ESCUELA SUPERIOR DE INGENIEROS
DEPARTAMENTO DE INGENIERÍA ELECTRÓNICA
UNIVERSIDAD DE SEVILLA

Implementación de un procesador empotrado para su estudio bajo inyección de fallos

Autor: Hipólito Guzmán Miranda
Tutor: Jon Tombs

Trabajo Fin de Máster
Máster en Electrónica,
Tratamiento de la Señal
y Comunicaciones

Universidad de Sevilla,
Sevilla, Marzo 2008

hipolito@gte.esi.us.es

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

Se ha investigado ya bastante sobre la tolerancia a fallos en circuitos digitales de poca complejidad, pero el estudio de cómo responden circuitos complejos como los microprocesadores y sus programas a estos efectos de la radiación es todavía un tema abierto a estudio y experimentación. Para poder investigar con comodidad y flexibilidad, y realizar experimentos exhaustivos y diversos, es necesario conocer el proceso de implementacion de un microprocesador empotrado. Y la necesidad de adquirir este conocimiento no termina en la implementacion: no sólo es importante implementar el microprocesador, sino conocer a la perfección el proceso, de forma que podamos realizar nuevas implementaciones, ya que la investigación sobre el estudio de tolerancia a fallos es una exploración constante y activa en la que seguramente habrá que realizar nuevas implementaciones del microprocesador, modificando algunos de los parámetros de la configuración. Se han escogido procesadores configurables sobre el que en un futuro se podrán realizar múltiples y diversos experimentos, estudiando el efecto de configurar el microprocesador con distintos periféricos.

1.1. Algunas cuestiones a estudiar

Algunas de las cuestiones que merece la pena estudiar en circuitos microprocesador son la sensibilidad a las faltas, y cómo podemos reducirla, ya sea a través de la redundancia de información o la replicación de los módulos sensibles del circuito. En el caso de no querer triplicar todos los módulos del circuito, debido al incremento en área y consumo de potencia en que se incurre, tanto por la triplicación del hardware como por el incremento de la complejidad del rutado, una técnica de nueva aplicación es el reducir consumo de potencia y área protegiendo al circuito de forma selectiva[7], para lo que necesitamos evaluar qué partes del procesador son las más sensibles. También, como un microprocesador es capaz

de aplicar detección y corrección de errores no sólo en hardware, sino como parte del programa que ejecuta (protección *software*), queremos poder evaluar ante qué fallos puede recuperarse el microprocesador y ante cuáles no, y en caso de que se recupere, evaluar a su vez el tiempo en ciclos necesario para que el microprocesador alcance un estado libre de errores, tiempo que dependerá de la técnica de protección software utilizada.

1.2. Estructura del presente documento

En el presente trabajo se estudian, en primer lugar, algunas cuestiones físicas y de motivación, para aclarar cómo se producen estos fallos y por qué necesitamos protección ante ellos. Seguidamente se entra a tratar el tema de cómo se pueden proteger a los diseños que realicemos de estos fallos, tanto mediante el uso de protecciones hardware como utilizando protecciones software. Más adelante tratamos el tema de la inyección de faltas: de dónde surge su necesidad, en qué consiste, y una breve revisión del estado del arte en inyección de faltas. Tras esto, se hará una revisión del sistema FT-Unshades y se expondrán las modificaciones propuestas y realizadas a la plataforma para tratar el problema de inyección de faltas en arquitecturas microprocesador. Una vez vistas las modificaciones a la plataforma, hablaremos de los procesadores con los que se ha trabajado: Leon3 y MicroBlaze. Hablaremos de las implementaciones realizadas y de las conclusiones obtenidas a partir del análisis de tolerancia a fallos. En el siguiente capítulo describiremos el programa de procesado de señal que corren ambos microprocesadores, y más adelante dedicaremos los dos últimos capítulos a las conclusiones y trabajos futuros del presente trabajo fin de máster.

Capítulo 2

Radiación y fallos en circuitos digitales

"Si las entiendes, las cosas son como son; si no las entiendes, las cosas son como son"
– Proverbio Zen

Cuando una partícula impacta en un circuito, puede producir, mediante distintos mecanismos que dependerán del tipo de partícula, generación de portadores en el interior de los transistores, lo que hace que los transistores conduzcan, y se cambien los estados de los biestables o las salidas de las puertas lógicas. En este caso, cambiaría el estado del circuito y podríamos tener salidas erróneas, comportamiento no predecible, o incluso por ejemplo llevar el circuito a un estado 'inexistente' que no fuera considerado durante la síntesis lógica del circuito. Esto es un problema considerable, ya que en el momento que nuestro sistema pasa a un estado indeterminado no podemos garantizar nada sobre él, ni siquiera una degradación no catastrófica.

2.1. Iones pesados y protones

Las partículas que pueden producir los cambios de estado mencionados anteriormente pueden ser, o bien protones de alta energía o bien iones pesados[1]. Aunque ambas pueden llegar a producir cambios en el estado de los circuitos digitales, el mecanismo a través del que se consigue este efecto es distinto dependiendo del mecanismo a través del cual la partícula radiada causa la anomalía. Para estimar la tasa de fallos, debemos considerar el mecanismo de la partícula concreta.

Los iones pesados causan ionización directa en el interior del dispositivo. Los protones, sin embargo, no suelen causar cambios en los circuitos por ionización

[1]Concretamente, la componente de iones pesados de los rayos cósmicos provenientes del sol u otras estrellas.

directa[2], sino a través de reacciones nucleares complejas en la cercanía de un nodo sensible (ver figura 2.1). Este proceso, en el que un núcleo pesado emite dos o más fragmentos o partículas como resultado de ser golpeado por un protón de alta energía se conoce como 'spallation[3]'. Al fragmentarse, el retroceso de los iones pesados es lo que causa los SEU.

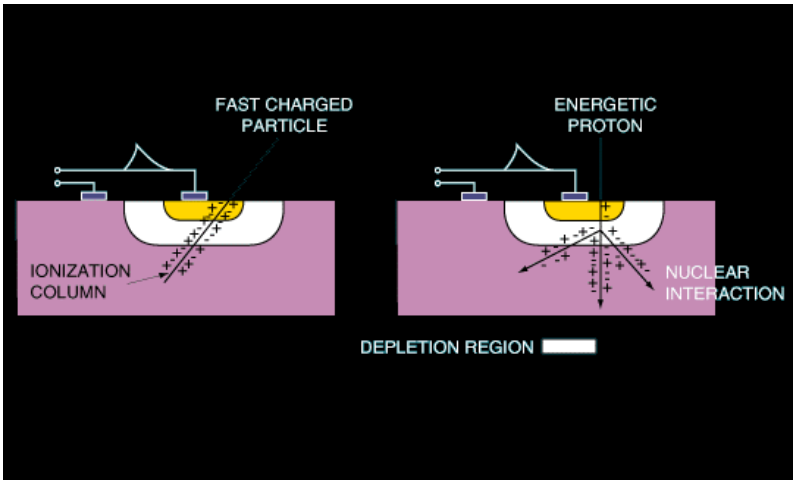

Figura 2.1: Esquema que muestra cómo los rayos cósmicos pueden depositar energía en un dispositivo electrónico. Fuente: "Spacecraft Anomalies due to Radiation Environment in Space", Lauriente y Vampola

[2]No obstante, cuando el tamaño de los transistores es menor de 0.3μm, los protones pueden provocar SEUs por medio de ionización directa.

[3]La traducción literal al español sería 'desconchado'.

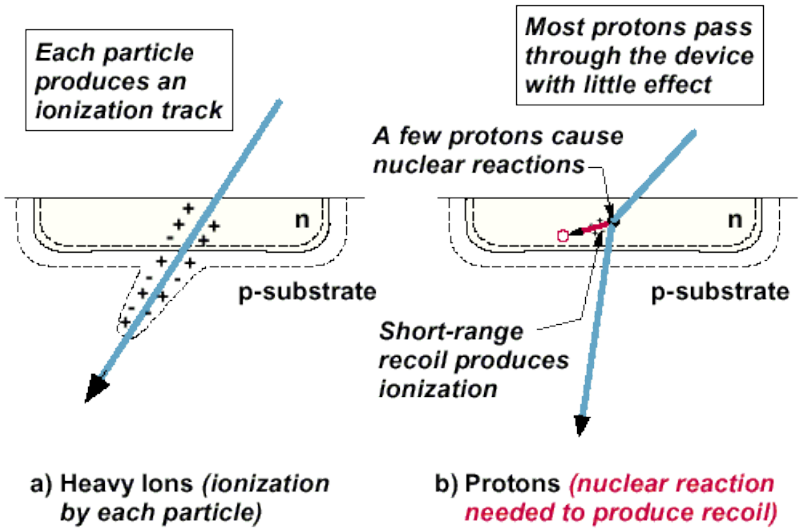

Figura 2.2: SEU por ión pesado o protón. Fuente: "Space Radiation Effects on Microelectronics", NASA Jet Propulsion Laboratory

2.2. LET y cross-section

LET son las siglas de Linear Energy Transfer. La LET es una medida de la energía transferida a un material a medida que una partícula ionizada lo atraviesa. Para el estudio de circuitos electrónicos en aplicaciones aeroespaciales, suele medirse en MeV * cm^2 / mg. Esta unidad surge al dividir de la energía perdida por la partícula por unidad de longitud (MeV * cm) entre la densidad del material (mg / cm^3)

La cross-section o sección cruzada es el cociente entre el número de eventos (cambios de estado o eventos transitorios) producidos por la radiación y la afluencia de partículas sobre el circuito (partículas/cm^2). Se mide en cm^{-2}.

2.3. Carga crítica

La carga crítica es la carga necesaria para producir un *bit-flip*, o lo que es lo mismo, convertir un '1' binario en un '0' (o viceversa). Es interesante el hecho de que esta carga es **menor** que la carga total almacenada. Concretamente, *Qcrit* es la diferencia entra la carga que almacena el nodo y la carga mínima requerida para que el amplificador de sensado de la siguiente etapa digital pueda leer el valor digital correctamente. En los circuitos SRAM, *Qcrit* depende no sólamente de la carga almacenada sino también de la forma temporal de los pulsos.

A medida que la tecnología sigue avanzando y haciéndose más y más pequeña, la carga crítica cada vez va siendo menor y se pueden provocar SEE con cada vez menos energía, incluso a altura de nivel del mar.

2.4. Modelado de las faltas

En un circuito digital, "cambia un poquito la corriente" no es una forma conveniente de modelar esos fallos: deberíamos saber cómo se traducen estos cambios en las corrientes que circulan por los dispositivos en parámetros que podamos considerar al menos a nivel de puerta lógica.

2.4.1. Faltas destructivas

Estas faltas han de combatirse con la tecnología. Existen tecnologías rad-hard (por ejemplo la de Atmel). Se distingue entre protecciones físicas (de la tecnología) y lógicas (de la información). Las faltas destructivas han de combatirse necesariamente con protecciones físicas.

2.4.1.1. Single Event Latch-up (SEL)

El single-event latchup (SEL) puede ocurrir en cualquier chip que tenga una estructura PNPN parásita. Un ión pesado o un protón con alta energía que atraviese una de los dos uniones de los transistores 'internos' puede encender la estructura tiristor, que tras encenderse, se queda cortocircuitado (un efecto conocido como latchup) hasta que se quita y vuelve a dar la alimentación del dispositivo[4]. Como este efecto puede ocurrir entre la fuente de alimentación y el substrato (tierra), se pueden crear como resultado muy altas corrientes capaces de destruir el dispositivo. Los dispositivos 'Bulk CMOS'[5] son los más susceptibles a este tipo de fallo.

[4]Lo que se conoce como un power-cycle.

[5]Es decir, aquellos implementados directamente sobre un substrato de Silicio en lugar de utilizar una capa de dieléctrico como es el caso de las tecnologías SOI.

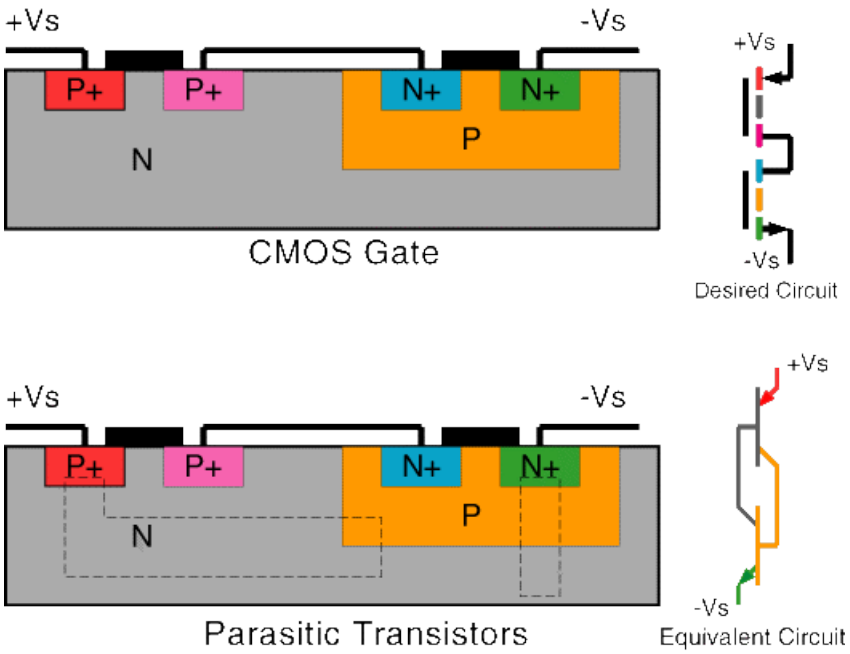

Figura 2.3: Transistores parásitos que producen el latchup

2.4.1.2. Single Event Snapback (SES)

Este evento es similar al Single Event Latchup (SEL), pero no requiere de la estructura PNPN, ya que puede ser inducido en transistores MOS de canal N que conmuten grandes corrientes, cuando un ión impacta en el drenador y causa la multiplicación por avalancha de los portadores de carga. El transistor entonces se corta y queda cortado.

2.4.1.3. Single Event Induced Burnout (SEB)

El Single Event Burnout (SEB) puede ocurrir en MOSFETs de potencia cuando la unión de la fuente con el substrato que está justo debajo se polariza en directa y la tensión drenador-fuente es mayor que la tensión de ruptura de las estructuras parásitas. La alta corriente resultante y el sobrecalentamiento local pueden destruir el dispositivo.

2.4.1.4. Single Event Gate Rupture (SEGR)

El Single-event gate rupture (SEGR) se puede observar en los MOSFETs de potencia cuando un ión pesado impacta en la región de puerta mientras una tensión alta es aplicada a la puerte. Una ruptura de dieléctrico local ocurre entonces en la capa aislante de dióxido de silicio, causando sobrecalentamiento global y

destrucción[6] de la región de puerta. Es interesante de este efecto que puede ocurrir incluso en celdas EEPROM durante la escritura o el borrado, cuando las celdas están sujetas a una tensión comparativamente alta.

2.4.1.5. Total Ionizing Dose (TiD)

Los dispositivos electrónicos también sufren efectos de la radiación a largo plazo, mayormente debidos a los electrones y a los protones. Las fuentes principales de estas partículas son los 'Solar Energetic Particle Events', que ocurren en asociación con las erupciones solares [7] y la Anomalía Atlántico Sur (South Atlantic Anomaly, ver 3.3.1), donde la magnetosfera terrestre está más cerca de la superficie de la tierra, por lo que los circuitos electrónicos encuentran más radiación atrapada. El daño acumulado a largo plazo debido a los protones y electrones puede hacer que los dispositivos sufran desviaciones en sus umbrales, incrementen el 'leakage' o corriente de fuga de los dispositivos (y consecuentemente su consumo de potencia), sufran cambios en los parámetros temporales (timing) y reduzcan, en general, sus prestaciones[8].

El 'shielding' de los dispositivos (escudos de plomo u otros materiales que frenen la radiación) puede ayudar, pero han de considerarse varios factores. La geometría del escudo, las técnicas de análisis utilizadas, y la composición de los dispositivos son todos parámetros relevantes para predecir la efectividad del escudo. Los electrones pueden ser efectivamente atenuados por escudos de aluminio incluso a altas energías. Sin embargo, mientras los escudos de aluminio son efectivos para protones de baja energía, son inefectivos para los protones de alta energía (>30 MeV).

La TiD, debida mayormente a electrones y protones, puede resultar en fallo de dispositivos o daño biológico en los astronautas. En ambos casos, la TiD se puede medir en términos de la dosis absorbida, que es una medida de la energía absorbida por unidad de masa. La dosis absorbida es cuantifica usando, o bien una unidad llamada rad (siglas de 'radiation absorbed dose') o la unidad del Sistema Internacional (SI) que es el gray (Gy). 1 Gy = 100 rad = 1 J/Kg.

[6] Como una explosión microscópica.

[7] Las erupciones solares son violentas explosiones en la atmósfera del Sol, que pueden liberar una energía de hasta 6×10^{25} Julios, equivalente a decenas de millones de bombas de hidrógeno.

[8] El primer satélite que sufrió un fallo debido a Total Dose fue el Telstar. El Telstar fue lanzado el día después de la prueba nuclear del Starfish del 9 de Julio de 1962. Starfish Prime era un misil nuclear de una fuerza de 1.4 Megatones y fue detonado a una altura de unos 400 km sobre Johnston Island en el Océano Pacífico. La explosión produjo partículas beta (electrones) que fueron inyectados en el campo magnético de la tierra y formaron un cinturón de radiación artificial. Este cinturón artificial de electrones se mantuvo hasta principios de los años 70. El Telstar experimentó una dosis total 100 veces superior a la esperada debido al test del Starfish. Starfish destruyó siete satélites en siete meses, principalmente a causa de daño en los paneles solares.

La TiD se calcula a partir de los protones y electrones atrapados, los fotones secundarios Bremsstrahlung (radiación de frenado) y los protones provenientes de erupciones solares, ya que la contribución de los iones provenientes de rayos cósmicos es despreciable en presencia de estas otras fuentes. Las fuentes principales de protones y electrones son

1. 'Solar Energetic Particle Events' (provenientes de erupciones solares), y

2. paso a través de la South Atlantic Anomaly.

En la órbitas LEO (Low Earth Orbit), la fuente principal de dosis viene de los protones y electrones del cinturón interior de Van Allen, mientras que en las órbitas geostacionarias (GEO: Geostationary Earth Orbit) las fuentes principales son los electrones del cinturón exterior y los protones provenientes del Sol.

