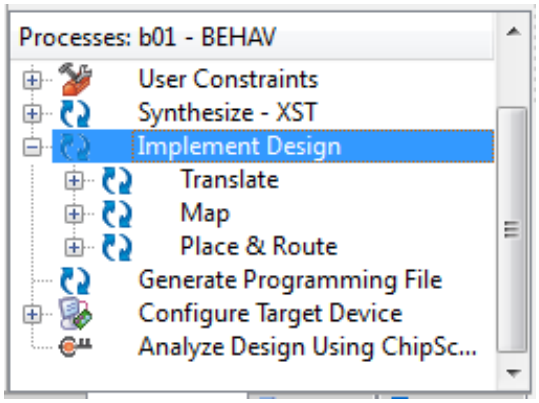


Figura 2.8: Cuadro de procesos de Xilinx™ ISE

El último paso es la generación del *bitstream*; fichero que contiene toda la información que se carga en la memoria para que la FPGA sea configurada por completo.

En nuestro proyecto, usaremos la herramienta para el desarrollo de los circuitos a probar en VHDL, para comprobar su correcto funcionamiento antes de crear el *bitstream* para la inyección de errores.

2.2.2. Xilinx™ EDK (*Embedded Development Kit*) 12.1

Esta herramienta permite diseñar sistemas complejos hw/sw y volcarlos sobre la FPGA. Este software integra a Xilinx™ ISE y usa sus procesos de síntesis e implementación. Permite también realizar aplicaciones software que irán mapeadas en memoria y serán ejecutadas por los distintos procesadores que implementemos en nuestro diseño.

Asimismo, proporciona un conjunto de módulos hardware de alto nivel para desarrollar los sistemas, a los que denomina *cores*. Permite desarrollar un sistema añadiendo y conectando dichos *cores*, los cuales pueden haber sido desarrollados por el propio usuario o por Xilinx™. Para cada uno de estos *cores* se debe seleccionar la región de memoria donde va a ir mapeado su software.

2.2.3. Xilinx™ SDK (*Software Development Kit*) 12.1

Es una versión modificada de Eclipse integrada con EDK, lo que lo hace muy atractivo para desarrollar y compilar las aplicaciones software en lenguaje C que se cargarán en el core del procesador de la FPGA, en nuestro caso el MicroBlaze.

Este software nos proporciona funciones de alto nivel adaptadas al hardware que utilizamos en el proyecto de EDK para poder utilizar los cores añadidos al sistema.

2.2.4. Xilinx™ iMPACT 12.1

Es un software empleado para programar las FPGAs. Permite la comunicación entre el PC y la misma a través del JTAG y tiene un funcionamiento bastante sencillo.

Una vez establecida la comunicación entre el programa y la FPGA sólo hay que seleccionar el dispositivo que queremos programar (ventana superior derecha de la figura 2.9) y elegir el *bitstream* que vamos a utilizar para que se cargue en el mismo.

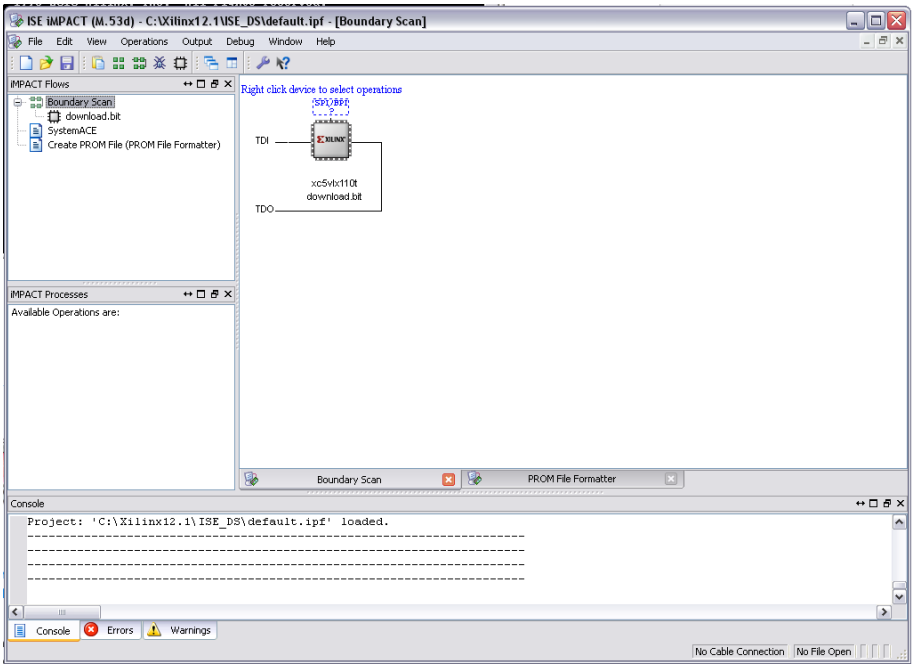


Figura 2.9: Interfaz gráfica de iMPACT12.1

Además se puede ejecutar en modo consola recibiendo un script; esta es la manera en la que este software se encuentra integrado en NESSY.

2.2.5. Xilinx™ PlanAhead 12.1

Este software permite crear fácilmente un *bitstream* parcial, se trata de un archivo *bitstream* que contiene información referente la configuración de la FPGA correspondiente con una parte del diseño a testear (no con todo el diseño completo). Esto resulta fundamental en nuestro proyecto, ya que la inyección de errores se acelera considerablemente si reconfiguramos sólo el componente a testear, en lugar de todo el *bitstream* del sistema.

A fin de poder utilizar esta herramienta para realizar reconfiguración parcial, ha sido necesaria una licencia especial de este software, la cual está disponible en la

Universidad. Gracias a la misma podemos decidir que módulos van a ser parcialmente reconfigurables y cuáles van a ser estáticos del sistema(es decir, no se van a modificar durante la ejecución o, lo que es lo mismo, no se verán afectados por la inyección de errores).

Los proyectos los creamos a partir de una netlist (fichero con extensión .ngc), resultado de la síntesis de EDK, y un fichero con extensión .ucf de restricciones para los pines de entrada/salida.

Tras seleccionar ambos archivos, nos permite elegir las coordenadas donde queremos situar nuestros componentes (área reconfigurable) lo que nos es muy útil para ajustar el área que queremos usar, haciendo que el porcentaje de utilización de la misma sea lo mayor posible, para que el resultado obtenido en la inyección sea fiable. La interfaz del mismo es bastante amigable, pudiendo verse fácilmente la situación de cada componente (figura 2.10) pero también se le puede proporcionar un script, forma en la que lo utiliza NESSY.

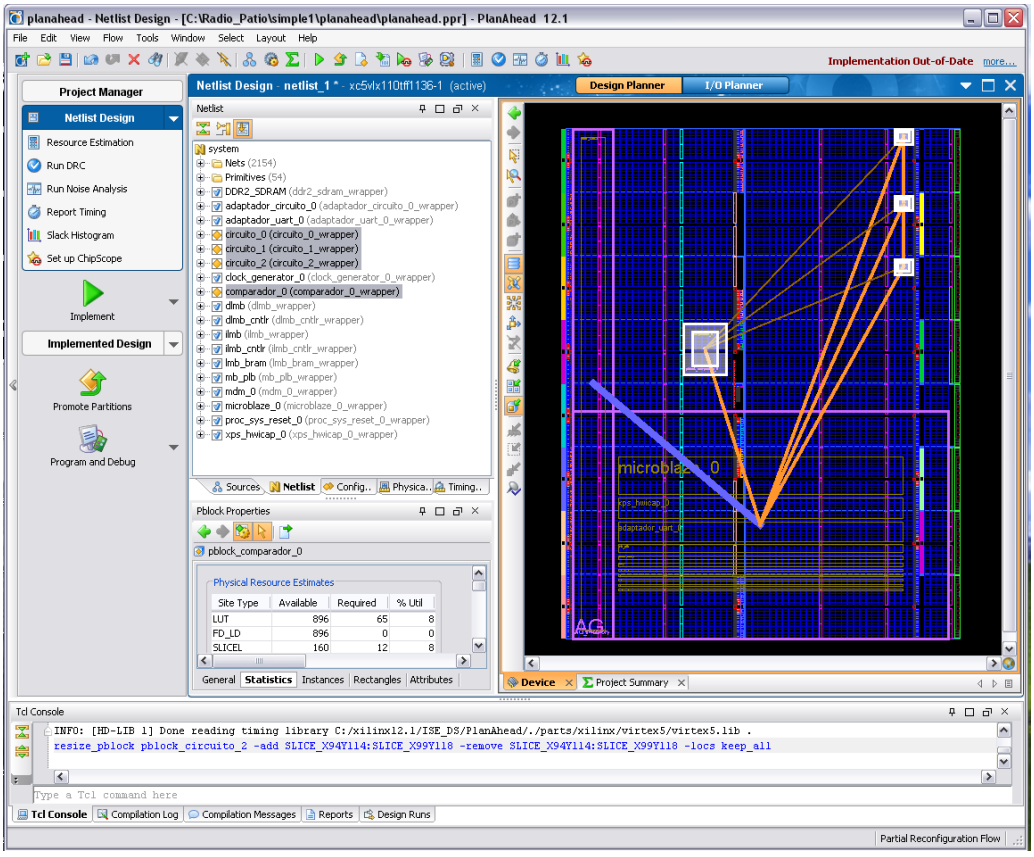


Figura 2.10: Interfaz gráfica de Xilinx™ PlanAhead 12.1

2.2.6. Xilinx™ FPGA Editor

Es una aplicación gráfica para la visualización y configuración de FPGAs.

Gracias a ella se puede ver de manera bastante clara cómo quedan ubicados los diferentes componentes del diseño dentro del dispositivo (figura 2.11). Se puede utilizar la aplicación antes de ejecutar la herramienta automáticamente para comprobar que no pisamos ningún cable estático del sistema, al situar nuestros circuitos en unas determinadas coordenadas.

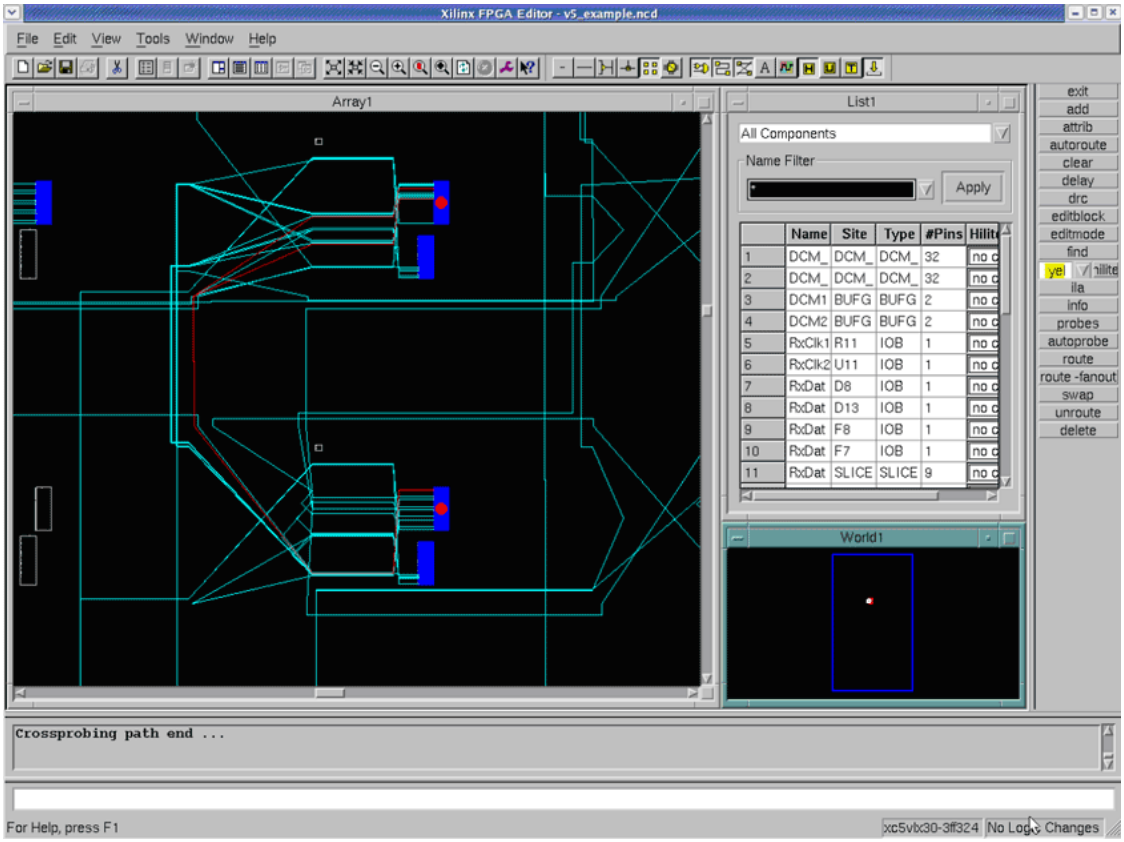


Figura 2.11: Interfaz gráfica de XilinxTM FPGA Editor

Este programa requiere una descripción proporcionada a través de un fichero con extensión .ngc, el cual contiene información acerca de la lógica que en un determinado diseño se asigna a los componentes (como CLBs y IOBs).

2.2.6. NetBeans

Es un entorno de desarrollo integrado libre para editar, compilar depurar y ejecutar programas escritos en código Java. Cuenta con muchas facilidades a la hora de generar una interfaz de usuario llamativa.

Capítulo 3: Funcionamiento de NESSY 5.0

Como ya se ha comentado en capítulos anteriores, nuestro sistema de protección de circuitos utiliza una plataforma de inyección de errores desarrollada en la Facultad de Informática de la Universidad Complutense de Madrid y presentada durante el trabajo fin de máster de Víctor Alaminos Benéitez bajo el título "Inyección de errores sobre FPGAs tipo Virtex 5"[Ala12].

El sistema ofrecía al usuario la posibilidad de emular los errores que se producirían con la colisión de partículas cósmicas sobre un determinado diseño cargado en la FPGA. Dichos resultados daban una orientación sobre qué regiones del circuito son más problemáticas y se deben proteger.

Para obtener estos resultados, la plataforma cumplía los siguientes requisitos:

- ✓ Aceptar cualquier diseño escrito en vhdl.
- ✓ Permitir al usuario elegir la zona de la FPGA donde quiere instanciar su circuito.
- ✓ Comunicarse con el PC para poder recibir *testbench*s y enviar los resultados de la inyección de errores. El *testbench* es el conjunto de datos de entrada que utilizará el circuito. Consiste en una matriz de datos en la que cada fila representa el conjunto de datos de entrada del circuito en un instante de tiempo (cada ciclo de reloj se pasará a la siguiente fila de datos).
- ✓ Ejecutar el *testbench* localmente en el circuito.
- ✓ Generar y almacenar en la placa un fichero *golden* (fichero con los resultados de una correcta ejecución del circuito) partiendo de la ejecución del *testbench* en el circuito original.
- ✓ Obtener y guardar en memoria local las salidas del circuito.
- ✓ Crear los ficheros *bitstream*s parciales modificados internamente desde la placa.
- ✓ Reconfigurar internamente sólo la parte de la FPGA donde está instanciado el circuito a testear.
- ✓ Comparar, desde la placa, las salidas del circuito con los resultados del *golden* para detectar errores.

- ✓ Poder almacenar un *testbench*, *golden* y *bitstream* parcial en la memoria de la placa.

A continuación, vamos a presentar tanto la arquitectura como el funcionamiento de dicha plataforma, para posteriormente poder explicar los cambios realizados sobre la misma con mayor detalle.

