
Diseño y análisis de un procesador tolerante a fallos transitorios compatible con ARM a nivel de instrucciones



TRABAJO FIN DE GRADO

Andrés Gamboa Meléndez

Grado en Ingeniería de Computadores

Facultad de Informática

Universidad Complutense de Madrid

Junio 2015

Documento maquetado con T_EX!S v.1.0.

Este documento está preparado para ser imprimido a doble cara.

Diseño y análisis de un procesador tolerante a fallos transitorios compatible con ARM a nivel de instrucciones

Trabajo fin de grado

Grado en Ingeniería de Computadores

Versión 1.0

Grado en Ingeniería de Computadores

Facultad de Informática

Universidad Complutense de Madrid

Junio 2015

Copyright © Andrés Gamboa Meléndez

Resumen

Los sistemas electrónicos son susceptibles a que partículas de energía colisionen contra los componentes nanométricos que los componen, interfieran en el proceso que estén llevando a cabo y modifiquen los datos internos. Esto incluye a los microprocesadores que se utilizan en los dispositivos de propósito general o específico.

En algunas áreas de trabajo estas partículas se encuentran en una mayor cantidad, y con una mayor energía. Lo que supone que aumente la probabilidad de que ocurran colisiones y el sistema funcione de forma impredecible.

Este proyecto consiste en implementar una unidad de control capaz de funcionar correctamente aún recibiendo interferencias de estas partículas. El proyecto ha abordado tres tareas:

1. Primero se implementará la unidad de control basada en la arquitectura DLX estudiada en los cursos de estructura y arquitectura de computadores. Adaptándola al repertorio de instrucciones elegido.
2. A continuación se aplicarán técnicas de tolerancia a fallos sobre la CPU.
3. Para finalizar se realizarán comprobará que la tolerancia a fallos se ha aplicado satisfactoriamente y aumenta el grado de fiabilidad del sistema.

Abstract

...

Índice

Resumen	v
Abstract	vii
1. Introducción	1
1.1. Introducción	1
1.2. Motivación	1
1.3. Planteamiento del problema	3
1.4. Objetivo	4
1.5. Estructura del documento	4
2. Introducción al procesador	7
2.1. Procesador	7
2.1.1. Arquitectura	8
2.1.2. Repertorio de instrucciones	8
2.1.3. Memoria	8
2.1.4. Segmentación	9
2.2. ARM	11
2.2.1. Arquitectura ARM	12
2.2.2. Repertorio de instrucciones ARM	13
2.2.3. Segmentación ARM	14
2.2.4. Memoria ARM	15
2.3. Field-Programmable Gate Arrays (FPGA)	15
2.3.1. Nexys 4	16
3. Introducción a los fallos y su tolerancia	19
3.1. Introducción a los fallos	19
3.1.1. Causas	19
3.1.2. Tipos de fallos	20
3.2. Tolerancia a Fallos	23
3.2.1. Redundancia en la información	24
3.2.2. Redundancia en el tiempo	25

3.2.3. Redundancia en el hardware	25
3.3. Tolerancia en microprocesadores	27
4. Procesador	31
4.1. Introducción	31
4.2. Arquitectura del procesador	31
4.2.1. Estructura	31
4.2.2. Repertorio de instrucciones	33
4.2.3. Segmentación	34
4.2.4. Memoria	35
4.3. Implementación	35
4.3.1. Banco de registros	35
4.3.2. Contador de programa	37
4.3.3. Unidad Aritmético-Lógica (ALU)	37
4.3.4. Control principal	38
4.3.5. Registros de control	39
4.3.6. Memoria de instrucciones	39
4.3.7. Memoria de datos	40
4.4. Formato de instrucciones	41
4.4.1. Accesos a Memoria	41
4.4.2. Procesamiento de datos	42
4.4.3. Operaciones de control	45
5. Aplicando tolerancia a fallos	47
5.1. Introducción	47
5.2. Redundancia modular	47
5.3. El Votador	47
5.3.1. Implementación	49
5.4. Aplicación	50
6. Comprobación Funcional	53
6.1. Introducción	53
6.2. Programa de control	53
6.3. Fallos introducidos	54
6.4. Procesador estándar	55
6.4.1. Recursos	55
6.4.2. Ejecución de control	57
6.4.3. Inserción de fallos	57
6.5. Procesador tolerante a fallos	58
6.5.1. Recursos	58
6.5.2. Ejecución de control	59

6.5.3. Inserción de fallos	59
7. Análisis de los resultados	61
7.1. Introducción	61
7.2. Comparativa de procesadores	61
7.3. Simulaciones	62
7.3.1. Procesador estándar	62
7.3.2. Procesador tolerante a fallos	64
8. Conclusiones y futuras líneas de trabajo	67
8.1. Conclusiones	67
8.2. Futuras líneas de trabajo	68
A. Código de simulación sin fallos	69
A.1. Testbench para ejecución de control	69
B. Código de simulación con fallos	75
B.1. Testbench para ejecución con inserción de fallos en CPU es- tándar	75
B.2. Testbench para ejecución con inserción de fallos en CPU to- lerante a fallos	82
C. Simulaciones	89
C.1. Procesador estándar sin fallos	90
C.2. Procesador estándar con fallos	91
C.3. Procesador tolerante a fallos sin fallos	92
C.4. Procesador tolerante a fallos con fallos	93
Bibliografía	95

Índice de figuras

1.1. Equipamiento de los hogares en algunos productos tecnológicos.	2
1.2. Flujo de neutrones a 40.000 pies de altitud [11].	3
2.1. Procesador	7
2.2. Arquitectura Von Neumann y Arquitectura Harvard	9
2.3. Ejecución secuencial comparada con ejecución segmentada . .	10
2.4. Procesador Qualcomm Snapdragon 810	12
2.5. Segmentación ARM	15
2.6. Arquitectura de una FPGA [2].	16
2.7. Distribución del uso de FPGAs en el año 2008.	17
2.8. Placa de prototipado Nexys 4	17
3.1. Single Event Upset en una FPGA [11].	20
3.2. Flujo de neutrones a 40.000 pies de altitud [11].	21
3.3. Fallos Transitorios	22
3.4. Fallo enmascarado por una puerta lógica.	23
3.5. Fallo enmascarado eléctricamente.	23
3.6. Fallo enmascarado por ventana de tiempo.	24
3.7. Aplicando Triple Modular Redundancy (TMR)	26
3.8. Tolerancia dinámica parcial [25].	27
4.1. Estructura del procesador diseñado	32
4.2. Diseño completo del procesador segmentado.	36
4.3. Banco de registros.	36
4.4. Contador de programa.	37
4.5. Unidad aritmético-lógica.	37
4.6. Control principal.	38
4.7. Registros de control	40
4.8. Memoria de Instrucciones.	40
4.9. Memoria de datos.	41
5.1. Sustitución de biestable.	48

5.2. Diseño de votador con puertas lógicas	49
5.3. Ejemplos de fallos en entradas.	49
5.4. Procesador con TMR en los registros de control.	51
6.1. Código Thumb-2 de programa de pruebas.	54
6.2. Pseudo-código de programa de pruebas.	55

Índice de Tablas

2.1. Segmentacion simple de 5 etapas	9
4.1. Instrucciones de acceso a memoria (bits 31..16)	41
4.2. Instrucciones de acceso a memoria (bits 15..0)	42
4.3. Instrucciones de procesamiento de datos con dos registros (bits 31..16)	43
4.4. Instrucciones de procesamiento de datos con dos registros (bits 15..0)	43
4.5. Operaciones con dos registros	44
4.6. Instrucciones de procesamiento de datos con un registro y un inmediato (bits 31..16)	44
4.7. Instrucciones de procesamiento de datos con un registro y un inmediato (bits 15..0)	45
4.8. Operaciones con un registro y un inmediato	45
4.9. Instrucciones de control (bits 31..16)	45
4.10. Instrucciones de control (bits 15..0)	46
5.1. Tabla de verdad del votador	48
7.1. Simulación de control sobre el procesador estándar.	63
7.2. Simulación con fallos sobre el procesador estándar.	64
7.3. Simulación de control sobre el procesador tolerante a fallos . .	65
7.4. Simulación con fallos sobre el procesador tolerante a fallos. . .	66

Capítulo 1

Introducción

1.1. Introducción

Esta monografía es el resultado del estudio e investigación realizados para la asignatura «Trabajo de Fin de Grado» del Grado en Ingeniería de Computadores que se ha llevado a cabo en el departamento de «Arquitectura de Computadores y Automática (DACYA)» de la Universidad Complutense de Madrid (UCM), bajo la dirección del Dr. José Miguel Montañana Aliaga.

El trabajo se centra en el desarrollo e implementación de un microprocesador tolerante a fallos transitorios, con un diseño que le permita ser compatible con las instrucciones ARM. La tolerancia a fallos aplicada ha sido el «modelo de replicado triple de módulos(TMR)» [8].

1.2. Motivación

Hoy en día, el uso de la tecnología y la informática se extienden a nivel mundial, con aplicación a cada vez, un mayor número de campos. La tecnología está cada vez más presente en nuestras vidas, ya no se concibe un hogar o puesto de trabajo sin un ordenador sobre la mesa. El uso de los dispositivos electrónicos de carácter personal va en aumento, convirtiéndose en elementos imprescindibles en nuestros hogares (figura 1.1). Las estadísticas publicadas por el Instituto Nacional de Estadística (INE) [4], muestran que en España más del 95 % de los hogares posee al menos un teléfono móvil, normalmente teléfonos inteligentes, y más del 70 % posee un ordenador personal, lo que es un indicativo de la necesidad y dependencia tecnológica existente en estos tiempos.

Estos dispositivos, y muchos otros presentes en nuestra vida cotidiana, son sistemas con componentes micro-electrónicos. Poseen un microprocesador que es el «cerebro» y responsable de dirigir el sistema ejecutando los programas.

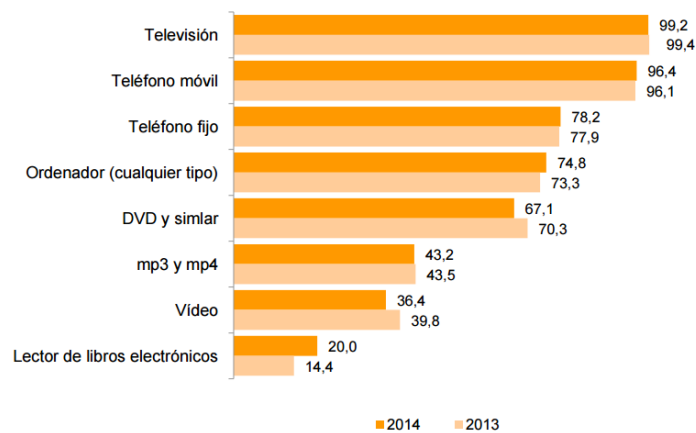


Figura 1.1: Equipamiento de los hogares en algunos productos tecnológicos.

Los microprocesadores, como todos los sistemas, son susceptibles de sufrir fallos y producir errores a varios niveles (se explican en el capítulo 3), errores que provocan comportamientos erráticos y no deseados.

Con un tamaño cada vez más pequeño, los sistemas electrónicos resultan más sensibles a los efectos de los «ruidos de transmisión»¹ producidos por la radiación y los rayos cósmicos.

Las radiaciones cósmicas puede provocar fallos en cualquier sistema electrónico, dañando el mismo permanente o temporalmente, por ello, los sistemas de los satélites que orbitan alrededor de la tierra o de los aviones que se desplazan a gran altura deben ser mucho más robustos que los sistemas que trabajan a nivel del suelo. En la figura 1.2, se representa la variación de la radiación con respecto a la longitud y latitud terrestres. Se observa que hay un mayor flujo de radiación cuanto mayor es la distancia al ecuador. Fundamentalmente debido a la menor protección que proporciona la atmósfera frente a la radiación y los rayos cósmicos provenientes del espacio.

Los fallos, en el terreno de la medicina, pueden provocar consecuencias fatales e incluso la pérdida de vidas humanas. Tal puede ser el caso en los sistemas biomédicos encargados de asistir a la vida de una persona, como un marcapasos o un equipo de respiración asistida.

En el sector del transporte, vehículos y aeronaves controlados por sistemas electrónicos pueden sufrir fallos y causar accidentes con pérdidas humanas y económicas.

En el campo aeroespacial si un sistema falla de forma irrecuperable, causará la pérdida del sistema completo con un coste económico muy elevado.

El día 7 de octubre de 2008, un avión Airbus A330-303 que había despegado de Singapur con destino Perth, Australia, sufrió dos descensos rápidos

¹Interferencias en la señal que tiende a enmascarar la información transmitida.

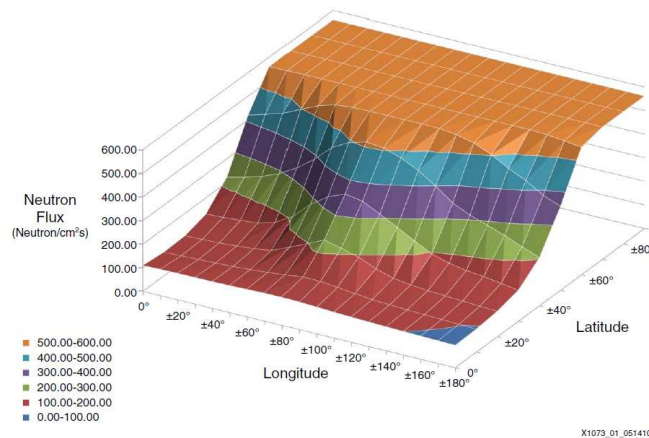


Figura 1.2: Flujo de neutrones a 40.000 pies de altitud [11].

de 650 y 400 pies. Tras las investigaciones, las autoridades australianas llegaron a la conclusión que la «Air Data Inertial Reference Unit» (ADIRU) podría haber sufrido un fallo provocado por radiaciones cósmicas [13].

Para garantizar el correcto funcionamiento de estos sistemas existen múltiples técnicas de tolerancia a fallos, técnicas que ayudan a detectar los fallos y a recuperar el sistema antes de que causen errores. Aplicando una o varias de estas técnicas se obtiene un sistema robusto y capaz de funcionar ante los fallos inducidos por la radiación u otros agentes externos.

1.3. Planteamiento del problema

Este trabajo tiene en cuenta que el motor principal de muchos sistemas es el procesador o microprocesador. El procesador de un sistema es su *cerebro*, más concretamente es el encargado de ejecutar las instrucciones que componen los programas.

Si no se toman medidas de prevención y tolerancia, este *cerebro* puede ver alterado su comportamiento por efectos externos, provocando errores de ejecución del programa. Errores que a su vez pueden ser causa de un comportamiento no deseado alterando los datos, modificando el funcionamiento del propio procesador o de otros componentes del sistema como las memorias o los controladores entrada/salida.

Con este trabajo se pretende ofrecer un medio para evitar estas situaciones concediendo un grado extra de fiabilidad a los sistemas basados en microprocesadores. Para ello se quiere diseñar e implementar un microprocesador sencillo capaz de ejecutar un conjunto reducido de instrucciones (RISC), al que posteriormente se le aplicarán las técnicas de tolerancia a fallos, aumentando así su capacidad de detectar e incluso recuperarse de los fallos.

1.4. Objetivo

Una vez planteado el problema señalado en el apartado anterior, el objetivo del presente trabajo es diseñar un procesador con un grado de fiabilidad mayor que un procesador convencional.

El proyecto se ha dividido en cuatro tareas dedicadas a la implementación del microprocesador y a la aplicación de la tolerancia a fallos.

1 Implementación del procesador.

Se ha implementado el procesador segmentado en 5 etapas. Para ello se ha partido de la arquitectura DLX vista en las asignaturas de computadores de nuestro grado.

2 Ruta de control

Se ha rediseñado la ruta de control y parte de la ruta de datos. El nuevo juego de instrucciones empleado, completamente distinto al que utiliza un DLX convencional, obliga a cambiar la ruta de control. Simulando unos pequeños programas se comprueba que el procesador es capaz de decodificar y ejecutar las nuevas instrucciones.

3 Diseño de tolerancia a fallos.

Una vez implementado el procesador completo y comprobado su funcionamiento se diseña y se incorpora la tolerancia a fallos. Para ello se **triplican** los módulos que pueden causar mayor número de fallos y se insertan **votadores** de mayoría.

4 Diseño del sistema de inserción de fallos.

Para finalizar se ha diseñado un sistema externo de inserción de fallos. Este sistema es capaz de alterar los valores de las salidas de los módulos triplicados, para comprobar después como afecta esto al funcionamiento del procesador.