La dosis total acumulada depende de la altitud de la órbita, la orientación y el tiempo transcurrido. Esta TID se puede calcular, pero necesitamos conocer el espectro integrado de energía de las partículas, phi(E), es decir, la fluencia como función de la energía de las partículas. Los satélites y las sondas espaciales encuentran típicamente TIDs entre 10 y 100krad. Los dispositivos se pueden probar, antes de su uso, en instalaciones de laboratorio específicas, como aceleradores de partículas[38]. En las órbitas LEO, los Single Event Effects (SEE) son un problema más significativo aunque en 1991 se descubrió que los dispositivos bipolares tendían a degradarse más al ser expuestos a las 'dosis pequeñas' (menores de 0.1 rad/sec) que se encontraban en estas órbitas que cuando eran sometidos a las dosis de radiación mayores (por ejemplo, entre 50 y 300 rad/sec) que se utilizan normalmente en los tests de laboratorios. Muchos dispositivos bipolares, notablemente los transistores PNP laterales, muestran esta sensibilidad mayor a las dosis bajas[9].

Al incrementarse la TID, los materiales se degradan progresivamente, por ejemplo, la energía que da una celda solar va reduciéndose[10] y, como se ha comentado anteriormente, la exposición a largo plazo de los dispositivos puede acarrear incrementos en las corrientes de fuga y consumo de potencia, y demás cambios en los parámetros electrónicos de los dispositivos.

[9]En la jerga técnica, se habla de ELDRS o Enhanced Low Dose Rate Sensitivity.

[10]Curiosamente, esto es una causa muy frecuente de cese de funcionamiento en los satélites.

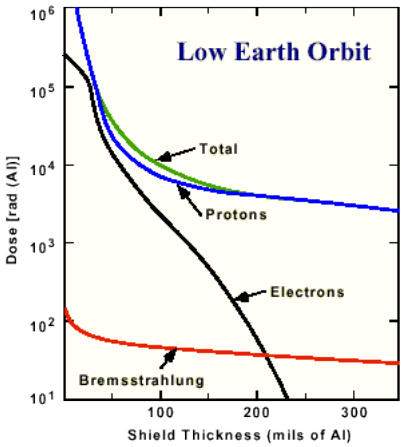

Figura 2.4: Fuente: "Space Radiation Effects on Microelectronics", NASA Jet Propulsion Laboratory

En los semiconductores, la ionización excita a los portadores de la banda de conducción a la banda de valencia. La radiación ionizante principalmente afecta a la puerta y al óxido de campo, SiO_2. La ionización produce pares electrón-hueco a una tasa de 8.1x1012 pares/rad(SiO2)cm3. Los electrones producidos se son barridos rápidamente, pero los huecos tienen una movilidad mucho menor. Una fracción (aproximadamente 1/5) de los huecos son transportados y quedan atrapados en el interfaz entre el Si y el SiO_2. La carga atrapada, en el óxido y en las regiones de interfaz, cambia la tensión umbral y movilidad de los transistores, modificando de esta forma sus características. Los huecos atrapados no son estables, y gradualmente se recombinan con el tiempo, aunque este tiempo puede variar entre horas y años. El efecto global depende de las condiciones de polarización y de la tecnología del dispositivo: un efecto típico es el desplazamiento de la tensión umbral en los transistores MOS, como se ve en la figura 2.5.

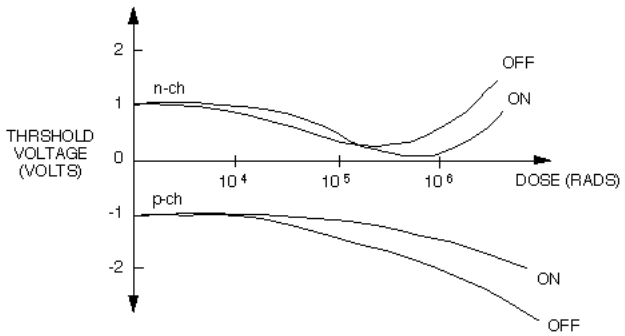

Figura 2.5: Fuente: "The NASA ASIC Guide: Assuring ASICS for Space"

Ya que tanto la TID como los SEE (single event effects) son causados por la

radiación ionizante, es importante comentar la diferencia entre los dos con respecto al diseño y análisis. La TID es un mecanismo de fallo a largo plazo frente a los SEE que son mecanismos de fallo instantáneo.

2.4.1.6. Displacement Damage (DD)

El Daño por Desplazamiento o Displacement Damage (DD) es el resultado del intercambio de energía cinética con las partículas energéticas que impactan en el dispositivo. Los efectos dañinos provocados por estos intercambios de energía son proporcionales a la cross section de daño por desplazamiento D. Esta cantidad es equivalente a la pérdida de energía no ionizante (NIEL o Non Ionizing Energy Loss). También existe otra métrica llamada KERMA (Kinetic Energy Released to MAtter), o Energía Cinética Liberada a la Materia. La cantidad, ya sea referida en D, NIEL o KERMA, es responsable del desplazamiento de los átomos en la red cristalina. Estos desplazamientos afectan al comportamiento del silicio cristalino, deteriorando las propiedades de los semiconductores.

2.4.2. Faltas no destructivas

Las faltas transitorias[11] se modelan, según si su efecto cambia el contenido de un biestable o altera de forma temporal la salida de una puerta lógica, respectivamente como:

2.4.2.1. Single Event Upsets (SEUs)

El contenido de un biestable cambia (de ‘0’ a ‘1’ o bien de ‘1’ a ‘0’) como resultado de la radiación.

[11]Las faltas se pueden repartir en tres categorías: Permanentes, si reflejan daño físico irreversible en el hardware, Transitorias, si son inducidas por condiciones externas no permanentes, e Intermitentes, si ocurren debido a la inestabilidad del hardware.

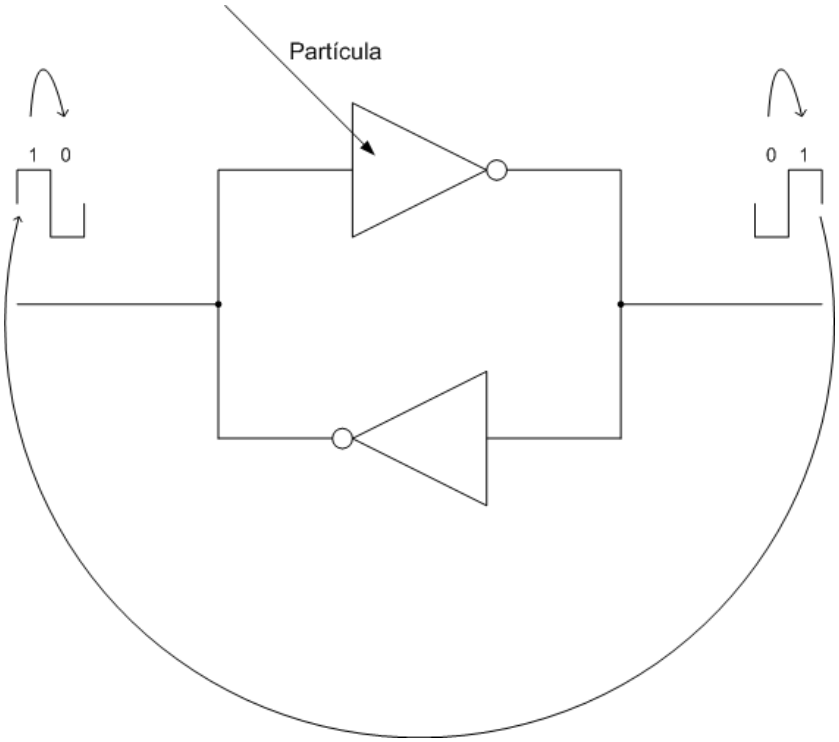

Figura 2.6: Single Event Upset. Efecto a nivel lógico

2.4.2.2. Single Event Transient (SETs)

La salida de una puerta lógica cambia temporalmente debido a que un transistor de la misma, que debería estar cortado según la entrada de la puerta, conduce a causa de los efectos de deposición de carga en el canal producidos por la radiación.

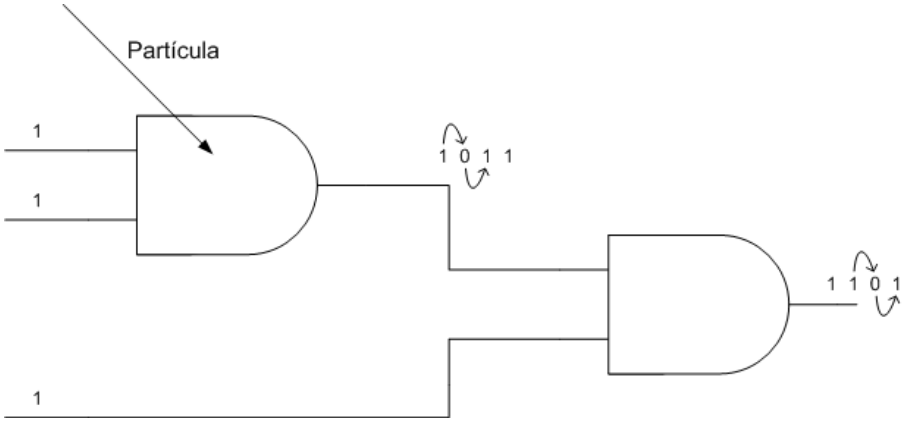

Figura 2.7: Single Event Transient

Para referirnos a las faltas SEU y SET indistintamente, hablaremos de SEE o Single Event Effects.

2.4.2.3. Multi Bit Upsets (MBUs)

Pasa la partícula por un sitio que está petao de transistores cual chinatown a la hora del almuerzo, por lo que petan todos a la vez. La ocurrencia de esta falta depende de la topología del circuito: ha de ocurrir que en el circuito se encuentren varios biestables en un área muy pequeña, como es el caso de RAMs, donde más suelen ocurrir esta clase de faltas.

2.4.2.4. Single Event Multi Upsets (SEMUs)

Un evento transitorio (SET) es capturado por varios biestables a la vez. Los biestables no tienen que estar cerca los unos de los otros, sólo debe cumplirse que el retraso de propagación del pulso transitorio (SET) a los biestables es parecido.

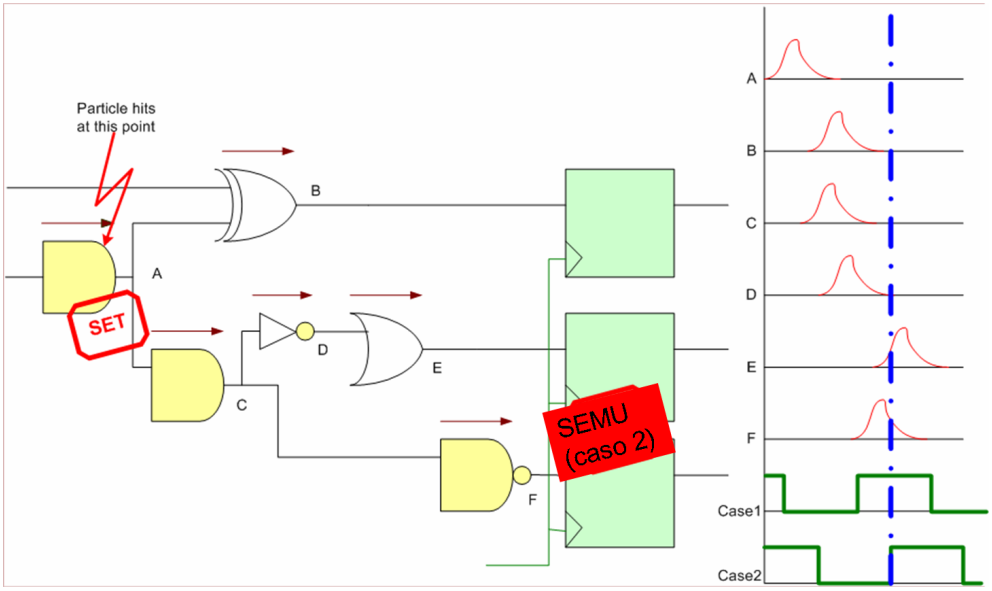

Figura 2.8: SET y SEMU. El Multi-Upset es producido al capturarse la falta transitoria por varios biestables simultáneamente

Capítulo 3

¿Por qué Tolerancia a Fallos?

"¿Quién es éste General Failure y qué hace leyendo mi disco duro?"
– Chascarrillo informático

Sabemos que la radiación puede producir fallos en los circuitos electrónicos, concretamente en los circuitos digitales, que serán el objeto de nuestro estudio. Ya sabemos también que podemos modelar estos efectos en dos categorías básicas: Single Event Upsets (SEUs) y Single Event Transients (SETs).

La siguiente pregunta que cabe plantearse es: ¿es realmente necesario diseñar e implementar protecciones ante estos fallos? Si no esperásemos un fallo catastrófico hasta más alla de la vida útil del producto, no tendría por qué ser necesario. Desafortunadamente, hay ciertas razones que hacen que la protección de los circuitos sea más que necesaria:

3.1. Aplicaciones Espaciales

En el momento que un grupo investigador o de desarrollo quiera plantearse una aplicación espacial, la tolerancia a fallos se convierte en un requisito indispensable de los sistemas a desarrollar, debido a que no tenemos atmósfera que proteja a nuestros circuitos de la radiación, y el número de fallos esperados durante la vida del circuito crece enormemente.

3.2. Miniaturización cada vez mayor

La miniaturización cada vez mayor (tecnologías de 90, 65, 45nm y bajando...), presenta las siguientes características en los nuevos circuitos:

- Baja tensión de alimentación

- Alta frecuencia de funcionamiento
- Tamaños de componentes reducidos
- Alta densidad de dispositivos

Todas estas características, sobre todo las dos últimas, hacen que los dispositivos sean más sensibles a las radiaciones ionizantes. Se piensa que en el futuro, con la reducción de los tamaños de las tecnologías, la cantidad de SEUs y SETs producidos en un circuito dentro de la atmósfera será significativa y habrá que diseñar los circuitos considerando estos fallos como eventos esperados durante la vida de los mismos.

3.3. Radiación a alturas cercanas al nivel del mar

Si bien el número de neutrones que llegan a los dispositivos a nivel del mar hace que los SEUs/SETs sean mucho menos probables, la constante miniaturización comentada anteriormente puede hacer vulnerable nuestros dispositivos a radiación de menor energía. Hoy día existen estudios que demuestran una correlación entre el número de fallos encontrados durante la vida de un circuito y la altura a la que funciona, y se encuentran fallos incluso cuando estos circuitos están 'supuestamente' protegidos contra las radiaciones por la atmósfera terrestre.

También tendremos que proteger los circuitos que, aunque no hayan de viajar al espacio, vayan a trabajar en entornos hostiles, como por ejemplo los circuitos de control de un reactor nuclear (este es un ejemplo particularmente crítico).

3.3.1. South Atlantic Anomaly (SAA)

Incluso si nos mantenemos a la altura del nivel del mar, existen zonas de la superficie terrestre que son más sensibles a la radiación, por ejemplo el South Atlantic Anomaly (SAA), la región en la que el cinturón interior de Van Allen está más cerca de la superficie de la tierra. Estos cinturones de Van Allen, que atrapan la radiación proveniente del espacio, llevan parte de la radiación atrapada a esta zona, en la cual se registra mucha más radiación a nivel del mar que en cualquier otra zona del mundo.

Además, no es una situación del todo estática: el SAA se mueve hacia el oeste unos 0.3 grados por año.

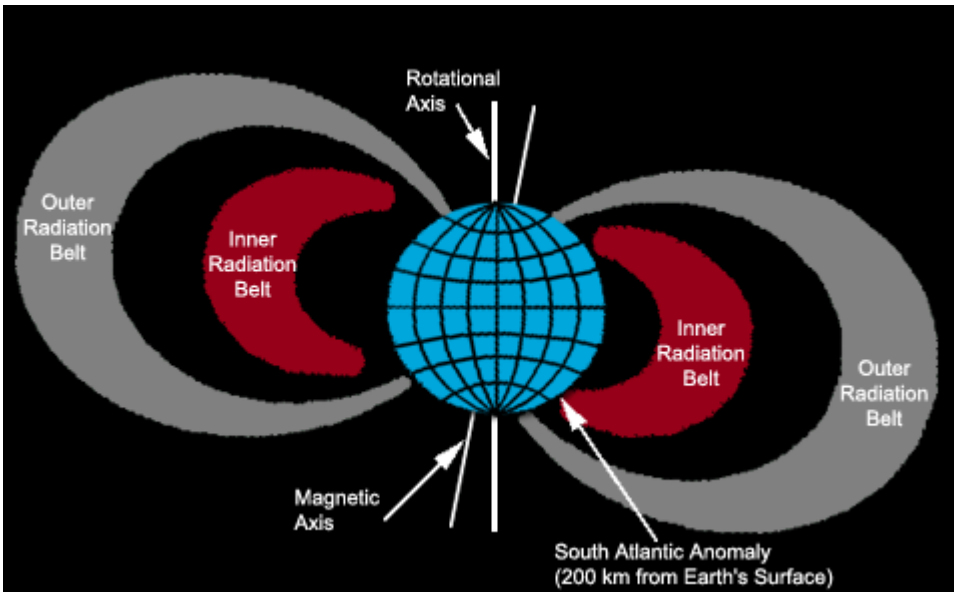

Figura 3.1: South Atlantic Anomaly. Cortesía de NASA

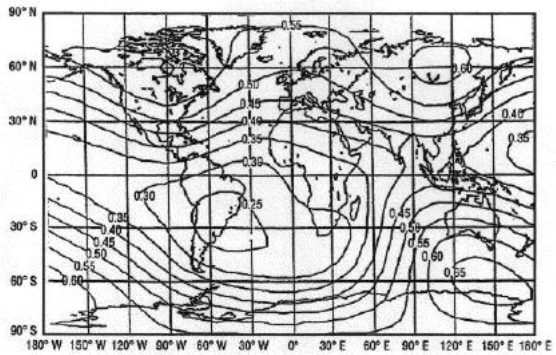

Figura 3.2: Geomagnetic field at sea level. Note the South Atlantic Anomaly (SAA) which is centered off the southeastern coast of South America. Fuente: from the Space Environments and Effects Program at NASA's Marshall Space Flight Center

Capítulo 4

¿Cómo protegernos?

"Cuando te enfrentes con un luchador más fuerte que tú, deberás moverte más rápidamente que él. Y si te alcanza... que Dios te ayude"
– Les Luthiers, Iniciación a las artes marciales

Básicamente, tenemos dos formas de protegernos de estos fallos: proteger el hardware y proteger el software. Es objeto de este trabajo el estudio de las protecciones software.