3.1. Hardware de NESSY 5.0

La arquitectura hardware que presentamos parte de una arquitectura anterior desarrollada en esta misma facultad en el trabajo de Fin de Carrera de Rubén Tarancón y Felipe Serrano para la inyección de errores sobre una Virtex 5 [SeTa11] .

Dicha arquitectura (figura 3.1) estaba formada por un subsistema software que se ejecutaba en el PC al que denominaremos *NessyJava* y un subsistema hardware implementado sobre una FPGA con un core MicroBlaze que ejecutaba un código al que denominaremos *NessyC*. La comunicación entre ambos subsistemas se establecía a través de dos canales: gracias al puerto JTAG para la configuración de la FPGA y todos sus componentes y la UART, mediante el protocolo RS232, para el envío y recepción de datos correspondientes al testeo de los circuitos.

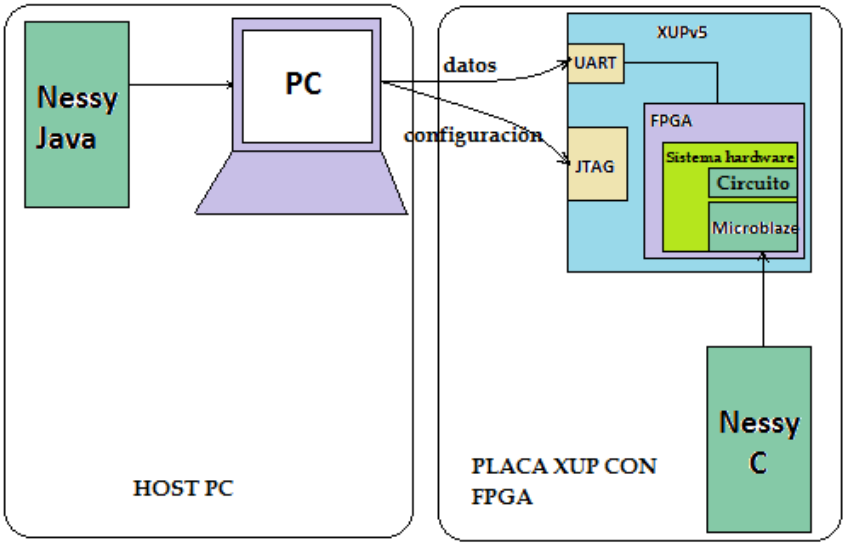


Figura 3.1: Arquitectura hardware NESSY 5.0

A continuación mostramos en detalle el subsistema hardware al que hacíamos referencia antes (figura 3.2); dicha arquitectura utilizaba cores propios de Xilinx™ EDK (color morado en la figura) y cores propios desarrollados para NESSY.

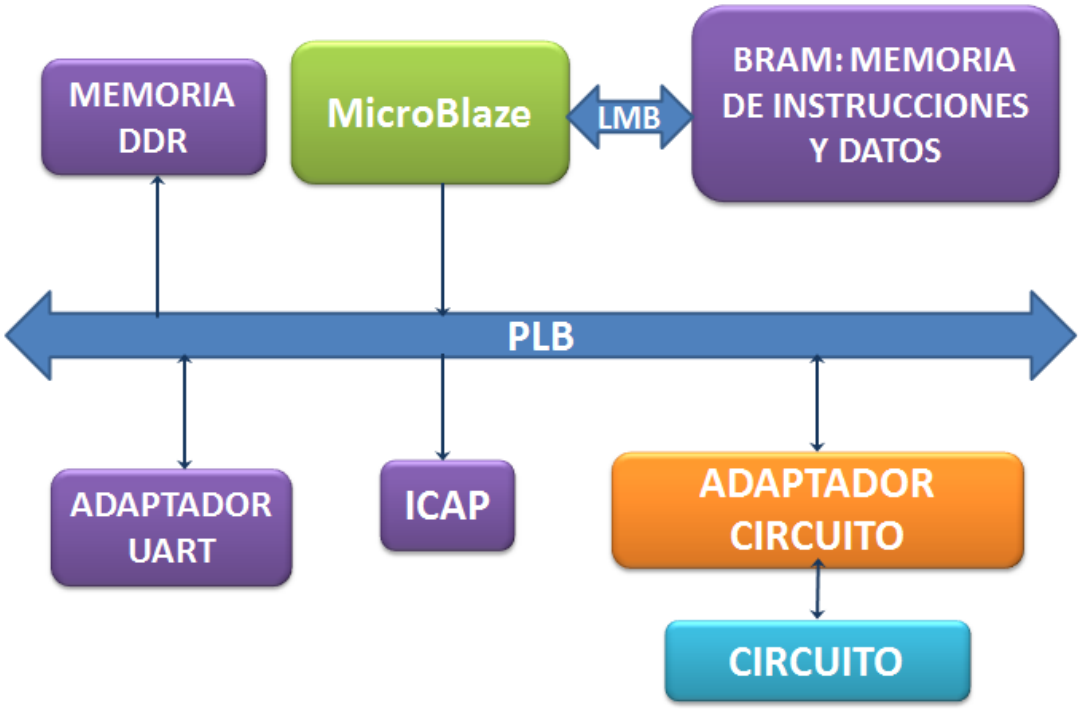


Figura 3.2: Subsistema hardware de la plataforma NESSY

La necesidad de un sistema hardware independiente surgió fruto del volumen de datos tan grande que maneja la aplicación NESSY. El motivo por el que se optó por la creación mediante EDK de un sistema independiente para la comunicación con el bus de datos PLB es limitar el protocolo RS232. Todos los módulos están conectados al bus PLB por el que se comunicarán a una frecuencia de reloj de 100 MHz. El adaptador circuito recibe dos entradas, una señal de reloj que suministra al circuito bajo test y una entrada de datos que administrará a cada instancia de circuito.

Vamos a describir los mencionados cores con más detalle en las siguientes subsecciones.

3.1.1. BRAM

Memoria RAM implementada dentro de la FPGA. En ella se encontraba mapeado el código fuente de *NessyC*.

3.1.2. MicroBlaze

Se trata de un *soft-processor* que implementa un conjunto de instrucciones reducido (RISC) optimizado para la ejecución en FPGAs de Xilinx™. Es un procesador altamente configurable, permite configurar diversas características parametrizables. Posee registros de propósito general de 32 bits, instrucciones de 32 bits con 3 operandos y 2 modos de direccionamiento y direcciones de bus de 32 bits. Podemos ver su estructura en la figura 3.3 [XiCo12].

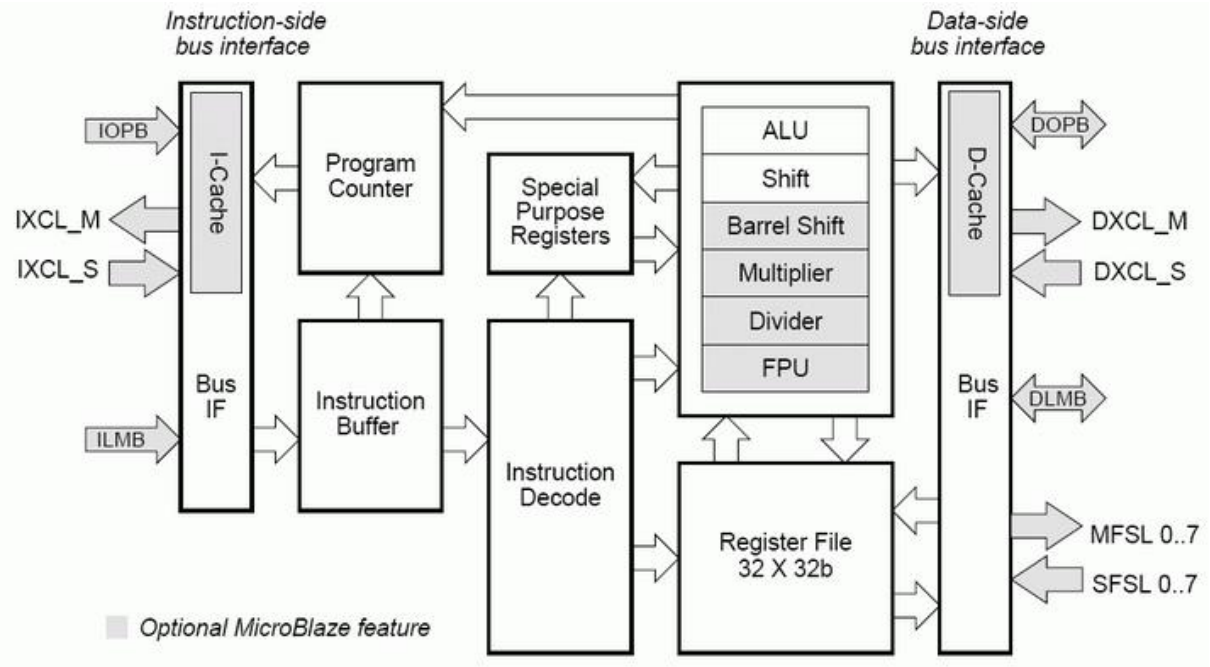


Figura 3.3: Arquitectura del procesador MicroBlaze

El MicroBlaze ejecutaba el programa *NessyC* mapeado en las memorias BRAM sirviendo de controlador del sistema.

3.1.3. Controlador DDR

La herramienta Xilinx™ EDK proporciona un controlador de memoria multi-puerto que permite acceder al chip de memoria DDR2 RAM existente en la placa XUP.

3.1.4. Controlador ICAP

A través del controlador del puerto ICAP se configuraba parcialmente la FPGA.
Este módulo estaba conectado al bus PLB (*Processor Local Bus*), por lo que podíamos comunicarnos con el ICAP a través del MicroBlaze escribiendo o leyendo a través de un conjunto de registros de configuración y datos

Para lograr la reconfiguración ser partía de un *bitstream* parcial enviado a través de la UART y almacenado en RAM local previamente. Para modificar el circuito de la FPGA, el MicroBlaze generaba un *bitstream* parcial modificado, que consistía en un archivo *bitstream* obtenido a partir del *bitstream* parcial del diseño original, al cual se le alteraba un bit que simulaba la inyección de un error, lo almacenaba en la BRAM y daba la orden de que iniciara la reconfiguración parcial.

3.1.5. Adaptador UART

Encargado de comunicar la placa con el exterior mediante el protocolo RS232. Este core posee una interfaz de comunicación con el bus PLB, de forma que mediante la escritura/lectura de diferentes registros, el MicroBlaze puede intercambiar datos con el exterior.

3.1.6. Adaptador circuito

Servía para poder conectar el circuito bajo test con el resto del sistema. Se utilizaba para que cualquier circuito que se quiera testear en la herramienta NESSY se pudiera conectar al bus PLB a través de dicho adaptador. Era un módulo genérico que tenía una entrada como señal de reloj para el circuito a testear, una entrada de 32 o 64 bits (según el diseño) como datos y una salida del circuito de 32 o 64 bits (según el diseño). Con este adaptador se podía trabajar con circuitos que tuvieran un reloj, una entrada y una salida de hasta 64 bits.

3.1.7. Circuito

Hacía de puente entre el sistema hardware y el circuito en el que se quería realizar la inyección de errores. Se trataba tan solo de una entidad en vhdl, con unas entradas y unas salidas generado por NESSY que adaptaban el circuito a testear introducido por el usuario al hardware de NESSY, convirtiendo el mismo en un archivo que la herramienta pudiera usar gracias a un parseador. Iba conectado al módulo Adaptador Circuito de forma bidireccional (figura 3.4) lo que le proporcionaba la comunicación con el resto del hardware.

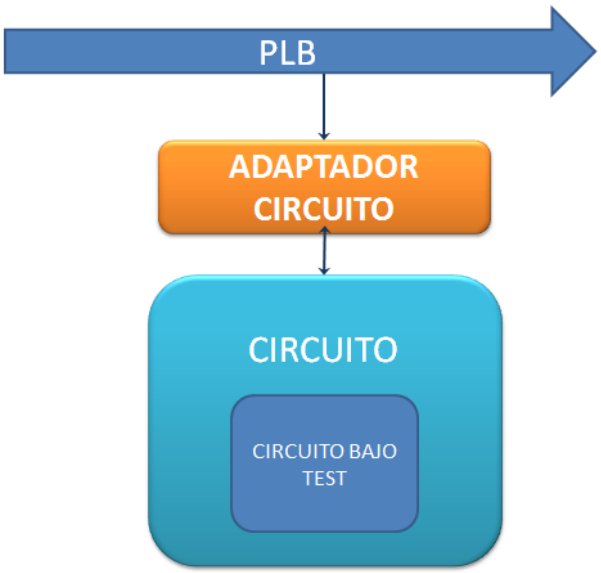


Figura 3.4: Detalle de la adaptación del circuito al sistema hardware

3.1.8. Buses PLB Y LMB

Por un lado, para la implementación del sistema hardware se utilizaba un bus PLB que conectaba el procesador con el resto de componentes del sistema como ya vimos en la figura 3.2.

Por otro lado, el bus LMB (*Local Memory Bus*) era un bus especial utilizado para conectar el MicroBlaze con los puertos de datos e instrucciones a dispositivos de alta velocidad.

3.2. Software de NESSY 5.0

3.2.1. *NessyJava*

El proceso de emulación de SEUs era controlado por el usuario mediante una interfaz gráfica que se ejecutaba en el PC y que fue desarrollada en lenguaje de programación Java. El aspecto de esta interfaz era simple e intuitivo (figura 3.5). [SeTa12]

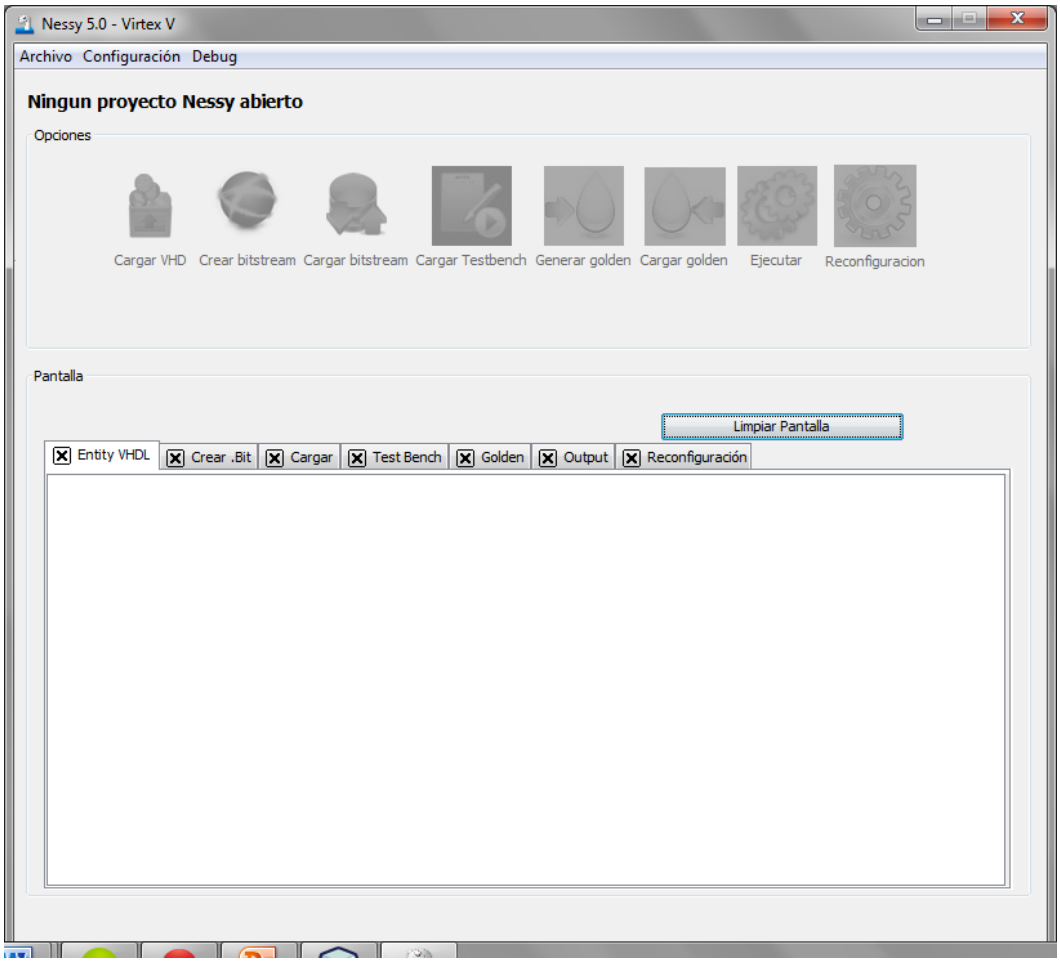


Figura 3.5: Interfaz gráfica de NESSY

El usuario podía ir ejecutando los distintos pasos en los que dividió el proceso de inyección desde esta interfaz, de manera que cuando se realizaba uno de estos pasos el programa lanzaba un comando a través del puerto serie, que era escuchado por el MicroBlaze de la FPGA y este ejecutaba la operación correspondiente. Dichos pasos se deben ejecutar secuencialmente, además no se debía empezar un paso sin haber acabado el anterior.