1.5. Estructura del documento

Capítulo 1 *Introducción:*

En el presente capítulo 1 se realiza la introducción al proyecto propuesto y realizado para el trabajo de fin de grado que se desarrolla en este documento.

Capítulo 2 *Introducción al procesador:*

En el capítulo 2 se realiza una introducción al diseño de un procesador y sus características.

Capitulo 3 *Introducción a los fallos y su tolerancia:*

En el capitulo 3 se introducen los fallos, y se definen técnicas de tolerancia frente a estos.

Capitulo 4 *Desarrollo del procesador:*

En el capitulo 4 se describe el diseño y arquitectura final del microprocesador y sus componentes.

Capitulo 5 *Proporcionando tolerancia a fallos transitorios:*

En el capitulo 5 se describe cómo se ha proporcionado la tolerancia a fallos y qué técnicas se han utilizado.

Capitulo 6 *Comprobación Funcional:*

En el capitulo 6 se muestran los resultados obtenidos de las síntesis, implementaciones y simulaciones realizadas.

Capitulo 7 *Análisis de los resultados:*

En el capitulo 7 se analizan los resultados mostrados en el capítulo anterior.

Capitulo 8 *Conclusiones:*

En el capitulo 8 se describen las conclusiones tras analizar los resultados.

Capítulo 2

Introducción al procesador

2.1. Procesador

El Diccionario de la Real Academia Española (DRAE) define el procesador como la «Unidad Central de Proceso (CPU), formada por uno o dos chips». Figura 2.1.

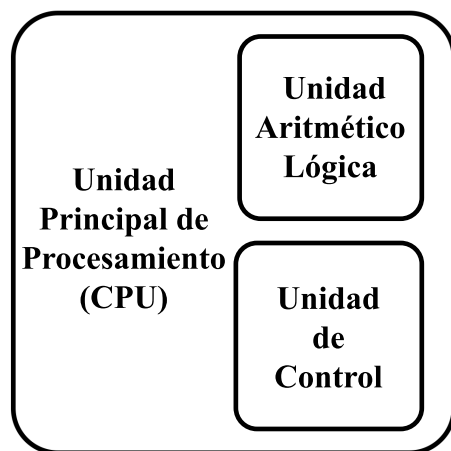


Figura 2.1: Procesador

La CPU es el circuito integrado encargado de acceder a las instrucciones de los programas informáticos y ejecutarlas. Para poder ejecutar un programa, el procesador debe realizar las siguientes tareas:

1. Acceder a las instrucciones almacenadas en memoria.
2. Analizar las instrucciones y establecer las señales de control internas.
3. Ejecutar operaciones sobre datos.

4. Almacenar los resultados en memoria.

A continuación se definen los elementos fundamentales para constituir un procesador.

2.1.1. Arquitectura

Un procesador está formado por una serie de módulos conectados entre sí, siendo la arquitectura del mismo la que define el diseño de los módulos que lo componen y de qué manera se conectan entre ellos.

La arquitectura del procesador diseñada por Von Neumann separa los componentes del procesador en módulos básicos. La CPU es el verdadero núcleo de los computadores, donde se realizan las funciones de computación y control. En la CPU se concentran todos los componentes, la memoria y los elementos de entrada/salida [9].

Según el juego de instrucciones que sea capaz de ejecutar un procesador, su arquitectura puede clasificarse como:

1. *Reduced instruction set computer (RISC)*. Utiliza un repertorio de instrucciones reducido, con instrucciones de tamaño fijo y poca variedad en su formato.
2. *Complex instruction set computer (CISC)*. Utiliza un repertorio de instrucciones muy amplio, permite realizar operaciones complejas entre las que se encuentran las de realizar cálculos entre los datos en memoria y los datos en registro.

2.1.2. Repertorio de instrucciones

El repertorio de instrucciones define todas las operaciones que el procesador es capaz de entender y ejecutar. Este juego de instrucciones incluye las operaciones aritmético-lógicas que pueden aplicarse a los datos, las operaciones de control sobre el flujo del programa, las instrucciones de lectura y escritura en memoria, así como todas las instrucciones propias que se hayan diseñado para el procesador.

2.1.3. Memoria

Los procesadores tienen una serie de registros donde se almacenan temporalmente los valores con los que está trabajando. El conjunto de estos registros se conoce como «banco de registros». Los registros de propósito general son muy limitados, por lo que el procesador necesita disponer de apoyo externo donde alojar la información, para ello tiene acceso a una memoria externa.

El modo de acceso a la memoria externa divide las arquitecturas en dos tipos, conocidas con los nombres de Von Neumann y Harvard. La arquitectura Von Neumann utiliza una única memoria para almacenar tanto los datos como las instrucciones. La arquitectura Harvard, sin embargo, separa la memoria de datos de la memoria de instrucciones. Figura 2.2.

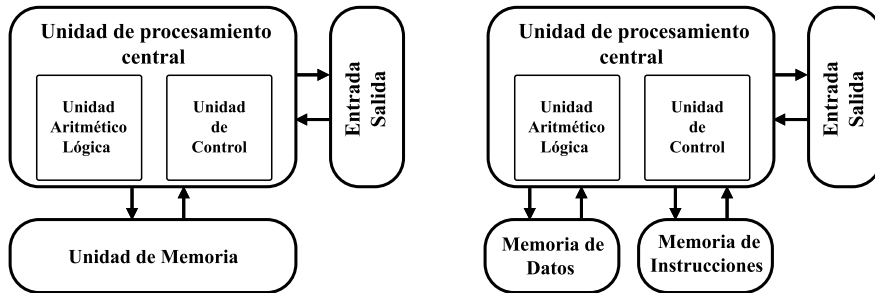


Figura 2.2: Arquitectura Von Neumann y Arquitectura Harvard

2.1.4. Segmentación

La segmentación consiste en dividir el procesador en etapas, de modo que en cada una de ellas se ejecute una parte de la instrucción. Al dividir las instrucciones se consigue que cada etapa procese de forma independiente una parte de las mismas.

Las instrucciones van avanzando de etapa en etapa hasta que se terminen de procesar. De este modo se pueden tener en el procesador varias instrucciones ejecutándose de forma simultanea en distintas etapas, lo que resulta en un aumento significativo del rendimiento del procesador.

Número de instrucción	Ciclo de reloj								
	1	2	3	4	5	6	7	8	9
i	IF	ID	EX	MEM	WB				
i + 1		IF	ID	EX	MEM	WB			
i + 2			IF	ID	EX	MEM	WB		
i + 3				IF	ID	EX	MEM	WB	
i + 4					IF	ID	EX	MEM	WB

Tabla 2.1: Segmentacion simple de 5 etapas

En la tabla 2.1 podemos ver cómo se procesan una serie de instrucciones en 5 etapas (IF, ID, EX, MEM, WB). Se observa cómo cada instrucción va ocupando una única etapa en cada ciclo de reloj, cómo cambian de una a otra al ser procesadas, permitiendo la ejecución de la siguiente instrucción.

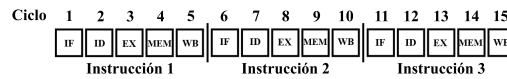
2.1.4.1. Reducción de ciclos por segmentación

La segmentación proporciona la ventaja de poder lanzar una instrucción por cada ciclo de reloj. Característica que aumenta el rendimiento del procesador al obtener un menor número total de ciclos por instrucción para un mismo programa. Para conocer los ciclos por instrucción que necesita un programa se utiliza la formula:

$$\text{Ciclos por instrucción (CPI)} = \frac{\text{Número de ciclos total}}{\text{Número de instrucciones}} \quad (2.1)$$

A modo de ejemplo, veamos que sucede al ejecutar un programa de 3 instrucciones sobre un procesador que emplee 5 ciclos de reloj en ejecutar cualquier instrucción, pero en un caso no segmentado, y en otro caso segmentado en 5 etapas de 1 ciclo cada una:

Ejecución Secuencial



Ejecución Segmentada

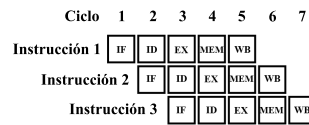


Figura 2.3: Ejecución secuencial comparada con ejecución segmentada

Como podemos ver en la figura 2.3, el procesador no segmentado tarda 15 ciclos en ejecutar las 3 instrucciones y aplicando la formula anterior se obtiene que el valor de CPI es 5. Al ejecutar el mismo programa en el procesador segmentado, este tarda 5 ciclos en ocupar las 5 etapas del procesador. A partir de ese momento con cada ciclo de reloj se completa una instrucción, completándose la ejecución del programa en 7 ciclos de reloj. El nuevo valor de CPI es de 2,33. Así pues, la segmentación ha reducido el número de ciclos por instrucción de este programa a menos de la mitad.

2.1.4.2. Inconvenientes de la segmentación

Un programa es un conjunto de instrucciones que ejecutadas en orden realizan una tarea específica. Al permitir ejecutar una instrucción sin terminar las anteriores pueden aparecer conflictos, denominados «riesgos de segmentación» y pueden ser de los siguientes tipos: [9]

1. *Riesgos estructurales*. Surgen cuando 2 o más instrucciones necesitan acceder a los mismos recursos.
2. *Riesgos de datos*. Surgen cuando la ejecución de una instrucción depende del resultado de una instrucción anterior, y este todavía no se ha escrito en el registro correspondiente. A su vez pueden ser:
 - *Lectura después de escritura (RAW)*. Una instrucción intenta leer un dato antes de que se escriba en el registro.
 - *Escritura después de lectura (WAR)*. La *instrucción i+1* escribe el resultado en el registro antes de que la *instrucción i* haya leído el dato del mismo registro. Esto solo ocurre con instrucciones que realicen una escritura anticipada como por ejemplo, las instrucciones de auto-incremento de direccionamiento.
 - *Escritura después de escritura (WAW)*. Ocurre cuando las escrituras se realizan en orden incorrecto. Por ejemplo, cuando en un mismo registro, la *instrucción i+1* escribe su resultado antes de que lo haga la *instrucción i*.
 - *Lectura después de lectura (RAR)*. Realmente no es un riesgo como tal, ya que no se modifica ningún dato.
3. *Riesgos de control*. Surgen a consecuencia de las instrucciones que afectan al registro del contador de programa (PC).

2.2. ARM

La arquitectura ARM fue originalmente desarrollada por Acorn Computer Limited, entre los años 1983 y 1985.

Actualmente la arquitectura ARM es el conjunto de instrucciones más extensamente utilizado en unidades producidas. Esto se debe a su amplio uso en los sectores de telefonía móvil, sistemas de automoción, computadoras industriales y otros dispositivos.

Lo que hace que esta arquitectura sea tan popular es la simpleza de sus núcleos, utilizan un número relativamente pequeño de transistores, permitiendo añadir funcionalidades específicas en otras partes del mismo chip [3]. Por ejemplo, uno de los procesadores de última generación de la empresa Qualcomm, el «Qualcomm Snapdragon 810» está compuesto por una CPU con 8 núcleos ARM, una unidad de procesamiento gráfico, controladores de pantalla, conectividad y cámara entre otros, todo ello en un único chip [21]. Figura 2.4.

Además de requerir poco espacio, los dispositivos ARM están diseñados con el objetivo de minimizar el consumo de energía, haciéndolo apropiado

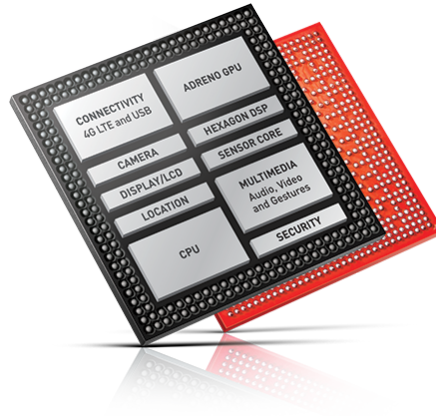


Figura 2.4: Procesador Qualcomm Snapdragon 810

para sistemas móviles empotrados¹ que dependen de una batería.

Por último, la arquitectura ARM es altamente modular, es decir, sus componentes como pueden ser la memoria caché o los controladores, se construyen como módulos independientes y opcionales..

Todo ello no impide que la arquitectura ARM resulte muy eficiente y proporcione un alto rendimiento.

2.2.1. Arquitectura ARM

La arquitectura ARM [19] deriva de la arquitectura RISC con las características propias de esta:

- Banco de registros uniforme.
- Instrucciones de tamaño fijo.
- Las instrucciones de procesamiento operan sobre los datos almacenados en los registros.
- Modos de direccionamiento simples.

La arquitectura ARM añade algunas características adicionales para proporcionar un equilibrio entre el rendimiento, el tamaño del código, el consumo y el silicio requerido, estas son:

- Actualización de flags en la mayoría de instrucciones.
- Ejecución condicional de instrucciones.

¹Sistema de computación diseñado para realizar una o pocas funciones dedicadas.

- Auto-incremento y auto-decremento para el direccionamiento.
- Instrucciones de carga y almacenamiento múltiple.

La arquitectura ARM contiene un banco de registros con 31 registros de propósito general. De estos sólo son visibles 16, a los cuales puede acceder cualquier instrucción. Los otros registros se utilizan para acelerar el procesamiento. Tres de los 31 registros tienen un uso especial y son el «puntero de pila (SP)», el «registro de enlace (LR)» y el «contador de programa (PC)».

2.2.2. Repertorio de instrucciones ARM

El repertorio de instrucciones se divide en seis categorías:

- **Salto**

Además de permitir que las instrucciones aritmético-lógicas alteren el flujo de control, almacenando sus resultados en el registro PC, se incluye una instrucción estándar capaz de aplicar un salto de hasta 32MB hacia delante o hacia atrás.

Otra instrucción de salto permite almacenar el valor del contador de programa en un registro para poder volver al mismo punto al finalizar el desvío. Esto es útil cuando se quiere llamar a una subrutina.

También es posible lanzar instrucciones de salto que realizan un cambio de juego de instrucciones, en caso de necesitar lanzar subrutinas en alguno de los otros juegos de instrucciones compatibles con la arquitectura, tal es el caso de Thumb o Jazelle.

- **Procesamiento de datos**

El procesamiento de datos se realiza mediante instrucciones aritmético-lógicas, operaciones de comparación, instrucciones sobre múltiples datos, instrucciones de multiplicación y operaciones diversas.

Las instrucciones aritmético-lógicas, como su nombre describe, ejecutan operaciones aritméticas o lógicas sobre dos operandos. El primer operando siempre será un registro, mientras que el segundo puede ser un inmediato, o un segundo registro. El resultado se almacena en un registro.

Como se ha comentado anteriormente, las operaciones de comparación aplican una operación aritmético-lógica. Sin embargo no escriben el resultado en un registro, actualizan los flags de condición.

- **Transferencia de registros de estado**

Estas instrucciones son capaces de transferir contenidos entre los registros especiales CPSR y SPSR, y los registros de propósito general.

Al escribir en el registro CPSR se consigue establecer los valores de los bits de condición, habilitar o deshabilitar interrupciones, cambiar el estado y el modo del procesador, y cambiar el modo de acceso a memoria entre «little endian» o «big endian».

- **Carga y almacenamiento**

Las instrucciones de carga y almacenamiento permiten transmitir datos entre los registros de propósito general y la memoria externa.

Se pueden cargar o almacenar los registros de forma individual, un solo dato por instrucción, o de forma colectiva, un bloque de datos con una sola instrucción.

- **Co-procesador**

Las instrucciones de co-procesador comunican el procesador principal con un co-procesador auxiliar para transmitir instrucciones o datos.

Existen tres clases de este tipo de instrucciones: Procesado de datos, comienza el trabajo específico del co-procesador. Transferencia de instrucciones, envía o recibe datos del procesador a la memoria. Transferencia de registro, envía o recibe datos entre los registros del microprocesador y el co-procesador.

- **Excepciones**

Las instrucciones de excepción generan interrupciones en el programa. Las instrucciones «Interrupción software» normalmente se utilizan para realizar peticiones al sistema operativo. Mientras que las instrucciones «Punto de interrupción software» generan excepciones abortando la ejecución del programa.

Los procesadores ARM son capaces de procesar instrucciones de tres repertorios diferentes. El repertorio ARM [26], el set Thumb/ Thumb-2 [1] y las instrucciones Jazelle [26].

2.2.3. Segmentación ARM

La evolución de los ARM ha significado un aumento en la cantidad de etapas en las que se divide el procesador. La familia «ARM7TDMI» consta de 3 etapas, mientras que la familia «ARM9TDMI» se divide en 5 etapas, y la familia «Cortex» se compone de 13 etapas.