Además de las protecciones en el uso de tecnología 4.1

4.1. Tolerancia a Fallos Implementada en Hardware

Las técnicas de protección del hardware no son el objeto principal de este estudio, pero de ellas podemos comentar que requieren recursos adicionales (tanto de área como de consumo de potencia) y han de contemplarse previamente a la fabricación del circuito integrado. Estas técnicas suelen basarse en la triple redundancia modular (TMR) y extensiones de este paradigma, y encarecen enormemente el coste de diseño y fabricación de los circuitos integrados.

- Protecciones físicas
- Protecciones lógicas

Además de las protecciones físicas, características de las tecnologías rad-hard, existen las llamadas protecciones lógicas.

Algunas protecciones físicas:

- A menudo, se fabrican chips endurecidos sobre substratos aislados en lugar de utilizar las obleas usuales de semiconductor. Las más comunes son las tecnologías SOI (sobre óxido de silicio) y la SOS (sobre óxido de zafiro). Mientras que los chips comerciales estándar pueden soportar entre 5 y 10 krad, los chips SOI y SOS cualificados para espacio pueden sobrevivir a dosis de radiación varios orden de magnitud mayores.

- Shielding del encapsulado contra radioactividad, reduciendo la sensibilidad que tendría el dispositivo sin encapsular.

- La DRAM basada en condensadores se reemplaza normalmente por SRAM, con mayor área y por ello más cara, pero con menor sensibilidad a los efectos de la radiación[30].

- Si se escoge un substrato con una amplia banda prohibida, como el *silicon carbide* o el *gallium nitride*, el circuito resultante tendrá mayor tolerancia a los 'deep-level effects'.

- Shielding de los dispositivos utilizando Boro empobrecido (que contiene únicamente el isótopo Boro-11) en la capa de cristal de borofosfosilicato que protege a los chips, en lugar del isótopo Boro-10, que captura fácilmente neutrones, emitiendo partículas alfa y produciendo 'soft errors'[1].

Algunas protecciones lógicas a nivel de sistema:

- Las memorias correctoras de errores utilizan bits adicionales de paridad para detectar y posiblemente corregir datos corruptos. Ya que los efectos de la radiación dañan los contenidos de memoria incluso cuando el sistema no está accediendo a la ram, un circuito debe barrer continuamente la memoria RAM, leyendo los datos, comprobando la paridad para saber si hay errores, y escribiendo cualquier corrección, en el caso en que sea posible, en la RAM. Esta técnica también que se puede utilizar para restaurar valores corrompidos en FPGAs basadas en memoria SRAM[11, 41] y se conoce como 'scrubbing'.

- Se pueden utilizar elementos redundantes a nivel de sistema, por ejemplo tres placas microprocesador que computen un resultado y comparen sus respuestas. Cualquiera de los sistemas, si produce un resultado minoritario, repetirá los cálculos. También se puede añadir lógica para desconectar uno de los sistemas si presenta errores repetidamente.

[1] Faltas no destructivas y de carácter transitorio, que veremos más adelante.

- Como último recurso, otra protección que se suele utilizar son los watchdog, timers que resetean el sistema salvo que se vuelvan a preinicializar antes de que su cuenta llegue a cero. Si la radiación hace que un microprocesador comience a funcionar incorrectamente o incluso deje de funcionar, al no reescribir el watchdog, la cuenta de éste llegará a cero y se realizará un 'hard reset' del sistema.

4.1.1. Triple redundancia modular

La triple redundacia modular es la elección común para las protecciones lógicas a nivel de circuito. Se basa en utilizar elementos redundantes, pero a nivel de circuito y no de sistema como en el apartado anterior. Cada bit se reemplaza por tres bits y una lógica de votación (*voting logic*), que determina su resultado. Esto incrementa el area de un chip por un factor de más de 3, por lo que para diseños especialmente grandes no es demasiado práctico[2]. Como ventaja de este esquema podemos señalar que, si se produce un fallo en un único bit de un dominio de triple redundancia[3], el error será corregido en tiempo real, es decir, sin divergir o retrasar el flujo de programa como ocurriría si utilizáramos un watchdog o las técnicas de protección del software que veremos en la próxima sección.

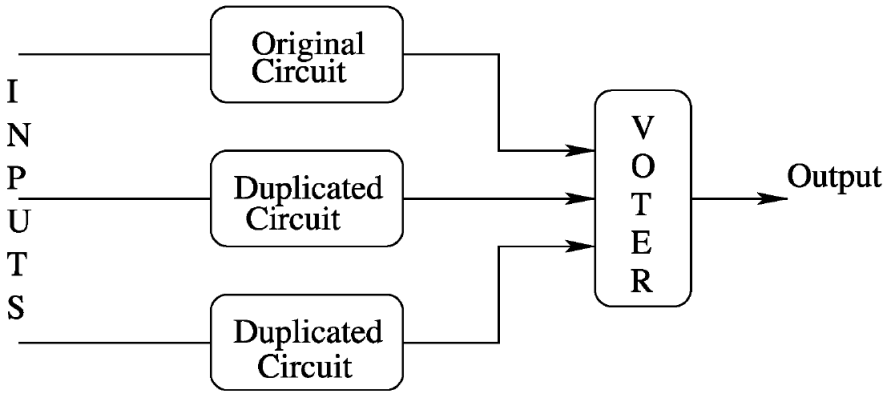

Figura 4.1: Triple redundancia modular

[2]Dependiendo del autor que estudie este problema, se pueden encontrar factores desde 3.2 hasta 5.

[3]Si el fallo se produce en dos bits del mismo dominio, se convierte en mayoritario y no sólo no se corrige, sino que se propaga a través de la lógica de votación. Esta técnica de protección, por tanto, se basa en que es relativamente raro que dos transistores del mismo dominio sufran un bit-flip en el mismo ciclo de reloj (si es necesario se utilizan técnicas de rutado especiales para garantizar que ésto).

4.2. Tolerancia a Fallos Implementada en Software

La gran ventaja de estas técnicas es el hecho de que se pueden realizar si tenemos dispositivos COTS (commercial off-the-shelf) aunque, como veremos más adelante, estas técnicas no protegen a un único procesador contra faltas en cualquiera de sus subsistemas (existen módulos críticos, como el de reset, que seguirán siendo sensibles aunque protejamos el software).

4.2.1. Técnicas SIFT

Existen muchas técnicas de protección del software: las técnicas SIFT proporcionan redundancia activa y/o pasiva para protegernos de los fallos que ocurran en los datos o el código del programa. Se realizan la detección, localización y recuperación de las faltas en software. De las distintas técnicas que existen, nos centraremos en las técnicas que permiten una protección automática de un software ya existente.

4.2.2. Técnicas de propósito general

Estas técnicas no son dependientes de que utilicemos un cierto algoritmo u otro. Son básicamente versiones software de técnicas ya conocidas para la protección del hardware.

4.2.2.1. N-version programming

Consiste en implementar el concepto de redundancia hardware en software: N equipos diferentes desarrollan N versiones de un software que se adhiere a una misma especificación. Para casos prácticos, resulta demasiado cara en términos de tiempo de desarrollo, tiempo de procesado y coste de ingeniería. Y tiene el problema de que si existe un problema en la especificación, las N versiones fallarán.

4.2.2.2. Recovery blocks

Consiste en implementar el concepto de redundancia híbrida en software: se tienen varias réplicas del mismo módulo, y unos módulos adicionales de control que se encargan de

- Guardar en memoria un estado correcto del sistema
- Escoger a uno de los N módulos para realizar el procesado
- Decidir si los resultados proporcionados por el módulo escogido son aceptables o no

- Restaurar el estado correcto del sistema si los resultados no son aceptables (es decir, ha habido una ejecución fallida)

No es una técnica sencilla de implementar, por ejemplo 'Decidir si los resultados proporcionados por el módulo escogido son aceptables o no' no es una acción trivial y dependerá de qué clase de problema encaremos. Todos estos módulos de ejecución, aceptación y guardar/restaurar estado necesitan ser desarrollados, validados y, a ser posible, protegidos a su vez contra fallos.

4.2.3. Técnicas específicas

4.2.3.1. ABFT: Algorithm-Based Fault Tolerance

Existen ciertas técnicas que únicamente se pueden utilizar si trabajamos con cierto tipo de algoritmos, o ciertos tipos de datos. Si bien estas técnicas carecen de generalidad, pueden ser muy útiles y eficientes para problemas específicos. Por ejemplo, ABFT se utiliza para hacer más robustas las operaciones entre matrices, y se basa en tener una fila y/o una columna adicionales en cada matriz. La fila y columna adicional guardan una suma de verificación de los datos de las columnas o filas de la matriz. Al multiplicar una matriz 3x3 + fila de checksum por otra matriz 3x3 + columna de checksum obtendremos una matriz 3x3 + fila de checksum + columna de checksum + casilla de checksum total. Esto nos permite detectar y corregir errores en la multiplicación final. Lamentablemente esta técnica no es general y tampoco es automática.

4.2.4. Protección de los datos

De las siguientes técnicas que veremos, unas están orientadas a proteger los datos que manejará nuestro programa, mientras que las otras se centran más en proteger el flujo de control de nuestro programa, esto es, que nuestro programa no haga saltos inesperados (como consecuencia de fallos en las instrucciones de salto o en el contador de programa).

4.2.4.1. SWIFT

SWIFT es una técnica automática y general, que permite detectar y localizar errores en los datos del programa. El funcionamiento de esta técnica es muy sencillo:

- Replicamos las variables 2 veces
- Las operaciones entre variables también se duplican

- Tras cada uso de una variable, se comprueba la consistencia de las 2 réplicas
- Si hay disparidad de valores, es que hemos tenido un error

Un ejemplo práctico de esto sería:

Código original:	**Código modificado**

a=b+c;	a0=b0+c0;
	a1=b1+c1;
	if(b1!=b0)
	  error(b0,b1);
	if(c1!=c0)
	  error(c0,c1);

Figura 4.2: Ejemplo de código SWIFT

Esta técnica nos permite introducir la redundancia automáticamente. Eso sí, doblamos los segmentos de datos y código, e incrementamos el tiempo de ejecución. Como sólo trabajamos a nivel de código, esta técnica es independiente del procesador que utilicemos.

4.2.4.2. SWIFT-FT

SWIFT-FT es una variante de SWIFT que, además de detectar y localizar fallos, es capaz de corregirlos. Esto se consigue teniendo un checksum del segmento de datos a proteger, de forma que podamos comparar cualquiera de los segmentos de datos con el checksum y así determinar cuál es el incorrecto:

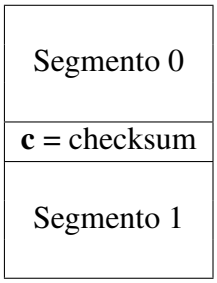

Figura 4.3: Datos replicados por SWIFT-FT

- Eliminamos a0 del checksum
- Duplicamos la variable a=b

- Añadimos a0 al checksum
- Si b0 no coincide con b1, comprobamos el checksum del segmento 0 (en este caso, b0)
- Si el checksum de b0 coincide con c, el segmento dañado es el 1, y lo sustituimos por el 0
- Si el checksum de b0 no coincide con c, el segmento 0 está dañado y lo sustituimos por el correcto. Además en este caso debemos regenerar el checksum correcto una vez que restauremos el segmento 0

Código original:	**Código modificado**

a=b;	c = c XOR a0;
	a0 = b0; a1 = b1;
	c = c XOR a0;
	if(b0!=b1)
	  if(f(Segmento 0)==c)
	  { b1 = b0; a1 = a0;
	  } else
	  { b0 = b1; a0 = a1;
	  c = calcula_checksum(); }

Figura 4.4: Ejemplo de código SWIFT-FT

Es importante comentar que tanto el SWIFT como el SWIFT-FT, las variables se comprueban cuando se acceden para su uso, y no cuando se generan: es decir, comprobaremos 'a' en el momento en que la vayamos a usar para alguna operación.

4.2.4.3. ED^{4}I

ED^{4}I, desarrollado en la Universidad de Stanford, se basa en computar, a la vez que el resultado normal de nuestro algoritmo, llamémosle S=f(x), una solución desplazada S'=f(x*k). Tras esto, debemos comprobar que los resultados S y S' son consistentes. Si bien tiene mucho sentido desde un punto de vista matemático, esta técnica no es automática por lo que no nos interesa estudiarla en profundidad.

4.2.5. Protección del control-flow

Estas técnicas nos ofrecen soluciones para proteger el flujo de programa o control-flow de nuestra aplicación. Se basan en particionar un programa en bloques básicos que forman un diagrama de flujo, y controlar en todo momento que las transiciones entre bloques son únicamente las permitidas por el diagrama de flujo (por ejemplo, que no vayamos del principio al final saltando pasos intermedios).

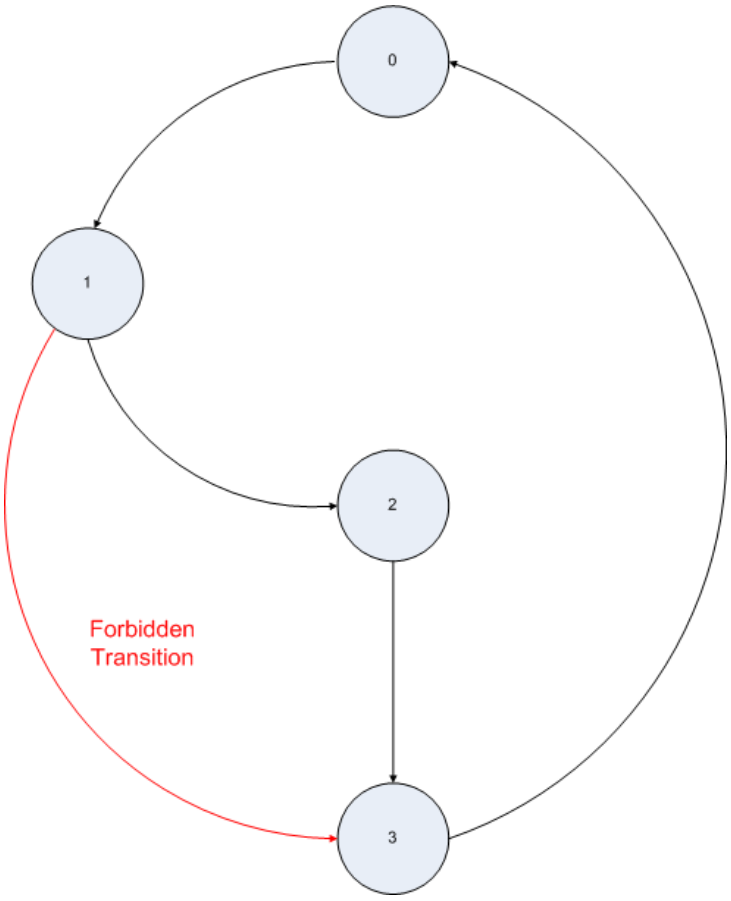

Figura 4.5: Transiciones permitidas y no permitidas

4.2.5.1. ECCA

Desarrollada en la UT (Universidad de Texas), es una técnica basada en realizar modificaciones sobre el código fuente y detectar saltos no permitidos mediante divisiones por cero:

- Asignamos una firma impar a cada bloque de ejecución básico del programa

- Se mantiene una firma (signature) de ejecución del programa
- Cuando entramos en un bloque básico, cambiamos la firma de ejecución del programa a la del bloque en el que entramos y comprobamos que el cambio está permitido
- Cuando salimos de un bloque básico, cambiamos la firma de ejecución del programa a la del bloque al que vamos

Al pasar del bloque 2 al bloque 4:
firma = BID2/((!(id %BID1))*(id %2));
código bloque 2;
firma = BID4+!!(id-BID2);

Figura 4.6: Código ECCA ejecutado al pasar del bloque 2 al bloque 4

Esta técnica es muy efectiva para detectar los errores en el flujo de programa, pero también es muy costosa ya que realiza todas las comprobaciones usando divisiones.

4.2.5.2. CFCSS

Es una técnica parecida a la anterior, pero en lugar de utilizar divisiones, se utilizan operaciones lógicas para comprobar que la ejecución es correcta, con lo cual se reduce muchísimo el coste de ejecución de la técnica.

4.2.5.3. YACCA

Es una mejora de CFCSS en la que tenemos dos firmas asignadas a cada bloque de ejecución básico: una firma de entrada y una firma de salida. Mantendremos constantemente una firma de ejecución, que nos indica en qué bloque de ejecución estamos.

Cuando entremos en un bloque básico:

- Comprobaremos que la firma de ejecución actual es correcta
- Cambiaremos la firma actual: ahora será la de entrada del bloque

Y cuando salgamos de un bloque básico:

- Comprobaremos que la firma actual es correcta
- Cambiaremos la firma actual a la de salida del bloque

Al pasar del bloque 2 (viniendo del bloque 1) al bloque 4:
ERR = firma XOR B12; (comprobamos que llegamos del bloque 1) firma = firma XOR M21; (firma = entrada del bloque 2 (B21)) código bloque 2; ERR = firma XOR B21; (comprobamos bloque 2 ejecutó correctamente) firma = firma XOR M22; (firma = salida del bloque 2 (B22))

Figura 4.7: Código YACCA ejecutado al pasar del bloque 2 al bloque 4

Capítulo 5

Inyección de faltas

"I felt like destroying something beautiful"
– Narrator, Fight Club

5.1. Necesidad

¿Para qué necesitamos la inyección de faltas? Surge por la necesidad de comprobar efectivamente que nuestro diseño se encuentra protegido. La inyección de faltas se basa en recrear las condiciones de fallo por radiación de nuestro circuito, ya sea sobre una simulación de nuestro circuito, sobre un prototipo o incluso sobre un sistema final, y comprobar si el circuito es capaz de recuperarse de los errores producidos o si por el contrario se ve afectado negativamente. Si bien existen tantas clasificaciones como autores, en el presente documento clasificaremos las técnicas de inyección de faltas de la siguiente manera[1]:

- Simulada
- Emuladas sobre FPGAs
  - Instrumentada
  - No Instrumentada
- Emuladas sobre prototipos
  - Pin-Level

[1]En la clasificación propuesta, no se consideran a los tests de radiación como inyección de faltas ya que estos tests de radiación son precisamente lo que se pretende 'emular' a través de estas diversas técnicas de inyección, y porque un test de radiación no da información del instante de tiempo ni del registro en el que se insertan los fallos, a diferencia de la mayoría de las técnicas que veremos a continuación.

- Code Event Upset
- Técnicas Láser

En las siguientes secciones se describirán cada una de las técnicas y al final del capítulo se mostará una tabla comparativa de las distintas técnicas y sus mejores ámbitos de aplicación.