Para entender mejor la ejecución de estos pasos, vamos a ir explicando los mismos con un ejemplo, mostrando además la salida que produce cada uno de ellos. El circuito elegido para los test será el denominado b13.

ABRIR O CREAR PROYECTO

Para empezar a utilizar el programa, el usuario debía crear un nuevo proyecto, dándole el nombre y la ubicación deseada, o abrir uno ya existente. NESSY guardaba la información de un proyecto en un fichero con extensión .nessy, extensión creada especialmente para tal fin.

En este caso crearemos un proyecto al que denominaremos b13 (al igual que el circuito que testearemos en el mismo) y lo ubicaremos en el escritorio (figura 3.6).

Figura 3.6: GUI para la creación de un nuevo proyecto en NESSY

Al crear un nuevo proyecto se creaba una jerarquía con las siguientes carpetas y archivos:

- Archivo "<proyecto>.nessy" con la metainformación del proyecto.
- Carpeta "edk" con los archivos del proyecto EDK.
- Carpeta "salidas" donde se crean los .log.
- Carpeta "vhdl" donde se copiaban los archivos vhdl del proyecto.

CARGAR VHD

Para elegir el circuito donde se iba a realizar la inyección de errores se pulsaba el botón "cargar vhd" de manera que el archivo .vhdl seleccionado se parseaba obteniendo la información necesaria sobre la entidad (entradas, salidas y sus nombres), para adaptar el sistema hardware a dicho circuito a través del módulo *adaptador circuito.*

En este caso, tras seleccionar el circuito b13, NESSY nos muestra el nombre de la entidad, sus 11 entradas (figura 3.7) y sus 10 salidas de forma detallada y por último el número total de las mismas (figura 3.8).

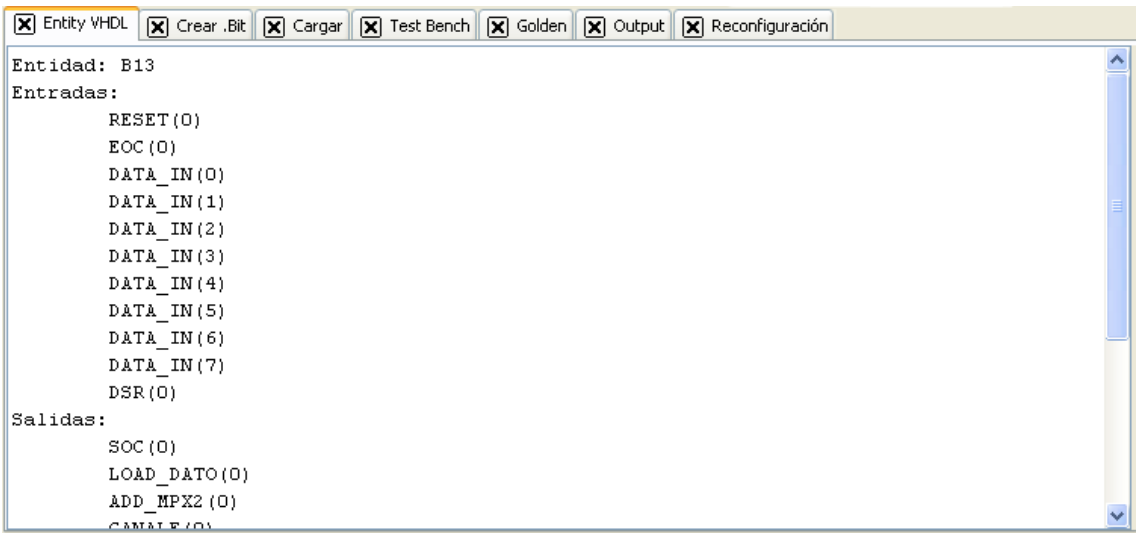


Figura 3.7: Información mostrada sobre una entidad al cargar su VHDL

Figura 3.8: Información mostrada sobre una entidad al cargar su VHDL

GENERAR BITSTREAM

Era el siguiente paso a seguir, al pulsar el botón con el mismo nombre que aparecía en interfaz nos encontrábamos con una ventana donde el usuario debía introducir las coordenadas inferior izquierda y superior derecha de la región donde se deseaba que el circuito fuera situado dentro de la FPGA (figura 3.5).

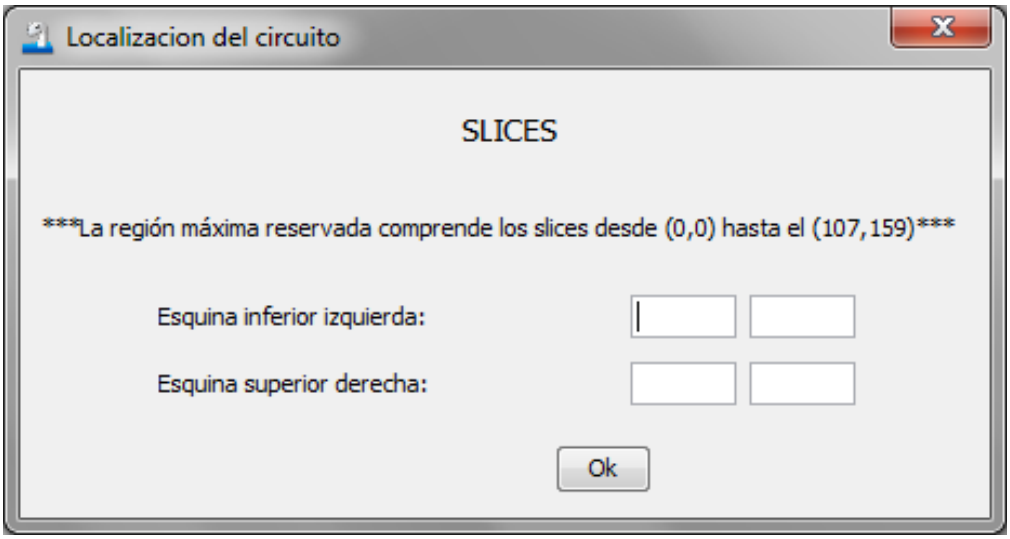


Figura 3.9: Selección de la posición donde se coloca el circuito en la FPGA

Tras la elección de las coordenadas, NESSY copiaba automáticamente el proyecto EDK que contenía todo el sistema hardware de la carpeta base a la carpeta donde se había creado el proyecto, copiando además los archivos VHDL del circuito a testear y el adaptador circuito correspondiente al número de bits que conforma la entrada/salida del circuito (el proyecto disponía de un adaptador para circuitos con E/S de menos de 32 bits y otro para circuitos con E/S de hasta 64 bits).

A continuación, NESSY modificaba el archivo .pao añadiendo una línea por cada archivo VHDL del circuito a testear. Finalmente se generaba el *bitstream* del sistema con el circuito a testear ya incorporado. Los pasos que NESSY realizaba para la ejecución de este último paso eran los siguientes:

- Generar netlist del sistema.
- Crear el script con las restricciones de área.
- Modificar el .ucf para el proyecto PlanAhead.
- Generar *bitstream*s totales y parciales con PlanAhead.
- Mapear el software en la BRAM con el programa data2mem.

CARGAR BITSTREAM

Durante este paso se configuraba la FPGA con el fichero *bitstream* generado por el EDK y el PlanAhead, haciendo uso del programa iMPACT, previamente descrito.

CARGAR TESTBENCH

Esto enviaba el testbench a NESSYC. Para ello se configuraba el puerto serie, asegurándose de que la comunicación con la FPGA era correcta.

En este ejemplo en concreto cargamos un *testbench* que hemos creado con 1000 entradas, cada una de las cuales está compuesta de los 11 bits correspondientes a la entrada del circuito. En la figura 3.10 podemos ver las entradas del *testbench* que había enviado NESSY a la FPGA numeradas.

Figura 3.10: Testbench introducido en la aplicación

GENERAR GOLDEN

Este paso enviaba a la FPGA la orden de generar el fichero golden, del que ya hemos hablado, a través del puerto serie.

En este caso NESSY muestra por la consola las salidas correspondientes a las entradas del *testbech* introducidas en el paso anterior. Podemos ver las últimas salidas del ejemplo que estamos ilustrando en la figura 3.11.

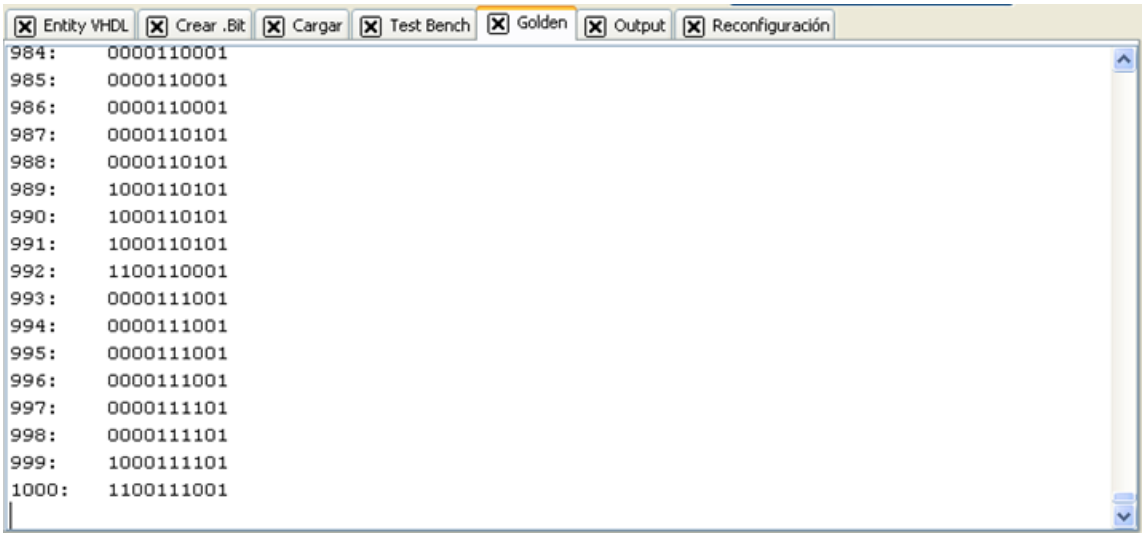


Figura 3.11: Golden generado por la aplicación

CARGAR GOLDEN

Se ejecutaba sólo en caso de tener el archivo *golden* previamente generado. Lo que hacía era leer el archivo de texto que lo contenía y enviaba éste por el puerto serie a la FPGA.

EJECUTAR

Ejecutaba el *testbench* almacenado en la memoria RAM de la FPGA sobre el circuito a testear y comparaba la salida obtenida con el *golden* previamente generado en el paso "Generar golden".

En nuestro ejemplo podemos ver que la aplicación muestra por consola el número de entrada del *testbench* correspondiente, seguido de la salida almacenada en el *golden*

para esa entrada, así como la salida que el circuito ha generado al ejecutar. Si ambas son iguales al final de la línea escribe OK, si por el contrario son diferentes, señala que ha habido un error. En este caso queda claro que la ejecución de todas las entradas del *testbench* ha sido correcta (figura 3.12).

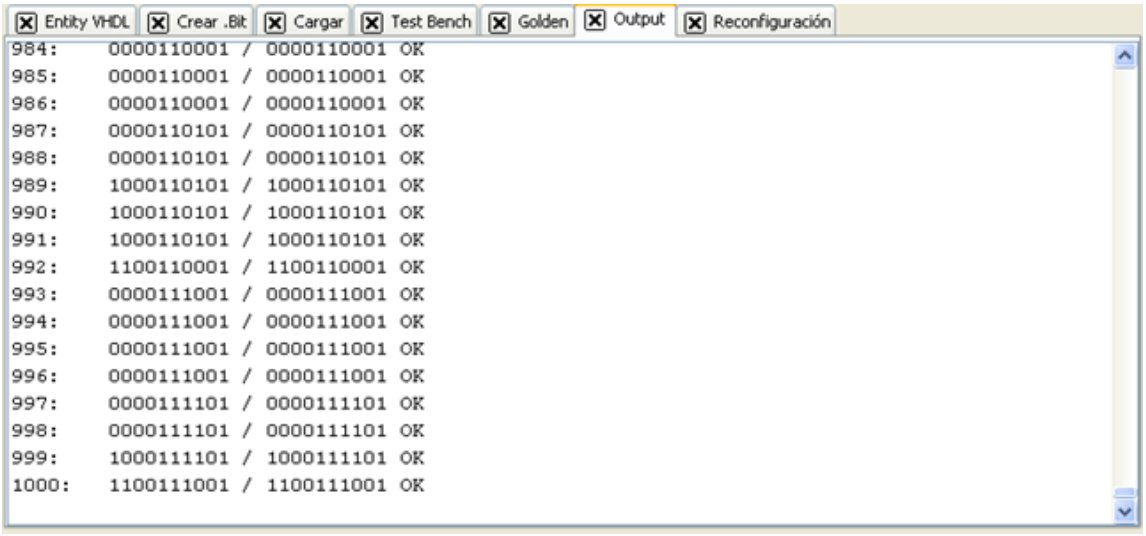


Figura 3.12: Comprobación de que las salidas al ejecutar son las mismas que las del golden.

RECONFIGURACIÓN

Era el último paso que se ejecutaba, la finalidad del mismo era emular SEUs, para ello se inyectaba un bitflip (cambio del valor de un bit de 1 a 0 o de 0 a 1) en todos los bits de la memoria de configuración asociados con el circuito a testear y se comprobaban las salidas generadas por el circuito al pasar el testbench con los resultados obtenidos previamente en el *golden*.

La comunicación por el puerto serie es extremadamente lenta por lo que era necesario limitar este tipo de comunicación entre el PC y la FPGA lo más posible. Es por este motivo por el cual la inserción de *bitflips* y el procesamiento de los datos recaía en el MicroBlaze de la FPGA, encargándose el PC únicamente de recibir los resultados de la inyección y procesarlos. Además, para insertar los *bitflips* en un tiempo mínimo se

hacía uso del ICAP (*Internal Configuration Access Port*) el cual sirve para reconfigurar internamente la FPGA sin necesidad de hacerlo por el puerto JTAG.

Al finalizar, se obtenían los resultados de dicha inyección de errores mostrándose tanto el tiempo transcurrido, como el número de reconfiguraciones realizadas, el número de reconfiguraciones erróneas y el porcentaje de reconfiguraciones erróneas respecto al total de reconfiguraciones realizadas.