La arquitectura del «ARM7TDMI», como se ha comentado, está segmentada en 3 etapas para aumentar la velocidad de flujo de entrada de las instrucciones en el procesador. Permite realizar varias operaciones al mismo tiempo y operar de forma continua. [18]

Las tres etapas en las que se divide la segmentación del «ARM7TDMI» son: (Figura 2.5)

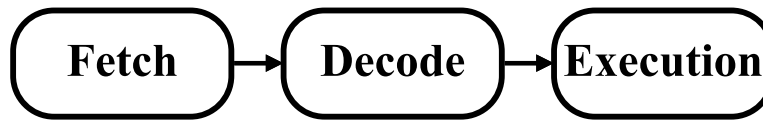


Figura 2.5: Segmentación ARM

1. Búsqueda de instrucción

Se accede a la memoria para extraer la instrucción.

2. Decodificación

Los registros utilizados son extraídos de la instrucción.

3. Ejecución

Los valores de los registros se extraen del banco de registros, se realizan las operaciones, y se almacenan los resultados en el banco de registros.

Mientras se ejecuta una instrucción, la siguiente es decodificada y una tercera es traída de memoria.

2.2.4. Memoria ARM

Se utiliza una arquitectura Von-Neumann con un único bus de 32 bits para acceder tanto a las instrucciones como a los datos.

El único tipo de instrucciones con acceso a memoria son las instrucciones de carga y almacenamiento. Puede transmitir datos de 8, 16 o 32 bits, alineados cada 1, 2 y 4 bytes respectivamente. [18]

2.3. Field-Programmable Gate Arrays (FPGA)

Las FPGAs, del inglés Field-Programmable Gate Arrays, consisten en bloques lógicos con conexiones programables para realizar diferentes diseños [22]. Figura 2.6.

Los bloques pueden ser tan simples como un transistor o tan complejos como un microprocesador. Las FPGAs comerciales suelen tener bloques basados en:

- Parejas de transistores.
- Puertas lógicas simples.
- Multiplexores.
- Look-up tables (LUT).
- Estructuras de puertas AND-OR.

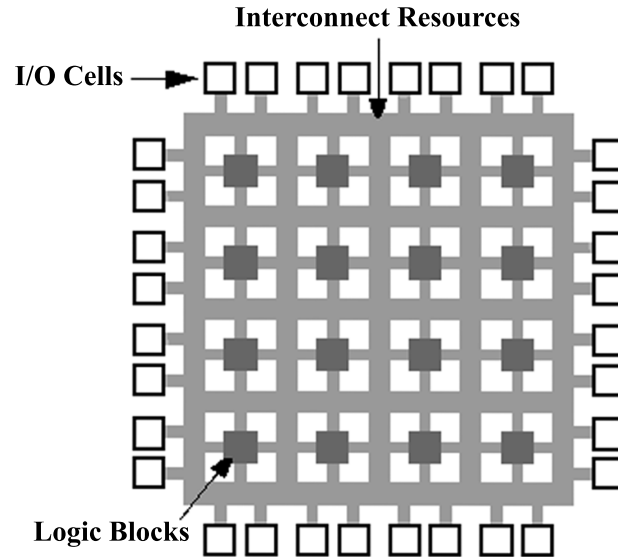


Figura 2.6: Arquitectura de una FPGA [2].

Debido a su naturaleza re-configurable, las FPGAs conllevan un mayor coste en área, retardos y consumo de energía: requieren un área 20 veces mayor, consume 10 veces más energía y trabaja 3 veces más lenta [17]. Estas desventajas son minimizadas por la ventaja de permitir que el sistema funcione de forma casi inmediata.

Las ventajas de este tipo de dispositivos residen en su gran versatilidad, flexibilidad, su alta frecuencia de trabajo, su capacidad de procesamiento en paralelo, y a su bajo precio en comparación con los «Circuitos Integrados para Aplicaciones Específicas (ASICs)»².

Es por ello que las FPGAs se utilizan en todo tipo de sectores desde el procesamiento de datos y las comunicaciones hasta el sector de la automoción, el sector militar y el sector aeroespacial. Ha sido el sector de comunicaciones, como vemos en la figura 2.7, el que más uso hacía de esta tecnología en 2008.

2.3.1. Nexys 4

Para el desarrollo de este proyecto se ha utilizado la placa de prototipado «Nexys 4» [5], figura 2.8. Ésta placa se basa en la FPGA Artix-7 de Xilinx, proporcionando a la misma acceso a memorias externas, puertos de entrada/-salida (USB, ethernet, etc), y una serie de sensores y periféricos integrados (acelerómetro, sensor de temperatura, micrófono, etc).

²Circuitos integrados hechos a la medida para un uso particular.

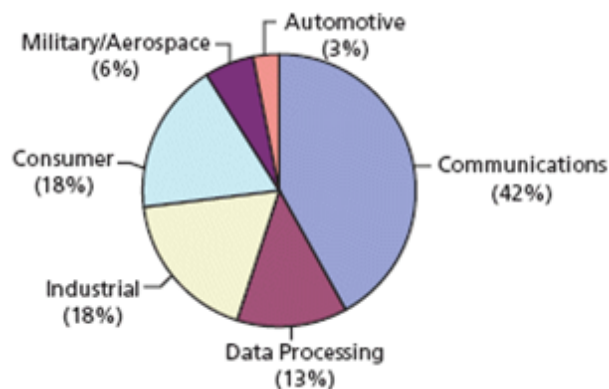


Figura 2.7: Distribución del uso de FPGAs en el año 2008.

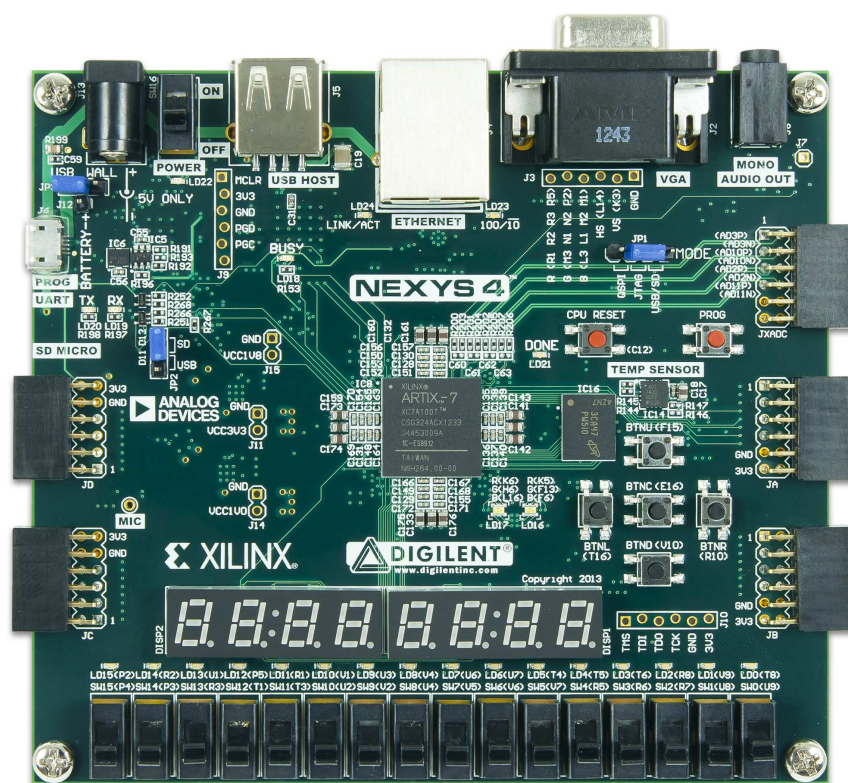


Figura 2.8: Placa de prototipado Nexys 4

La FPGA Artix-7 (XC7A100T-1CSG324C) [29], está optimizada para la lógica de altas prestaciones y ofrece más recursos y mejor rendimiento que sus predecesoras. Esta FPGA consta de:

- 15,850 bloques lógicas (cada porción contiene 6 LUTs y 8 biestables).
- 4,860 Kb de bloques RAM.
- 6 líneas de reloj.
- Velocidad interna de reloj superior a 450MHz.
- 240 bloques DSP.
- Conversor analógico-digital integrado (XADC).

Algunos periféricos incluidos en la placa son: 16 interruptores, 16 LEDs, puerto USB-UART, lector de tarjeta microSD, salida de audio, salida VGA, acelerómetro, sensor de temperatura, 16MB de memoria CellularRAM y puerto ethernet 10/100.

Capítulo 3

Introducción a los fallos y su tolerancia

3.1. Introducción a los fallos

Un fallo ocurre cuando un sistema no ha funcionado correctamente. Se pueden encontrar desde fallos en la definición de requisitos que se propagan hasta la fase de producción, hasta fallos producidos en el sistema por agentes externos, como la radiación. En un sistema electrónico pueden ocurrir fallos que se clasifican en «*soft errors*» o *fallos transitorios* y «*hard errors*» o *fallos permanentes*.

Cuando el fallo ocurrido afecta a los elementos de memoria alterando sus valores, lo que incluye tanto a los datos como a las instrucciones, se conoce como «*soft error*» o *fallo transitorio*. Sin embargo, si el fallo daña o altera el funcionamiento del chip, se conoce como «*hard error*» o *fallo permanente*.

En esta sección no se contemplan los fallos que se producen a partir de una mala implementación, únicamente se centra en los fallos producidos por agentes externos que no se pueden evitar en las fases de diseño, y que afectan al hardware, dañando sus componentes o alterando los valores de las señales con las que trabaja.

3.1.1. Causas

En general, estos fallos se conocen como «*Single-Event Effects (SEEs)*». Se deben al choque de una partícula de energía contra un elemento del circuito integrado. Figura 3.1.

Las partículas de energía pueden ser:

- **Los rayos cósmicos:** Si poseen suficiente carga pueden depositar energía suficiente para invertir un bit en un elemento de memoria, en una puerta lógica, o en una sección del circuito. Estos rayos pueden tener

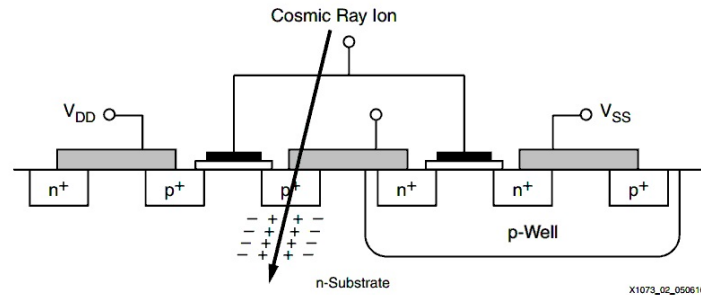


Figura 3.1: Single Event Upset en una FPGA [11].

un origen galáctico o solar.

- Los **protones y neutrones de alta energía**: Bien sean de origen radiactivo o solar, pueden provocar una reacción radiactiva ionizando elementos en el chip y provocando un SEE.

Los rayos cósmicos y las partículas solares reaccionan con la atmósfera provocando un efecto de lluvia de partículas. La atmósfera actúa a modo de filtro contra estas partículas. Este efecto se distribuye de manera diferente alrededor de la tierra debido a la densidad de la atmósfera, variando la proporción de partículas que llegan a nivel de suelo y las que quedan bloqueadas.

Como ya se adelantó en la introducción, los efectos varían según la latitud, la longitud y la altitud. Figura 3.2. Al entrar en contacto con la atmósfera las partículas colisionan contra estas y pierden energía. Cuanto menor sea la densidad, mayor será el número de partículas que llega al nivel del suelo manteniendo su energía, mayor será el número de partículas que puedan colisionar contra un chip y en consecuencia mayor la probabilidad que se produzca un fallo. Para más información sobre lluvias de partículas véase el trabajo de W.K. Melis[20].

3.1.2. Tipos de fallos

Los fallos se clasifican los fallos en dos tipos: Fallos transitorios o no destructivos, y fallos permanentes o destructivos.

3.1.2.1. Fallos Transitorios

Los *fallos transitorios*, también llamados «*soft errors*», son aquellos que cambian el estado del dispositivo o celda sin afectar a su funcionalidad.

Los principales tipos de fallos transitorios son [14]:

- **Single-Event Upset (SEU)**

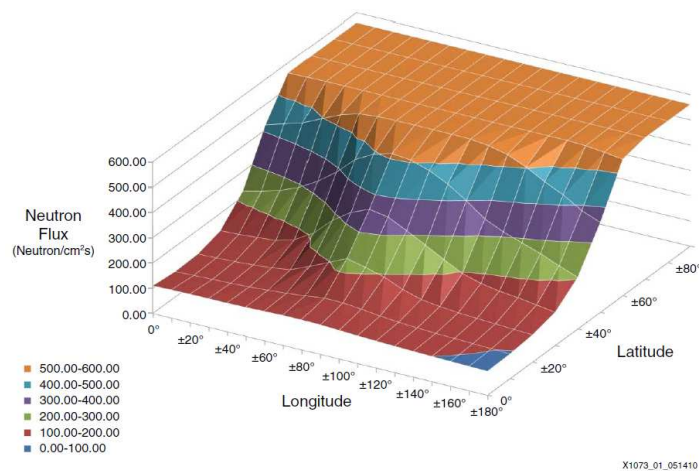


Figura 3.2: Flujo de neutrones a 40.000 pies de altitud [11].

Aquellos fallos que afectan a los elementos del chip invirtiendo su valor: memoria, celdas de memoria o registros. En un microprocesador se pueden corromper los datos del banco de registro, o los datos y las señales de control entre las etapas de segmentación. Figura 3.3a.

■ Single-Event Functional Interrupt (SEFI)

Fallos que producen una pérdida temporal de la funcionalidad del dispositivo, provocando un mal funcionamiento detectable, que no requiere reiniciar el sistema para recuperar su funcionalidad. Normalmente se asocia con un SEU en los registros de control.

■ Single-Event Transient (SET)

Picos de energía provocados por una partícula en un nodo de un circuito integrado. Pueden propagarse y almacenarse en un biestable si se produce en un flanco de reloj. Figura 3.3b.

El sistema sufre las consecuencias como un cambio de valor en un bit. Si se produce un fallo de este tipo en una celda de memoria o en un registro de un microprocesador se corromperá el dato almacenado. Si afecta a un biestable en cualquier etapa de la segmentación, puede alterar el comportamiento de la instrucción, siendo más o menos grave según el lugar donde se produzca el fallo.

3.1.2.2. Fallos Permanentes

Los *fallos permanentes* o «*hard errors*» son los que afectan a la funcionalidad del dispositivo y lo dañan permanentemente. Pueden producir cambios

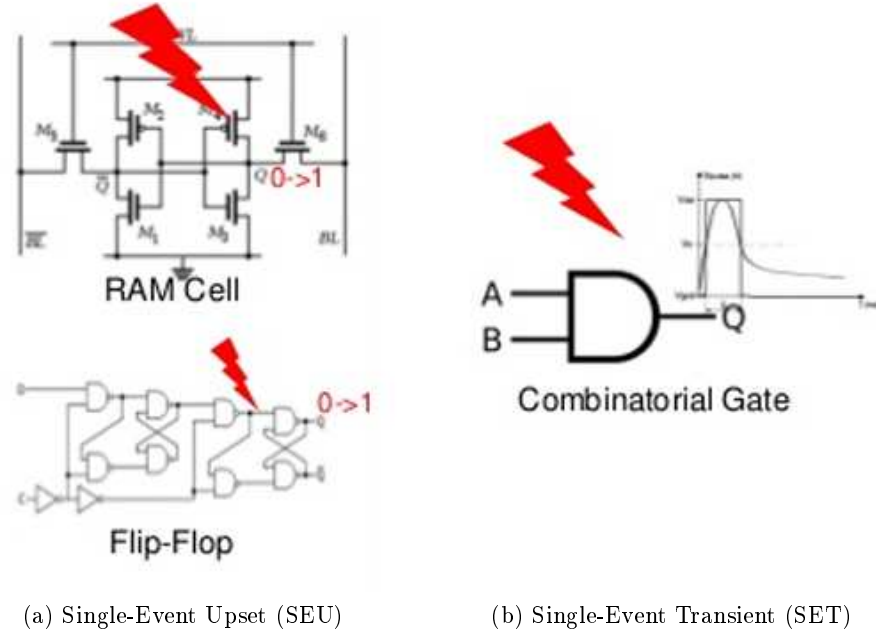


Figura 3.3: Fallos Transitorios

en el diseño que impiden el correcto funcionamiento del módulo o circuito que lo sufre. [14]

Los principales tipos de fallos permanentes son:

- **Single-Event Latch-up (SEL)**

Corto-circuito en un transistor que provoca el mal funcionamiento del mismo. En algunos casos pueden ser reparados reiniciando el sistema.

- **Single-Event Hard Errors (SHE)**

Este fallo se identifica por causar que las celdas afectadas no puedan cambiar de estado.