5.2. Técnicas de inyección de faltas

5.2.1. Simulada

Esta técnica se conoce como SBFI o Simulation-Based Fault Injection, y consiste en simular un modelo del MUT (Module Under Test) en un ordenador. Los fallos se inyectan modificando los niveles lógicos del modelo durante la simulación[2]. Estas técnicas permiten realizar estudios de tolerancia a fallos en la fase de diseño del DUT, cuando todavía no existen prototipos del mismo. Tienen la gran ventaja de que la observabilidad y controlabilidad de los registros se maximizan, si bien los tiempos de computación hacen que sea muy lenta en relación a otro tipo de técnicas.

El método más extendido consiste en realizar modelos VHDL del MUT. Este lenguaje permite la descripción del sistema en varios niveles de abstracción, dotando de gran flexibildad a la técnica.

Entre las plataformas SBFI consolidadas se encuentran MEFISTO[10], desarrollado por la Universidad Chalmers de Goteborg (Suecia) y el LAAS Research Centre de Toulouse (Francia), y AMATISTA[17, 27], desarrollado por el Politéctnico de Turín (Italia), la Universidad Carlos III de Madrid (España) y Alcatel Espacio (España) .

Además, se puede señalar la relevancia de SST[16], una herramienta de inyección de faltas desarrollada por la Agencia Espacial Europea (ESA), concebida originalmente para validar el funcionamiento del primer prototipo de FT-UNSHADES (ver sección 5.2.2.2) y actualmente utilizada por la Universidad Antonio de Nebrija para sus estudios de tolerancia a fallos[33].

5.2.2. Emuladas sobre FPGA

Estas técnicas se basan en emular el circuito a probar en una FPGA. Realmente, se sintetizan dos copias del modelo del MUT, una sobre la que se inyectarán los fallos (Seu MUT), y otra que servirá de referencia de una ejecución correcta, para

[2]Por ejemplo, forzando con Modelsim el cambio en el valor lógico de una señal interna del circuito.

poder comparar (Gold MUT)[3]. Para ello es necesario disponer de una descripción HDL del mismo. Debido a que el circuito emulado es programado directamente en una FPGA, no es necesario simular su funcionamiento (el circuito funciona en tiempo real en la FPGA), estas técnicas son varios órdenes de magnitud más rápidas que las simuladas aunque, a diferencia de éstas, presentan un problema no trivial de controlabilidad (para insertar las faltas) y observabilidad (para conocer cómo se propagan) de los bits del sistema. Según cómo se resuelva este problema, los sistemas de inyección de faltas emulados sobre FPGA pueden clasificarse en dos subcategorías: Instrumentados y No Instrumentados, dependiendo de si se modifica o no la netlist del MUT.

5.2.2.1. Instrumentada

Si modificamos la netlist del circuito a probar para poder tener acceso en lectura y escritura a los registros del mismo, estamos ante un esquema instrumentado. La idea básica es añadir circuitería extra para resolver el problema de controlabilidad y observabilidad en todos los bits del circuito. Esto tiene dos grandes desventajas: la primera, que el tamaño del circuito se multiplica por un factor de aproximadamente 4, por lo que la técnica es poco práctica para circuitos complejos, y la segunda y más importante, que al haber instrumentado la netlist, el circuito que se está probando es diferente del que pretendíamos probar originalmente[4]. Además, en el caso de que se usen primitivas internas de la FPGA, como las memorias empotradas, hay que duplicar e instrumentar dichas primitivas de forma específica, multiplicando el número de recursos utilizados[15]. La ventaja principal de esta técnica es que es la más rápida, pudiendo conseguir velocidades de inyección de hasta 1μs por falta[31].

Este es el caso del sistema autónomo desarrollado por la Universidad Carlos III de Madrid[26],

[3]Podemos ver un diagrama de éste modelo en la sección 6.2.

[4]Este problema tiene graves implicaciones, puesto que de forma rigurosa no serían válidos para el circuito sin instrumentar los resultados obtenidos sobre el circuito instrumentado.

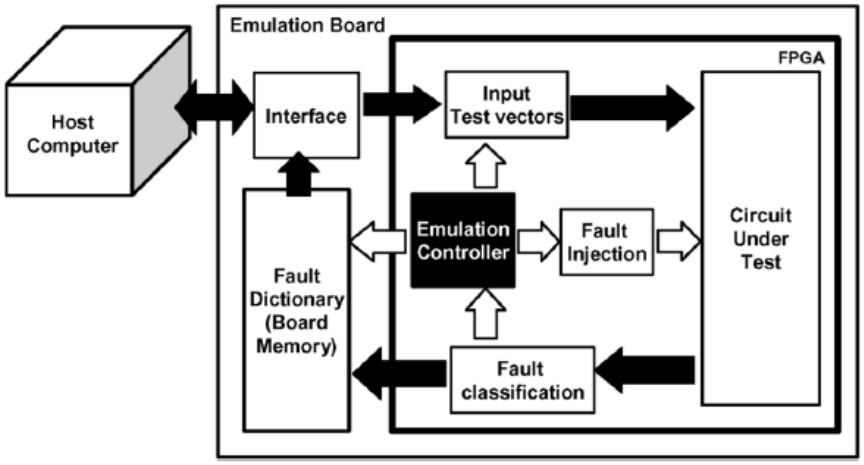

Figura 5.1: Arquitectura del Sistema Autónomo de Inyección de faltas desarrollado por la Universidad Carlos III de Madrid

El esquema de instrumentación de los Flip-Flops es el siguiente:

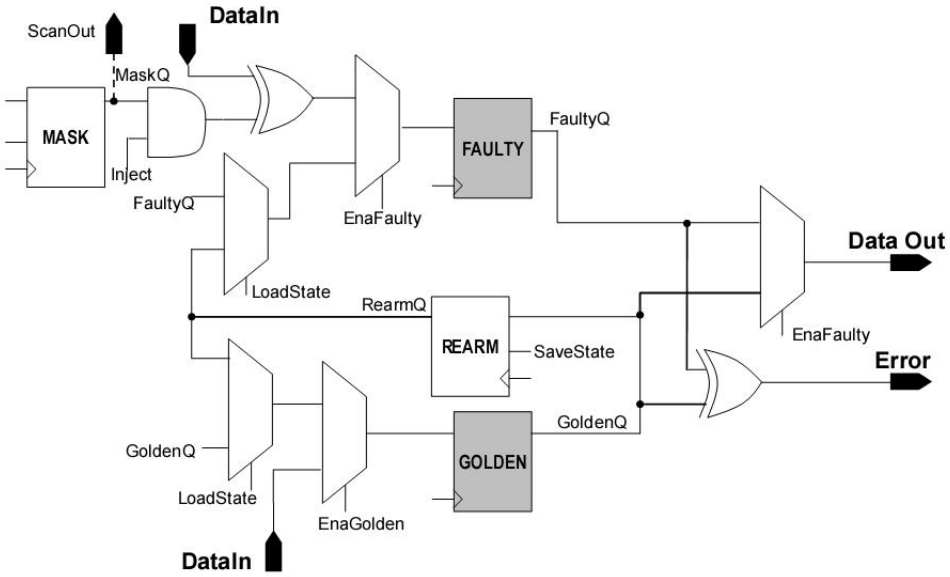

Figura 5.2: Modelo de instrumentación para los Flip-Flops

Mientras que para instrumentar las memorias de bloque es necesario duplicarlas y además añadir circuitería extra:

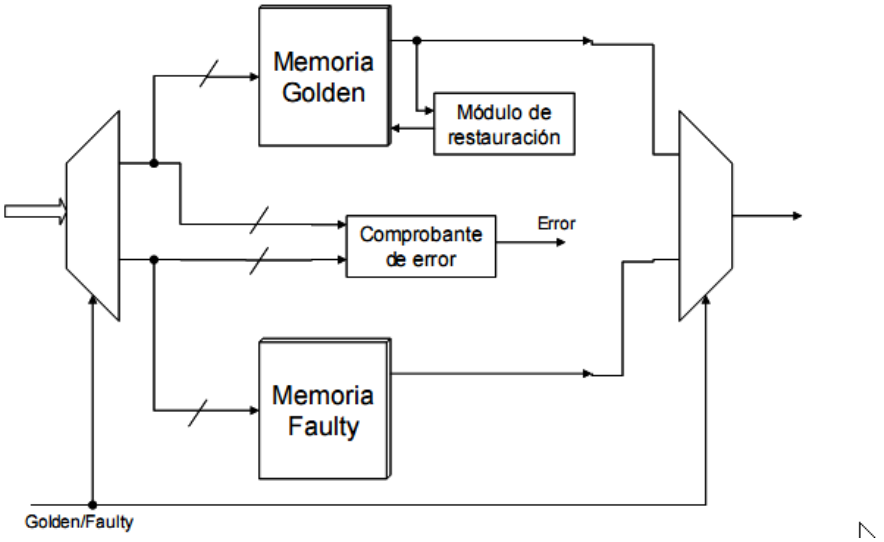

Figura 5.3: Modelo de instrumentación para las memorias empotradas

5.2.2.2. No Instrumentada

Algunos sistemas de inyección de faltas aprovechan circuitería de depuración ya existente en la FPGA donde se implementa el prototipo para resolver el problema de controlabilidad y observabilidad de los bits del circuito. La principal ventaja de estos sistemas es que no modifican la netlist original del MUT, pero a la vez mantienen una alta velocidad de test, como es característico en los sistemas de inyección de faltas emulados sobre FPGA. Los sistemas No Instrumentados, si bien a día de hoy no llegan a la velocidad alcanzada por los sistemas Instrumentados, pueden alcanzar velocidades de hasta 10μs por falta, como es el caso del sistema FT-UNSHADES[5], desarrollado por el Departamento de Ingeniería Electrónica de la Universidad de Sevilla y que sirve de punto de partida para el presente trabajo. Otra ventaja adicional de los sistemas No Instrumentados es que se reduce el tiempo de preparación del diseño para el test, puesto que no es necesario introducir modificaciones en la netlist del mismo. Tampoco necesitamos una descripción HDL del MUT, ya que al no tener que modificarlo nos basta con un fichero EDIF o NGC post-síntesis, lo cual permite realizar pruebas sobre circuitos cuyo código fuente no sea accesible por razones de propiedad intelectual (como es el caso del microprocesador empotrado MicroBlaze, con el que se ha realizado una parte del presente trabajo).

Al ser la herramienta FT-UNSHADES la que se ha utilizado como base para el presente trabajo, se hará una descripción más detallada de la misma en el capítulo 6

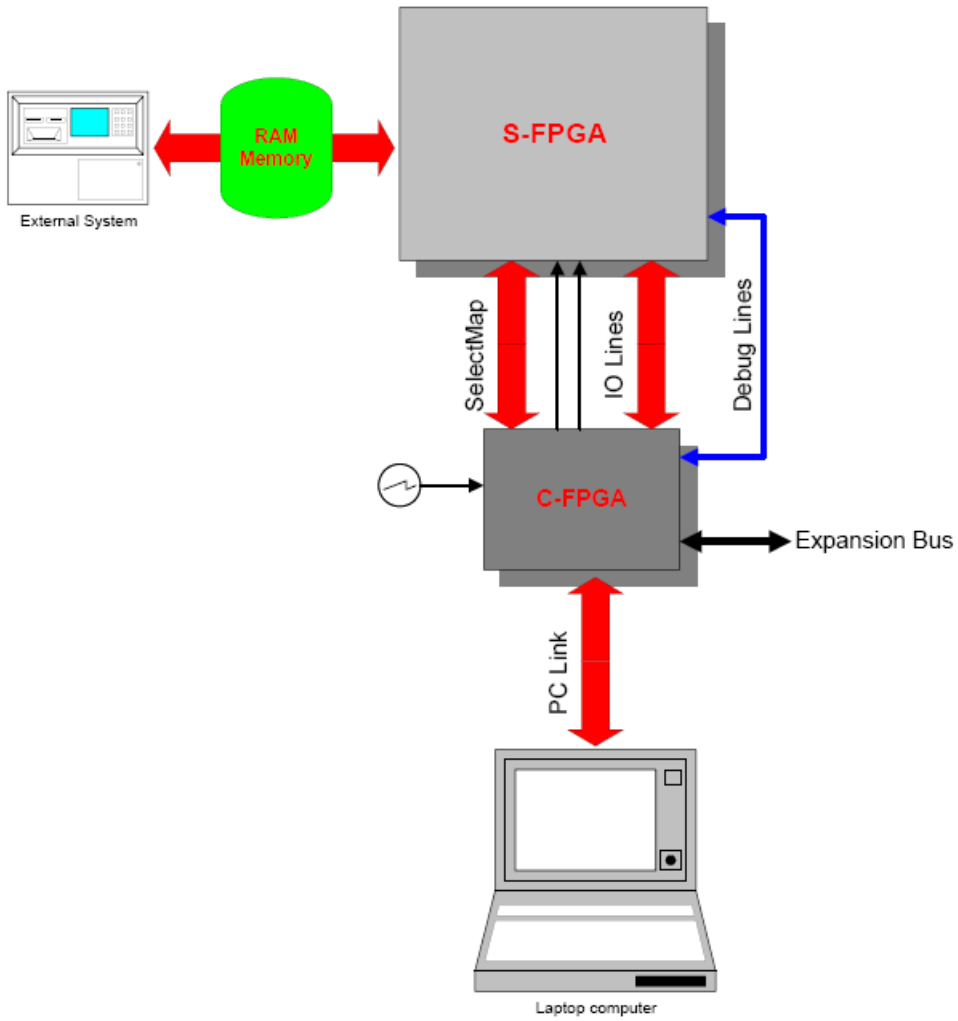

Figura 5.4: Framework del sistema FT-UNSHADES. El sistema aprovecha las capacidades de reconfiguración parcial y 'capture and readback' para realizar la inyección de las faltas y lectura del estado del diseño

Una comparación entre un sistema de emulación instrumentado y uno no-instrumentado se puede encontrar en [25].

5.2.3. Emulada sobre Prototipos

La inyección de faltas emulada sobre prototipos consiste en introducir las faltas sobre dispositivos fabricados en silicio. Debido a la menor flexibilidad, controlabilidad y observabilidad inherente a los prototipos en silicio en comparación con las FPGAs, es más difícil inyectar las faltas, y en general no es posible inyectarlas en todos los bits del circuito. Como ventaja principal tenemos el hecho

de que estamos probando efectivamente el circuito real, el mismo que trabajará en el entorno expuesto a la radiación, junto con la gran velocidad con la que se ejecuta la campaña de inyección de fallos. No obstante, el 'setup' de las pruebas es bastante trabajoso, ya que para cada circuito a probar será necesario desarrollar una plataforma hardware específica que nos permita tanto inyectar los fallos como realizar un posterior examen y clasificación de los efectos producidos. A continuación veremos algunas técnicas que se pueden utilizar para realizar la inyección de faltas en estos prototipos.

5.2.3.1. Pin-Level

Algunas de las plataformas de inyección de fallos que existen se basan en el 'stuck-at' o 'bridging' de los pines del chip de nuestro prototipo. A este enfoque se le denomina 'Pin-Level', ya que sólo podemos inyectar las faltas en los pines del chip. Algunos ejemplos de esta técnica son MESSALINE [8], desarrollado por el LAAS Research Centre de Toulouse (Francia) o RIFLE [20] desarrollado en la Universidad de Coimbra (Portugal).

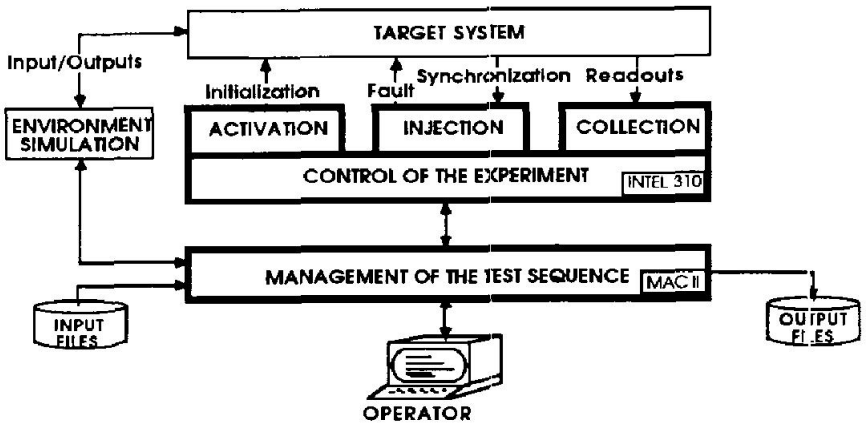

Figura 5.5: Arquitectura general de MESSALINE

5.2.3.2. Code Event Upset (C.E.U.)

La idea básica de los Code Event Upset, es que, si el circuito que estamos probando es un microprocesador, podemos modificar el código fuente que ejecuta el mismo, de forma que podemos instrumentarlo para que inserte fallos. Normalmente esto se consigue utilizando interrupciones software activadas por un timer, o bien interrupciones hardware activadas a través de un pin dedicado. Estas interrupciones, sea cual sea el mecanismo a través del que se activen, leen una posición de memoria e insertan en ella un bit-flip (cambiando uno de los bits leídos y volviendo a escribir la palabra entera).

La principal desventaja de este sistema es que no podemos acceder a todos los bits del diseño, tan sólo a los accesibles por software[5]. La técnica se considera altamente intrusiva ya que se está cambiando el flujo de programa por lo que se distorsiona la realidad del mismo (por ejemplo, su temporización). Por último, no podemos inyectar las faltas en cualquier instante, ya que esto viene limitado por los tiempos de ejecución de cada instrucción. No obstante, la técnica tiene ciertas ventajas: es una técnica con muy bajo coste de implementación que además es muy flexible en cuanto a controlabilidad y observabilidad[6].

Un par de ejemplos de plataformas de inyección de faltas basadas en C.E.U. son FERRARI[23], de la Universidad de Texas, y XCEPTION[12], de la Universidad de Coimbra.

5.2.3.3. Láser Pulsado

La última técnica de inyección de faltas que veremos es la técnica de láser pulsado. Esta técnica está considerada como la más realista, en el sentido de que se acerca más a las condiciones reales de un circuito que esté funcionando bajo radiación. Sus principales ventajas son, junto con el hecho importantísimo de que el circuito probado es exactamente el mismo que queremos enviar al entorno radioactivo, la velocidad de ejecución de las pruebas, y el hecho de que podemos alcanzar cualquier bit del circuito[7]. La principal desventaja es que es una técnica lenta y costosa, ya que para preparar la prueba es necesario realizar un decapado del circuito, para eliminar la parte del encapsulado y de la capa de pasivación del lado del circuito que vayamos a atacar, y además necesitamos utilizar unas instalaciones que dispongan de un láser con las características adecuadas, ya que no todos son capaces de producir SEE. Si no conocemos el layout del circuito esta técnica no se puede utilizar, ya que para insertar el fallo necesitamos apuntar al láser al biestable que queremos atacar: gran parte de la superficie del chip no es sensible al láser.