Podemos ver en la figura 3.13 las estadísticas obtenidas para el ejemplo que hemos tomado, el porcentaje de reconfiguraciones erróneas para este circuito en concreto sería de 6,21%, habiéndose ejecutado la prueba en 5 minutos y 11 segundos.

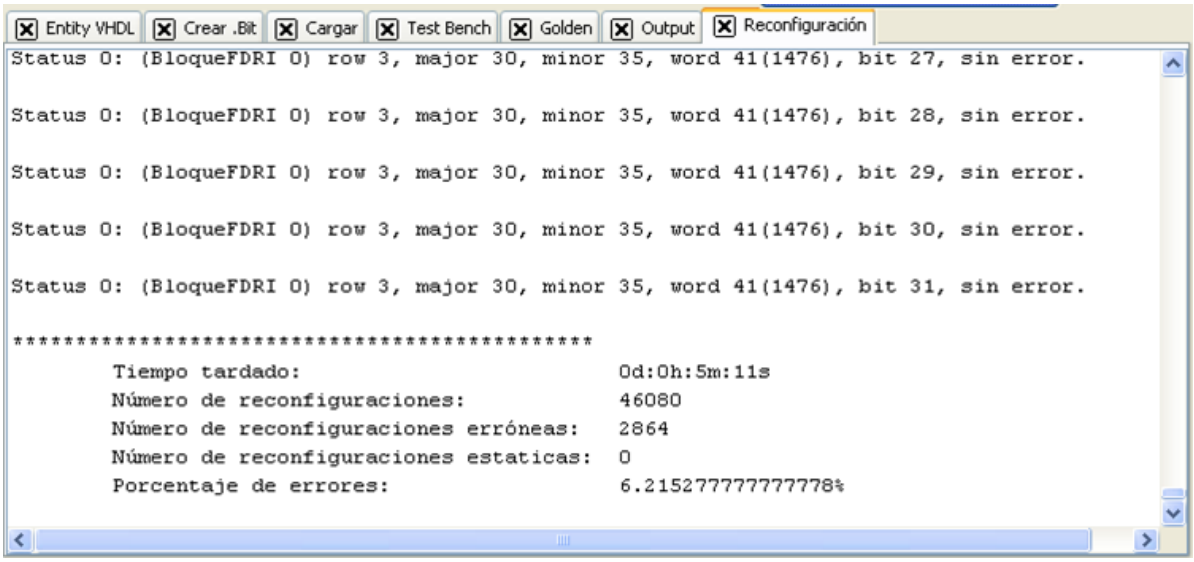


Figura 3.13: Resultados de una inyección

Además NEESSY generaba un archivo detallando lo ocurrido para cada bit en la inyección, así como otro archivo matlab para que la interpretación de dichos resultados fuera más fácil y poder comprobar de manera visual cuáles eran las zonas más vulnerables del circuito introducido.

CONFIGURACIÓN

En la pantalla principal existía una pestaña con opciones para la configuración del programa. La primera de dichas opciones era la de "configuración"; dado que para la ejecución de NESSY era imprescindible conocer la ruta donde están instalados algunos de los programas de Xilinx™, nos aparecía una ventana como la mostrada en la figura 3.14 para introducir las mismas:

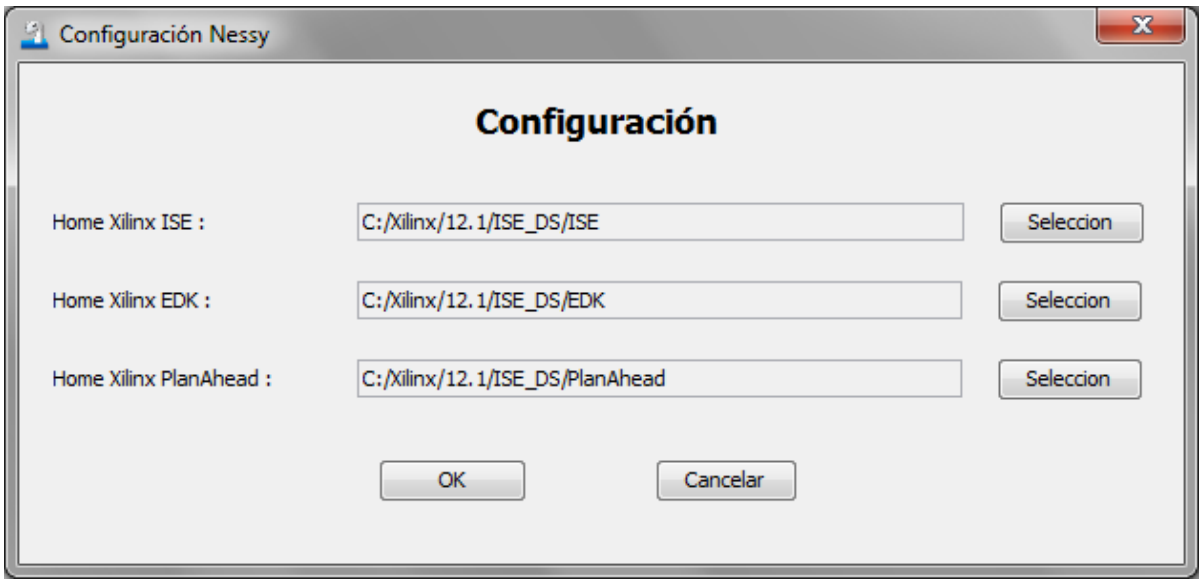


Figura 3.14: Interfaz de selección de las rutas de los programas de Xilinx™

Por otra parte, existía un archivo de configuración "config.properties" donde se podían configurar los siguientes parámetros para la aplicación:

- Proyecto por defecto.
- Creación del *bitstream* automáticamente, es decir, sin necesidad de pulsar el botón.
- Reconfiguración automática, para que la FPGA se configure automáticamente al iniciar la aplicación.
- Puerto serie por defecto.
- *Testbench* por defecto.

Al pulsar la opción "cargar fichero" de configuración estos parámetros se cargaban en la aplicación, lo que permitía un trabajo más rápido con un determinado proyecto.

Por último "configurar puerto serie" permitía elegir el puerto del PC al que se encontraba conectado el puerto serie de la FPGA (figura 3.15):

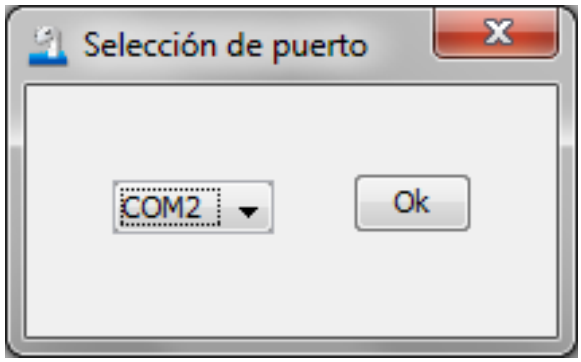


Figura 3.15: Interfaz de configuración del puerto serie

3.2.2. *NessyC*

Se trata del programa que se ejecutaba en el MicroBlaze y estaba mapeado en las memorias de la FPGA. Su función principal era estar escuchando el puerto serie a la espera de los comandos enviados desde *NessyJava*.

A continuación describimos los principales comandos que ejecutaba:

Cargar *testbench*: leía un *testbench* por la entrada de la UART y lo almacenaba en la memoria DDR.

Generar *golden*: leía los datos del *testbench* de la memoria DDR, los escribía en la entrada del circuito, ejecutaba éste y almacenaba las salidas de esta ejecución en el fichero *golden*, el cual se guardaba en la memoria DDR.

Ejecutar: leía el *testbench* colocando cada dato a la entrada del circuito y ejecutaba y comparaba las salidas del circuito con las del *golden*. En caso de que dichos datos no coincidieran, se notificaba qué bits habían fallado y para qué número de entrada del *testbench*.

Enviar golden: leía de la memoria DDR los datos correspondientes al *golden* y los enviaba por el puerto serie a NESSY.

Cargar golden: recibía por el puerto serie un *golden* obtenido durante una ejecución anterior y lo escribía en la memoria DDR.

Reconfiguración: se trata de la opción que más carga de trabajo tenía y más tiempo necesitaba para su ejecución. Dicho proceso se dividía en las siguientes etapas:

1. Recibir el *bitstream* parcial original del circuito por el puerto serie y almacenarlo en la memoria DDR de la placa.
2. Analizar el *bitstream* parcial para conocer su estructura.
3. Para cada uno de los bits de la memoria de configuración que se iba a testear ejecutar los siguientes pasos:
   3.1. Generar un *bitstream* parcial modificado (el cual incluía el *bitflip*) partiendo del original y guardarlo en las Block RAMs de la FPGA.
   3.2. Realizar una reconfiguración parcial del *bitstream* parcial modificado usando el puerto de configuración ICAP de la FPGA.
   3.3. Usar la función "ejecutar" para así probar el *testbench* con el circuito modificado y enviar los errores detectados a través del puerto serie.
   3.4. Restaurar el circuito original (sin errores), realizando la reconfiguración parcial del bitstream original.

Una vez se había realizado una inyección de SEUs completa, *NessyC* volvía al estado inicial a la espera de más comandos.

Comprobar comunicación: leía una palabra de la entrada del puerto serie y, si coincidía con el comando asignado para la sincronización (6), enviaba una respuesta predeterminada (0xf0f0f0f0) por el puerto serie. De esta manera se sincronizaban FPGA y computador, lo que era de gran importancia durante toda la ejecución.

Capítulo 4: NESSY 6.0, una plataforma con protección de circuitos

Basándonos en todo lo explicado hasta ahora acerca de las distintas técnicas de protección de circuitos, vamos a centrar nuestro trabajo de investigación en el estudio y obtención de resultados a partir de la técnica del TMR, técnica de redundancia hardware. Hemos seleccionado esta técnica debido a que tiene una alta tolerancia a los SEUs, problema al que principalmente se enfrentan los dispositivos reconfigurables en el mundo aeroespacial.

Aunque la técnica del TMR surgió hace años y había resultados de su aplicación sobre ASICs, nunca antes había sido aplicada sobre dispositivos reconfigurables tales como las FPGAs. Por este motivo empezamos a estudiar esta línea de investigación novedosa para analizar el comportamiento que tendría sobre FPGAs y así obtener conclusiones.

Originalmente partimos de la plataforma desarrollada por Víctor Alaminos Benéitez en su trabajo de fin de máster [Ala12]. Dicha plataforma no aplica redundancia de ningún tipo: carga sobre una FPGA de tipo Virtex 5 una única instancia del circuito a testear en una determinada región dada por el usuario, sobre la cual se emulan los errores que producirían la colisión de partículas cósmicas para determinar cuál es el porcentaje de errores asociado a ese circuito en concreto, tal y como se explicó en el capítulo 3 (Funcionamiento anterior de NESY 5.0).

El porcentaje promedio de los errores obtenidos sin aplicar redundancia es de 2,7122% tal como se puede calcular a partir de la tabla de resultados mostrada en la figura 5.1 (capítulo 5). El mundo aeroespacial es exigente y no puede permitir un tanto por ciento tan alto por lo que nuestro objetivo es realizar una protección más efectiva de los circuitos que permita reducir el porcentaje de error asociado a cada diseño. Para probar la efectividad de estas técnicas, modificaremos la plataforma presentada en [SeTa11] para adaptarla a nuestras necesidades.

Basándonos en la citada técnica del TMR, hemos efectuado diferentes variantes que explicaremos a continuación. Aplicando estas variantes de la técnica a distintos ejemplos de circuitos, hemos probado su efectividad introduciéndolos en la plataforma modificada (NESSY 6.0) para analizar los resultados.

4.1. Redundancia simple

La primera técnica de protección de circuitos digitales propuesta en este proyecto consiste en hacer la triplicación de los circuitos y la posterior selección mediante el votador en una única instancia implementada en lenguaje VHDL. Ésta técnica no es transparente al usuario, pues es él mismo quien efectúa la redundancia: debe introducir a NESSY un VHDL que contiene tres veces la funcionalidad del circuito que desee, así como un votador. El cometido del mismo es comparar las salidas de las tres instancias del circuito dentro del mismo archivo VHDL y seleccionar aquélla que se supone que es la correcta. La salida del votador será la salida total del diseño. Dicha estructura se muestra en la figura 4.1.

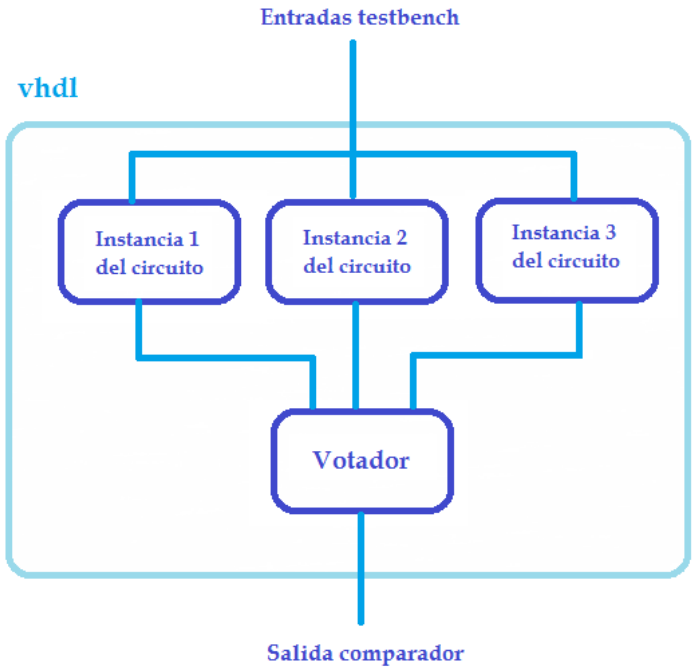


Figura 4.1: Estructura VHDL para triplicación simple

La aplicación de NESSY se ve inalterada al aplicar esta primera técnica. El principal inconveniente es que por el hecho de tener tres veces la misma funcionalidad del

circuito dentro de la misma instancia, el número de recursos compartidos al situar el diseño en la FPGA es muy elevado. Esto quiere decir que el circuito no tiene el tamaño que ocuparía realmente en la placa, si no que existen recursos entre las componentes idénticas del circuito que se comparten y regiones de las mismas que se solapan.

Seguimos con el ejemplo usado en el capítulo 3 para completar la explicación (figuras 3.4, 3.6, 3.7, 3.8 y 3.9). En la figura 4.2 se observan las estadísticas que se obtienen al finalizar la inyección de errores sobre el circuito.

```
**********************************************
    Tiempo tardado:              0d:0h:5m:11s
    Número de reconfiguraciones:        46080
    Número de reconfiguraciones erróneas:   223
    Número de reconfiguraciones estaticas:  0
    Porcentaje de errores:          0,483217592%
```

Figura 4.2: Estadísticas de la inyección mediante la técnica de triplicación simple

El porcentaje de errores obtenido sin aplicar redundancia (figura 3.9) se reduce notablemente al aplicar la triplicación simple y así lo podemos observar en la gráfica de la figura 4.3, la cual compara los porcentajes obtenidos mediante la aplicación de dichas técnicas.