Existen otros tipos de fallos permanentes, *Single-Event Burnout (SEB)* y *Single-Event Gate Rupture (SEGR)*, que destruyen el transistor a nivel físico.

Los fallos permanentes, una vez detectados, únicamente pueden solucionarse sustituyendo el chip o modificando la configuración interna del propio chip. Véase el apartado 3.2.3.2.

3.2. Tolerancia a Fallos

La tolerancia a fallos se define como la capacidad de un sistema para funcionar correctamente, incluso si se produce un fallo o anomalía en el sistema.

En ocasiones se producen fallos que no llegan a propagarse por el sistema y no producen errores en su funcionamiento, algo que ocurre cuando los cambios sufridos en un sistema debidos a un fallo, se ven enmascarados. Pueden deberse a alguna de las siguientes razones:

■ Enmascarado lógico

Se evita el error en una puerta lógica, gracias a que el valor del dato no es necesario para estimar la salida. En la figura 3.4 vemos que el valor de la señal invertida es indiferente para calcular el resultado ya que el resultado de una puerta «or» es «1» siempre que una de sus entradas sea «1».

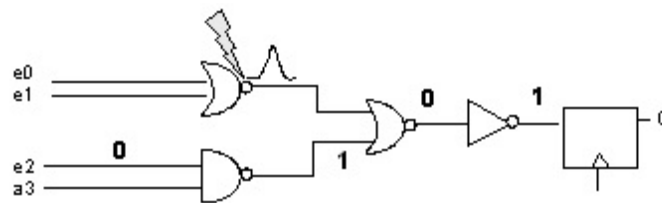


Figura 3.4: Fallo enmascarado por una puerta lógica.

■ Enmascarado eléctrico

El fallo producido pierde intensidad en el recorrido lógico y no tiene efecto al llegar al elemento de memoria donde se almacenaría. Figura 3.5.

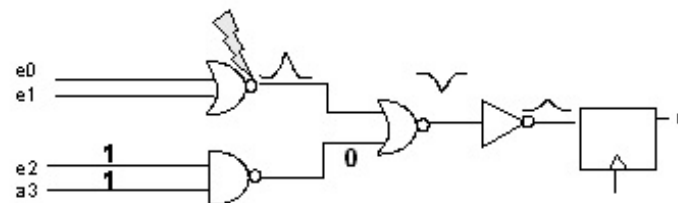


Figura 3.5: Fallo enmascarado eléctricamente.

■ Enmascarado temporal

El fallo se propaga con suficiente energía hasta el biestable, sin embargo, ocurre fuera de la ventana crítica de tiempo y la señal puede

estabilizarse a su valor correcto antes de almacenarse en el biestable. Figura 3.6.

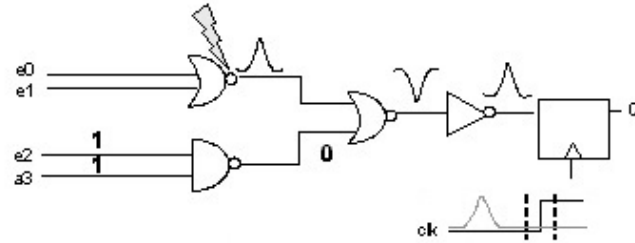


Figura 3.6: Fallo enmascarado por ventana de tiempo.

Dependiendo de la aplicación del sistema, se distinguen diferentes grados de tolerancia:

- **Tolerancia completa (fail operational)**

El sistema puede seguir funcionando sin perder funcionalidad ni prestaciones.

- **Degradación aceptable (failsoft)**

El sistema continua funcionando parcialmente hasta la reparación del fallo.

- **Parada segura (failsafe)**

El sistema se detiene en un estado seguro hasta que se repare el fallo.

La tolerancia a fallos hardware se resuelve principalmente aplicando la redundancia en una o varias de sus modalidades:

3.2.1. Redundancia en la información

La redundancia de datos se basa en mantener varias copias de todos los datos en diferentes ubicaciones junto a códigos de detección y corrección de errores. La replicación de datos consigue que la pérdida o daño de una memoria no implique la pérdida de los datos que almacena, mientras que los códigos de detección y corrección permiten comprobar los datos en busca de errores y corregir los datos si fuese necesario.

El ejemplo más claro de este tipo de tolerancia es el conocido como *conjunto redundante de discos independientes* o *redundant array of independent disks (RAID)*. Las diferentes clases de RAID proporcionan un acceso a los datos rápido y transparente para el sistema operativo [27]. Por ejemplo:

- **RAID 1:** Se basa en la utilización de discos adicionales sobre los que se realiza una copia de los datos que se están modificando.
- **RAID 5:** Reparte la información en bloques con bits de paridad, que se guardan en diferentes discos.

3.2.2. Redundancia en el tiempo

La redundancia en el tiempo es efectiva contra los fallos transitorios. Consiste en ejecutar parte de un programa o el programa completo varias veces. Los fallos transitorios, como se ha explicado anteriormente, se producen en zonas aleatorias del chip, siendo poco probable que aparezca el mismo error en el mismo lugar.

Aunque este tipo de redundancia requiere una menor cantidad de hardware y de software, obliga a ejecutar varias veces el programa, con lo que se produce una reducción en el rendimiento del sistema.

Algunas técnicas de redundancia en el tiempo se basan en «puntos de control» o «checkpoints». Consisten en almacenar los datos con los que se está trabajando cada cierto tiempo, se crea así un «punto de control». Una vez se detecta un error se recurre al último «checkpoint» en lugar de tener que reiniciar el programa completo [15].

3.2.3. Redundancia en el hardware

La redundancia hardware se basa en la inserción de módulos extra para la detección y corrección de los fallos. Aunque su objetivo es el de reducir el número de fallos que provocan errores, la inserción de módulos extra implica un aumento en la complejidad del sistema, paradójicamente, con ello aumenta la posible aparición de nuevos fallos.

La tolerancia con hardware redundante se clasifica en:

- **Tolerancia estática:** Se hace uso de varias unidades que realizan la misma función en paralelo.
- **Tolerancia dinámica:** Consiste en mantener una unidad en funcionamiento y varias de repuesto para sustituirla si fuera necesario.
- **Tolerancia híbrida:** Combinan tolerancia estática con tolerancia dinámica.

Algunas técnicas se detallan a continuación [16].

3.2.3.1. Redundancia modular

La redundancia modular consiste en replicar N veces el bloque al que se desea aplicar la tolerancia, siendo N un número impar, y a través de una

votación de mayoría de las salidas extraer el valor correcto del módulo. Al aplicar la redundancia modular es posible corregir los fallos producidos en $\frac{N}{2}$ de los módulos redundantes.

El votador de mayoría es un componente de lógica combinacional que determina el valor más repetido en sus entradas. Actúa recibiendo tanto las salidas del bloque original como de cada una de las réplicas y determinando cual es el valor más repetido. De este modo los fallos quedan enmascarados.

Este método es conocido como «*N-Modular Redundancy (NMR)*», y el uso más común de esta técnica es la «*Triple Modular Redundancy (TMR)*», con $N = 3$.

En la figura 3.7 se observa el resultado de aplicar la TMR a un bloque.

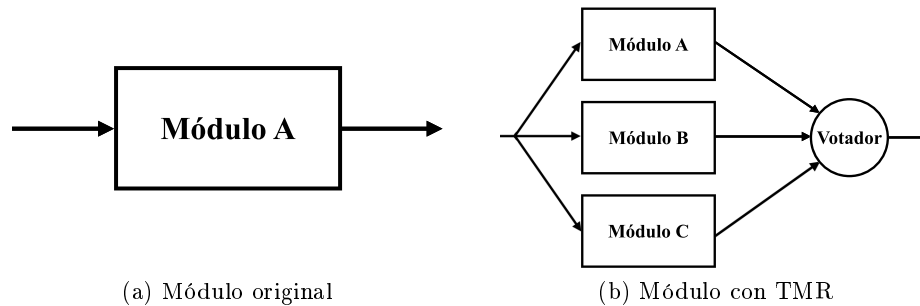


Figura 3.7: Aplicando Triple Modular Redundancy (TMR)

Esta técnica permite evitar los fallos producidos dentro de los bloques, sin embargo inserta un nuevo punto crítico. Si el votador, un circuito combinacional, se ve afectado por un SET, este puede propagarse y afectar a los siguientes bloques, generando un error en la ejecución.

3.2.3.2. Re-configuración

La re-configuración de un sistema consiste en cambiar su implementación en el momento deseado. Por ejemplo, cuando nuestro sistema empieza a fallar debido a que parte del chip se ha dañado, en vez de eliminar el chip y sustituirlo por otro, se puede configurar el mismo circuito de manera que se eviten las zonas dañadas.

En el apartado 2.3 de este mismo capítulo se han introducido las FPGAs, sistemas re-programables que permiten al diseñador re-configurar su estructura para realizar diferentes tareas, o la misma con una nueva implementación u organización de los componentes.

En la figura 3.8 podemos ver un ejemplo de re-configuración, concretamente de re-configuración en un caso de tolerancia dinámica parcial.

Hay dos formas de aplicar la re-configuración a un sistema [7]:

- **Re-configuración estática**

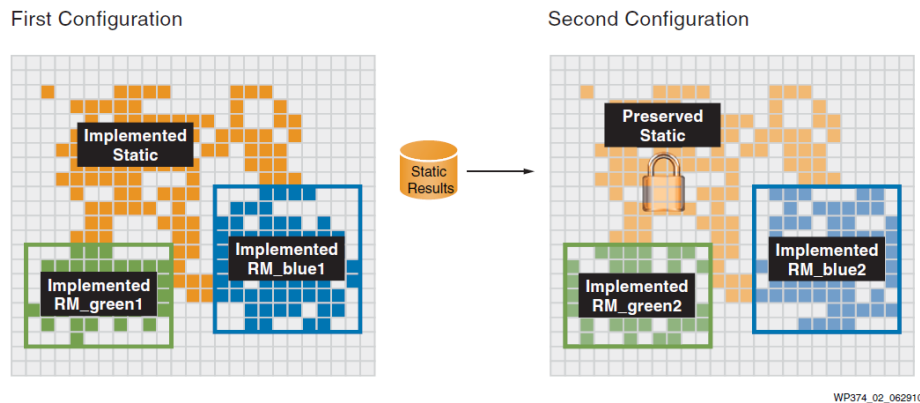


Figura 3.8: Tolerancia dinámica parcial [25].

Aquella que requiere detener el sistema completamente e iniciarlo con la nueva configuración.

- **Re-configuración dinámica**

Aquella que en tiempo de ejecución es capaz de sustituir parcial o completamente el diseño del sistema.

3.3. Tolerancia en microprocesadores

La importancia de aplicar estas técnicas a los microprocesadores viene condicionada por la utilización final que se haga de los mismos. Inicialmente, los microprocesadores nacen sin un uso específico, es precisamente su empleo final el que determina la necesidad de emplear técnicas de tolerancia. Tal puede ser el caso en misiones espaciales o como controladores de sistemas vitales.

La NASA utiliza microprocesadores para sus misiones espaciales. Los incluye en sus sistemas de asistencia a la vida y sistemas de experimentación. Tradicionalmente la NASA ha utilizado técnicas de tolerancia estática, como las técnicas de NMR por su buena confiabilidad [24]. En estos casos se hacen más necesarias por la mayor tasa de fallos que se da fuera de la atmósfera terrestre.

Algunas técnicas aplicadas a microprocesadores, sin modificar su diseño interno, requieren de un sistema externo conectado al microprocesador, que sea capaz de comprobar los valores internos, detectar los fallos y recuperar el sistema, relanzando las instrucciones o recuperando los valores correctos. Para prevenir los casos en los que los fallos se producen en el circuito de comparación y detección, se recurre principalmente a técnicas de NMR en estos circuitos, evitando tener que modificar la estructura interna del micro-

procesador, se replica el sistema de comprobación, que normalmente tendrá una implementación más simple que el propio microprocesador. [28]

Otras técnicas utilizadas sobre microprocesadores evitan modificar o añadir sistemas auxiliares para comprobar y corregir los fallos. Se centran en la «*tolerancia en el tiempo*» esto es, duplicando el programa y lanzando ambos en el mismo procesador de forma simultánea. Se propone esta técnica por considerar que las técnicas de «*tolerancia hardware*» son demasiado intrusivas para el diseño, insuficientes para cubrir los fallos lógicos o demasiado costosa para la computación de propósito general[23].

Existen procesadores tolerantes a fallos en el mercado, tales como el «LEON3FT», una implementación tolerante a fallos de la tercera versión del procesador «LEON», el «IBM S/390 G5» o el «Intel Itanium» [6].

En concreto el «LEON3FT», fue diseñado para misiones militares y espaciales. Tiene cuatro modos de tolerancia que dependen de la tecnología utilizada y de la cantidad de bloques RAM disponibles. El modo de tolerancia se selecciona a la hora de sintetizar el diseño y estos modos pueden ser: [12]

- **Biestables resistentes a la radiación o TMR.**

Se utilizan registros compuestos de biestables reforzados para resistir la influencia de la radiación o se utiliza la técnica TMR.

- **Paridad de 4-bits con reinicio.**

Se utiliza un código «*checksum*» de 1 bit por cada byte, 4 bits por palabra. Se reinicia la cola de segmentación para corregir los fallos.

- **Paridad de 8-bits sin reinicio.**

Se utiliza un código «*checksum*» de 8 bits por palabra y permite corregir 1 bit por byte, puede llegar a corregir 4 bits por palabra. La corrección se realiza sin reiniciar la cola de segmentación.

- **Código BCH de 7 bits con reinicio.**

Se utiliza un código «*BCH checksum*» de 7 bits, que permite detectar fallos en 2 bits y corrige 1 bit por palabra. La cola de segmentación se reinicia al aplicar la corrección.

El procesador «IBM S/390 G5» duplica la cola de segmentación hasta la etapa de escritura, lanzando la misma instrucción dos veces. En la etapa de escritura se comparan los resultados, en caso de discrepancia no se escribe el resultado y se reinicia la ejecución desde la instrucción fallida. La ventaja proporcionada por este método reside en que el tiempo de propagación de las señales no se ve afectado por la inserción la lógica del votador. En caso contrario, el reiniciar la cola de segmentación puede costar miles de ciclos de reloj. [6]

Por último, la implementación de Intel en el «Intel Itanium» incluye una combinación de códigos de corrección de errores y códigos de paridad en las memoria caché y TLB. [6]

Capítulo 4

Procesador

4.1. Introducción

Basado en la arquitectura de los procesadores DLX estudiados durante el grado en ingeniería de computadores [10], se ha diseñado e implementado un procesador con arquitectura RISC. Se trata de un procesador con un ancho de palabra de 32 bits y una segmentación en 5 etapas.

La implementación ha sido adaptada para poder ejecutar instrucciones del repertorio ARM. En concreto, se permite ejecutar un subconjunto del juego de instrucciones THUMB-2 que es utilizado principalmente por los procesadores de la gama ARM CORTEX M.

Para el desarrollo de este proyecto se ha utilizado la tecnología de las FPGAs, en concreto la placa «Nexys 4» que hace uso de la FPGA «Artix 7» de Xilinx.

Para la implementación se ha utilizado el lenguaje de diseño hardware VHDL junto al software «ISE Design Suite 14.4» de Xilinx y el software «ModelSim PE Student Edition» de Altera para las simulaciones. Con ello se ha conseguido probar el diseño y la implementación de forma rápida y realizar las correcciones necesarias.

4.2. Arquitectura del procesador

El microprocesador es una implementación modificada de la arquitectura DLX para permitir ejecutar instrucciones del repertorio ARM. A continuación se describen sus características completas.

4.2.1. Estructura

El procesador se compone de los siguientes elementos (figura 4.1):

- **Banco de registros:** Dispone de 16 registros (R0, R1, ..., R15) de

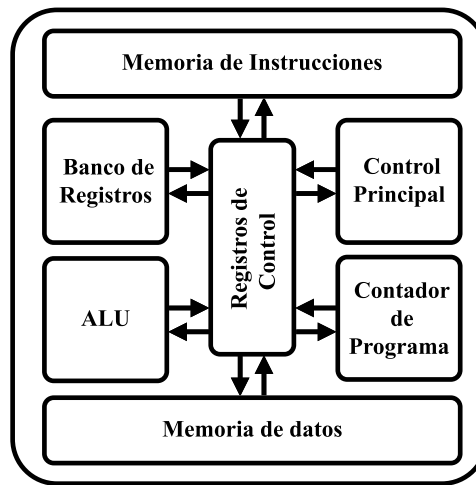


Figura 4.1: Estructura del procesador diseñado

propósito general con un tamaño de 32 bits. Estos registros se pueden utilizar tanto para guardar datos leídos de memoria como enviar los valores a memoria. Igualmente se puede trabajar con los valores que tengan almacenados ejecutando operaciones sobre ellos. El registro R15 es accesible de forma limitada puesto que el identificador de este registro se utiliza para diferenciar unas instrucciones de otras.