5.3. Tabla Comparativa

La presente comparativa no pretende quitar valor a ningún sistema de inyección concreto, sino analizar las ventajas y desventajas características de cada uno de forma que podamos elegir apropiadamente cuál utilizar dependiendo de cada situación. Según en qué parte del flujo de diseño nos encontremos nos será más útil utilizar un sistema de inyección de faltas u otro: mientras más nos vayamos

[5] Y aún así, hay que tener cuidado con las memorias caché

[6] Por supuesto, esto sólo aplica a los registros accesibles por software.

[7] Salvo que las capas de metal interfieran. Esto puede llegar a ser un problema.

alejando de la fase de planteamiento inicial y más nos acerquemos al sistema final, más ‘hardware’ tenderá a ser nuestra inyección de faltas.

Técnica		Velocidad	Alcance[a]	Coste	Info necesaria[b]	Precisión[c]	Dist de flujo de programa o timing	Tipos de circuito[d]	Notas
Simulada		bad	ok	great	bad	ok	bad	ok	La precisión del análisis depende de la precisión del modelo de simulación
Emulada FPGA	Instrumentada	great	ok	good	bad	worst	good	ok	Al instrumentar el MUT estamos realmente atacando un circuito distinto
	No Instrumentada	good	great	good	ok	great	good	great	
Emulada sobre Prototipo	Pin-Level	good	worst	good	great	great	great	great	Pequeño alcance, muy baja controlabilidad para insertar la falta
	C.E.U.	good	bad	good	great	great	ok	worst	El circuito a probar debe ser un microprocesador con soporte para interrupciones
	Láser	good	great	bad	bad	great	good	great	La velocidad puede ser bastante alta si se ha utilizado previamente un emulador para localizar las áreas sensibles

Tabla 5.1: Comparativa de las distintas técnicas de inyección de faltas

[a] ¿Podemos atacar a todos los bits del diseño?

[b] ¿Necesitamos información específica sobre el diseño (Por ejemplo su descripción VHDL, una netlist post-síntesis o incluso su layout)?

[c] ¿El circuito sobre el que inyectamos los fallos, es exactamente el mismo que el circuito real? ¿Si no es exactamente igual, difiere mucho o poco de éste?

[d] Tipos de circuito que podemos probar. Por ejemplo, si utilizamos C.E.U. sólo podremos probar circuitos microprocesador.

Capítulo 6

Plataforma FT-Unshades

6.1. Revisión del sistema FT-Unshades

El sistema FT-UNSHADES, descrito en profundidad en [29], es una plataforma basada en FPGA dedicada al estudio de fiabilidad de netlists mediante técnicas de reconfiguración parcial. Esta plataforma está basada en una suite de herramientas software y una plataforma hardware dedicada basada en una FPGA Xilinx de la familia Virtex-II. En la forma original del sistema FT-UNSHADES, las faltas (SEUs) son inyectadas como bit-flips en uno o más registros de usuario del diseño. Paquetes de bits de configuración que contienen el valor actual de un registro de usuario concreto son leídos de la plataforma hardware, tras lo cual son procesados (por ejemplo para escribir el bit-flip) y nuevamente cargados en la plataforma con estos nuevos valores. El software de FT-UNSHADES[1] lee el fichero .ll de asignación lógica (*logic location*) que se obtiene a través del flujo de diseño estándar de Xilinx. De este fichero se puede extraer la posición física de cada registro del diseño dentro de la FPGA y asociar esta información con el nombre lógico del registro que resulta del proceso de síntesis a alto nivel.

El registro objetivo se elige de entre el conjunto completo de registros de usuario que componen el diseño antes de la inyección de la falta, a diferencia de otras herramientas de inyección en las que la inyección se realiza de forma ciega y es necesario realizar un análisis sistemático[2] tras el cual identificar las faltas[3].

[1] Concretamente, el programa TNT: Test aNalysis Tools.

[2] Una campaña de análisis en la que atacamos a todos los bits en todos los posibles instantes de tiempo.

[3] Digamos que las campañas en estos sistemas son 'atómicas': si no terminamos la campaña o no es práctico hacerla porque el circuito sea especialmente grande, no podemos obtener ninguna información sobre la tolerancia a fallos del circuito.

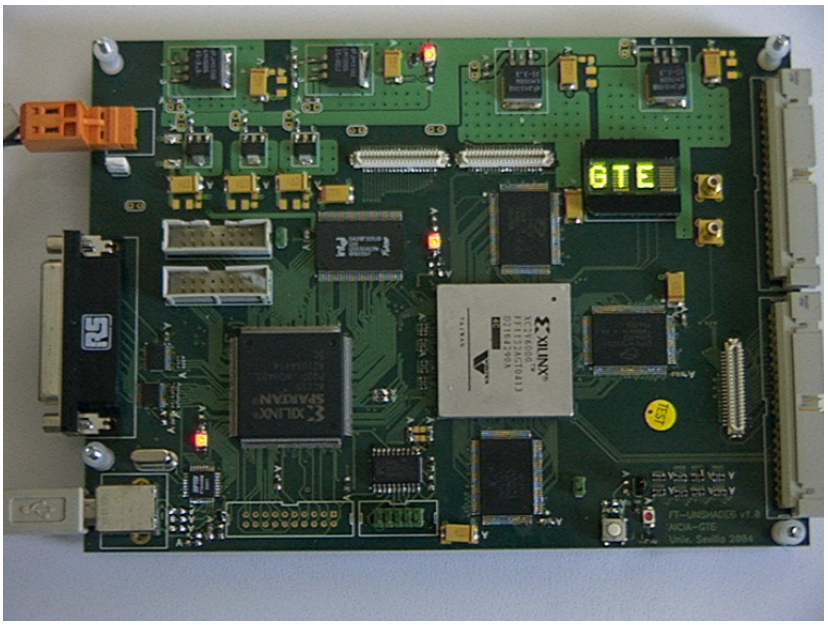

Figura 6.1: FT-UNSHADES. Universidad de Sevilla

6.2. Modelo de inyección de faltas

En FT-UNSHADES el diseño bajo estudio, llamado MUT (Module Under Test) se instancia por partida doble, pero sólamente una de las dos instancia es atacada (Seu MUT): la otra corre en paralelo y se considera como referencia para comparación (Gold MUT). La figura 6.2 muestra el modelo de inyección. En este figura pueden verse las dos instancias junto con el controlador de reloj de sistema, que nos sirve para determinar cuándo vamos a inyectar el bit-flip. La detección de eventos permite saber cuándo la falta se ha propagado a una salida primaria, ya que en ese caso, alguno de los bits de la salida no coincidirán en el Seu MUT y el Gold MUT.

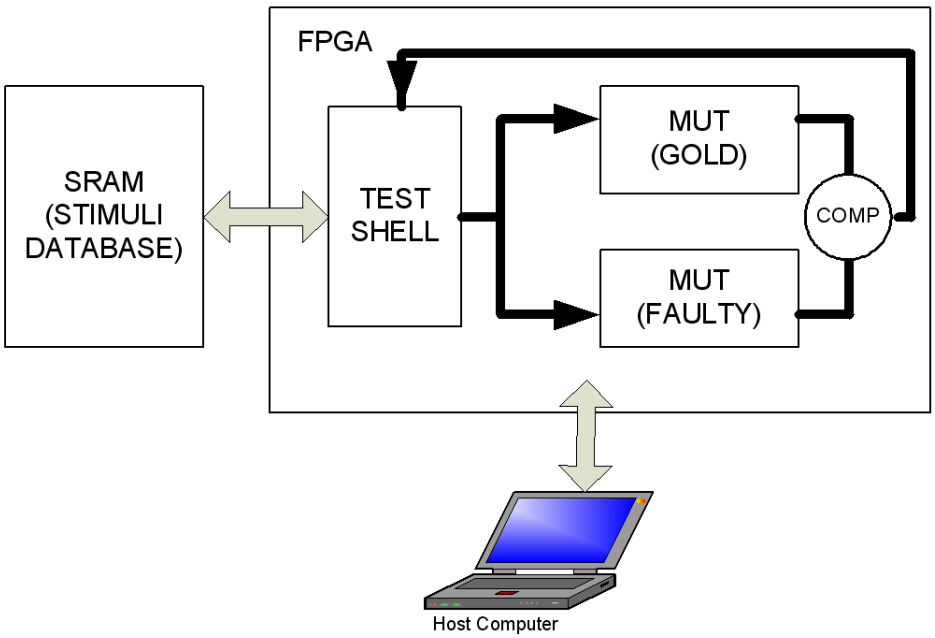

Figura 6.2: Arquitectura de la plataforma FT-UNSHADES

6.2.1. Ciclos de Inyección

Una campaña de test estará compuesta por un número arbitrario de ciclos de inyección. Un ciclo de inyección es una ejecución completa de nuestro circuito en la FPGA, durante la cual hemos insertado al menos un bit-flip. Cada ciclo de inyección se lleva a cabo siguiendo estos pasos:

1. Se escoge el tiempo de inyección. Se escogen uno o más registros para cambiar su valor cuando se llegue al tiempo de inyección.

2. Se programa al sistema para que se pare en el tiempo de inyección.

3. El sistema llega al tiempo de inyección y para el reloj del diseño bajo test. Se inducen los bit-flips en los registros seleccionados. Se continúa la ejecución del sistema hasta que se detecte un evento en las salidas o terminen los vectores de test sin haber detectado daño.

4. Se clasifica la falta.

El sistema presenta la ventaja de que no hay restricciones de tiempo, lugar o número de fallos que se inyectan.

Capítulo 7

Propuesta: FT-Unshades-uP

En el presente capítulo (y trabajo) se propone una extensión y adaptación de la herramienta FT-Unshades para el estudio de tolerancia a fallos microprocesadores empotrados. Se expondrá por qué un circuito microprocesador necesita un tratamiento distinto a otros tipos de circuito, se estudiarán los problemas que pueden presentar dichos circuitos a la hora de estudiar su tolerancia a fallos con las herramientas actuales, y por último se propondrá un modelo nuevo adaptado a los sistemas microprocesador para dar solución a dichos problemas.

7.1. Motivación

Medir la fiabilidad de sistemas microprocesador no es una tarea fácil, debido a que los microprocesadores son circuitos con una arquitectura muy particular. Los SEE pueden ocurrir en diferentes partes del diseño y afectar al sistema de diferentes formas. Aplicar la triple redundancia modular [34] es la primera solución que se plantea[1] y la que resulta más sencilla, ya que puede recuperar faltas dentro del procesador, y existen técnicas de detección de errores para corregir SEEs en las memorias. La segunda solución es la protección de memoria, utilizando técnicas EDAC (Error Detection And Correction) que incrementan la memoria necesaria para el sistema. De todas formas ambas técnicas incrementan los recursos necesarios para implementar el sistema, tanto en una FPGA como en un ASIC. El área de silicio, las prestaciones del circuito y el consumo de potencia son los primeros parámetros afectados. En último lugar están las técnicas de protección software que implementan redundancia en los datos e instrucciones de forma que se pueda recuperar la ejecución normal en caso de que los errores hayan afectado a las variables o al flujo de programa.

[1] Tiene, además, la ventaja de que puede hacerse sin incrementar el tiempo de ejecución

Al estudiar la tolerancia a fallos de circuito microprocesador podemos encontrarnos con cualquiera de estas técnicas, incluso todas a la vez.

7.1.1. Desalineamiento de los microprocesadores

Uno de los problemas que encaramos con esta extensión de FT-UNSHADES es al problema de 'desalineamiento' de los procesadores. Esto es, básicamente, que creamos que se ha propagado un error por el microprocesador ya que su estado es diferente al de un procesador 'GOLD' en el que no se han introducido faltas, pero en realidad nuestro procesador 'FAULTY' no está sino recuperándose activamente del fallo, y tras cierto número de ciclos, alcanzará un estado fault-free de nuevo. Para este momento, el procesador GOLD irá en su ejecución de programa varios ciclos por delante del FAULTY, y al comparar observaremos grandes diferencias entre ambos, cuando en realidad los dos procesadores están realizando una ejecución libre de fallos del mismo programa (el problema es este desfase temporal).

*** SI PUEDO METER AQUI UNA IMAGEN DEL DESALINEAMIENTO ***

Una de las primeras soluciones que se proponen para encarar y mitigar este problema, aunque una de las opciones más sencillas es congelar el procesador GOLD tras realizar la inserción del fallo, y ver si el procesador FAULTY es capaz de alcanzar el estado congelado del GOLD tras cierto número de ciclos (si no es así, supondremos que no ha sido capaz de recuperarse de la falta). El problema de esta técnica es que, para que sea practicable, se necesita a priori conocer al menos una estimación del *tiempo de recuperación* del estado *fault-free*[2], para lo que es necesario conocer en profundidad los procedimientos de corrección de errores y recuperación de flujo de programa. Pero en un circuito que implemente distintas técnicas de corrección puede ser muy complicado predecir qué técnica concreta utilizará para corregir cada fallo que insertemos, ya que ésto dependerá del registro y del ciclo en el que insertemos el mismo.

7.2. Descripción

La aplicación práctica que aquí se propone es la extensión de la herramienta de análisis e inyección de faltas FT-UNSHADES[37] para su uso en el estudio de tolerancia a fallos en sistemas microprocesador.

Al principio de la presente investigación, se plantearon varias tareas a realizar para hacer posible esta extensión de FT-UNSHADES: entre otras, ha sido necesario añadir a FT-UNSHADES soporte para poder acceder en lectura y escritura

[2]Estado libre de fallos.

a las BRAMs (Block RAMs) de la FPGA, y resolver el problema de backannotation de los bits de las LUTRAMs y los registros de desplazamiento SRL16[3]. El siguiente paso ha sido diseñar un *modelo de inyección* apropiado para sistemas microprocesador, ya que el modelo Seu + Gold no es demasiado práctico, dando lugar al *Modelo de Tabla Inteligente* descrito en la sección 7.6.2.

7.3. Nuevas técnicas de inserción de SEUs en FPGAs

Previo a este trabajo, se han realizado estudios de tolerancia a fallos sobre el microprocesador Leon2[7][4], pero accediendo únicamente a los flip-flops, ya que no se podían insertar SEUs en todos los bits del diseño.

El desarrollo de nuevas técnicas para la inserción de SEUs en FPGAs Virtex-II ha sido necesario para la presente extensión de FT-UNSHADES. Debido a que un procesador softcore tendrá, normalmente, algo de memoria RAM y/o caché instanciada en memorias de bloque (BRAMs) dedicadas, y también puede utilizar LUTs y SRL16, la primera necesidad que surge es la posibilidad de insertar SEUs en otros bits de la FPGA que no sean los Flip-Flops estándar. En el caso del procesador MicroBlaze, el Register File (array de registros internos del microprocesador) se implementa utilizando memoria distribuida basada en LUTs (LUTRAM), y el contador de programa se implementa utilizando primitivas SRL16. Además, el programa que correrá el micro está almacenado en memorias de bloque[5], por lo que atacar a estos tipos de elementos es vital para poder realizar un análisis amplio y profundo del microprocesador. Las herramientas de la plataforma FT-UNSHADES han sido extendidas para permitir inserción de faltas en todos estos bits de la FPGA (BlockRAMs, LUTs y SRL16)

[3]El acceso a los bits de las LUTRAMs y SRL16 no difiere en esencia del acceso a los Flip-Flops, pero el fichero .ll (logic location) generado por el software de Xilinx no da suficiente información para relacionar el bit modificado en la FPGA con los bits del diseño.

[4]Utilizando el modelo de implementación Seu + Gold.

[5]En el caso de los dos procesadores estudiados en el presente trabajo, se han hecho diseños totalmente System on Chip, en los que se corre un pequeño programa empotrado desde las memorias de bloque internas a la FPGA. En casos en los que el programa a ejecutar se encuentre en una memoria externa, como sería el caso si fuéramos a ejecutar, por ejemplo, una versión de Linux para sistemas empotrados, en las BlockRAMs internas de la FPGA los normal es almacenar algún programa boot-loader como puede ser FS-Boot. La inserción de SEUs en memorias externas, si bien se está considerando para posteriores ampliaciones de la herramienta, no es trivial y escapa al objetivo del presente trabajo.

7.4. Análisis jerárquico y 'backannotation'

Para poder realizar un análisis jerárquico del diseño, no sólo necesitamos la capacidad de insertar físicamente los bit-flips, sino que además necesitamos saber a qué parte del diseño estamos atacando, esto es, a qué submódulo del diseño pertenece el bit atacado. A esto se le llama *backannotation*. Para realizar el backannotation de los bits del diseño que pertenecen a primitivas del tipo LUT, BRAM y SRL16, necesitamos información adicional, ya que no hay suficiente información en el fichero de *logic allocation* (o asignación lógica, con extensión .ll) que produce el flujo de Xilinx ISE, por lo que las herramientas de FT-UNSHADES han de generar un fichero de *ram allocation* o asignación de ram (con extensión .rl) para proporcionar la información que falta.

7.4.1. Utilidad generateBMM

La utilidad generateBMM lee el map report (el log del resultado de la etapa de technology mapping), y a partir de la información contenida en este fichero genera un fichero llamado ftunshades_top.bmm. Hay que añadir este .bmm al diseño en ISE y volver a realizar la implementación. El flujo de diseño ISE genera un nuevo fichero bmm (ftunshades_top_bd.bmm), añadiendo la información necesaria para poder acceder desde TNT a las memorias internas de bloque (BlockRAMs). Actualmente la utilidad generateBMM está obsoleta debido a que la utilidad generateRL, creada posteriormente, nos permite acceder, además de a las memorias de bloque, a las LUTs y a los registros de desplazamiento SRL16.

7.4.2. Utilidad generateRL

Es una ampliación de generateBMM. Deja obsoloeta a generateBMM. Tiene la ventaja de que no necesitamos realizar la implementación del DTE dos veces, ya que no se genera ningún fichero que deba añadirse al proyecto. La herramienta utiliza únicamente el .ncd del DTE del diseño. Esta nueva herramienta tiene dos ventajas principales: la primera, que nos permite acceder, además de a las memorias de bloque, a las LUTs y a los registros de desplazamiento SRL16; y la segunda, es que no es necesario añadir ningún fichero al proyecto ISE por lo que los pasos de síntesis y de implementación no necesitan repetirse, se realizan una sóla vez. Esta herramienta genera un fichero un xdl utilizando la herramienta de Xilinx de mismo nombre (xdl), y analiza los contenidos de este fichero para producir el *ram allocation file* o fichero .ll que leerá TNT.