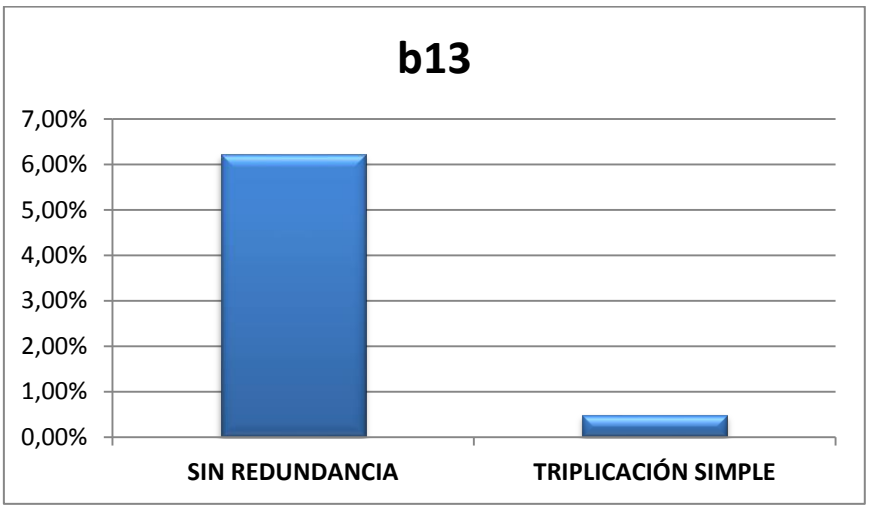


Figura 4.3: Gráfica comparativa

Esta aproximación no es suficiente, ya que al compartirse los recursos de los circuitos dentro de la misma instancia, un fallo en uno de ellos puede estar afectando a su vez a otro circuito y esto provocar un error que no puede ser enmascarado con la triple redundancia. Este es el motivo por el cual recurrimos a la técnica que se explica a continuación con el fin de obtener mejores resultados.

4.2. Redundancia triple modular + aislamiento

Debido a que el TMR simple tenía el inconveniente de la compartición de recursos que hacía que un fallo pudiera estar afectando a más de una copia del circuito, provocando así un error en el sistema imposible de enmascarar, hemos ideado una mejora de la técnica anterior que llamaremos redundancia triple modular + asilamiento.

Esta técnica consiste en triplicar los circuitos igual que en el caso anterior, pero aislando cada una de las instancias. De esta manera, se impide que los circuitos compartan recursos entre sí y que un error pueda estar afectando a más de uno. A su vez, el votador también ha sido situado aislado de los circuitos por el mismo motivo. Así conseguimos una técnica transparente para el usuario, el cual sólo se encargará de asignar la región de la FPGA donde quiere situar cada una de las tres instancias del circuito.

Por tanto, la actual técnica propuesta permitirá la triplicación automática del diseño: el usuario introducirá un diseño VHDL que contiene una única instancia del circuito sobre el cual desea inyectar errores, como ocurría en la versión anterior; sin embargo, de forma automática se generan tres instancias del circuito, cuyas salidas serán las entradas de un votador que permanecerá fijo en la FPGA, cuyo resultado será la salida global del sistema.

La nueva plataforma multitarea NESSY va a aceptar cualquier diseño escrito en VHDL que tenga un tamaño desde 32 hasta 128 bits y va a permitir al usuario elegir las zonas de la FPGA donde quiere colocar las tres instancias del circuito generadas.

A continuación, se detallan los cambios efectuados en la plataforma anterior con el fin de poder simular los circuitos protegidos con las técnicas presentadas. Estos cambios nos han llevado a la obtención de una nueva versión de la plataforma que llamaremos NESSY 6.0.

4.2.1. Hardware

El hardware de la aplicación ha sido modificado para que el adaptador circuito sea capaz de suministrar la entrada de datos a las tres instancias del circuito.

La salida cada circuito está conectada a la entrada del votador el cual conecta su salida de nuevo al adaptador circuito. Dicha salida será considerada como la salida del sistema. En todo momento, el adaptador está comunicado con el bus de datos PLB y en consecuencia, con todos los elementos del sistema hardware.

En la figura 4.4 se muestra se observa dicha estructura.

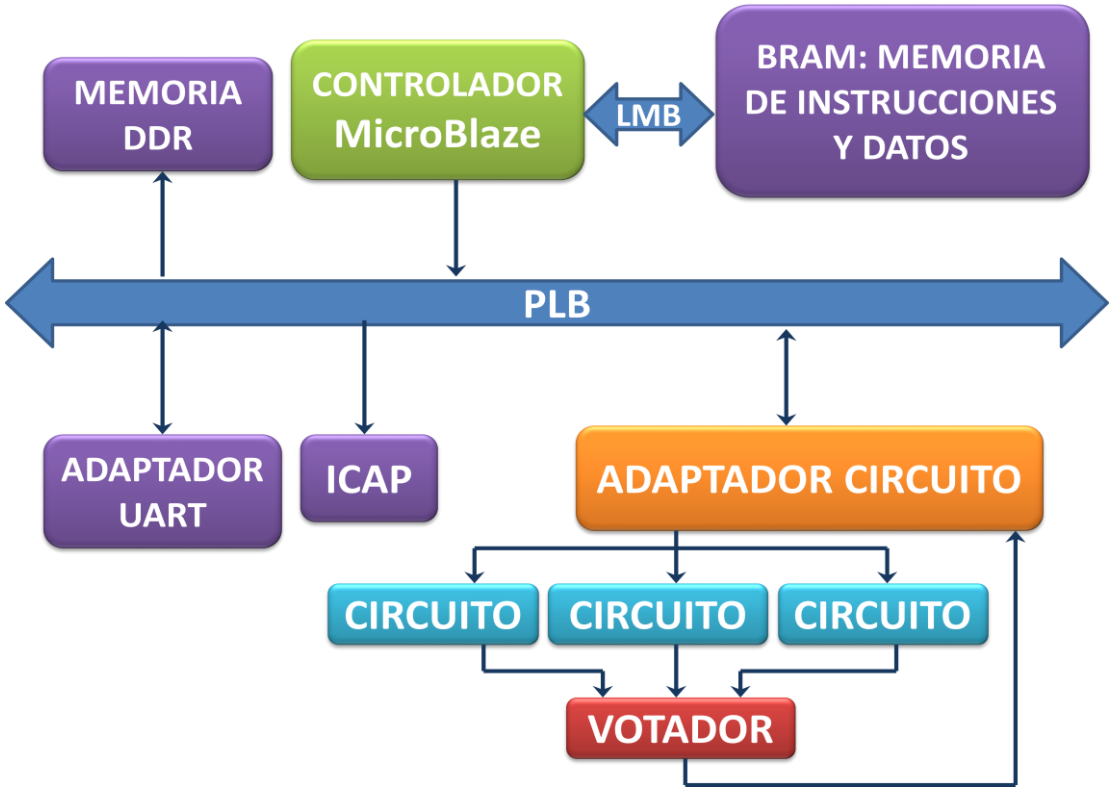


Figura 4.4: Arquitectura hardware de la plataforma NESSY 6.0

Implementamos la estructura hardware mediante la herramienta EDK. Hacemos uso de los periféricos creados con NESSY 5.0 y modificamos el proyecto añadiendo tres instancias del periférico "circuito" en lugar de una sola.

En la figura 4.5 se muestra el proyecto EDK: en la parte de la izquierda, dentro del apartado "*Project local pcores*" se observan los periféricos creados; en la parte de la derecha y dentro de la pestaña "*Bus interfaces*", se pueden ver todas las instancias de periféricos creadas, tres para el periférico circuito, todas conectadas al bus de datos PLB. NESSY 6.0 realiza este proceso de forma automática.

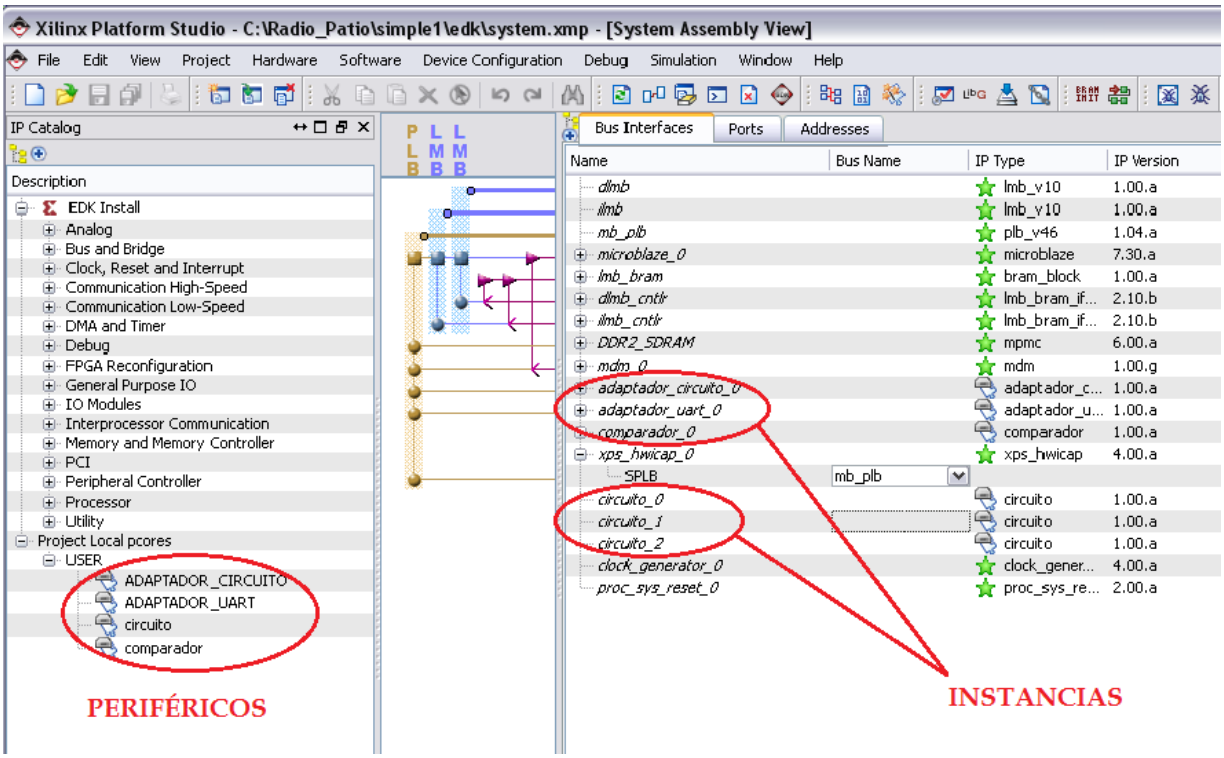


Figura 4.5: Periféricos e instancias en el proyecto EDK

En el proyecto EDK se generan las conexiones entre las instancias de los periféricos creados. La entrada y salida global del sistema serán la entrada y salida del adaptador circuito.

Dato listo es una señal interna que dosifica la señal del reloj puesto que se pone a 1 cada X ciclos originando así que con cada señal a 1 se escriban en un registro interno de

las instancias de circuito la entrada del *testbench* que corresponda, comenzando así el funcionamiento del diseño.

En la figura 4.6 se observa cómo se establece esta conexión dentro de EDK. Pulsando sobre la pestaña "*Ports*" observamos la siguiente configuración.

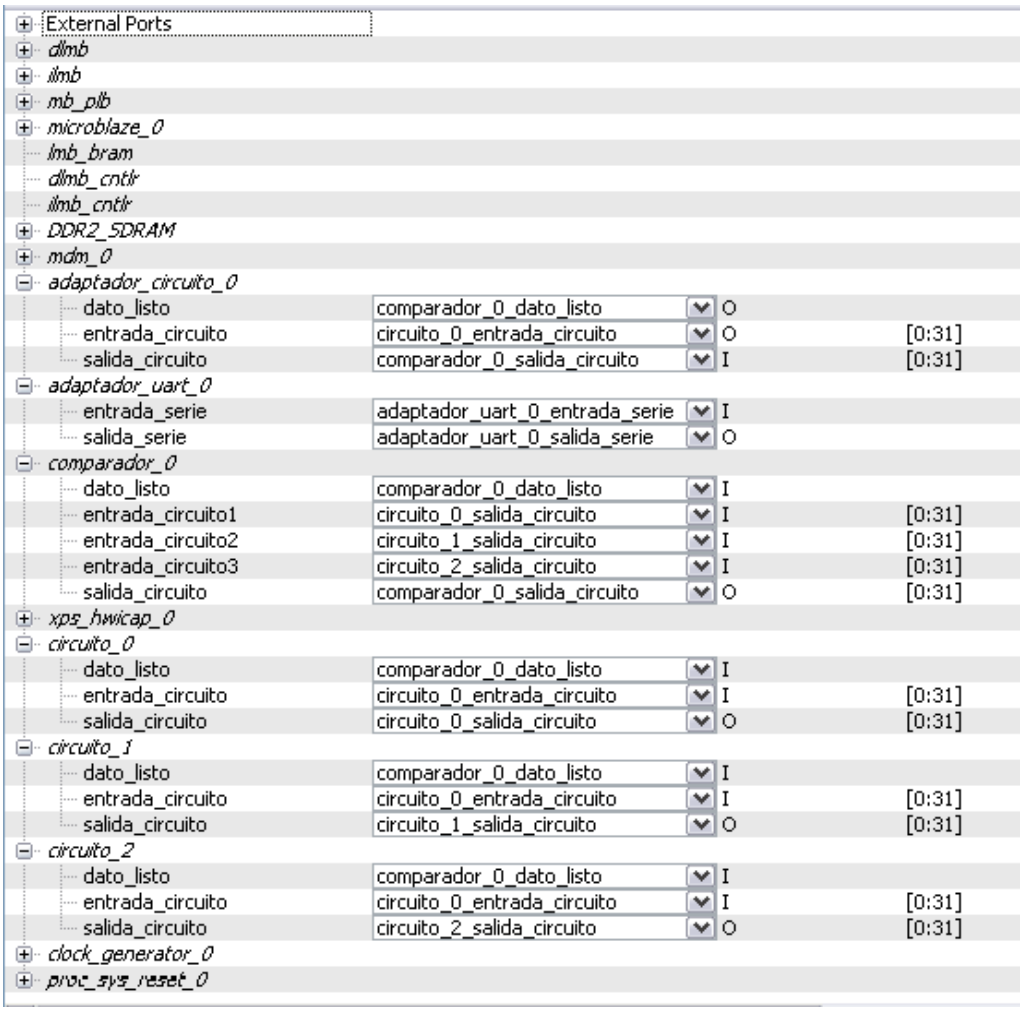


Figura 4.6: Configuración de los puertos de las instancias de los periféricos

Tal y como podemos observar en la figura anterior, la entrada de todas las instancias de circuito se fija como "circuito_0_entrada_circuito" porque serán las entradas del *testbench* comunes a los tres circuitos. La salida de cada instancia de circuito se fija como entrada del votador.

Por tanto, entendemos el diseño total como una caja negra que contiene las tres instancias de los circuitos y el votador en su interior. Se muestra su estructura en la siguiente figura 4.7.