- **Contador de programa (PC):** Este registro especial almacena la dirección de memoria de la instrucción que debe ejecutarse a continuación. Se incrementa automáticamente en 4 cada ciclo y solo se puede alterar este mecanismo por medio de instrucciones de control.
- **Memoria de instrucciones:** Memoria que almacena el código de programa, al cual accede el procesador para analizar y ejecutar las instrucciones.
- **Memoria de datos:** Esta memoria conserva los valores de los datos que son accesibles por el procesador mediante instrucciones de carga y almacenamiento.
- **Control principal:** Analiza las instrucciones para extraer la información necesaria que permita ejecutarles adecuadamente. Esta información se propaga mediante señales de control al resto de componentes.
- **Unidad Aritmético-Lógica (ALU):** Es el componente encargado de realizar las operaciones aritméticas o lógicas sobre los operandos.
- **Registros de control:** Almacena las señales de control, y las señales internas entre las diferentes etapas de la segmentación.

4.2.2. Repertorio de instrucciones

El procesador implementado es capaz de ejecutar 3 tipos de instrucciones:

- Accesos a memoria
- Procesamiento de datos
 - a). Operaciones con dos registros
 - b). Operaciones con un registro y un inmediato
- Operaciones de control

A continuación se explican brevemente los diferentes tipos de instrucciones. Más adelante se expondrán las instrucciones con más detalle, explicando los campos de cada una.

4.2.2.1. Accesos a memoria

Las instrucciones de acceso a memoria son necesarias cuando se requiere cargar (load) un dato desde la memoria al banco de registros, o almacenar (store) el valor de un registro en la memoria.

Aunque es posible acceder a las direcciones de memoria direccionadas por media palabra. En esta implementación se está obligado a cargar valores de tamaño 4 bytes (tamaño de palabra), siendo por tanto recomendable utilizar direcciones de memoria que sean múltiplos de 4.

Para el cálculo de la dirección efectiva de carga o almacenamiento se ha implementado un único modo de direccionamiento. Registro base $R_n + \text{imm12}$ ", es decir, la dirección base se obtiene del registro R_n , y se suma un inmediato de 12 bits extraído de la instrucción.

4.2.2.2. Procesamiento de datos

Las instrucciones de procesamiento realizan cálculos aritméticos y lógicos. Se aplican sobre dos operandos y el resultado (si existe) se almacena en un registro.

Dependiendo de la instrucción los operandos pueden ser:

- **Operaciones con dos registros:**

Los datos de trabajo se extraen de dos registros.

Al utilizar el registro R15 se deben tener en cuenta ciertas restricciones ya que se utiliza para diferenciar unas operaciones de otras. Por ejemplo, si el código de operación es "0010", el registro origen R_n es R15 ("1111") entonces la operación ejecutada será la operación "MOVE". Si el registro R_n es cualquier otro, se ejecutará una "Ó lógica"(operación or).

- **Operaciones con un registro y un inmediato:**

El conjunto de operaciones con inmediato se limita a cuatro. Se permite mover un inmediato a la mitad más significativa, o a la menos significativa, de un registro. Y se permite sumar o restar un inmediato al valor de un registro.

4.2.2.3. Operaciones de control

Las operaciones de control intervienen en la ejecución normal del programa y se utilizan para modificar el valor del registro del contador de programa. Esta operación se conoce como «instrucción de salto».

Existen dos tipos de instrucciones de salto. La primera es el salto incondicional y permite sumar un entero al valor del contador de programa y almacenar el resultado en el mismo.

La segunda operación de control es el salto condicional. Previamente a un salto condicional se debe ejecutar una operación de comparación para actualizar los flags de comparación de la ALU. Los flags se comparan a la condición de salto y en caso de coincidir, se efectúa el salto. Si no se ejecuta la comparación, el estado de los flags es desconocido y el procesador se comportará de manera no controlada.

4.2.3. Segmentación

El microprocesador ha segmentado en 5 etapas, en cada una de las cuales realiza una parte fundamental en la ejecución de las instrucciones. Las etapas en las que se divide el procesador son:

1. **Búsqueda de instrucción (IF):**

La primera etapa es la encargada de cargar las instrucciones de memoria y transmitir las a la siguiente, simultáneamente se calcula la dirección de la siguiente instrucción.

2. **Decodificación de instrucción (ID):**

En la etapa de decodificación se analiza la instrucción y se obtienen los datos necesarios para realizar las operaciones correctamente.

3. **Ejecución (EX)**

En esta etapa se realizan los cálculos aritméticos o lógicos sobre los datos obtenidos del banco de registro y del circuito de extensión de signo.

4. **Acceso a Memoria (MEM)**

En la etapa de memoria se realizan intercambios de datos con la memoria principal.

5. Escritura en registros (WB)

Es la etapa final del procesador en la que se escriben los resultados calculados por la ALU o los datos cargados de memoria en el banco de registros.

4.2.4. Memoria

El diseño del procesador hereda el sistema de memoria de la arquitectura Harvard, es decir, tiene acceso a dos memorias diferentes:

- **Memoria de instrucciones:**

Memoria ROM donde se almacenan todas las instrucciones del programa a ejecutar.

- **Memoria de datos:**

Memoria RAM accesible en modo lectura y en modo escritura para almacenar los datos con los que trabaja el programa.

4.3. Implementación

La implementación del proyecto se ha realizado utilizando la herramienta «ISE Design Suite» de Xilinx y el lenguaje de diseño hardware «VHDL».

En la figura 4.2 se muestra el diseño completo del microprocesador segmentado. En color negro se muestra la ruta de datos y los componentes. En color azul se destaca la ruta de control y el módulo de control principal. Y en color rojo aparecen los registros de control.

En esta sección se describen los componentes principales del procesador descritos en la figura 4.1.

4.3.1. Banco de registros

El banco de registros es el componente de memoria con el que opera principalmente el procesador. De este obtiene los datos con los que realiza la mayoría de las operaciones.

Internamente se compone de 16 registros que almacenan valores de 32 bits. El acceso a estos se codifica en 4 bits de modo que el registro accedido con el valor «1010» es el registro «R10».

Sus señales externas son (figura 4.3):

- **Señales de entrada**

- **Registro A:** Registro origen del primer operando de la instrucción.

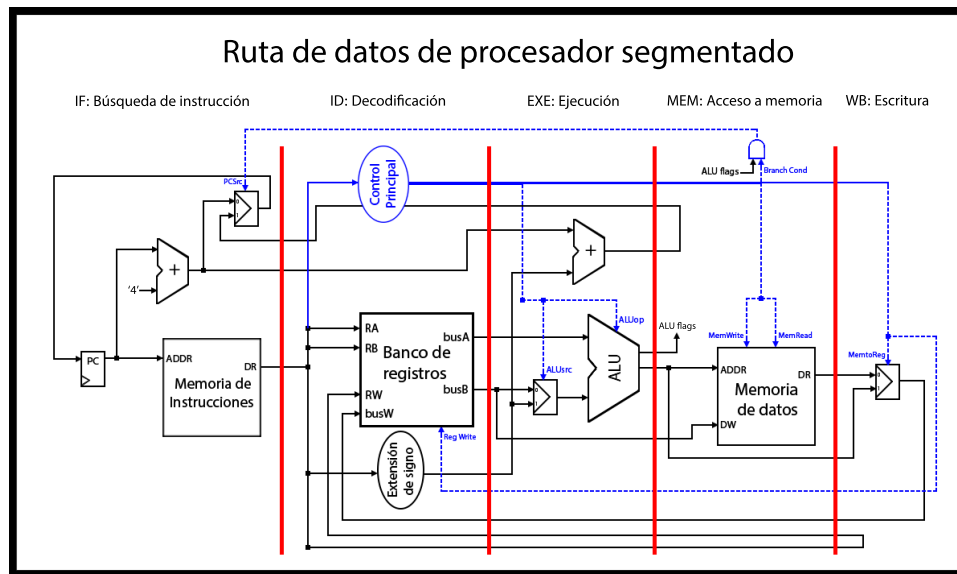


Figura 4.2: Diseño completo del procesador segmentado.

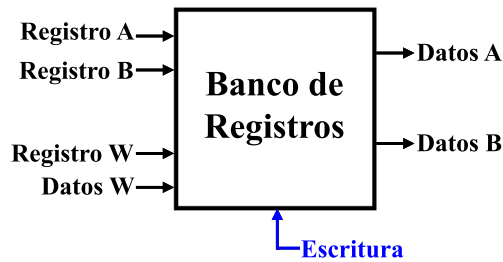


Figura 4.3: Banco de registros.

- **Registro B:** Registro origen del segundo operando de la instrucción.
- **Registro W:** Registro donde se almacenará el resultado de la instrucción.
- **Datos W:** El valor que se almacenará en el registro W.
- **Escriutura:** Habilita la escritura en el registro W.

■ Señales de salida

- **Datos A:** Valor del registro A, primer operando de la instrucción.
- **Datos B:** Valor del registro B, segundo operando de la instrucción.

4.3.2. Contador de programa

El contador de programa es un registro común de 32 bits encargado de almacenar la dirección de memoria que indica donde se encuentra la siguiente instrucción del programa. Su valor se actualiza en cada ciclo de reloj.

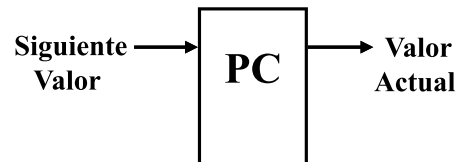


Figura 4.4: Contador de programa.

Sus señales externas son (figura 4.4):

- **Señales de entrada**

- **Siguiente Valor:** El valor de la siguiente instrucción. Puede ser calculado de forma secuencial, o por medio de un salto efectivo.

- **Señales de salida**

- **Valor actual:** Dirección origen de la instrucción que debe ejecutarse.

4.3.3. Unidad Aritmético-Lógica (ALU)

La unidad aritmético-lógica aplica ciertas operaciones sobre los datos extraídos previamente del banco de registros y de la instrucción. Así mismo, y dependiendo del resultado, puede modificar el valor de los flags de comparación que se modifican solo si la instrucción así lo requiere.

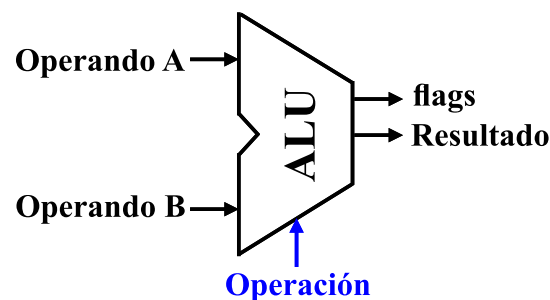


Figura 4.5: Unidad aritmético-lógica.

Las señales externas son (figura 4.5):

- **Señales de entrada**

- **Operando A:** Primer operando de la operación.
- **Operando B:** Segundo operando de la operación.
- **Operación:** Operación que debe aplicarse sobre los operadores.

- **Señales de salida**

- **flags:** Indican si el resultado de la operación cumple ciertas condiciones¹.
- **Resultado:** Valor de aplicar la operación a los operandos.

4.3.4. Control principal

El control principal (figura 4.6) del microprocesador es el encargado de analizar la instrucción y establecer las señales de control que permitirán a los módulos realizar la función adecuada. Es un módulo combinacional, por lo tanto analiza y establece los valores de las señales de control dentro de un único ciclo de reloj.

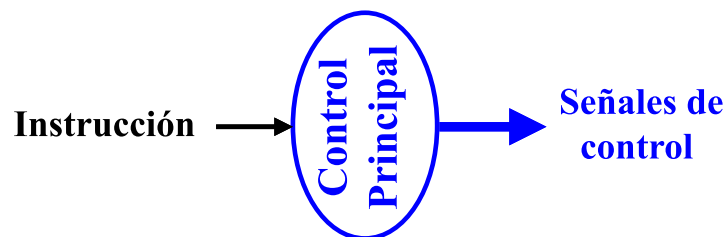


Figura 4.6: Control principal.

La instrucción es la única entrada para este módulo, sin embargo, las señales de control son varias y se transmiten de etapa a etapa de la segmentación hasta que son consumidas. A continuación se explican por el orden en el que son utilizadas (figura 4.2):

1. Ejecución (EX)

- **ALUsrc:** Establece el origen del segundo operando de la instrucción, este puede ser un registro o la propia instrucción.
- **ALUop:** Asigna a la ALU la operación que debe aplicar a los operandos.

¹Si el resultado de la operación es igual a cero se activa el flag correspondiente.

2. Acceso a Memoria (MEM)

- **Branch Cond:** Junto a los flags de la ALU, establece la señal **PCSrc** encargada de efectuar un salto en el código del programa.
- **MemWrite:** Indica al módulo «memoria de datos» si debe acceder a memoria en modo escritura².
- **MemRead:** Indica al módulo «memoria de datos» si debe acceder a memoria en modo lectura².

3. Escritura en registros (WB)

- **MemtoReg:** Establece el origen de los datos que deben almacenarse en el banco de registros, puede ser la ALU o la memoria de datos.
- **RegWrite:** Indica al banco de registros si el resultado de la instrucción debe almacenarse en un registro.

4.3.5. Registros de control

Los registros de control (figura 4.7) son todos aquellos que almacenan la información entre las etapas, se muestran como líneas rojas en la figura 4.2.

Esta colección de registros son fundamentales para que exista la segmentación. Separan las etapas de forma que los cambios en una de ellas no se propaguen y no interfieran con las demás etapas.

Para que esto no suceda se inserta una serie de registros entre, por ejemplo, los componentes de las etapas de decodificación y ejecución. Éstos registros se actualizan al final de cada ciclo de reloj, conservando los datos de forma estable para que se puedan utilizar correctamente en la siguiente etapa. Así se permite que ambas etapas trabajen de forma independientemente.

4.3.6. Memoria de instrucciones

La memoria de instrucciones es una memoria de solo lectura (ROM) Este tipo de memorias, como su nombre indica, solo permite la lectura de sus datos, y no permite que se modifiquen. Este módulo permite que se consulte el valor de la dirección en un único ciclo de reloj.

Las señales externas de esta memoria son (figura 4.8):

- **Señales de entrada**
 - **Dirección:** Dirección de la instrucción a la que se accede.
- **Señales de salida**
 - **Instrucción:** Instrucción leída de memoria.

² El modo lectura y el modo escritura en memoria son incompatibles, solo debe activarse una de estas señales.

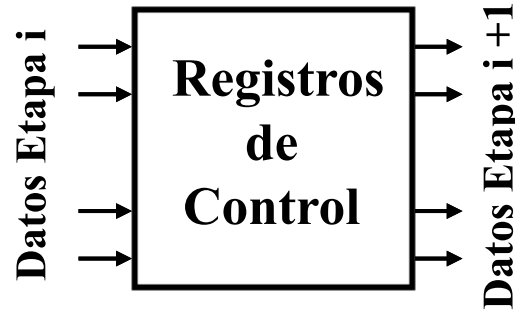


Figura 4.7: Registros de control

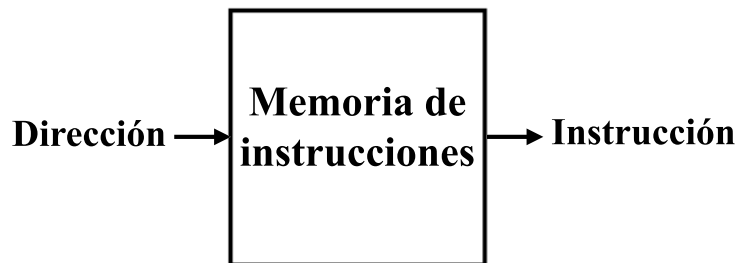


Figura 4.8: Memoria de Instrucciones.

4.3.7. Memoria de datos

El módulo de la memoria de datos es el encargado de almacenar aquellos datos que no son inmediatamente necesarios para ejecutar las instrucciones. Son de un mayor tamaño que el banco de registros y, por lo general, más lentos a la hora de transmitir la información.

Este módulo se ha implementado como un banco de registros de mayor tamaño que el módulo con el mismo nombre. Su acceso se completa en un mismo ciclo de reloj.

Las señales externas de esta memoria son (figura 4.9):

■ Señales de entrada

- **Dirección:** Dirección de acceso a memoria, ya sea en modo lectura o escritura.
- **Datos Escritura:** Datos que deben almacenarse en la memoria.
- **MemWrite:** Indica que los datos deben escribirse en memoria³.