7.5. Restauración de memoria no volátil dañada

Otro problema que ha habido que resolver es la necesidad de restaurar los elementos de memoria no volátil, por ejemplo, en el caso de que un SEU alcance la ROM de programa, que en nuestro experimento también se almacena en una BlockRAM. Durante una campaña de inyección de faltas, debido a que la entrada de reset del módulo bajo test no restaura la ROM de programa a sus contenidos originales, se puede acumular daño de un ciclo de inyección a los siguientes. La solución propuesta aquí es que, cuando se detecta daño, todos los registros y RAM se reinicializan y el ciclo de inyección que dió fallo se repite. Si en la repetición del ciclo de inyección no se detecta daño, sabremos que el error a la salida en el primer ciclo de inyección fue causado por daño acumulado en los elementos de memoria volátiles.

7.6. Modelos de Implementación

La última modificación introducida en la plataforma FT-UNSHADES-uP es el modelo de implementación. La figura 7.1 presenta la implementación utilizando la comparación estándar (Seu y Gold), apropiada para enfoques ASIC. Se muestra que se pueden comparar las salidas de los MUT, ciclo por ciclo. Como se ha comentado anteriormente, al estudiar microprocesadores, ya que pueden ejecutar técnicas de redundancia y protección software que producirían un desalineamiento de los microprocesadores Gold y Seu, con lo que el sistema de comparación detectaría una discrepancia y pararía el sistema, clasificando como daño un estado transitorio que podría corregirse, gracias a estas técnicas de redundancia software, si se le diera más tiempo de proceso. Este tiempo será, al menos, de varios ciclos de reloj y se le llama tiempo de recuperación o *recovery time*.

7.6.1. Módulos SEU y GOLD

La implementación estándar.

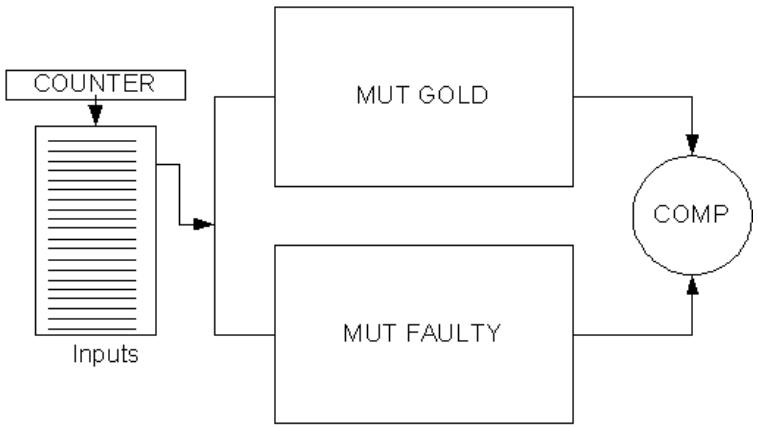

Figura 7.1: Arquitectura de Implementación con modelo SEU y GOLD

7.6.2. Modelo de Tabla Inteligente

La Smart Table o Tabla Inteligente aprende la secuencia correcta de salidas durante una ejecución sin fallos del programa (fault-free execution o golden run), y durante los siguientes ciclos de inyección comprueba que las salidas cambian en el orden correcto y que no toman valores extraños.

En esta arquitectura, la tabla inteligente sustituye al GOLD MUT, liberando de esta forma recursos ya qe no necesitamos implementar dos microprocesadores completos en la misma FPGA.

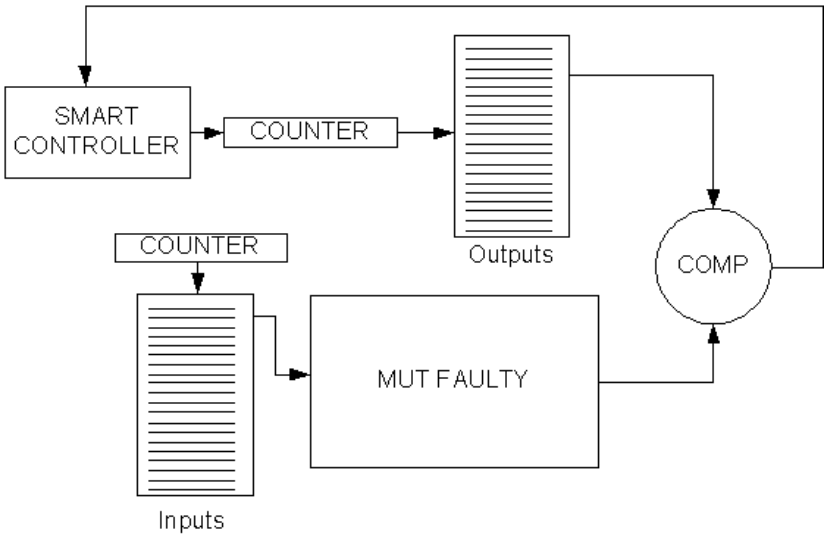

Figura 7.2: Arquitectura de Implementación con modelo de Tabla Inteligente

7.6.2.1. Tabla sin aprendizaje

La primera implementación del modelo de tabla inteligente es diseño que tiene memorizado, previamente a la síntesis, la secuencia esperada de salidas correctas del circuito. No es práctica para aplicaciones cuya complejidad no sea trivial, pero sirve como primera aproximación para comprobar la factibilidad y viabilidad del enfoque de tabla inteligente para sistemas microprocesador.

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

entity expertsystem is
    Port ( clk : in  STD_LOGIC;
           rst : in  STD_LOGIC;
           MUTinputs : in  STD_LOGIC_VECTOR (31 downto 0);
           SEUoutputs : in  STD_LOGIC_VECTOR (0 to 31);
           testcompleted : out STD_LOGIC;
           errordetected : out  STD_LOGIC
  );
end expertsystem;

architecture Behavioral of expertsystem is

type outputtable is array (31 downto 0) of std_logic_vector(0 to 31);
type mem_type is array (4 downto 0) of std_logic_vector(0 to 31);

signal mem : mem_type;
signal mode, n_mode: std_logic; --mode_t;
signal raddr, n_raddr, waddr, n_waddr : std_logic_vector(4 downto 0);

signal expectedMUToutputs: outputtable;
signal previousSEUoutputs: STD_LOGIC_VECTOR (0 to 31);
signal estado, p_estado, estado_final: integer range 0 to 31;
signal error, p_error: std_logic;

begin

expectedMUToutputs(0)<=x"00000001";
expectedMUToutputs(1)<=x"00000000";
expectedMUToutputs(2)<=x"E38E38E4";
expectedMUToutputs(3)<=x"87654321";
estado_final <= 4;


fsm: process(error, previousSEUoutputs, SEUoutputs,
             expectedMUToutputs, testcompletedi)
begin
  p_testcompleted <= testcompletedi;
  errordetected<=error;
  p_error <= error;
  if (SEUoutputs/=previousSEUoutputs) then
    if(SEUoutputs=expectedMUToutput) then
      p_estado<=estado+1;
    else
      p_error<='1';
      errordetected<='1';
    end if;
  end if;
  if (estado=estado_final) then
```

```
    testcompleted<='1';
    p_estado<=estado_final;
  end if;
end process;

sinc: process(clk, rst)
begin
  if (rst='0') then --reset activo a nivel bajo, pero dependera de cada MUT!
    error<='0';
    estado<=0;
    previousSEUoutputs<=(others=>'0');
  elsif (clk='1' and clk'event) then
    error<=p_error;
    estado<=p_estado;
    previousSEUoutputs<=SEUoutputs;
  end if;
end process;

end Behavioral;
```

7.6.2.2. Tabla con aprendizaje

La segunda versión de la tabla inteligente es un autómata que puede funcionar en dos modos: aprender (*learn*) y comparar (*compare*). En modo aprendizaje, la tabla captura la secuencia de salidas del módulo bajo test, mientras que en modo comparación, se comparan las salidas del MUT y se da un error si la secuencia no coincide, es decir, si en algún momento la salida cambia a un valor que no sea el esperado según la secuencia aprendida.

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

entity expertsystem is
  Generic ( WIDTH : integer := 10;
            DEPTH : integer := 10);
  Port ( clk : in  STD_LOGIC;
         rst : in  STD_LOGIC;
         SEUoutputs : in  STD_LOGIC_VECTOR (0 to WIDTH-1);
         testcompleted : out STD_LOGIC;
         errordetected : out  STD_LOGIC
        );
end expertsystem;

architecture Behavioral of expertsystem is

type mem_type is array (DEPTH-1 downto 0) of std_logic_vector(0 to WIDTH-1);
--type mode_t is (learn, compare);
CONSTANT learn: std_logic:='0';
CONSTANT compare: std_logic:='1';

signal mem : mem_type;
signal mode, n_mode: std_logic; --mode_t;
signal raddr, n_raddr, waddr, n_waddr : INTEGER range 0 to DEPTH-1;

--signal expectedMUToutputs: outputtable;
signal expectedMUToutput: STD_LOGIC_VECTOR (0 to WIDTH-1);
signal previousSEUoutputs: STD_LOGIC_VECTOR (0 to WIDTH-1);
```

```
--signal estado, p_estado, estado_final: integer range 0 to 31;
signal error, p_error, we: std_logic;
signal p_testcompleted, testcompletedi: std_logic;

begin

testcompleted<=testcompletedi;

fsm: process(error, previousSEUoutputs, SEUoutputs, expectedMUToutput,
             waddr, raddr, mode, testcompletedi)
begin
p_testcompleted <= testcompletedi;
errordetected<=error;
p_error <= error;
we<='0';
n_raddr<=raddr;
n_waddr<=waddr;
n_mode<=mode;
  if(mode=compare AND waddr>=1) then
    n_mode<=compare;
    if(raddr=0) then
    if(SEUoutputs=expectedMUToutput) then
      n_raddr<=raddr+1; --No detectamos errores hasta ver el vector 0
    end if;
    else
     if (SEUoutputs/=previousSEUoutputs) then
       if(SEUoutputs=expectedMUToutput) then
         n_raddr<=raddr+1;
       else
         p_error<='1';
         errordetected<='1';
       end if;
     end if;
    end if;
    if (raddr=waddr) then   --estado final = waddr
      p_testcompleted<='1';
      --n_raddr<=waddr; --p_estado<=estado_final;
    end if;
  else  --modo learn
    n_mode <= learn;
    p_testcompleted<='1';
    if(raddr=0) then
      n_raddr<=raddr+1;
      n_waddr<=0;
    elsif(SEUoutputs/=previousSEUoutputs) then
      if(waddr<DEPTH) then
        we<='1';
        n_waddr<=waddr+1;
      end if;
    end if;
  end if;
end process;

sinc: process(clk, rst)
begin
  if (rst='0') then --reset activo a nivel bajo, pero esto dependera de cada MUT!
    raddr<=0;
    error<='0';
    previousSEUoutputs <= (others=>'0');
    mode<=compare; --si quitas init value se optimiza mode y falla al implementar
    testcompletedi<='0';
  elsif (clk='1' and clk'event) then
```

```
    raddr <= n_raddr;
    error<=p_error;
    previousSEUoutputs <= SEUoutputs;
    mode<=n_mode;
    testcompletedi<=p_testcompleted;
  end if;
end process;

process(clk)
begin
if (rising_edge(clk)) then
  waddr <= n_waddr;
  if (we = '1') then
    mem(conv_integer(waddr)) <= SEUoutputs;
  end if;
end if;
end process;
process(raddr, waddr, mem)
begin
expectedMUToutput <= mem(conv_integer(raddr));
end process;

end Behavioral;
```

La variable *estado* de la tabla cambia al estado *learn* si waddr=0. Cuando estado = *learn*, no puede seguir siendo *learn* para siempre: necesitamos una condición para ir de *learn* a *compare* y así 'engañar' al sintetizador, de tal forma que no optimice la variable estado.

Para que la tabla inteligente aprenda la secuencia correcta de salidas de nuestro diseño, hay que realizar un run completo sin errores[6] con la tabla en estado en modo *learn*. Esto se consigue, en FT-UNSHADES, dando la siguiente secuencia de comandos en el programa TNT[4]:

```
FT-U>define w=[GOLD*waddr*]
FT-U>pulserst
FT-U>writeb w=h0
FT-U>pulsersm
```

Figura 7.3: Secuencia de comandos para entrenar a la tabla inteligente

Podemos comprobar que la tabla ha aprendido comprobando que la dirección de escritura en la misma es distina de cero, que es su valor inicial. Para ello hacemos readback del registro w y leemos su valor actual:

```
FT-U>readb w
```

Figura 7.4: Readback de un registro con TNT

[6]Lo que se conoce como un *golden run*.

La mayor ventaja de que la tabla realice por sí misma el aprendizaje de la secuencia correcta de salidas es que no hay que introducir esta secuencia manualmente, lo cual sería muy tedioso en el caso de vectores de tests de larga duración y limitaría de una forma práctica la clase de experimentos y pruebas a las que podríamos someter a nuestros circuitos microprocesador.

7.6.2.3. Tabla con aprendizaje de salidas y ciclos

Se hicieron dos versiones, una con el tiempo de recuperación (*recovery time*) fijo, a través de un GENERIC VHDL, y otra con un tiempo de recuperación programable, diseñando la tabla de forma que el registro en el que se almacena el tiempo de recuperación no sea optimizado por el sintetizador: de esta forma, se puede cambiar su valor aprovechando las capacidades de reconfiguración parcial de FT-UNSHADES:

```
FT-U>define rt=[GOLD*recovery_time*]
FT-U>writeb rt=h7
```

Figura 7.5: Reconfiguración del tiempo de recuperación

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

entity expertsystem_t is
  Generic ( WIDTH : integer := 10;
            DEPTH : integer := 10;
            RECOVERY_TIME : integer := 15);
  Port ( clk : in  STD_LOGIC;
         rst : in  STD_LOGIC;
         SEUoutputs : in  STD_LOGIC_VECTOR (0 to WIDTH-1);
         testcompleted : out STD_LOGIC;
         errordetected : out  STD_LOGIC;
         timeerrordetected : out STD_LOGIC
       );
end expertsystem_t;

architecture Behavioral of expertsystem_t is

type mem_type is array (DEPTH-1 downto 0) of std_logic_vector(0 to WIDTH-1);
type time_type is array (DEPTH-1 downto 0) of std_logic_vector(31 downto 0);
--type mode_t is (learn, compare);
CONSTANT learn: std_logic:='0';
CONSTANT compare: std_logic:='1';

signal recovery_time_s, recovery_time_ff: std_logic_vector(15 downto 0);
signal mem : mem_type;
signal mem_t : time_type;
signal cycle, p_cycle: std_logic_vector(31 downto 0);
signal mode, n_mode: std_logic; --mode_t;
signal raddr, n_raddr, waddr, n_waddr : INTEGER range 0 to DEPTH-1;
```

```
--signal expectedMUToutputs: outputtable;
signal expectedMUToutput: STD_LOGIC_VECTOR (0 to WIDTH-1);
signal previousSEUoutputs: STD_LOGIC_VECTOR (0 to WIDTH-1);
signal expectedoutputcycle: std_logic_vector (31 downto 0);
--signal estado, p_estado, estado_final: integer range 0 to 31;
signal error, p_error, we: std_logic;
signal p_testcompleted, testcompletedi: std_logic;

begin

testcompleted<=testcompletedi;
--The condition is not supposed to happen, it is here so ISE doesn't optimize
--away the regs and we can reconfigure them with TNT:
recovery_time_s<=
             recovery_time_ff + 1 when (cycle=X"FFFFFFFF" AND testcompletedi='1')
             else recovery_time_ff;

fsm: process(error, previousSEUoutputs, SEUoutputs, expectedMUToutput,
             expectedoutputcycle, cycle, recovery_time_ff, waddr, raddr,
             mode, testcompletedi)
begin
p_testcompleted <= testcompletedi;
errordetected<=error;
p_error <= error;
we<='0';
n_raddr<=raddr;
n_waddr<=waddr;
n_mode<=mode;
timeerrordetected<='0';
  if(mode=compare AND waddr>=1) then
    n_mode<=compare;
    if(raddr=0) then
      if(SEUoutputs=expectedMUToutput) then
        n_raddr<=raddr+1; --No detectamos errores hasta ver el vector 0
      end if;
    else
      if (SEUoutputs/=previousSEUoutputs) then
        if(SEUoutputs=expectedMUToutput) then
          n_raddr<=raddr+1;
        else
          p_error<='1';
          errordetected<='1';
        end if;
      end if;
      if(cycle = expectedoutputcycle + recovery_time_ff + 1)then
        timeerrordetected<='1';
      end if;
    end if;
    if (raddr=waddr) then   --estado final = waddr
      p_testcompleted<='1';
    end if;
  else  --modo learn
    n_mode <= learn;
    p_testcompleted<='1';
    if(raddr=0) then
      n_raddr<=raddr+1;
      n_waddr<=0;
    elsif(SEUoutputs/=previousSEUoutputs) then
      if(waddr<DEPTH) then
        we<='1';
        n_waddr<=waddr+1;
```