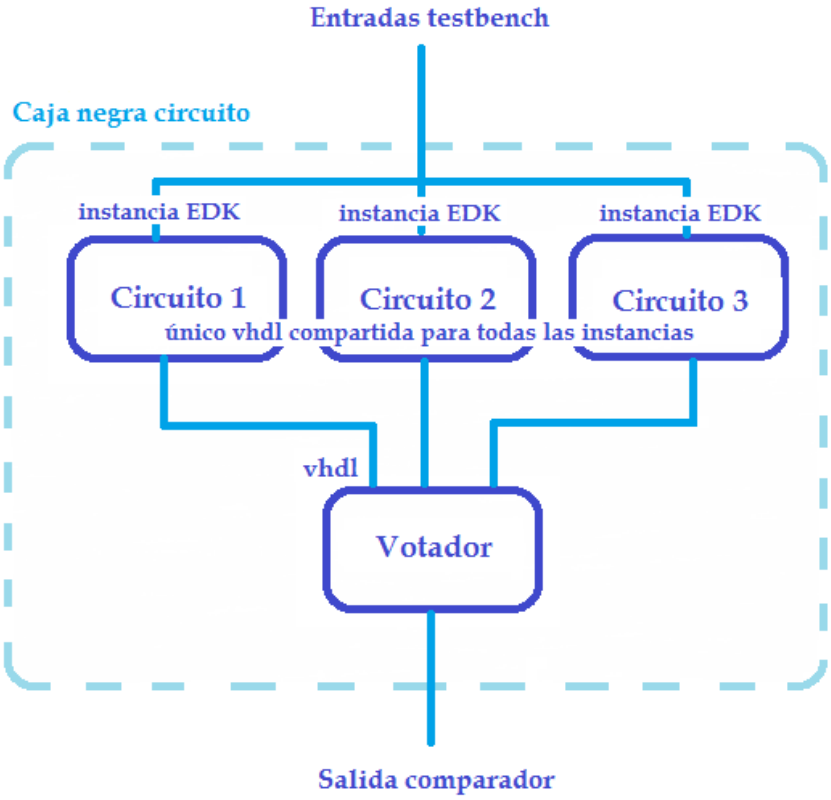


Figura 4.7: Estructura generada automáticamente a partir del circuito introducido

La señal de datos que introduce el adaptador circuito a las instancias puede ser desde 32 hasta 128 bits, es decir, NESSY 6.0 acepta circuitos de mayor tamaño. Esta es una novedad con respecto a NESSY 5.0, ya que sólo estaba permitido que el usuario introdujera diseños con tamaño inferior o igual a 32 bits. Actualmente el adaptador circuito está preparado para recibir circuitos de 32, 64 y hasta 128 bits. Si el número de bits de la salida del diseño introducido por el usuario es menor o igual a 32 bits entonces el adaptador será de 32 bits; si es mayor que 32 y menor o igual a 64 bits el tamaño del adaptador será de 64 bits; si por el contrario es mayor que 64 y menor o igual que 128 el tamaño del adaptador circuito será de 128 bits. Ocurre lo mismo con el votador, que tendrá el mismo tamaño que el adaptador circuito.

4.2.2. Software

NessyJava

La interfaz gráfica de NESSY 6.0 es la misma que la que usaba la anterior plataforma. Sin embargo se llevan a cabo una serie de modificaciones en el software, concretamente en los procesos adjuntos a los botones "Crear *bitstream*" y "Reconfiguración", para conseguir aplicar con éxito la técnica redundancia triple modular + aislamiento.

En primer lugar, cuando el usuario selecciona el botón "Crear *bitstream",* le aparece la ventana mostrada en la figura 4.8 donde deberá introducir las coordenadas inferior izquierda y superior derecha de la región de la FPGA donde desea situar cada una de las tres instancias del circuito. Tras pulsar sobre el botón "Aceptar" el programa se encarga de hacer una primera validación para asegurarse de que las coordenadas no se salen de los límites establecidos. En caso de que los datos introducidos no sean correctos se devuelve al usuario a la misma ventana para que vuelva a introducirlas.

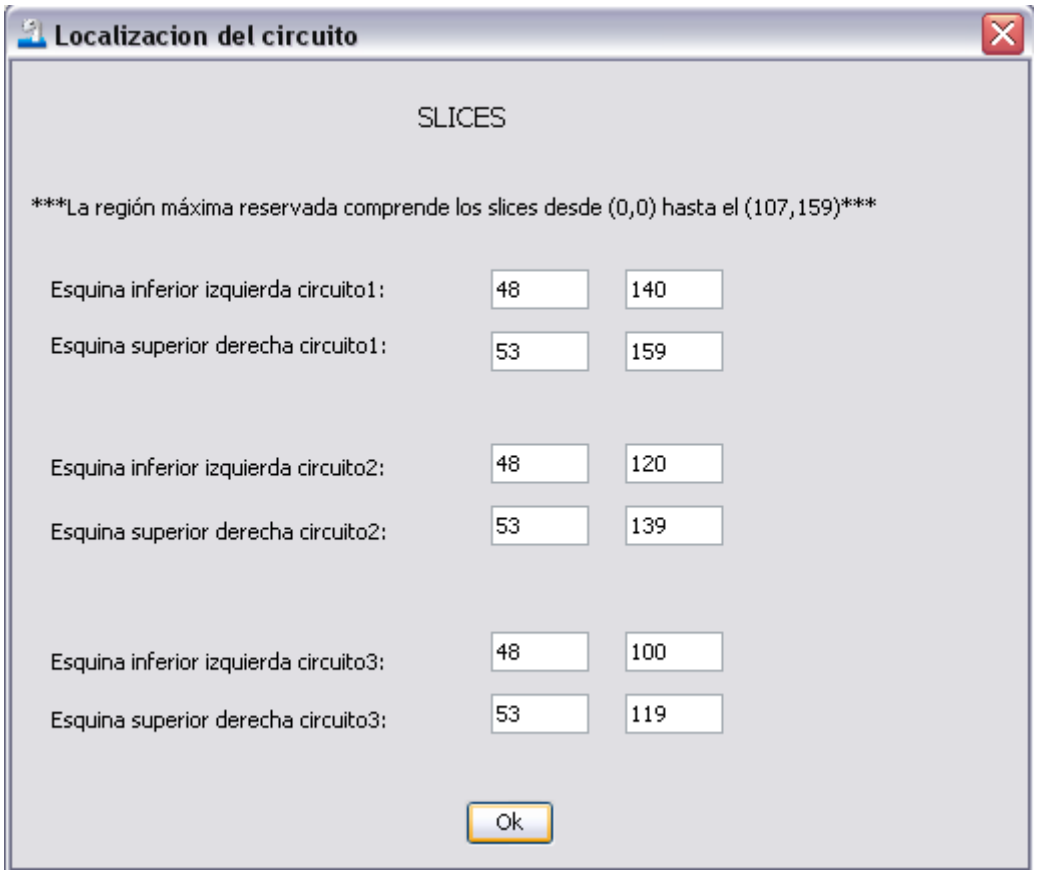


Figura 4.8: Ventana botón Crear bitstream

NESSY 6.0 sitúa a las tres instancias de circuito en la región de la FPGA correspondiente a las coordenadas introducidas por el usuario. El votador permanece anclado en una región fija de la FPGA. Mediante la herramienta PlanAhead se puede observar la situación de las tres instancias y el votador dentro de la placa de la FPGA y cómo están conectados, como se muestra en la figura 4.9.

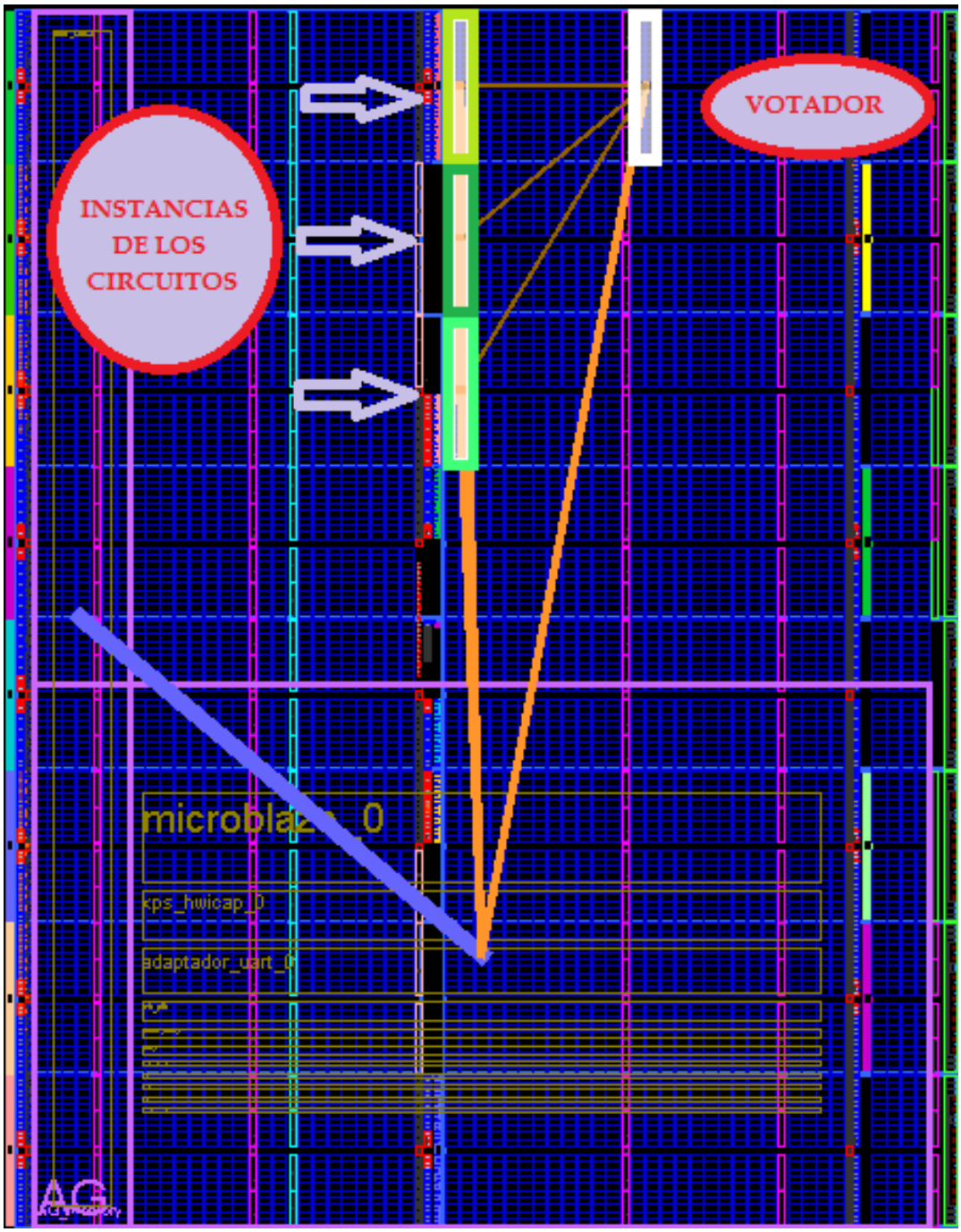


Figura 4.9: Las tres instancias del circuito y votador en la región de la FPGA

Una vez han sido situados los componentes del diseño en la placa, se modifica el archivo .ucf generado por el EDK añadiendo las tres instancias junto con las coordenadas introducidas por el usuario. En el caso del votador las coordenadas están fijadas en *NessyJava* de tal forma que se sitúe siempre en la misma región de la FPGA. La figura 4.10 muestra las modificaciones introducidas en un archivo .ucf.

```
269  INST "circuito_2" AREA_GROUP = "pblock_circuito_2";
270  AREA_GROUP "pblock_circuito_2" RANGE=SLICE_X48Y100:SLICE_X53Y119;
271  INST "comparador_0" AREA_GROUP = "pblock_comparador_0";
272  AREA_GROUP "pblock_comparador_0" RANGE=SLICE_X72Y140:SLICE_X74Y159;
273  INST "circuito_1" AREA_GROUP = "pblock_circuito_1";
274  AREA_GROUP "pblock_circuito_1" RANGE=SLICE_X48Y120:SLICE_X53Y139;
275  INST "circuito_0" AREA_GROUP = "pblock_circuito_0";
276  AREA_GROUP "pblock_circuito_0" RANGE=SLICE_X48Y140:SLICE_X53Y159;
```

Figura 4.10: Coordenadas de los circuitos y el votador en el ucf

En PlanAhead se llevan a cabo los siguientes procesos: se abre la netlist del proyecto generada con el EDK; se añaden al proyecto como módulos reconfigurables las tres instancias del circuito y el votador; se sitúan los módulos en la región de la FPGA especificada por el usuario en el caso de las instancias del circuito y en la región fijada en *NessyJava* para el caso del votador; se realiza la implementación y la síntesis de los módulos reconfigurables.

Es necesario tener en cuenta que tanto la región de la FPGA que contiene componentes estáticos (desde la mitad de la FPGA hasta abajo) como la región reservada para la memoria (un fragmento que va desde arriba hasta abajo en la parte izquierda del todo de la FPGA) deben ser respetadas por el usuario a la hora de colocar las instancias del circuito.

Tras la síntesis e implementación automática del sistema hardware junto con el circuito introducido por el usuario se obtienen cinco *bitstreams* (en la plataforma anterior se obtenían dos, uno parcial para el circuito y uno global): cuatro de ellos parciales, correspondientes a las tres instancias del circuito y al votador; y el quinto y último, el global del sistema, que sirve para futuras cargas del proyecto.

Por último, el proceso que se lleva a cabo cuando el usuario pulsa el botón "Reconfiguración", consiste en el envío e inyección de los *bitstream*s parciales además de la evaluación del impacto de las inyecciones. Vamos a analizar más en profundidad qué ocurre durante este proceso.

En el momento en el que el usuario pulsa este botón Reconfiguración y siempre y cuando la comunicación con la FPGA esté sincronizada, se procede al envío de los cuatro *bitstream*s parciales generados en el segundo paso de *NessyJava* (Crear *bitstream*). Para cada bit parcial se envían a la FPGA a través del puerto serie su tamaño y un fichero que contiene el bit parcial completo. Este fichero se almacena en la RAM de la FPGA, en la región reservada para ese mapa de bits (cada *bitstream* parcial tiene un rango de direcciones de memoria reservadas, como se explicará más adelante en el apartado correspondiente a *NessyC*).

Tras el envío de los cuatro bit parciales se procede a la inyección de los bitflips en cada *bitstream*. La inyección se ejecuta en orden, de tal forma que no se empezará a inyectar en el segundo *bitstream* hasta no acabe la inyección en el primero. Cuando comienza la inyección en el primer mapa de bits parcial se crea el archivo "reconfiguracion". De la misma forma, cada vez que comienza la inyección de algún *bitstream* parcial se crea un fichero "reconfiguración_matlab" de la instancia correspondiente en la carpeta "salidas" del proyecto NESSY. Estos ficheros contienen toda la información referente al proceso de inyección, así como las estadísticas obtenidas durante el proceso, una vez que finaliza el proceso sobre el cuatro *bitstream*.

La inyección de un *bitflip* en un *bitstream* genera los siguientes resultados: si ha habido fallo, obtenemos el ciclo en el que la ejecución del circuito comenzó a fallar, además de

cuál es el resultado erróneo y cuál el resultado que se debería haber obtenido para compararlos; si no ha habido fallo, se notifica que no ha habido error.

Aunque un *bitstream* falle no implica un error del diseño total, puesto que el votador selecciona como correcta la salida del circuito que se repita más, la cual se supone que es la correcta, no produciéndose un error. Por tanto, obtenemos fallo del diseño cuando falle el votador, cuando fallen al menos dos de las tres instancias del circuito a la vez, o bien cuando el *bitflip* insertado caiga justo en la entrada de las instancias, puesto que, al variar la entrada, el *golden* ya no coincidiría al producirse otro comportamiento distinto. Se reduce notablemente el abanico de casuística de error con respecto a la plataforma anterior, en la que obteníamos error siempre que fallaba la instancia única del circuito.

En caso de que haya habido error en la ejecución del proceso, se procede a la reconfiguración de esa parte de la FPGA en concreto, es decir, cada vez que hay un fallo en un *bitstream* sólo se reconfigura aquel que produjo fallo por medio del ICAP, tal como explicamos anteriormente.