³ El modo lectura y el modo escritura en memoria son incompatibles, solo debe estar activa una de estas señales.

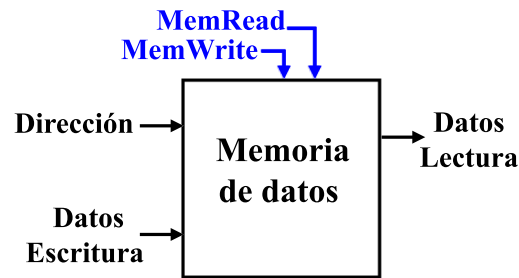


Figura 4.9: Memoria de datos.

- **MemRead:** Indica que los datos deben leerse de memoria³.
- **Señales de salida**
 - **Datos Lectura:** Datos que se han leído de la dirección indicada de memoria.

4.4. Formato de instrucciones

En esta sección se exponen el formato de las instrucciones que es capaz de ejecutar el microprocesador con todos sus campos, y el significado de estos.

4.4.1. Accesos a Memoria

Estas instrucciones permiten al microprocesador acceder a los valores almacenados en la memoria de datos así como almacenar datos en ella. Las instrucciones de carga o almacenamiento se identifican por los 7 bits más significativos de la instrucción, estos deben ser «1111100».

Se ha implementado un único tipo de instrucción de acceso a memoria. Éste, dependiendo del valor de sus campos, se utiliza para cargar de o almacenar datos en memoria.

La instrucción permite realizar transferencias de datos entre el banco de registros y la memoria de datos del microprocesador. Permite aplicar un desplazamiento de hasta 4KB al valor base del registro.

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Formato general	1	1	1	1	1	0	0									
Rn + imm12								S	1	Size	L	Rn				

Tabla 4.1: Instrucciones de acceso a memoria (bits 31..16)

Los campos de la instrucción «Rn + imm12» representados en las tablas 4.1 y 4.2 son los siguientes:

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Formato general																
Rn + imm12	Rd								imm12							

Tabla 4.2: Instrucciones de acceso a memoria (bits 15..0)

- **Extensión de signo (S):** Indica si se debe extender el signo del valor inmediato (S=1).
- **Tamaño (Size):** Indica el tamaño del valor que debe cargar de o almacenar en memoria. Puede cargar datos de tamaño 1, 2 o 4 bytes. No se utiliza.
- **Cargar/Almacenar (L):** Este bit indica si la operación debe leer un dato de memoria (L=1) o escribirlo (L=0).
- **Dirección (Rn):** Indica el registro con la dirección base de acceso a memoria.
- **Dato (Rt):** Indica el registro dónde se debe almacenar el dato en caso de carga, o el registro cd dónde debe extraerse el dato en caso de almacenamiento.
- **Desplazamiento (imm12):** Es el desplazamiento que debe aplicarse a la dirección base para obtener la dirección efectiva.

4.4.2. Procesamiento de datos

Las operaciones se distinguen según el tipo de operandos que se apliquen. El operando A siempre es obtenido de un registro. Mientras que el operando B puede ser el valor de un segundo registro o puede formar parte de la instrucción.

4.4.2.1. Operaciones con dos registros

Las operaciones que hacen uso de dos registros son aritméticas (sumar, restar y mover), lógicas (and, or y or exclusiva) y de comparación que activan diferentes flags (Negativo, Cero). En el caso de una comparación no se modifican los registros. Las instrucciones de operación con dos registros se identifican por los 7 bits más significativos. Éstos deben ser «1110101».

Se ha implementado un único tipo de instrucción de procesamiento con dos registros. Dependiendo del valor de sus campos se aplicará una operación u otra sobre los operandos.

Los campos de la instrucción «Data Processing» representados en las tablas 4.3 y 4.4 son los siguientes:

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Formato general	1	1	1	0	1	0	1									
Data Processing								OP				S	Rn			

Tabla 4.3: Instrucciones de procesamiento de datos con dos registros (bits 31..16)

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Formato general																
Data Processing	SBZ	imm3			Rd				imm2		type	Rm				

Tabla 4.4: Instrucciones de procesamiento de datos con dos registros (bits 15..0)

- **Código de operación (OP):** Indica la operación que debe realizarse en la fase de ejecución sobre los operandos.
- **Activar flags (S):** Indica si deben activarse los flags de salto al ejecutar la operación.
- **Registro origen A (Rn):** Indica el registro origen del primer operando.
- **Should be Zero (SBZ):** Este campo debe tener un valor de 0.
- **Inmediato (imm3:imm2):** Indica el desplazamiento que debe aplicarse al segundo operando. No se utiliza.
- **Registro destino (Rd):** Indica el registro destino donde se almacenará el resultado de la operación.
- **Tipo de desplazamiento (type):** Indica el tipo de desplazamiento aplicado. No se utiliza.
- **Registro origen B (Rm):** Indica el registro origen del segundo operando.

Las operaciones implementadas junto con su respectiva codificación se muestran en la tabla 4.5.

4.4.2.2. Operaciones con un registro y un inmediato

Las operaciones que hacen uso de un registro y un inmediato únicamente pueden ser aritméticas (sumar, restar y mover). Estas instrucciones se dividen en dos tipos según el tamaño del inmediato utilizado. Para identificar este tipo de instrucciones se utilizan los 5 bits más significativos, que deben ser «11110» y el bit 15 que debe valer «0».

Operación	Código	Restricciones
ADD	1 0 0 0	(Rd==«1111»,S==1)
AND	0 0 0 0	
CMP	1 1 0 1	
EOR	0 1 0 0	
MOV	0 0 1 0	(Rn==«1111»)
ORR	0 0 1 0	
SUB	1 1 0 1	

Tabla 4.5: Operaciones con dos registros

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Formato general	1	1	1	1	0											
Add, Subtract, plain 12-bit immediate						i	1	0	OP	0		OP2			Rn	
Move, plain 16-bit immediate						i	1	0	OP	1		OP2			imm4	

Tabla 4.6: Instrucciones de procesamiento de datos con un registro y un inmediato (bits 31..16)

Los campos de las instrucciones con inmediato, representados en las tablas 4.6 y 4.7 son:

- **«Add, Subtract, plain 12-bit immediate»**
 - **Código de operación (OP:OP2):** Indica la operación que debe aplicarse en la fase de ejecución sobre los operandos.
 - **Registro origen (Rn):** Indica el registro origen del primer operando.
 - **Inmediato (i:imm3:imm8):** Contiene el inmediato que se utiliza como segundo operando.
 - **Registro destino (Rd):** Indica el registro destino donde se almacenará el resultado de la operación.
- **«Move, plain 16-bit immediate»**
 - **Código de operación (OP:OP2):** Indica la operación que debe aplicarse en la fase de ejecución sobre los operandos.
 - **Inmediato (imm4:i:imm3:imm8):** Contiene el inmediato que se utiliza como segundo operando. Utilizado en las operaciones mover.
 - **Registro destino (Rd):** Indica el registro destino donde se almacenará el resultado de la operación.

Las operaciones implementadas junto con su respectiva codificación se muestran en la tabla 4.8.

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Formato general	0															
Add, Subtract, plain 12-bit immediate		imm3			Rd			imm8								
Move, plain 16-bit immediate		imm3			Rd			imm8								

Tabla 4.7: Instrucciones de procesamiento de datos con un registro y un inmediato (bits 15..0)

Operación	Código
ADD	0 0 0
SUB	1 1 0
MOVT	1 0 0
MOV	0 0 0

Tabla 4.8: Operaciones con un registro y un inmediato

4.4.3. Operaciones de control

También conocidas como instrucciones de salto, estas instrucciones son aquellas capaces de alterar el contador de programa. Para identificar este tipo de instrucciones se utilizan los 5 bits más significativos, que deben ser «11110» y el bit 15 que debe tener un valor de «1».

Se han implementado dos tipos de instrucciones de salto: salto incondicional, y salto condicional. El salto incondicional se realiza siempre que esté presente la instrucción. La operación de salto condicional sólo se efectúa cuando coinciden las condiciones de la instrucción con los flags previamente calculados de la ALU.

En el caso del salto incondicional se permite realizar un salto de 16MB por el código. El salto condicional, debido a necesitar un campo que indique la condición, puede realizar un salto de 1MB.

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Formato general	1	1	1	1	0											
Branch						S	offset[21:12]									
Conditional Branch						S	cond	offset[17:12]								

Tabla 4.9: Instrucciones de control (bits 31..16)

Los campos de las instrucciones de control representados en las tablas 4.9 y 4.10 son:

- **Salto (Branch)**

- **Extensión de signo (S):** Indica si se debe extender el signo del desplazamiento (S=1).
- **Desplazamiento (offset):** Contiene el inmediato que se suma al registro PC para calcular la dirección efectiva del salto. Los

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Formato general	1															
Branch		0	I1	1	I2	offset[11:1]										
Conditional Branch		0	J1	0	J2	offset[11:1]										

Tabla 4.10: Instrucciones de control (bits 15..0)

campos «I1» e «I2» son respectivamente los bits 23 y 22 del desplazamiento.

■ **Salto condicional (Conditional Branch)**

- **Extensión de signo (S):** Indica si se debe extender el signo del desplazamiento (S=1).
- **Condición de salto (cond):** Indica la condición necesaria para que el salto deba realizarse.
- **Desplazamiento (offset):** Contiene el inmediato que se suma al registro PC para calcular la dirección efectiva del salto. Los campos «J1» e «J2» son respectivamente los bits 19 y 18 del desplazamiento.

Capítulo 5

Aplicando tolerancia a fallos

5.1. Introducción

Los fallos más comunes en los sistemas electrónicos son los conocidos como fallos transitorios, estos no dañan el sistema de forma permanente pero provocan un cambio de valor en los elementos de memoria (SEU) o interferencias en las conexiones internas (SET), que si no se estabilizan a tiempo pueden llegar a propagarse hasta una celda de memoria y almacenarse provocando el mismo efecto que un SEU. Con esta información se ha implementado un método para paliar y reducir los efectos tanto de los SEU como de los SET.

5.2. Redundancia modular

La técnica utilizada para paliar los fallos transitorios ha sido la redundancia modular (NMR) con un valor de $N = 3$, conocida como redundancia modular triple (TMR). Consiste en triplicar cada componente de memoria y conectar «votadores» a sus salidas. Los «votadores» permiten enmascarar una cantidad de fallos igual a $\frac{N}{2}$. En nuestro caso significa que el sistema podrá tolerar un fallo en cada conjunto de módulos triplicados.

Para proporcionar al sistema tolerancia frente a los SEU al sistema, se han sustituido todos los biestables por un conjunto compuesto por tres biestables más un votador. Si el componente inicial era el representado en la figura 5.1a, aplicando lo anterior obtendremos el conjunto representado en la figura 5.1b.

5.3. El Votador

El votador es un circuito combinatorial, y su único objetivo es filtrar los valores de entrada, para dar un valor de salida igual al de la mayoría de sus entradas. Esta propiedad hace del votador el componente principal utilizado

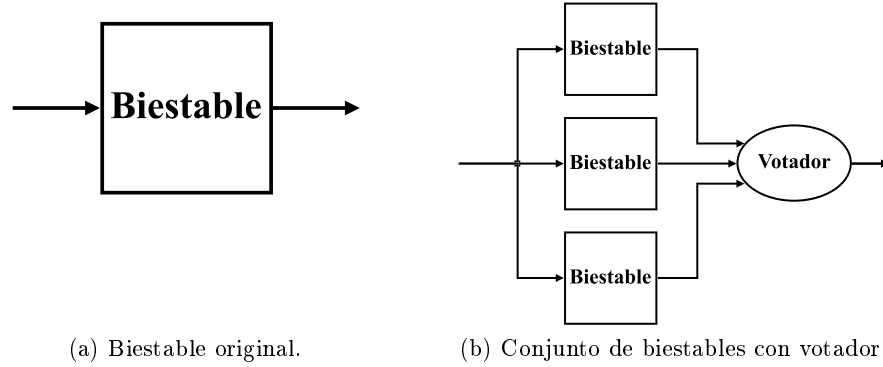


Figura 5.1: Sustitución de biestable.

para otorgar la capacidad de tolerar los fallos transitorios de tipo SEU al sistema.

Como se vé en la tabla 5.1, el valor de salida de un votador es igual al valor que más se repite en sus entradas. Con este método puede enmascarse un fallo en cualquiera de los módulos que alimentan sus entradas.

Entradas			Salida
A	B	C	Z
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Tabla 5.1: Tabla de verdad del votador

La implementación del votador puede realizarse de diferentes formas siempre que cumpla con las restricciones de la tabla 5.1. El votador de este proyecto se ha diseñado siguiendo la fórmula lógica 5.1, utilizando 4 puertas lógicas: 3 puertas «AND» de dos entradas y 1 puerta «OR» de tres entradas. Como se puede observar en la figura 5.2, se introduce únicamente un retardo de 2 puertas lógicas, haciendo que las puertas lógicas «AND» funcionen en paralelo.

$$Z = (A * B) + (B * C) + (A * C) \quad (5.1)$$

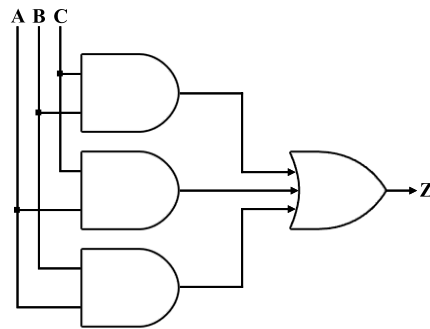


Figura 5.2: Diseño de votador con puertas lógicas

Para exponer cómo funciona el votador se muestran las figuras 5.3a y 5.3b. En la figura de la izquierda observamos cómo la entrada B es diferente al resto, eso quiere decir que el módulo origen de esa señal ha sufrido un fallo. Se observa cómo tal fallo queda enmascarado por las puertas «AND» por su funcionalidad ($0 * 1 = 1 * 0 = 0$). En el caso de la figura derecha, el fallo ocurre en la entrada C, en este caso queda enmascarado por la puerta «OR» ($0 + 0 + 1 = 1$).

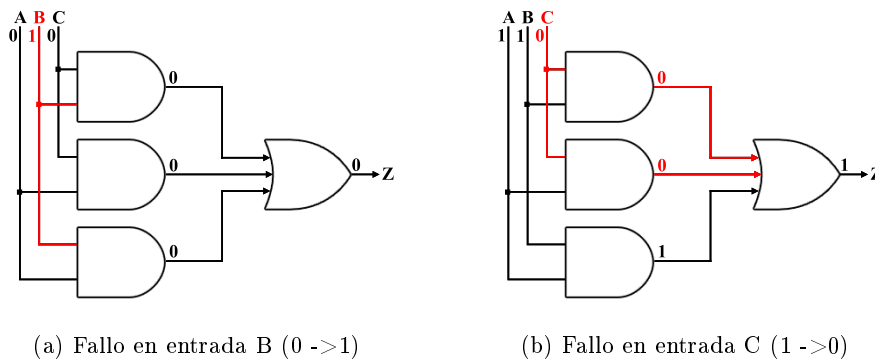


Figura 5.3: Ejemplos de fallos en entradas.

5.3.1. Implementación

La implementación de este módulo se divide en dos secciones:

■ Registros triplicados

Al aplicar la redundancia tipo TMR a los registros, estos deben triplicarse. Para ello se define una constante «N_Tolerancia» para establecer el número de replicas que se desea implementar, en nuestro

caso este número es 3. Después se declara un conjunto de tres registros de un tamaño variable determinado por una variable. Esto facilita utilizar el componente para datos tanto de tamaño 32 como de tamaño 1. A continuación se declara un proceso por el cual se actualizan los tres registros al mismo tiempo y con los mismos datos.

```

constant N_Tolerancia : integer := 3;
type tipo_conjunto_registros is array (0 to N_Tolerancia
    -1) of STD_LOGIC_VECTOR (tamaño-1 downto 0);
signal registros : tipo_conjunto_registros;

[... ]

p_regs: process (clk, rst)
begin
    if rst='0' then
        for i in 0 to N_Tolerancia-1 loop
            registros(i) <= (others=>'0');
        end loop;
    elsif rising_edge(clk) then
        for i in 0 to N_Tolerancia-1 loop
            registros(i) <= dato_entrada;
        end loop;
    end if;
end process;

```

■ Votador

Aplicando la fórmula 5.1 a los registros triplicados, el votador obtiene el valor con un mayor número de repeticiones.

```

dato_salida <= (registros(0) and registros(1)) or
    (registros(0) and registros(2)) or
    (registros(1) and registros(2));

```

5.4. Aplicación

La segmentación del procesador requiere una cantidad importante de biestables para almacenar todos los datos que deben transmitirse de una etapa a la siguiente. Debido al enorme aumento del número de lugares críticos donde puede ocurrir un SEU, se ha visto necesario aplicar la técnica TMR especialmente en esta parte del microprocesador.