```
      end if;
    end if;
  end if;
end process;

sinc: process(clk, rst)
begin
  if (rst='0') then --reset activo a nivel bajo, pero esto dependera de cada MUT!
    cycle<=(others=>'0');
    raddr<=0;
    error<='0';
    previousSEUoutputs <= (others=>'0');
    mode<=compare; --si quitas init value se optimiza mode y falla al implementar
    testcompletedi<='0';
  elsif (clk='1' and clk'event) then
    cycle <= cycle + 1;
    raddr <= n_raddr;
    error<=p_error;
    previousSEUoutputs <= SEUoutputs;
    mode<=n_mode;
    testcompletedi<=p_testcompleted;
    recovery_time_ff<=recovery_time_s;
  end if;
end process;


process(clk)
begin
if (rising_edge(clk)) then
  waddr <= n_waddr;
  if (we = '1') then
    mem(conv_integer(waddr)) <= SEUoutputs;
    mem_t(conv_integer(waddr)) <= cycle;
  end if;
end if;
end process;

process(raddr, waddr, mem)
begin
expectedMUToutput <= mem(conv_integer(raddr));
expectedoutputcycle <= mem_t(conv_integer(raddr));
end process;

end Behavioral;
```

Capítulo 8

Procesadores trabajados

Parte de la necesidad de realizar un trabajo como éste es el hecho de que, en un futuro próximo, se notarán los efectos de la radiación en la electrónica de consumo, a nivel del mar. Esto será posible debido a la reducción de los tamaños de los dispositivos, por lo que al escoger microprocesadores para estudiar, debemos estudiar el estado del arte de los microprocesadores actuales. Es por esta razón por la que se han escogido los microprocesadores Leon3 y MicroBlaze para el presente trabajo.

8.1. Leon3 con modelo SEU y GOLD

Para este trabajo, y como extensión del trabajo anteriormente citado, la primera implementación trabajada ha sido del microprocesador Leon3, de Gaisler Research [3, 22]. Leon3 es un microprocesador RISC de 32 bits sintetizable basado en la arquitectura SPARC V8. El core es altamente configurable y se pueden añadir periféricos de forma modular, lo que lo hace muy útil para aplicacones System On Chip. Los cambios más importantes de Leon3 con respecto a la versión anterior (Leon2), son el pipeline de 7 etapas (anteriormente eran 5) y el soporte multiprocesador. Un trabajo anterior a ésta implementación se puede encontrar en [7], con una implementación basada en el procesador Leon2.

8.1.1. Preparación del MUT o módulo bajo test

Lo primero en esta implementación es preparar un MUT, el cual instanciaremos dos veces en nuestro entorno de test. Para realizar la implementación hay que descargar las fuentes HDL del microprocesador, configurarlo adecuadamen-

te[1], eliminar las señales triestado[2] y crear un proyecto ISE, en el que deseleccionaremos la opción *Synthesis Options ->Xilinx Specific Options ->Add I/O Buffers*.

8.1.2. Preparación del DTE o entorno de test del diseño

Para preparar el DTE (Design Test Environment) utilizamos la versión para Linux de la herramienta *generateDTE*, que es parte del conjunto de herramientas software de FT-UNSHADES. En el proyecto ISE que genera esta herramienta hemos de marcar las siguientes opciones de Bitgen: *Create Readback Data Files, Allow SelectMAP Pins to Persist, Create Logic Allocation File*.

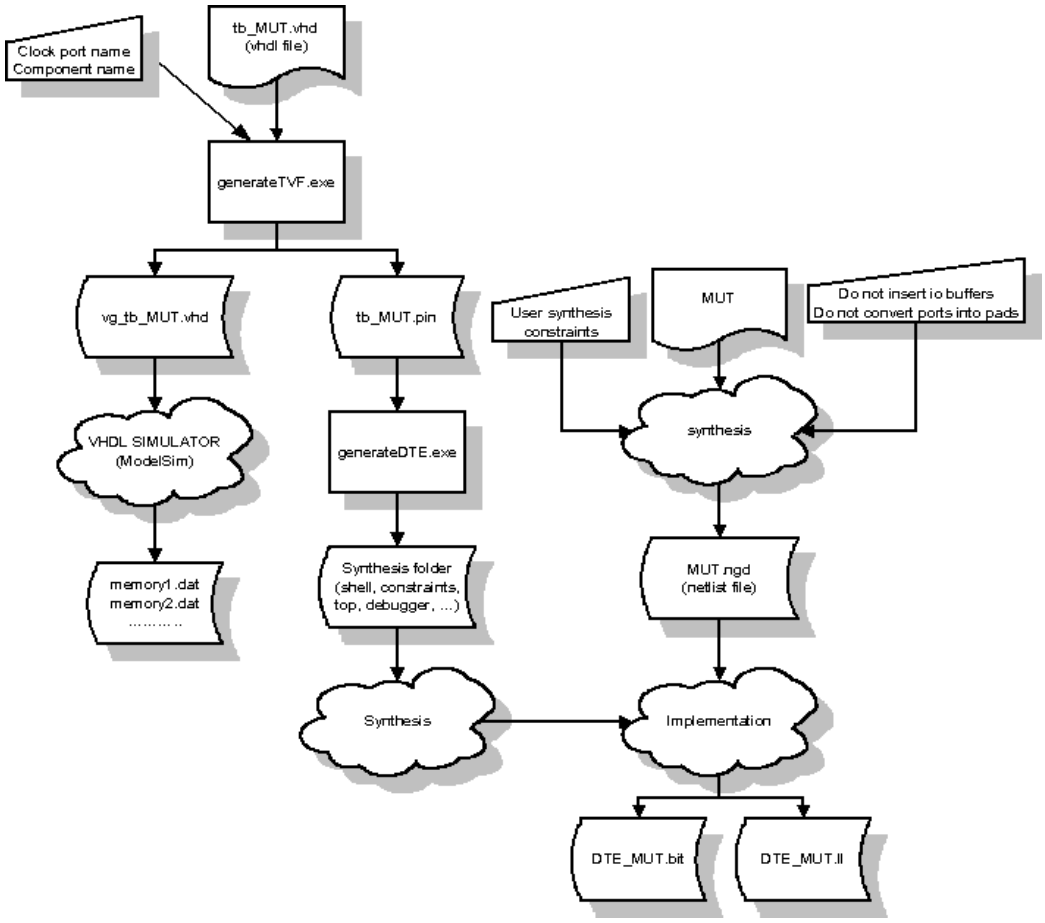

Figura 8.1: Preparación del entorno de diseño en FT-UNSHADES

[1]Parte de la configuración del microprocesador se ha realizado a mano, modificando directamente el código VHDL, sin utilizar las herramientas gráficas proporcionadas por Gaisler Research

[2]Se descomponen en señales de entrada, salida y control. Si bien FT-UNSHADES puede trabajar con circuitos que tengan puertos triestado, el número de puertos es limitado

8.1.3. Implementación y resultados

Se ha conseguido implementar el microprocesador Leon3 utilizando el modelo de implementación SEU + GOLD. Se han comparado los resultados de una simulación realizada con GHDL[1] con los resultados del microprocesador ejecutando los vectores de test en FT-Unshades, comprobando el correcto funcionamiento del microprocesador implementado en el sistema de emulación. Si bien este modelo no es tan útil para el estudio de tolerancia a fallos como el modelo de Tabla Inteligente, resulta un buen punto de partida sobre el que probar variantes de implementación.

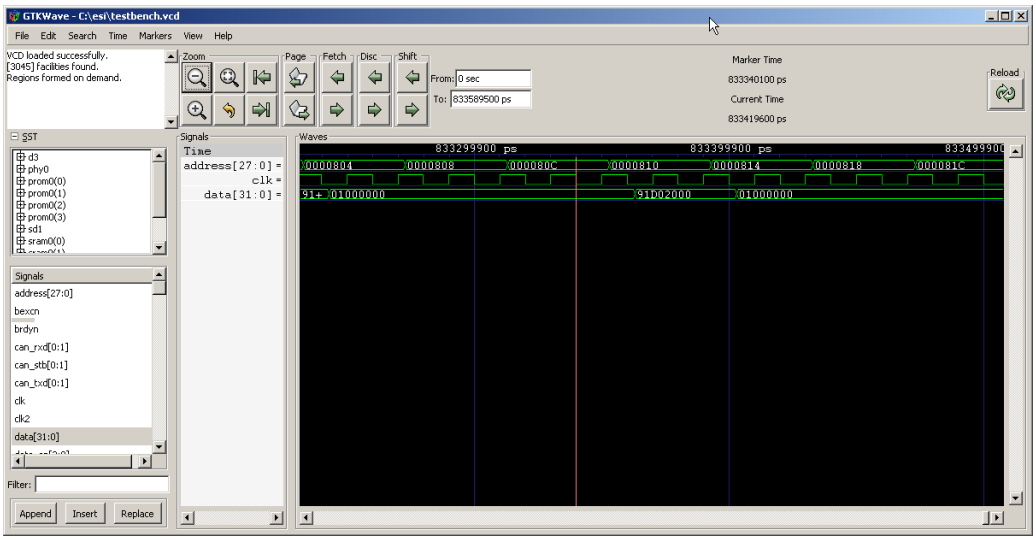

Figura 8.2: Resultados de simulación con GHDL. Captura del visor de ondas GTKWAVE

8.2. Leon3 con modelo de tabla inteligente

8.2.1. Preparación del MUT o módulo bajo test

La segunda implementación realizada ha sido el Leon3 con modelo SEU + Tabla Inteligente. Este diseño se ha hecho desde cero, configurándolo con 'make xconfig' e implementándolo

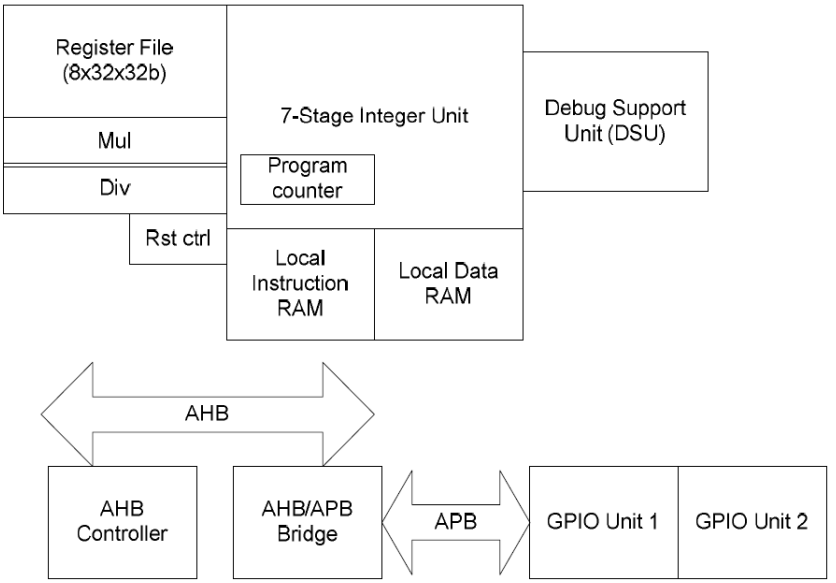

Figura 8.3: Diagrama de bloques de la configuración de Leon3 Implementada. El GPIO 1 actúa como puerto de entrada de los datos y el 2 como puerto de salida de los resultados

8.2.2. Implementación y Resultados

Como veremos en la próxima sección, para MicroBlaze es muy sencillo compilar el software y escribirlo en las memorias internas de la FPGA, ya que las herramientas están integradas en EDK y el usuario no ha de preocuparse de los detalles. Sin embargo, parece que Leon3 no está pensado para funcionar como System On Chip, con RAM y ROM de programa embebidas dentro de la FPGA. La documentación al respecto es escasa y confusa, y la única posibilidad de resolver dudas es la lista de correo *[leon-sparc]* y el contacto directo con otros desarrolladores.

Finalmente se ha logrado implementar el microprocesador y programar las memorias de bloque con el programa empotrado. Los resultados de la campaña de inyección de faltas se pueden encontrar en el capítulo 10.

8.3. MicroBlaze

El procesador MicroBlaze[2] es un microprocesador de 32 bits con juego de instrucciones reducido (RISC) desarrollado y mantenido por Xilinx, que posee una arquitectura flexible que puede ser expandida con distintos periféricos. Este microprocesador está optimizado para su implementación en FPGAs de Xilinx, lo

que lo hace un candidato perfecto para nuestro estudio. La figura 8.4 muestra la arquitectura del microprocesador:

8.3.1. Preparación del procesador y sus periféricos on-chip

Como se explica en la documentación de MicroBlaze, una buena parte de las funcionalidades de la arquitectura son configurables. Xilinx proporciona una herramienta software para definir la arquitectura deseada del microprocesador, el Embedded Development Kit (EDK). En la implementación realizada, de las funcionalidades opcionales sólo se ha implementado la unidad multiplicadora, por lo que el estudio realizado comprende únicamente una versión básica de MicroBlaze más un multiplicador.

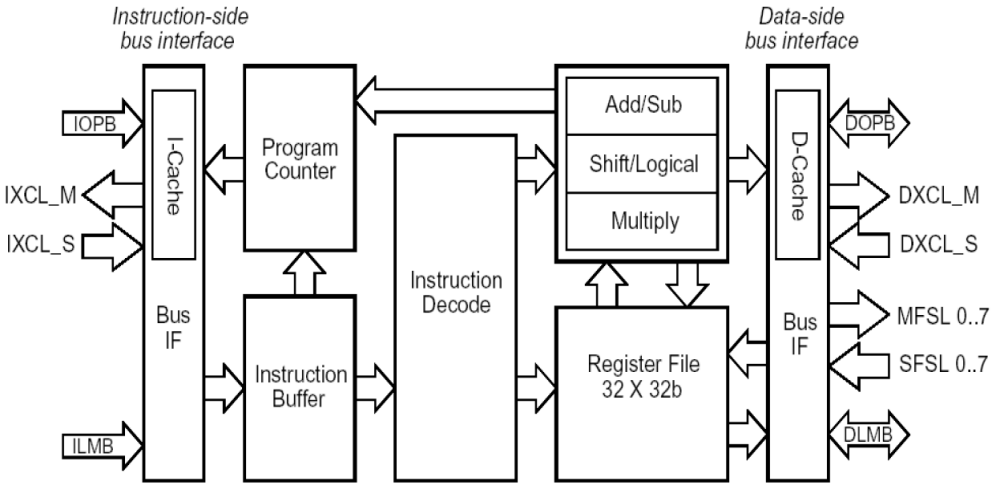

Figura 8.4: Diagrama de bloques del procesador MicroBlaze

8.3.2. Implementación y Resultados

La implementación del MicroBlaze ha sido mucho más sencilla de realizar y programar que la de Leon3. Microblaze tiene ciertas particularidades interesantes, como el hecho de que el contador de programa es redundante 16 veces: cada bit está implementado en una primitiva SRL16. Es posible que esto sea para enventanado y recuperación rápida del valor del contador de programa cuando se vuelve de una o varias subrutinas, tardando sólo un ciclo en recuperar dicho valor en lugar de tener que realizar una lectura de la pila, que está en memoria. Un dato interesante es que, en nuestro programa empotrado, al no utilizar subrutinas pero tener el contador de programa redundante, tenemos una incidencia de fallos especialmente baja al insertar faltas en el contador de programa (ver sección 10).

8.4. Conclusiones

Entre las conclusiones que se pueden obtener de las implementaciones realizadas, es que si queremos utilizar un enfoque SEU + GOLD es más sencillo utilizar Leon3, ya que EDK no te deja implementar directamente dos microprocesadores HDL en un mismo proyecto, pero es más complicado sintetizar el MUT, ya que hemos de eliminar los puertos bidirecciones y tener mucho cuidado con la configuración del microprocesador y sus periféricos. MicroBlaze da muchos problemas para implementar más de uno en un proyecto ISE: como el programa no te deja hacerlo directamente, habría que hacerlo instanciando los MicroBlaze como *cajas negras*, pero en este caso para actualizar el programa empotrado tendremos problemas, ya que no podemos usar en el DTE la opción de EDK *Update Bitstream with Processor Data*. La alternativa sería resintetizar los MUTs y reimplementar el DTE cada vez que cambiemos el programa, con la consiguente pérdida de tiempo, o utilizar a mano el programa Data2Mem. Es por todas estas razones por la que se ha descartado realizar una implementación SEU + GOLD de MicroBlaze, ya que de todas formas en este trabajo se propone un modelo de implementación nuevo.

Capítulo 9

Programa empotrado

9.1. Descripción

El programa implementado es un sencillo programa que realiza un mínimo tratamiento de señal. Recibe datos por el puerto de entrada (GPIO 1) y calcula la raíz cuadrada del valor recibido, valor que saca por la salida (GPIO 2).

9.2. Versión estándar

A continuación se muestra el código fuente del programa embebido. Por comodidad se muestra únicamente la versión para Leon3 y no ambas (Leon3 y MicroBlaze), aunque el código es portable cambiando adecuadamente las direcciones del GPIO puesto que son distintas en ambos microprocesadores.

```
#include "stdio.h"
//**********************
//NON FAULT TOLERANT VERSION
//**********************
#define GPIO1 0x80000B00   //Input
#define GPIO2 0x80000900   //Output
#define GPIO_IN 0x00
#define GPIO_OUT 0x01
#define GPIO_DIR 0x02

//-----------------------------------------------------------------------------
////////////////////////////////////////////////////////////////////////////////
//Definiciones asociadas al GPIO
////////////////////////////////////////////////////////////////////////////////
#define gpio_data_out_reg GPIO_OUT //El puerto 2 es de salida. La direcc 2
                                   //(en incrementos de 32 bits)
                                   //corresponde a este puerto.
#define gpio_data_in_reg GPIO_IN //El puerto 1 es de entrada.
////////////////////////////////////////////////////////////////////////////////
//-----------------------------------------------------------------------------

//Como son globales se auto-inicializan a cero
```

```
  volatile unsigned int dato, datoo, ultimo_dato;

int rootq(unsigned int val){
  int i;
  unsigned int ret=0;
  for (i=15;i>=0;i--){
    ret |= 1 << i;
    if((ret*ret)>val)
      ret &= ~(i << i);
  }
  return ret;
}

int main(){
  volatile unsigned int *ip;
  volatile unsigned int *ip2;
  ip=(unsigned int *)GPIO1;
  ip2=(unsigned int *)GPIO2;
  ip[GPIO_DIR]=0;
  ip2[GPIO_DIR]=0xFFFFFFFF;
  ip2[GPIO_OUT]=0xFFFFFFFF;

  dato=ip[gpio_data_in_reg];

  while (1){  //Funcion raiz cuadrada
  datoo=rootq(dato);
  ip2[gpio_data_out_reg]=datoo;
  ultimo_dato=dato;
  do {
    dato=ip[gpio_data_in_reg];
  } while(ultimo_dato==dato);  //Esperamos a que la entrada cambie
  }
}
```

9.3. Versión tolerante a fallos

Hay que comentar que, dado que la tolerancia a fallos se ha implementado de forma manual, sólo se han protegido los datos, por lo que los errores que ocurran en el flujo de control de programa (por ejemplo, como consecuencia de faltas inyectadas en el contador de programa) no serán detectados ni corregidos. Dada la simplicidad del problema, esto provoca un comportamiento interesante (ver capítulo 11).