Tras inyectar errores en los cuatro mapa de bits parciales con éxito, la información que se obtiene es el tiempo transcurrido desde el comienzo de la inyección en el primer *bitstream* hasta la finalización del proceso en el cuarto, el número de reconfiguraciones realizadas en total, el número de reconfiguraciones erróneas, el número de reconfiguraciones estáticas y el porcentaje de reconfiguraciones erróneas respecto del total de reconfiguraciones realizadas.

NessyC

En el programa que ejecuta el MicroBlaze de esta nueva versión de NESSY es necesario reservar tres regiones más de memoria: necesitamos almacenar en distintas direcciones cada *bistream* parcial. Junto con la región de memoria donde anteriormente se almacenaba el que era el único *bitstream* parcial, ahora contamos con cuatro regiones diferenciadas.

En cada una se almacenará el *bitstream* parcial original que es enviado desde el PC hasta la FPGA por el puerto serie tras el envío del comando correspondiente. Además,

como consecuencia de la reserva de tres nuevos rangos de memoria creamos tres nuevos punteros, cada uno de los cuales apuntarán al comienzo de la región de memoria que corresponda a cada *bitstream* parcial. En la figura 4.11 se muestran las regiones de memoria reservadas para el almacenamiento de los cuatro mapas de bits parciales.

```
58  ////////////////////////////////////RAM/////////////////////////////////////
59  #define RAM_TESTBENCH              (XPAR_DDR2_SDRAM_MPMC_BASEADDR + 0x00000000)
60  #define RAM_GOLDEN                 (XPAR_DDR2_SDRAM_MPMC_BASEADDR + 0x00100000)
61  #define RAM_ERRORES                (XPAR_DDR2_SDRAM_MPMC_BASEADDR + 0x00200000)
62  #define RAM_BITSTREAM_ORIGINAL1    (XPAR_DDR2_SDRAM_MPMC_BASEADDR + 0x01000000)
63  #define RAM_BITSTREAM_ORIGINAL2    (XPAR_DDR2_SDRAM_MPMC_BASEADDR + 0x01500000)
64  #define RAM_BITSTREAM_ORIGINAL3    (XPAR_DDR2_SDRAM_MPMC_BASEADDR + 0x02000000)
65  #define RAM_BITSTREAM_ORIGINAL4    (XPAR_DDR2_SDRAM_MPMC_BASEADDR + 0x02500000)
66  #define RAM_BITSTREAM_MODIFICADO   (XPAR_DDR2_SDRAM_MPMC_BASEADDR + 0x03000000)
67  #define RAM_BITSTREAM_RESTORER     (XPAR_DDR2_SDRAM_MPMC_BASEADDR + 0x04000000)
68  #define RAM_VARIABLES              (XPAR_DDR2_SDRAM_MPMC_BASEADDR + 0x05000000)
69  #define RAM_READBACK               (XPAR_DDR2_SDRAM_MPMC_BASEADDR + 0x06000000)
```

Figura 4.11: Regiones de memoria de los cuatro bit parciales originales

Además hemos generado un nuevo script que contiene el nuevo tamaño de la memoria RAM de la FPGA, el cual se ha visto aumentado tras la reserva de espacio para los bitstreams parciales. La figura 4.12 muestra el fragmento del script mencionado en el que se observa el tamaño total que ocupa la memoria DDR2 RAM de la FPGA.

```
18 MEMORY
19 {
20    ilmb_cntlr_dlmb_cntlr : ORIGIN = 0x00000050, LENGTH = 0x0003FFB0
21    DDR2_SDRAM_MPMC_BASEADDR : ORIGIN = 0x90000000, LENGTH = 0x10000000
22 }
```

Figura 4.12: Script SDK con el nuevo tamaño de la memoria DDR2 RAM

Es necesario almacenar todos los *bitstreams* parciales originales porque si no se perderían y no podríamos inyectar errores sobre ellos. Cada vez que se inyecta sobre un *bitstream* parcial original, se genera su correspondiente *bitstream* parcial modificado y su correspondiente *bitstream* restorer, los cuales se almacenan en memoria hasta que finaliza la inyección en ese mapa de bits.

Una vez comienza la siguiente inyección, las regiones de memoria reservadas para el modificado y el restorer se sobrescriben con los *bitstreams* correspondientes al nuevo mapa de bits parcial original. Los *bitstreams* parcial modificado y restorer se desechan porque no será necesario usarlos más. Éste es el motivo por el cual sólo es necesario almacenar los mapas de bits originales.

Algunos de los comandos que se envían por el puerto serie y que activan la ejecución del código C en el MicroBlaze se han visto modificados. Los cambios introducidos en los mismos para la aplicación de la técnica que hemos explicado son:

- ✓ Comando ***"enviar los bit parciales originales"***: después de que el PC envíe el comando correspondiente por el puerto serie, *NessyJava* envía de forma ordenada el tamaño y el fichero que contiene cada mapa de bits parcial original. El MicroBlaze permanece a la escucha tras la recepción del comando y recibe el pack tamaño-fichero de cada *bitstream* parcial. Estos ficheros los almacena en la memoria DDR2 RAM donde permanecerán para llevar a cabo el proceso de la inyección.

- ✓ Comando ***"recibir los bit parciales originales"***: proceso inverso al acontecido durante la ejecución del comando anterior. Envío por parte del controlador de la FPGA y recibo del PC a través del puerto serie de los cuatro pack tamaño-fichero de cada *bitstream* parcial original.

- ✓ Comando ***"inicializar variables"***: el PC envía, tras el comando correspondiente, los tamaños de los cuatro *bitstream*s parciales originales, uno tras otro. Además también se envían el número de bits de la entrada y la salida del circuito, el número de entradas del *testbench* y el tipo de adaptador que se usa, si el de 32, el de 64 o el de 128 bits. El MicroBlaze de la FPGA recibe todos los datos y los almacena.

- ✓ Comando ***"scan"***: la llamada al método *scan()* dentro de *NessyJava* se realiza una vez por cada *bitstream* parcial original durante su inyección. Al comienzo del método se envía a través del puerto serie el número del *bitstream* en el que se va a realizar la inyección actual (recordar que los tres primeros corresponderán a las tres instancias de circuito y el cuarto y último al votador).

- ✓ Comando ***"inyectar errores"***: se envía desde el PC hasta la FPGA el número de *bitstream* sobre el que se va a realizar la inyección actual.

Para justificar la técnica aplicada, observamos en la figura 4.13 las estadísticas obtenidas al finalizar la inyección en el ejemplo b13, del que estamos haciendo un seguimiento más exhaustivo.

```
**********************************************
    Tiempo tardado:              0d:0h:41m:47s
    Número de reconfiguraciones:         230400
    Número de reconfiguraciones erróneas:   191
    Número de reconfiguraciones estaticas:  0
    Porcentaje de errores:           0.08289930555555555%
```

Figura 4.13: Estadísticas de la inyección mediante la técnica de redundancia triple modular + aislamiento

El porcentaje de errores obtenido al aplicar la técnica anterior, triplicación simple, (figura 4.2) se reduce notablemente al aplicar el aislamiento.

Podemos observarlo en la siguiente gráfica (figura 4.14), la cual compara los porcentajes obtenidos al aplicar las técnicas ya estudiadas.

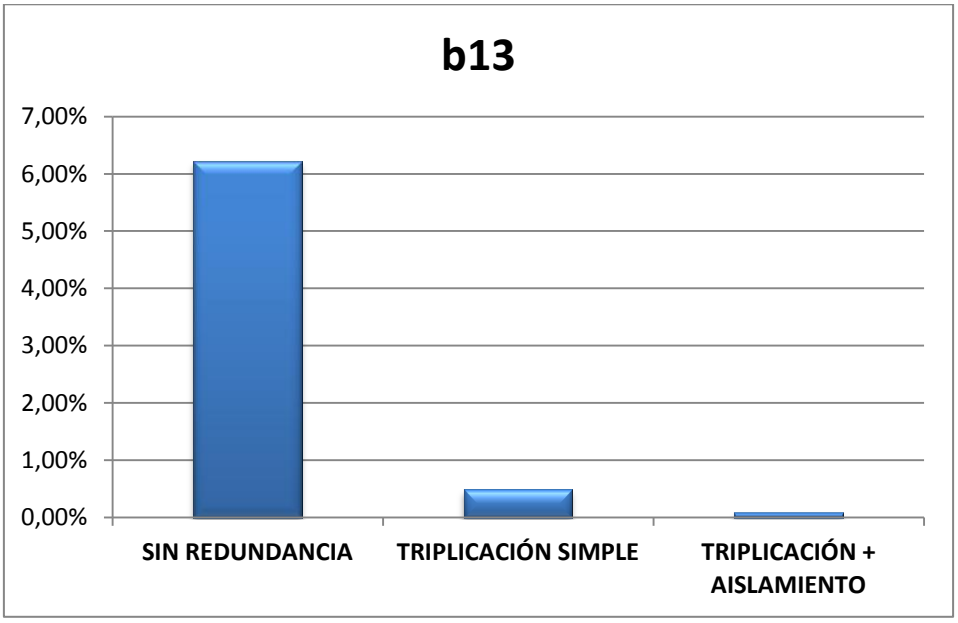


Figura 4.14: Gráfica comparativa

Además es importante el hecho de que la reconfiguración tras un fallo sólo implique al circuito o región de la FPGA que ha fallado ya que en el caso real aeroespacial no será necesaria la reconfiguración total de todo el sistema sino sólo de aquella parte fallida.

Este aislamiento nos ha permitido reducir el porcentaje de errores significativamente considerando ésta técnica una buena idea que aún no había sido probada sobre este tipo de dispositivos. A pesar de esto, ya que el objetivo que buscamos es obtener un porcentaje de error cada vez más próximo a cero, entra en juego la posibilidad de endurecer alguna región de la FPGA, de tal forma que obtengamos mejores resultados. Recurrimos así a la técnica que se expone a continuación.

4.3. Redundancia triple modular + aislamiento + votador blindado

Dado que el TMR no proporciona absoluta fiabilidad a nuestros diseños, por los fallos que recaen sobre el votador, hemos planteado como alternativa la utilización de un votador totalmente tolerante ante los efectos de los citados SEUs, lo que denominamos votador blindado.

Dicho votador formaría parte de la FPGA como elemento no reconfigurable y de esta manera las partículas que impactaran contra el mismo no producirían error alguno. En nuestro experimento se ha barajado esa posibilidad, simulando que las partículas que colisionan con el votador no producen fallo por el blindaje de éste.

Para el desarrollo de esta técnica, la plataforma permanece con todas las modificaciones incluidas para el desarrollo de la técnica anterior, la técnica de triplicación + aislamiento. La única diferencia es que únicamente se inyectarán errores sobre los *bistreams* parciales originales de las instancias del circuito, no sobre el mapa de bits del votador. Ha sido añadido un nuevo botón a la interfaz gráfica de la aplicación (figura 3.5) que permite al usuario la simulación de errores mediante la técnica de triplicación + aislamiento + votador blindado.

Según se ha explicado, el cometido del votador es evitar los errores que se producen en las tres instancias del circuito. Por tanto, sólo obtendremos fallo del diseño cuando el bitflip en cierta inserción caiga sobre la entrada de las tres instancias idénticas del circuito, pues en este caso la entrada ya no sería correcta y en consecuencia el funcionamiento del diseño tampoco.

En la figura 4.15 se muestran las estadísticas que se obtienen tras finalizar la inyección en los *bitstreams* de los circuitos aplicando la técnica del votador blindado.

```
**********************************************
    Tiempo tardado:              0d:0h:25m:3s
    Número de reconfiguraciones:        138240
    Número de reconfiguraciones erróneas:   4
    Número de reconfiguraciones estaticas:  0
    Porcentaje de errores:          0.002893518518518519%
```

Figura 4.15: Estadísticas de la inyección mediante la técnica de redundancia triple modular + aislamiento + votador blindado

El porcentaje de errores nuevamente se ha reducido, siendo muy próximo a cero. Podemos observarlo en la figura 4.16, la cual compara los porcentajes obtenidos al aplicar todas las técnicas ya estudiadas.

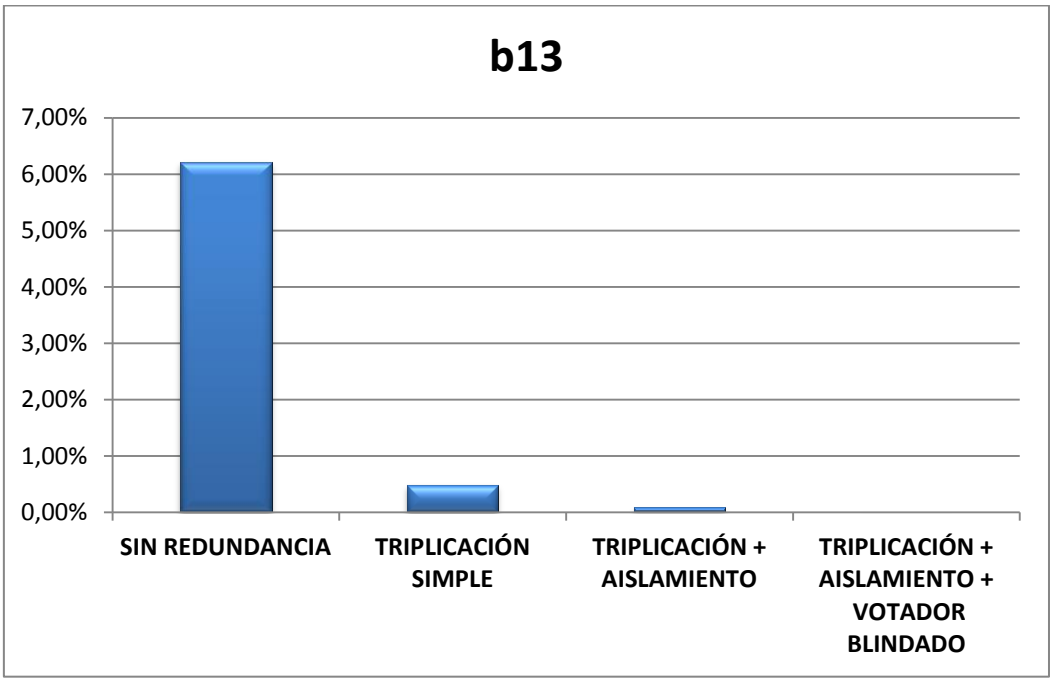


Figura 4.16: Gráfica comparativa

En el capítulo 6 (Trabajo futuro) se comentan algunas de las técnicas a explotar que pueden implicar una mejora en el descenso del porcentaje de errores frente a esta nueva plataforma NESSY 6.0.

Capítulo 5: Resultados

En este capítulo expondremos los resultados obtenidos al probar experimentalmente la eficiencia de las distintas técnicas de protección de circuitos propuestas en este trabajo. Así, a lo largo del capítulo, las técnicas propuestas se evaluarán al introducirlas de manera incremental en NESSY.

Para comprobar la eficiencia de los tres métodos propuestos, hemos utilizado un conjunto de *benchmarks* realistas, incluidos en el conjunto de *benchmarks* ITC'99, desarrollados por [CoReSq09].

En estos experimentos, hemos seleccionado todos aquellos *benchmarks* cuya implementación en FPGA no utiliza ninguno de los bloques de memoria BRAM que se encuentran embebidos en la FPGA. El motivo es, que la actual versión de NESSY aún no es capaz de inyectar SEUs en los bits de configuración que configuran este tipo de bloques de memoria.

Para la realización de estos experimentos, hemos debido tener en cuenta que Xilinx™ PlanAhead no puede asegurar que los cables que comunican el sistema hardware estático no crucen la región parcialmente reconfigurable. Para evitar este problema hemos seleccionado en todos los circuitos de manera manual una región libre de este tipo de cables para situar los circuitos a testear. También es importante destacar que a la hora de reservar una región de slices para un circuito, hemos intentado ajustar al máximo el tamaño reservado para el mismo, para así minimizar el número de slices en los que se inyectan errores, pero no se utilizan para instanciar el circuito. De esta manera aumentamos la precisión de los resultados obtenidos y al mismo tiempo disminuimos el tiempo de ejecución de la herramienta.