Esta técnica se ha aplicado a todos los registros de control para evitar que un fallo no controlado produzca un error en medio de la ejecución de una instrucción en cualquiera de sus etapas. Por ejemplo: cambiando la dirección de un salto, cambiando el dato que debe almacenarse en memoria o habilitando la escritura en registro cuando no debería almacenarse el resultado de la instrucción.

Sin incluir las memorias de datos e instrucciones que normalmente son externas, el componente «registros de control» es el mayor elemento de memoria dentro del microprocesador. Un fallo en este componente altera de forma imprevisible la ejecución de las instrucciones que estén en ejecución dentro del procesador en ese momento, evitar este problema es el objetivo de este trabajo y por ello nos centraremos en modificar este componente para ofrecer un mayor grado de confiabilidad. En su caso, también se podría aplicar este método a otros componentes como el banco de registros y el registro «contador de programa».

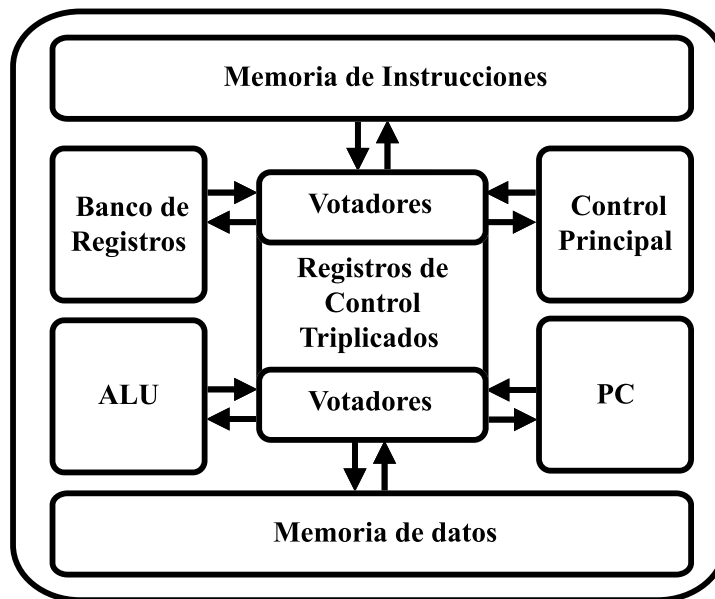


Figura 5.4: Procesador con TMR en los registros de control.

El resultado de aplicar este método sobre los registros de control se puede observar en la figura 5.4. Obtenemos un componente de mayor tamaño, pero que proporciona una mayor fiabilidad en que las instrucciones se ejecutarán correctamente.

Capítulo 6

Comprobación Funcional

6.1. Introducción

La herramienta software «PlanAhead» de Xilinx es capaz de sintetizar e implementar el diseño VHDL. Se utilizará esta herramienta para analizar los cambios que ocurren en el procesador al insertar los módulos de tolerancia a fallos.

Para realizar las simulaciones de ejecución de programas sobre las implementaciones del procesador se utilizará el software "ModelSim PE Student Edition 10.4". A continuación se analizan ambos diseños.

6.2. Programa de control

Para probar el funcionamiento del microprocesador se han implementado las memorias de datos e instrucciones y se han utilizado las mismas en ambos diseños del procesador:

- La **memoria de datos** se ha implementado como una memoria RAM de 128 bytes capaz de almacenar 32 palabras.
- La **memoria de instrucciones** contiene un programa sencillo de 20 líneas de código THUMB-2 en el que se realiza una multiplicación mediante un bucle de sumas.

Las pruebas consistirán en la ejecución del programa (figura 6.1) un total de 4 veces:

1. La primera ejecución se realizará sobre el procesador sin tolerancia a fallos y sin inserción de fallos. Este caso se considera de control.
2. La segunda ejecución será realizada sobre el procesador sin tolerancia a fallos pero insertando fallos.

3. La tercera prueba se ejecutará sobre el procesador con tolerancia a fallos, sin inserción de fallos. Con esto se quiere probar que se han introducido satisfactoriamente los componentes tolerantes a fallos sin modificar la funcionalidad.
4. La cuarta prueba se volverá a ejecutar en el procesador tolerante a fallos, pero insertando fallos. Así probaremos que insertando los mismos fallos que en la segunda prueba, se vuelve a ejecutar el programa satisfactoriamente.

```

1  LDR R2, R0, #5
2  MOV R1, #25
3  MOV R3, #0
4  MOV R5, #1
5  MOV R4, R2
6  NOP
7  NOP
8  NOP
9  CMP R4, R0
10 BEQ #24
11 NOP
12 NOP
13 NOP
14 ADD R3, R3, R1
15 SUB R4, R4, R5
16 B #-32
17 NOP
18 NOP
19 NOP
20 STR R0, R3, #0

```

Figura 6.1: Código Thumb-2 de programa de pruebas.

El código de la figura 6.1 es el cálculo de multiplicar 25 por el dato almacenado en la dirección 5 de memoria (en este caso 5) para después almacenar el resultado en la dirección 0 de memoria. En primer lugar se carga el dato de memoria a un registro, $R2 = 5$. A continuación se inicializan las variables que ayudarán a ejecutar el bucle: R1 con el valor 25, R3 se inicializa a 0 y acumulará el resultado de la ejecución, R5 conserva la constante 1, R0 vale 0 y R4 contará cuantas iteraciones quedan por ejecutar el bucle. En pseudo-código estas acciones se traducen en la figura 6.2.

6.3. Fallos introducidos

Con la intención de probar la fiabilidad del sistema, se insertan ciertos fallos en los componentes que deseamos poner a prueba, en este caso en los registros de control que almacenan los datos entre etapas.

```

1  R1 = 25;
2  R2 = Memoria(R0+5);
3  R3 = 0;
4  R4 = R2;
5  R5 = 1;
6  Si R4 = 0:
7      saltar a "instruccion 12";
8  En otro caso:
9      R3 = R3 + R1;
10     R4 = R4 - R5;
11     saltar a "instruccion 6";
12 Memoria(R0+0) = R3;

```

Figura 6.2: Pseudo-código de programa de pruebas.

1. **Fallo A:** Se introduce un cambio de valor del bus del dato leído de memoria de la etapa de escritura en registro. El bit 0 de este registro se invierte resultando en 0.
2. **Fallo B:** Se activa la señal de control «MemWrite» en la etapa «Memory».
3. **Fallo C:** Se modifica el valor del operando B en la instrucción de comparación entre los registros R4 y R0.

6.4. Procesador estándar

En esta sección se exponen los recursos utilizados por el procesador estándar, sin tolerancia a fallos, después de sintetizar e implementar el diseño. Como primer paso de la prueba funcional se exponen los resultados de las simulaciones de la prueba de control y la prueba de inserción de fallos.

6.4.1. Recursos

Los informes de las herramientas de síntesis e implementación de Xilinx nos proporcionan una serie de informes entre los que se encuentran los recursos consumidos por el diseño, el mapeado realizado sobre la FPGA y las limitaciones de tiempos. A continuación se presentan los datos más relevantes:

6.4.1.1. Map report

Design Summary	
Number of errors:	0
Number of warnings:	0

Slice Logic Utilization:		
Number of Slice Registers:	1,952	out of
126,800	1%	
Number used as Flip Flops:	1,952	
Number used as Latches:	0	
Number used as Latch-thrus:	0	
Number used as AND/OR logics:	0	
Number of Slice LUTs:	1,032	out of
63,400	1%	
Number used as logic:	1,015	out of
63,400	1%	
Number using O6 output only:	873	
Number using O5 output only:	28	
Number using O5 and O6:	114	
Number used as ROM:	0	
Number used as Memory:	0	out of
19,000	0%	
Number used exclusively as route-thrus:	17	
Number with same-slice register load:	16	
Number with same-slice carry load:	1	
Number with other load:	0	
Slice Logic Distribution:		
Number of occupied Slices:	603	out of
15,850	3%	
Number of LUT Flip Flop pairs used:	2,362	
Number with an unused Flip Flop:	492	out of
2,362	20%	
Number with an unused LUT:	1,330	out of
2,362	56%	
Number of fully used LUT-FF pairs:	540	out of
2,362	22%	
Number of unique control sets:	50	
Number of slice register sites lost		
to control set restrictions:	8	out of
126,800	1%	

6.4.1.2. Timing report

Timing summary:	
Timing errors: 0 Score: 0 (Setup/Max: 0, Hold: 0)	
Constraints cover 27520 paths, 0 nets, and 8309 connections	
Design statistics:	
Minimum period:	6.435 ns{1} (Maximum frequency: 155.400MHz)

6.4.2. Ejecución de control

Se procede a realizar la primera prueba, o prueba de control, que consiste en ejecutar el programa explicado en la sección 6.2 sobre el procesador estándar. La simulación se ha realizado ejecutando el código adjunto en el apéndice A.

Los cálculos realizados por el procesador han sido monitorizados y comprobados. Se han controlado los registros de trabajo R0, R1, R2, R3, R4 y R5 para monitorizar que han sido actualizados en los momentos adecuados y con los valores esperados. Igualmente, se ha comprobado el resultado obtenido en la dirección 0 de la memoria de datos. En nuestro caso es 125.

6.4.3. Inserción de fallos

Realizada la prueba de control, se procede con la segunda prueba, en este caso con la inserción de fallos. Con el objetivo de comprobar si alteran el comportamiento del procesador, se han insertado 3 fallos en los registros de control entre etapas. A continuación se describen los fallos insertados y sus consecuencias. El resultado completo de la simulación se incluye en la sección C.1 del apéndice C.

Además de las señales monitorizadas en la ejecución de control se han analizado las señales en donde se han insertado los fallos, así como los registros y direcciones de memoria donde se espera que provoquen consecuencias haciendo así efectivo un error en la ejecución del programa.

El simulador nos facilita la siguiente información alertando que los datos almacenados en los registros no son los esperados después de ejecutar las instrucciones:

```

1 # ** Error: ERROR: LDR R2, R0, #5
2 #   Time: 150 ns   Iteration: 0   Instance: /tb_ejecucion_fallos
3 # ** Error: ERROR: MOV R4, R2
4 #   Time: 190 ns   Iteration: 0   Instance: /tb_ejecucion_fallos
5 # ** Error: ERROR: SUB R4, R4, R5(4)
6 #   Time: 300 ns   Iteration: 0   Instance: /tb_ejecucion_fallos
7 # ** Error: ERROR: SUB R4, R4, R5(3)
8 #   Time: 410 ns   Iteration: 0   Instance: /tb_ejecucion_fallos
9 # ** Error: ERROR: SUB R4, R4, R5(2)
10 #   Time: 520 ns   Iteration: 0   Instance: /tb_ejecucion_fallos
11 # ** Error: ERROR: PC 204
12 #   Time: 620 ns   Iteration: 0   Instance: /tb_ejecucion_fallos
13 # ** Error: ERROR: ADD R3, R3, R1(100)
14 #   Time: 620 ns   Iteration: 0   Instance: /tb_ejecucion_fallos
15 # ** Error: ERROR: PC 208
16 #   Time: 630 ns   Iteration: 0   Instance: /tb_ejecucion_fallos
17 # ** Error: ERROR: PC 248
18 #   Time: 730 ns   Iteration: 0   Instance: /tb_ejecucion_fallos
19 # ** Error: ERROR: ADD R3, R3, R1(125)
20 #   Time: 730 ns   Iteration: 0   Instance: /tb_ejecucion_fallos
21 # ** Error: ERROR: PC 252

```

```

22 #      Time: 740 ns   Iteration: 0   Instance: /tb_ejecucion_fallos
23 # ** Error: ERROR: SUB R4, R4, R5(0)
24 #      Time: 740 ns   Iteration: 0   Instance: /tb_ejecucion_fallos
25 # ** Error: ERROR: PC 296
26 #      Time: 860 ns   Iteration: 0   Instance: /tb_ejecucion_fallos
27 # ** Error: ERROR: STR R3, R0, #0
28 #      Time: 860 ns   Iteration: 0   Instance: /tb_ejecucion_fallos

```

6.5. Procesador tolerante a fallos

Siguiendo el mismo proceso anterior, en esta sección se exponen los recursos utilizados por el procesador tolerante a fallos después de sintetizar e implementar el diseño. De igual modo se exponen los resultados obtenidos al realizar las simulaciones de la prueba de control y la prueba de inserción de fallos sobre este microprocesador.

6.5.1. Recursos

Los informes de las herramientas de síntesis e implementación de Xilinx nos proporcionan una serie de informes entre los que se encuentran los recursos consumidos por el diseño, el mapeado realizado sobre la FPGA y las limitaciones de tiempos. A continuación se presentan los datos más relevantes:

6.5.1.1. Map report

```

1 Design Summary
2 -----
3 Number of errors:      0
4 Number of warnings:   6
5 Slice Logic Utilization:
6   Number of Slice Registers:      2,798 out of
6       126,800      2%
7   Number used as Flip Flops:      2,798
8   Number used as Latches:         0
9   Number used as Latch-thrus:     0
10  Number used as AND/OR logics:    0
11 Number of Slice LUTs:      1,539 out of
11     63,400      2%
12   Number used as logic:          1,464 out of
12     63,400      2%
13   Number using O6 output only:    1,328
14   Number using O5 output only:    28
15   Number using O5 and O6:         108
16   Number used as ROM:              0
17   Number used as Memory:           0 out of
17     19,000      0%
18   Number used exclusively as route-thrus: 75

```

19	Number with same-slice register load:	74
20	Number with same-slice carry load:	1
21	Number with other load:	0
22		
23	Slice Logic Distribution:	
24	Number of occupied Slices:	858 out of
	15,850 5%	
25	Number of LUT Flip Flop pairs used:	3,146
26	Number with an unused Flip Flop:	844 out of
	3,146 26%	
27	Number with an unused LUT:	1,607 out of
	3,146 51%	
28	Number of fully used LUT-FF pairs:	695 out of
	3,146 22%	
29	Number of unique control sets:	71
30	Number of slice register sites lost	
31	to control set restrictions:	42 out of
	126,800 1%	

6.5.1.2. Timing report

1	Timing summary:
2	
3	
4	Timing errors: 0 Score: 0 (Setup/Max: 0, Hold: 0)
5	
6	Constraints cover 163797 paths, 0 nets, and 10410 connections
7	
8	Design statistics:
9	Minimum period: 6.819ns{1} (Maximum frequency: 146.649MHz)
)

6.5.2. Ejecución de control

Se ha ejecutado el programa de control sobre el procesador tolerante a fallos monitorizandose los registros relevantes para la ejecución del programa así como las direcciones de memoria accedidas por el mismo.

Durante la ejecución se comprueba que la evolución de los registros es la esperada y al finalizar la ejecución del programa se vuelve a comprobar el valor almacenado en la dirección 0 de memoria. Los resultados de la simulación se incluyen en el apartado C.3 del apéndice C.

6.5.3. Inserción de fallos

Se ha ejecutado la simulación con inserción de fallos, esta simulación se puede consultar en la sección C.4 del apéndice C.

Junto a los registros monitorizados en la prueba de control se presentan los registros donde se insertan los fallos y así como los registros y las direc-

ciones de memoria que se espera se verán afectadas por los fallos si estos no se toleran.

Capítulo 7

Análisis de los resultados

7.1. Introducción

El presente capítulo se inicia comparando las implementaciones de ambos procesadores, estándar y tolerante a fallos, exponiendo sus principales diferencias. Posteriormente se analizan y comparan los resultados obtenidos al ejecutar las simulaciones sobre las dos implementaciones del microprocesador.

7.2. Comparativa de procesadores

El procesador estándar requiere un total de 1,952 registros para ser utilizados como biestables, un lo que supone aproximadamente el 1,5 del total disponible. De los utilizados 420 se dedican a los registros de control y el resto se utilizan para el banco de registros así como para las memorias de instrucciones y de datos.

En cuanto a la lógica combinacional del microprocesador se utilizan un total de 1,015 LUTs para los elementos como la ALU, el control principal y los sumadores entre otros. Requiere un 1 % de las LUTs disponibles en la FPGA.

Los informes de restricción de tiempo indican que el diseño requiere un periodo mínimo de reloj de 6.435 ns, alcanzando una frecuencia máxima de 155.400 MHz.

El procesador tolerante a fallos se deriva del procesador estándar aplicando la técnica TMR sobre los registros de control entre etapas.