```
#include "stdio.h"
//**********************
//FAULT TOLERANT VERSION
//**********************
#define GPIO1 0x80000B00   //Input
#define GPIO2 0x80000900   //Output
#define GPIO_IN 0x00
#define GPIO_OUT 0x01
#define GPIO_DIR 0x02

//-----------------------------------------------------------------------
////////////////////////////////////////////////////////////////////////
//Definiciones asociadas al GPIO
```

```
////////////////////////////////////////////////////////////////////////////
#define gpio_data_out_reg GPIO_OUT //El puerto 2 es de salida. La direcc 2
                                   //(en incrementos de 32 bits)
                                   //corresponde a este puerto.
#define gpio_data_in_reg GPIO_IN //El puerto 1 es de entrada.
////////////////////////////////////////////////////////////////////////////
//--------------------------------------------------------------------------

//Como son globales se auto-inicializan a cero
  volatile unsigned int dato, datoo, ultimo_dato;
  volatile unsigned int dato1, datoo1, ultimo_dato1;
  volatile unsigned int dato2, datoo2, ultimo_dato2;

int rootq(unsigned int val){
  int i;
  unsigned int ret=0;

  for (i=15;i>=0;i--){
    ret |= 1 << i;
    if((ret*ret)>val)
      ret &= ~(i << i);
  }
  return ret;
}

int main(){

  volatile unsigned int *ip;
  volatile unsigned int *ip2;

  ip=(unsigned int *)GPIO1;
  ip2=(unsigned int *)GPIO2;
  ip[GPIO_DIR]=0;
  ip2[GPIO_DIR]=0xFFFFFFFF;

  ip2[GPIO_OUT]=0xFFFFFFFF;

  dato=ip[gpio_data_in_reg];
  dato1=ip[gpio_data_in_reg];
  dato2=ip[gpio_data_in_reg];

  while (1){  //Funcion raiz cuadrada
  datoo=rootq(dato);
  datoo1=rootq(dato1);
  datoo2=rootq(dato2);
  if(datoo==datoo1 || datoo==datoo2)  //no fallo, o fallo en datoo1 o datoo2
    ip2[gpio_data_out_reg]=datoo;
  else //fallo en datoo, datoo1 == datoo2 es correcto
    ip2[gpio_data_out_reg]=datoo1;

  ultimo_dato=dato;
  ultimo_dato1=dato1;
  ultimo_dato2=dato2;

  do {
    dato=ip[gpio_data_in_reg];
    dato1=ip[gpio_data_in_reg];
    dato2=ip[gpio_data_in_reg];
  }
  while((ultimo_dato==dato && ultimo_dato1==dato1) ||
        (ultimo_dato==dato && ultimo_dato2==dato2) ||
        (ultimo_dato1==dato1 && ultimo_dato2==dato2) );
```

```
        //Esperamos a que la entrada cambie, requerimos
        //que cambien dos de 3 para evitar cuelgues
  }
}
```

Capítulo 10

Resultados experimentales

A continuación se muestran los resultados de las campañas de inyección de faltas sobre los microprocesadores Leon3 y MicroBlaze, corriendo los programas normal y protegido contra fallos:

Proporción de faltas que causan daño	Diseño desprotegido	Diseño protegido por software	Tamaño Target (bits)
All registers and BRAMs	0.7	1.8	80331
All registers	2.7	3.0	11211
BRAMs	0.3	1.1	73728
dlmb module	3.4	2.7	18
gpio module	1.0	0.0	733
gpio output pins	85.9	70.4	32
ilmb module	12.8	3.5	18
mb_opb module	28.0	17.2	125
microblaze_0 core	2.7	1.8	5677
register file	3.2	3.0	4096
program counter	1.0	0.2	512

Tabla 10.1: Resultados experimentales sobre MicroBlaze, obtenidos con FT-UNSHADES-uP

Proporción de faltas que causan daño	Diseño desprotegi-do	Diseño protegido por software	Tamaño Target (bits)
All registers and BRAMs	0,1	0,2	999034
instcache	0.5	0,5	294912
datacache	0	0	294912
gpio1 (input)	0	0	624
gpio2 (output)	0	0	41
gpio output pins	92,2	91,6	32
reset module	80,5	81,9	6
timer module	0	0	162
jtag module	0	0	111
u0: leon3 + cache	0.1	0,2	665650
p0: leon3 core	3,0	2,9	1841
iInteger unit	3,9	3,5	1448
div0	0	0	85
mul0	0	0	105
register file	0,4	0,3	36864
rfo_data	3,5	2,0	64
tbo_data	0	0	128
program counter	8,1	9,4	184

Tabla 10.2: Resultados experimentales sobre Leon3, obtenidos con FT-UNSHADES-uP

10.1. Clasificación de los fallos

Como consecuencia de los análisis realizados, en caso de fallo se pueden clasificar las faltas en dos categorías:

- El programa calcula un resultado incorrecto (los datos han sido corrompidos)
- El programa deja de dar resultados a la salida y nunca termina (el flujo de programa ha divergido a un estado desconocido)

Si no se detecta ninguno de estos fallos a la salida, no podemos saber si la falta ha quedado latente o se ha corregido, ya que al utilizar el modelo de Tabla Inteligente no tenemos una implementación GOLD con la que comparar bit a bit[1].

[1]De todas formas, si utilizamos la opción CLEAR@DAMAGE, el hecho de que los errores se puedan propagar de un ciclo de inyección a otro no nos causa problemas

Capítulo 11

Conclusiones y Trabajos Futuros

"Futuro, s. Época en que nuestros asuntos prosperan, nuestros amigos son leales y nuestra felicidad está asegurada"
– Ambrose Bierce, Diccionario del Diablo

11.1. Conclusiones

Se ha desarrollado una plataforma para el análisis y el endurecimiento óptimo de sistemas microprocesador. Se ha presentado y demostrado con dos microprocesadores de amplio uso en la industria que corren un programa de ejemplo, en su versión estándar y en su versión tolerante a fallos con redundancia en las variables.

Se ha implementado un microprocesador embebido que corre como un System on Chip (SoC), con memorias internas dentro de la FPGA. Realmente se han implementado dos microprocesadores, MicroBlaze y Leon3, para poder realizar experimentos y comparaciones sobre distintas arquitecturas microprocesador.

Se ha ampliado la herramienta FT-Unshades, añadiéndole múltiples funcionalidades, creando así la plataforma FT-Unshades-uP.

Una de las conclusiones que surgen de nuestro estudio es la necesidad de protección híbrida Hardware-Software. Si quieres utilizar SIFT, hay que realizar esta protección híbrida necesariamente ya que la protección del software no te protege de fallos en cualquier sitio, únicamente te protege de fallos en:

- Los registros del microprocesador
- Las memorias caché
- La memoria externa

Cabe plantearse: ¿Qué ocurre si una partícula nos produce un fallo en el módulo de reset de nuestro microprocesador? Desgraciadamente, toda la protección que hayamos hecho en el software no hará que podamos recuperarnos de un reset espúreo del microprocesador. La conclusión de todo esto es que podemos realizar una protección selectiva de los módulos críticos del hardware, y utilizar técnicas SIFT para la protección de los registros del microprocesador, memorias caché y memoria externa. Esto nos dará una buena protección del conjunto, equilibrando el gasto en área y el gasto en tiempo de procesado. Por último, comentar que también tenemos la opción de proteger todo el hardware y **además** utilizar SIFT, lo cual redundará en un sistema muy protegido contra radiación a cambio de un mayor coste (de área, consumo de potencia, tamaño de datos y programa, y tiempo de procesado).

Otra observación que puede hacerse es que la adición de las protecciones software mejora la tolerancia de ciertas partes del diseño, pero causa el efecto opuesto en otros submódulos. Aparte del hecho de que no se tomaron medidas para dar protección al *control flow* del programa, parte de explicación para este comportamiento se puede encontrar considerando que, ya que el programa de ejemplo es muy simple, estimula una parte muy pequeña del diseño completo, pero al añadir las instrucciones de redundancia, se incrementa la actividad del circuito, haciendo que faltas que antes quedaban latentes se manifiesten como fallos del sistema.

Los pines de salida del GPIO son la parte más sensible del diseño, lo cual es lógico, considerando que son las salidas finales del diseño y no han sido triplicadas mediante TMR, pero su sensibilidad mejora con el software protegido.

La baja incidencia de fallos en el contador de programa probablemente se deba al hecho de que sus bits ya son redundantes en el propio MicroBlaze: hay una primitiva SRL16 por cada bit del contador de programa, de forma que la mayoría de las faltas inyectadas no afectan a los bits que de verdad se usan. En el procesador Leon3, hay un contador de programa diferente para cada etapa del pipeline por lo que no todas las faltas inyectadas pueden corromper el flujo de programa.

Cuando se analizan las diferencias en los porcentajes totales de faltas que causan daño en ambos procesadores, hay que tener en cuenta el hecho de que su tamaño (tanto en registros como en bits de memoria) es distinto. Para análisis futuro se considerará la eliminación selectiva de memoria que no está siendo usada, de forma que los tamaños de ambas implementaciones coincidan en la medida de lo posible.

11.2. Trabajos futuros

La presente investigación pretende ser una infraestructura que sirva como punto de partida para nuevos trabajos en materia de estudio de la tolerancia a fallos en sistemas microprocesador. Entre otras cosas, como trabajos futuros podemos considerar:

- Utilizar esquemas automáticos de protección del SW para implementar Software endurecido tanto en variables como en flujo de programa y comparar la tolerancia a fallos de las versiones protegidas con las de las versiones sin proteger
- Implementación de otros microprocesadores HDL como OpenRisc, haciendo especial hincapié en aquellos con más salida para aplicaciones espaciales o para entornos radioactivos
- Realizar una implementación de PicoBlaze y comparar sensibilidades con MicroBlaze para distintos tipos de tareas. Es posible que, dado el tamaño reducido del PIC, obtengamos mejor tolerancia a fallos para tareas sencillas de control que si utilizamos un microprocesador completo
- Realizar tests de radiación sobre microprocesadores y comparar datos con las campañas de inyección realizadas con FT-UNSHADES-uP
- Creación de una plataforma de inyección de faltas que permita almacenar el programa en una memoria externa pero permitiendo insertar faltas en esta memoria, de forma que se pueda realizar la emulación de un microprocesador que corra un programa de alta complejidad como por ejemplo Linux empotrado

Bibliografía

[1] GHDL user guide. http://ghdl.free.fr/ghdl/index.html.

[2] Microblaze processor reference guide. Xilinx User Guide 081 (v 7.0).

[3] Página web de gaisler research. http://www.gaisler.com.

[4] M. Aguirre and J. Tombs. FT-UNSHADES user manual, Oct 2004. GTE-AICIA. Proyecto FTUNSHADES.

[5] M. Aguirre, J. N. Tombs, F. Muñoz, V. Baena, A. Torralba, A. Fernández-León, and F. Tortosa-López. FT-UNSHADES: A new system for SEU injection, analysis and diagnostics over post synthesis netlist. In *8th Military and Aerospace Programmable Logic Device (MAPLD) International Conference*, Sept. 2005.

[6] M. A. Aguirre. Página web del proyecto Unshades. http://www.gte.us.es/˜aguirre/Web_unshades/index.

[7] M. A. Aguirre, J. N. Tombs, F. Muñoz, V. Baena, H. Guzman, J. Napoles, A. Torralba, A. Fernandez-Leon, F. Tortosa-Lopez, and D. Merodio. Selective protection analysis using a SEU emulator: Testing protocol and case study over the leon2 processor. *IEEE Transactions on Nuclear Science*, 54:951–956, Aug. 2007.

[8] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J. C. Fabre, J. C. Laprie, E. Martins, and D. Powell. Fault injection for dependability validation: a methodology and some applications. *IEEE Transactions on Software Engineering*, 16(2):166–182, Feb. 1990.

[9] J. Arlat, Y. Crouzet, J. Karlsson, P. Folkesson, E. Fuchs, and G. H. Leber. Comparison of physical and software-implemented fault injection techniques. *IEEE Trans. Computers*, 52(9):1115–1133, 2003.

[10] J. Boue, P. Petillon, and Y. Crouzet. MEFISTO-l: a VHDL-based fault injection tool for the experimental assessment of fault tolerance. In *Fault-Tolerant Computing, 1998. Digest of Papers. Twenty-Eighth Annual International Symposium on*, pages 168–173, Munich, June 1998.

[11] C. Carmichael. Correcting single-event upsets through virtex partial configuration, Jun 2000. Xilinx Application Note XAPP216.

[12] J. Carreira, H. Madeira, and J. G. Silva. Xception: a technique for the experimental evaluation of dependability in modern computers. *IEEE Transactions on Software Engineering*, 24(2):125–136, Feb. 1998.

[13] D. J. Cochran, S. P. Buchner, C. Poivey, K. A. LaBel, R. L. Ladbury, M. O'Bryan, J. W. Howard, A. Sanders, and T. Oldham. Compendium of current total ionizing dose results and displacement damage results for candidate spacecraft electronics for NASA. In *Radiation Effects Data Workshop, 2007 IEEE*, volume 0, pages 146–152, July 2007.

[14] C. Fuhrman, S. Chutani, and H. Nussbaumer. Hardware/software fault tolerance with multiple task modular redundancy. In *Hardware/software fault tolerance with multiple task modular redundancy*, June 27 1995.

[15] M. Garcia-Valderas, M. Portela-Garcia, C. Lopez-Ongil, and L. Entrena. An extension of transient fault emulation techniques to circuits with embedded memories. In *Design and Diagnostics of Electronic Circuits and systems, 2006 IEEE*, pages 216–217, Prague.

[16] D. González. The SEUs Simulation Tool (SST), functional description, Apr 2004. European Space Agency (ESA). Document Reference TEC-EDM/DGG-SST2.

[17] I. González and L. Berrojo. Supporting fault tolerance in an industrial environment: the AMATISTA approach. In *On-Line Testing Workshop, 2001. Proceedings. Seventh International*, pages 178–183, Taormina, July 2001.

[18] R. Guerraoui and A. Schiper. Software-based replication for fault tolerance. *Computer*, 30(4):68–74, Apr. 1997.

[19] H. Guzmán-Miranda. Investigación sobre herramientas de Codiseño Hardware-Software, May 2006. Proyecto de Fin de Carrera.

[20] F. M. Henrique Madeira, Mário Rela and J. G. Silva. RIFLE: A general purpose pin-level fault injector. In *Lecture Notes in Computer Science*, volume 852/1994. Springer Berlin / Heidelberg, 1994. ISSN 0302-9743 (Print) 1611-3349 (Online).

[21] J. Hudak, B. H. Suh, D. Siewiorek, and Z. Segall. Evaluation and comparison of fault-tolerant software techniques. *IEEE Transactions On Reliability*, 42(2):190–204, 1993.

[22] E. C. Jiri Gaisler, Sandi Habinc. Grlib ip library user manual, 2007.

[23] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham. FERRARI: a tool for the validation of system dependability properties. In *Fault-Tolerant Computing, 1992. FTCS-22. Digest of Papers., Twenty-Second International Symposium on*, pages 336–344, Boston, MA, July 1992.

[24] F. Lima, S. Rezgui, L. Carro, R. Velazco, and R. Reis. On the use of vhdl simulation and emulation to derive error rates. *VI European Conference on Radiation and its Effects on Components and Systems*, 2001.

[25] C. Lopez-Ongil, L. Entrena, M. Garcia-Valderas, M. Portela, M. A. Aguirre, J. Tombs, V. Baena, and F. Muñoz. A unified environment for fault injection at any design level based on emulation. *IEEE Transactions on Nuclear Science*, 54:946–950, Aug. 2007.

[26] C. Lopez-Ongil, M. Garcia-Valderas, M. Portela-Garcia, and L. Entrena. Autonomous fault emulation: A new FPGA-based acceleration system for hardness evaluation. *IEEE Transactions on Nuclear Science*, 54:252–261, Feb. 2007.

[27] A. Manzone and D. De Costantini. Fault tolerant insertion and verification: a case study. In *Memory Technology, Design and Testing, 2002. (MTDT 2002). Proceedings of the 2002 IEEE International Workshop on*, pages 44–48, July 2002.

[28] J. M. Mogollón. Síntesis del estado del arte: Emulación hardware de seu y transitorios mediante inyección de errores, Mar 2007. Proyecto Emuláser.

[29] J. Napoles, H. Guzman, M. Aguirre, J. N. Tombs, F. Muñoz, V. Baena, A. Torralba, and L. G. Franquelo. Radiation environment emulation for VLSI designs: A low cost platform based on xilinx FPGA's. In *Industrial Electronics, 2007. ISIE 2007. IEEE International Symposium on*, pages 3334–3338, Vigo, June 2007.

[30] M. Ohlsson, P. Dyreklev, K. Johansson, and P. Alfke. Neutron single event upsets in SRAM-based FPGAs. In *Radiation Effects Data Workshop, 1998. IEEE*, pages 177–180, Newport Beach, CA, July 1998.

[31] M. Portela-García. Técnicas de inyección de fallos basadas en FPGAs para la evaluación de la tolerancia a fallos de tipo SEU en circuitos digitales, 2007. Tesis Doctoral, Universidad Carlos III de Madrid.

[32] L. L. Pullum. *Software fault tolerance techniques and implementation*. Artech House, 2001.

[33] P. Reyes, P. Reviriego, J. A. Maestro, and O. Ruano. New protection techniques against SEUs for moving average filters in a radiation environment. *IEEE Transactions on Nuclear Science*, 54:957–964, Aug. 2007.

[34] P. K. Samudrala, J. Ramos, and S. Katkoori. Selective triple modular redundancy (STMR) based single-event upset (SEU) tolerant synthesis for fpgas. *IEEE Trans. Nuclear Science*, 51, 2004.

[35] E. Shokri and H. Hecht. Matching software fault tolerance with application needs. In *HASE*, page 248. IEEE Computer Society, 1998.

[36] L. Sterpone and M. Violante. An analysis of SEU effects in embedded operating systems for real-time applications. *IEEE International Symposium on Industrial Electronics, ISIE 2007*, 2007.

[37] J. Tombs, F. Muñoz, V. Baena, A. Torralba, L. Franquelo, A. Fernández-León, D. González-Gutiérrez, and F. Tortosa-López. A hardware approach for SEU immunity verification using Xilinx FPGAs. *XIX Conference on Design of Circuits and Integrated Systems DCIS'2004*, 2004.

[38] A. P. Vega-Leal, F. R. Palomo, A. Rodríguez, H. Guzmán-Miranda, J. Nápoles, J. N. Tombs, M. Aguirre, J. M. Mogollón, J. García, Y. Morilla, and F. Garcia. the experience of starting-up a radiation test at the 18 mev cyclotron in the spanish national accelerators center. *9th European Conference Radiation and Its Effects on Components and Systems (RADECS)*, Sept. 2007.

[39] M. Violante. Fault tolerant systems: Modelling, analysis and design, 2006. Apuntes del curso de doctorado, Politecnico di Torino.

[40] Wikipedia, the Free Enciclopedia. www.wikipedia.org.

[41] C. C. Yui, G. M. Swift, C. Carmichael, R. Koga, and J. S. George. SEU mitigation testing of xilinx virtex II FPGAs. In *Radiation Effects Data Workshop, 2003. IEEE*, pages 92–97, July 2003.