En la tabla 5.1 y gráficamente en la figura 5.1 se presentan los resultados obtenidos al realizar inyecciones exhaustivas en los circuitos considerados, para los siguientes cuatro casos, coincidentes con las técnicas presentadas a lo largo de este proyecto:

- ✓ Sin ningún tipo de redundancia, en la tabla y figura 5.1, por simplicidad, sin redundancia.
- ✓ Con Redundancia Triple Modular Simple, en la tabla y figura 5.1, por simplicidad, triplicación simple.
- ✓ Con Redundancia Triple Modular + Aislamiento, en la tabla y figura 5.1, por simplicidad, triplicación + aislamiento.
- ✓ Con Redundancia Triple Modular + Aislamiento y utilizando un votador blindado, en la tabla y figura 5.1, por simplicidad, triplicación + aislamiento + votador blindado.

Tabla 5.1: Tabla comparativa de resultados obtenidos

CIRCUITOS	SIN REDUNDANCIA	TRIPLICACION SIMPLE	TMR + ASILAMIENTO	TMR + AISLAMIENTO + VOTADOR BLINDADO
b01	1,146%	0,157%	0,040%	0,021%
b02	0,414%	0,061%	0,028%	0,014%
b03	2,609%	0,322%	0,043%	0,000%
b04	6,426%	1,024%	0,057%	0,003%
b05	0,041%	0,026%	0,013%	0,000%
b06	0,818%	0,166%	0,084%	0,010%
b07	0,143%	0,073%	0,033%	0,000%
b08	3,715%	0,361%	0,072%	0,024%
b09	2,826%	0,107%	0,018%	0,005%
b10	4,049%	0,726%	0,068%	0,001%
b11	4,765%	0,240%	0,052%	0,010%
b12	2,093%	0,234%	0,028%	0,002%
b13	6,293%	0,483%	0,083%	0,003%

Como podemos observar, la triplicación simple consigue disminuir notablemente el porcentaje de SEUs no enmascarables (que producen un error en la salida del sistema), a menos de 1.1% en prácticamente todos los casos. Es decir, el 98.9% de los SEUs que se simulan mediante la inyección de un *bitflip* no producen error en la salida.

Además, podemos ver, que los circuitos b08 hasta el b13, circuitos en los que cuando no hay redundancia experimentan un mayor número de errores, con la aplicación de nuestras técnicas, ofrecen picos de disminución mayores que los anteriores, debido a

que son circuitos más grandes en los que se puede apreciar mejor la efectividad de las técnicas desarrolladas. La siguiente gráfica (figura 5.1) muestra los resultados obtenidos.

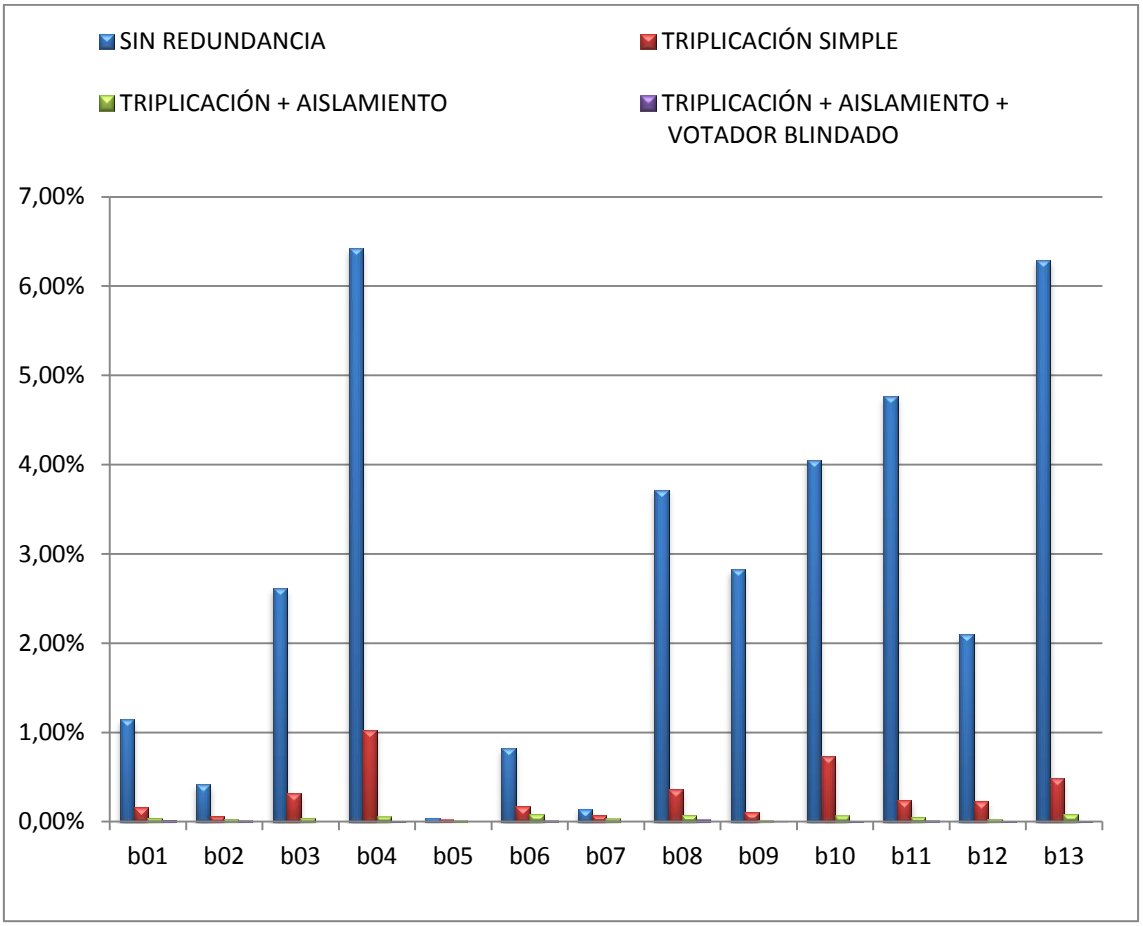


Figura 5.1: Porcentaje de SEUs que han provocado error para cada una de las técnicas de protección propuestas, en comparación con un sistema equivalente sin redundancia

Para estudiar en mayor detalle la mejora introducida por las técnicas propuestas, en la figura 5.2, se presentan las mismas gráficas exceptuando la opción sin redundancia. Así, podemos observar que, utilizando aislamiento de las instancias, el porcentaje de errores no enmascarables cae por debajo del 0,085%, siendo un 0,04% de media; es decir, se ha reducido su incidencia en un 96% frente al caso sin aislar. Por último, usando el votador blindado los errores no enmascarables quedan por debajo del 0,024%, siendo un 0,007% de media.

Podemos observar, que en algunos casos, gracias a la protección del votador mediante el blindaje conseguimos hasta un 0% de errores.

Por tanto, sin necesidad de invertir un alto presupuesto, somos capaces de conseguir resultados fiables que hacen que la técnica de triplicación con aislamiento de las instancias del circuito o el blindaje del votador sean interesantes para su utilización en misiones aeroespaciales reales.

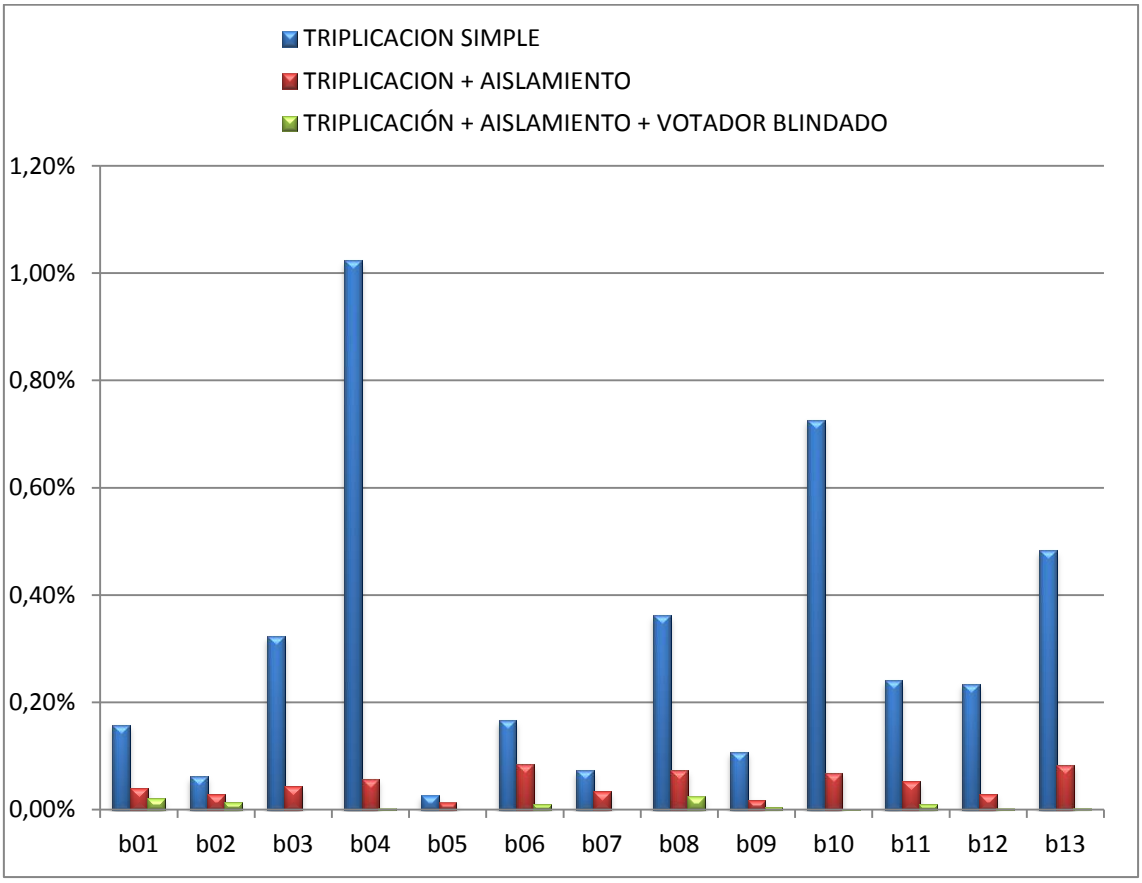


Figura 5.2: Comparativa de la eficiencia frente a SEUs de las tres técnicas de protección de circuitos presentadas en este trabajo

Capítulo 6: Conclusiones y trabajo futuro

Durante este proyecto se ha realizado un estudio comparativo de dos técnicas basadas en TMR para la protección de circuitos implementados en FPGAs. Se ha demostrado que, a diferencia de otras tecnologías, en las FPGAs utilizar TMR sólo no es suficiente, ya que hasta un 1% de los SEUs sigue produciendo error. Para reducir este porcentaje es preciso garantizar el aislamiento físico de las distintas instancias del circuito mediante su implementación en diferentes regiones parcialmente reconfigurables de la FPGA. Esto reduce la media de errores encontrados a la salida del circuito al 0,05%.

Finalmente, también se presenta una posible mejora arquitectónica que consiste en proteger el votador, causante de la mayor parte de los errores no enmascarables. Esta última mejora arquitectónica permite reducir SEUs que producen error al 0,007% de media (0,03% máximo).

Nuestro trabajo es sólo un paso en las mejoras de esta plataforma de inyección de errores. En el futuro se quieren realizar más avances y evoluciones para incluir nuevas funcionalidades convirtiendo a NESSY en una plataforma más completa.

Entre estas mejoras, se quiere incluir la posibilidad de emular fallos en los que el impacto de una partícula cósmica afecte a más de una celda de memoria, efecto cada vez más probable, al disminuir el tamaño de integración de los transistores en las últimas tecnologías.

Otra de las posibles mejoras pasaría por depurar estas técnicas usando un TMR selectivo, de forma que sólo se tripliquen aquellos recursos más vulnerables. De esta forma, sin incrementar el número de errores no enmascarables, conseguiremos soluciones menos costosas.

Además, nuestro trabajo se ha centrado solamente en la inyección de errores en los CLBs, ya que en el script desarrollado para el PlanAhead forzamos a las herramientas a

que no utilicen otros elementos lógicos para la síntesis e implementación de los circuitos. Sin embargo, como mejora en un futuro podría introducirse la inyección de errores tanto en los DSPs como en las BlockRams embebidas en la FPGA.

Bibliografía

[Ala12] Alaminos Víctor *Inserción de Bitflips en mapas de configuración de FPGAs*, 2012.

[Ald09] Alderighi M [y otros] *Experimental Validation of Fault Injection Analyses by the FLIPPER Tool :* Radiation and Its Effects on Components and Systems (RADECS), 2009.

[Ald07] Alderighi M**.** *Evaluation of Single Event Upset Mitigation Schemes for SRAM based FPGAs using the FLIPPER Fault Injection Platform :* IEEE International Symposium on, 2007.

[BaStVi08] Battezzati N, Sterpone L. y Violante M. *A new low-cost non intrusive platform for injecting soft errors in SRAM-based FPGAs* : IEEE International Symposium on, 2008.

[Ber10] Berg Melanie *Complexity Management and Design Optimization Regarding a Variety of Triple Modular Redundancy Schemes through Automation* : Marlug, 2010.

[CoReSq09] Corno F, Reorda M y Squillero G *RT-Level ITC'99 Benchmarks and First ATPG Results, Design Test of Computers* : Design & Test of Computers, IEEE , 2009.

[Guz12] Guzmán-Miranda H [y otros] *FT-UNSHADES2: A Platform for early evaluation of ASIC and FPGA dependability using partial reconfiguration* : Jornadas de Computación Reconfigurable, 2012.

[JoAd89] Johnson B.W. y Addison Wesley *Design and analysis of fault tolerant digital systems,* 1989.

[Ka05] Kakarla Sujana *Partial Evaluation Based Triple Modular Redundancy For Single Event Upset Mitigation* : Graduate School Theses and Dissertations, 2005.

[KaKa05] Kakarla Sujana y Katkoori Srinivas *Partial Evaluation Based Redundancy for SEU Mitigation in Combinational Circuits* : MAPLD, University of South Florida, 2005.

[Liu11] Liu S.F. [y otros] *Increasing Reliability of FPGA-Based Adaptive Equalizers in the Presence of Single Event Upsets*: Nuclear Science, IEEE Transactions on, 2011.

[Pra06] Pratt Brian *Improving FPGA Design Robustness with Partial TMR : Reliability Physics Symposium Proceedings*: IEEE International, 2006.

[SeTa11] Serrano Felipe y Tarancón Rubén *Desarrollo de una plataforma de inyección de errores sobre hardware reconfigurable,*2011

[Sol91] Sole Antonio Creus *Fiabilidad y seguridad. Su aplicación en procesos industriales*, 1991.

[StVi07] Sterpone L y Violante M *A new Partial Reconfiguration-Based Fault-Injection System to Evaluate SEU Effects in SRAM-Based FPGAs*: Nuclear Science, IEEE Transactions on, 2007.

[St08] Sterpone L. [y otros] *On the design of tunable fault tolerant circuits on SRAM-based FPGAs for safety critical applications* : Design, Automation and Test in Europe, 2008.

[XiCo12] Corporation Xilinx *MicroBlaze Processor Reference Guide*, 2012.

[Xtp029] Overview of Xilinx JTAG Programming Cables and Reference Schematics for Legacy Parallel Cable III