Al incluir una mayor lógica, el procesador tolerante a fallos requiere de 2,798 registros (poco más del 2 %) de los cuales 1,260 se utilizan para los registros de control. Como se ha diseñado el procesador aplicando la técnica TMR, este es el resultado esperado.

La lógica combinacional también se ve afectada ya que se insertan una

gran cantidad de votadores, uno por cada 3 biestables de los registros de control. En total se utilizan 1,464 LUTs (2 %), 449 más que en el procesador estándar. Esto incluye los 420 votadores necesarios.

La lógica de los votadores que se inserta a continuación de los registros tiene su incidencia pues ralentiza ligeramente el proceso. El procesador tolerante a fallos requiere un periodo mínimo de reloj de 6.819 ns alcanzando una frecuencia máxima de 146.649 MHz, lo que significa una reducción de la frecuencia máxima de funcionamiento en casi 10 MHz.

7.3. Simulaciones

A continuación se analizan las gráficas generadas por las simulaciones de los apéndices A y B.

7.3.1. Procesador estándar

En esta sección se analizan los resultados obtenidos de la ejecución sobre la implementación estándar del microprocesador.

7.3.1.1. Prueba de control

En la gráfica C.1 del apéndice C se observa paso a paso la ejecución del programa de control sobre el procesador estándar. A continuación se listan los eventos significativos por orden cronológico.

Durante la ejecución de la prueba de control, resumida en la tabla 7.1, se observa cómo se inicializan, en los ciclos 5 a 9, los valores de las variables de la operación (R1, R2), las variables del control del bucle (R4, R5) y la variable resultado (R3). A continuación se observa cómo se ejecuta el bucle acumulando las sumas en la variable resultado y el decremento en el indicador de iteración. El bucle se ejecuta 5 veces dejando un resultado final de 125 en el registro resultado. Para finalizar el resultado se almacena en memoria (M0) en el ciclo 75.

Ciclo	Señales	Evento
0	-	Inicio de la ejecución
5	R2	Se lee de memoria el valor 5 y se almacena en el registro R2.
6	R1	Se carga el valor 25 en el registro R1.
7	R3	Se inicia el registro R3 a 0 para acumular el resultado.
8, 9	R4, R5	Se inician las variables de control del bucle. R4 almacena el número de iteraciones.
18	R3	Se realiza la primera acumulación.
19	R4	Se decrementa el contador de iteraciones.
25	PC	Se realiza un salto al inicio del bucle.
...	R3, R4, PC	Se realizan las iteraciones del bucle, acumulando el resultado en R3 hasta que R4 valga 0.
65	PC	Se termina de ejecutar el bucle, se sigue avanzando por el código.
75	M0	Se almacena el resultado en la memoria de datos.

Tabla 7.1: Simulación de control sobre el procesador estándar.

7.3.1.2. Prueba con fallos

Al volver a ejecutar el mismo código de prueba, esta vez insertando fallos (Apéndice C, sección C.2), lo primero que se observa es que el resultado final de M0 no es el correcto, se ha almacenado el valor 75 en vez de el 125 correcto. A continuación analizamos por qué.

El primer fallo ocurre en el bus de memoria de una etapa interna (MEMbus_reg), y se propaga hasta almacenarse en el registro R2, es cuando se actualiza este registro cuando vemos los efectos del fallo, en vez del 5 esperado se ha almacenado un 4. Esto reduce el número de iteraciones que se realizarán.

Se continúa la ejecución del programa de forma normal hasta que ocurre el segundo fallo, esta vez se activa la señal «MEMWrite» encargada de indicar al módulo de memoria que almacene los datos. Esto se refleja en la señal «M3» que cambia de valor a 1. Este fallo no altera el funcionamiento de este programa pero la escritura en memoria podría alterar datos de otros programas si los hubiera.

El tercer fallo ocurre en el operando B de la ALU en una comparación. En vez de comparar la variable con la constante 0, se ha comparado con la constante 1, finalizando el bucle de ejecución antes de tiempo.

Ciclo	Señales	Evento
0	-	Inicio de la ejecución.
5	R2	Se lee de memoria el valor 5 y se almacena en el registro R2. Debido al primer fallo se almacena el valor 4.
6	R1	Se carga el valor 25 en el registro R1.
7	R3	Se inicia el registro R3 a 0 para acumular el resultado.
8, 9	R4, R5	Se inician las variables de control del bucle. R4 almacena el número de iteraciones.
17	-	Ocurre el segundo fallo. Se almacena un dato en la dirección 3 de memoria.
18	R3	Se realiza la primera acumulación.
19	R4	Se decrementa el contador de iteraciones.
25	PC	Se realiza un salto al inicio del bucle.
...	R3, R4, PC	Se realizan las iteraciones del bucle, acumulando el resultado en R3 hasta que R4 valga 0.
38	-	Ocurre el tercer fallo. Se termina el bucle antes de tiempo.
42	PC	Termina de ejecutar el bucle a falta de una iteración y se sigue avanzando por el código.
53	M0	Se almacena el resultado incorrecto en la memoria de datos.

Tabla 7.2: Simulación con fallos sobre el procesador estándar.

El resultado final es un programa mal ejecutado, que ha visto reducido las iteraciones que debería haber realizado, almacenando en memoria un dato incorrecto y además ha modificado elementos en memoria que no deberían haberse visto afectados.

7.3.2. Procesador tolerante a fallos

En esta sección se analizan los resultados obtenidos de la ejecución de las mismas pruebas anteriores sobre la implementación tolerante a fallos del microprocesador.

7.3.2.1. Prueba de control

En la gráfica del apéndice C sección C.3 se observa la evolución del programa a lo largo de su ejecución. El proceso sigue los mismos pasos seguidos al ejecutar las pruebas en el procesador estándar obteniendo los mismos resultados (tabla 7.3). Por lo cual se comprueba que la funcionalidad e integridad

Ciclo	Señales	Evento
0	-	Inicio de la ejecución
5	R2	Se lee de memoria el valor 5 y se almacena en el registro R2.
6	R1	Se carga el valor 25 en el registro R1.
7	R3	Se inicia el registro R3 a 0 para acumular el resultado.
8, 9	R4, R5	Se inician las variables de control del bucle. R4 almacena el número de iteraciones.
18	R3	Se realiza la primera acumulación.
19	R4	Se decrementa el contador de iteraciones.
25	PC	Se realiza un salto al inicio del bucle.
...	R3, R4, PC	Se realizan las iteraciones del bucle, acumulando el resultado en R3 hasta que R4 valga 0.
65	PC	Se termina de ejecutar el bucle, se sigue avanzando por el código.
75	M0	Se almacena el resultado en la memoria de datos.

Tabla 7.3: Simulación de control sobre el procesador tolerante a fallos .

del procesador se conservan al introducir la tolerancia a fallos.

7.3.2.2. Prueba con fallos

Para finalizar las pruebas, se realiza la simulación con fallos sobre el procesador tolerante a fallos. Lo primero que observamos en la simulación (apéndice C sección C.4) es que esta vez sí se ha ejecutado el programa de forma correcta, el resultado almacenado en memoria es el correcto y no se han modificado otras secciones de la memoria. A continuación analizamos la gráfica de la simulación, resumida en la tabla 7.4, para ver las diferencias.

El primer fallo se produce en uno de los registros del bus de memoria. El fallo es enmascarado gracias a que los otros dos registros han mantenido el valor correcto y a que el votador ha realizado su tarea, permitiendo que sea el valor más repetido el que continúe hasta el siguiente componente. El registro R2 almacena el valor 5 correctamente.

En el ciclo 17 se produce el segundo fallo. Este provocaría que se modificara la memoria de datos, sin embargo el sistema de tolerancia realiza su tarea y evita que esto ocurra.

El tercer fallo ocurre en el operando B durante el ciclo 38 de la ALU en una comparación. Este fallo, al haberse tolerado el primero, no habría afectado al programa ya que el valor de R4 es 2 y se compara con el valor 1, y no afecta a la ejecución ya que se realiza una comparación de igualdad. En cualquier caso, el fallo se ha enmascarado debido a la lógica del votador

Ciclo	Señales	Evento
0	-	Inicio de la ejecución
5	R2	Se lee de memoria el valor 5 y se almacena en el registro R2. Se almacena correctamente a pesar del primer fallo.
6	R1	Se carga el valor 25 en el registro R1.
7	R3	Se inicia el registro R3 a 0 para acumular el resultado.
8, 9	R4, R5	Se inician las variables de control del bucle. R4 almacena el número de iteraciones.
17	-	Ocurre el segundo fallo. Se tolera y no se almacena ningún dato en memoria.
18	R3	Se realiza la primera acumulación.
19	R4	Se decrementa el contador de iteraciones.
25	PC	Se realiza un salto al inicio del bucle.
...	R3, R4, PC	Se realizan las iteraciones del bucle, acumulando el resultado en R3 hasta que R4 valga 0.
38	-	Ocurre el tercer fallo. Se enmascara el fallo y no afecta a la ejecución.
65	PC	Se termina de ejecutar el bucle, se sigue avanzando por el código.
75	M0	Se almacena el resultado en la memoria de datos.

Tabla 7.4: Simulación con fallos sobre el procesador tolerante a fallos.

que siempre está activo.

Capítulo 8

Conclusiones y futuras líneas de trabajo

8.1. Conclusiones

El objetivo de este proyecto era crear una CPU segmentada compatible con ARM y dotarla de tolerancia a fallos transitorios con la intención de conseguir una ejecución correcta en sistemas donde los fallos fuesen algo común, como en sistemas internos de satélites o aeronaves.

En primera instancia se pretendía compatibilizar la CPU con un repertorio de instrucciones ampliamente usado. Se decidió utilizar el repertorio ARM ya que forma parte de una de las arquitecturas más utilizadas en todo tipo de sistemas. Para conseguir ejecutar estas instrucciones hubo que implementar una nueva unidad de control capaz de descifrar las instrucciones e integrarla en la CPU.

Una vez se tuvo el procesador con un repertorio de instrucciones capaz de ejecutar programas sencillos, se quería asegurar que este funcionaría bajo condiciones donde los fallos aparecieran con una alta probabilidad. Se decidió que los registros internos de la CPU eran los elementos que provocarían un mayor impacto, estos son muy numerosos y cada ciclo de reloj almacenan nueva información, si uno fallase se propagaría alterando la instrucción y el programa que se estuviera ejecutando.

Se decidió aplicar la técnica de «Triple Modular Redundancy» por ser una técnica efectiva y ampliamente utilizada en sistemas que requieren un mayor grado de tolerancia frente a fallos transitorios.

Una vez implementado el diseño y añadida la tolerancia a fallos realizaron simulaciones de situaciones controladas para comprobar si esta técnica realmente permitía al procesador funcionar bajo las condiciones deseadas.

Se llevaron a cabo simulaciones funcionales donde se ejecutaba un programa sobre ambas CPUs, el programa se ejecutaba

Se llevaron a cabo simulaciones funcionales sobre ambas CPUs. En unas simulaciones se ejecutaba un programa sin interferencias, mientras que en otras se ejecutaba el mismo programa insertando fallos en algunos registros. Con estas simulaciones pudimos determinar que:

1. Los resultados de la prueba de control eran idénticos por lo tanto ambas CPUs ejecutaban correctamente las instrucciones del programa en un entorno sin fallos.
2. Al insertar los mismos fallos en ambas CPUs, se observaban resultados diferentes. Comparando los resultados de la CPU estándar con la prueba anterior se comprobó que los fallos habían provocado errores graves en la ejecución del programa. Mientras que insertando los mismos fallos en la CPU tolerante a fallos daba los mismos resultados que las pruebas de control.

Gracias a estas simulaciones se pudo demostrar que se ha diseñado una unidad de procesamiento capaz de ejecutar programas sencillos y de tolerar fallos transitorios individuales de manera satisfactoria.

8.2. Futuras líneas de trabajo

Algunas posibilidades para continuar con este trabajo podrían incluir ampliaciones en la funcionalidad del sistema o aumentar la tolerancia a fallos.

Si deseamos ampliar la funcionalidad se puede conseguir aumentando el repertorio de instrucciones ya que solo se han implementado una pocas instrucciones que permiten ejecutar programas básicos en el diseño actual. Modificar las memorias de datos e instrucciones para permitir el uso de memorias externas e insertando memorias caché. O bien se puede integrar la CPU junto a controladores de diferentes tipos.

Por otro lado, si se desea aumentar la tolerancia a fallos, se puede aplicar la técnica TMR sobre otros componentes de la misma forma que se han aplicado sobre los registros de control. O se pueden aplicar nuevas técnicas para tolerar otros tipos de fallos, como podría aplicarse la re-configuración estática o dinámica para que el sistema fuese capaz de tolerar fallos permanentes además de los fallos transitorios.

Apéndice A

Código de simulación sin fallos

Este código se compone de 3 partes:

- **Declaración de componentes y señales:** Se declaran los componentes y las señales necesarias para controlar la simulación. Esto incluye las señales de entrada y salida del procesador, junto a las señales espía¹ que mostrarán los valores de señales internas del sistema.
- **Proceso de conexión:** Se establece la conexión entre las señales espía y las señales internas del sistema.
- **Proceso de control:** Se realizan las comprobaciones para asegurar el correcto funcionamiento del procesador.

A.1. Testbench para ejecución de control

```
1  —————
2  — Company: Universidad Complutense de Madrid
3  — Engineer: Andres Gamboa Melendez
4  —
5  — Module Name: TB_ejecucion_normal – Testbench
6  — Project Name: ARM compatible micro-processor
7  — Target Devices: Nexys4
8  — Tool versions: Xilinx ISE Webpack 14.4
9  — Description: Prueba de control para la ejecucion del programa
10 —                   multiplicacion basado en bucle de sumas.
11 —
12 — VHDL Test Bench Created by ISE for module: cpu
13 —
14  —————
15  library modelsim_lib;
16  use modelsim_lib.util.all;
17
```

¹Estas señales solo se pueden utilizar con el software «ModelSim» de Altera.

```

18 LIBRARY ieee;
19 USE ieee.std_logic_1164.ALL;
20 USE ieee.numeric_std.ALL;
21
22 ENTITY TB_ejecucion_normal IS
23 END TB_ejecucion_normal;
24
25 ARCHITECTURE behavior OF TB_ejecucion_normal IS
26
27     — Component Declaration for the Unit Under Test (UUT)
28     COMPONENT cpu
29     PORT(
30         clk : IN    std_logic;
31         rst : IN    std_logic;
32         led : OUT   std_logic_vector(15 downto 0)
33     );
34     END COMPONENT;
35
36     — Inputs
37     signal clk : std_logic := '0';
38     signal rst : std_logic := '0';
39
40     — Outputs
41     signal led : std_logic_vector(15 downto 0);
42
43     — Spy signals
44     signal spy_R0 : std_logic_vector(31 downto 0);
45     signal spy_R1 : std_logic_vector(31 downto 0);
46     signal spy_R2 : std_logic_vector(31 downto 0);
47     signal spy_R3 : std_logic_vector(31 downto 0);
48     signal spy_R4 : std_logic_vector(31 downto 0);
49     signal spy_R5 : std_logic_vector(31 downto 0);
50     signal spy_M0 : std_logic_vector(31 downto 0);
51     signal spy_M5 : std_logic_vector(31 downto 0);
52     signal spy_PC : std_logic_vector(31 downto 0);
53     signal spy_INSTR : std_logic_vector(31 downto 0);
54
55     — Clock period definitions
56     constant clk_period : time := 10 ns;
57
58 BEGIN
59
60     — Instantiate the Unit Under Test (UUT)
61     uut : cpu PORT MAP (
62         clk => clk ,
63         rst => rst ,
64         led => led
65     );
66
67     — Clock process definitions
68     clk_process : process
69     begin
70         clk <= '0';
71         wait for clk_period/2;

```

```

72     clk <= '1';
73     wait for clk_period/2;
74 end process;
75
76 — Configurar señales espías
77 spy_init : process
78 begin
79     init_signal_spy("/TB_ejecucion_normal/uut/i_ID/i_pID/
80         i_RegisterBank/regs(0)", "/TB_ejecucion_normal/spy_R0",
81         1);
82     init_signal_spy("/TB_ejecucion_normal/uut/i_ID/i_pID/
83         i_RegisterBank/regs(1)", "/TB_ejecucion_normal/spy_R1",
84         1);
85     init_signal_spy("/TB_ejecucion_normal/uut/i_ID/i_pID/
86         i_RegisterBank/regs(2)", "/TB_ejecucion_normal/spy_R2",
87         1);
88     init_signal_spy("/TB_ejecucion_normal/uut/i_ID/i_pID/
89         i_RegisterBank/regs(3)", "/TB_ejecucion_normal/spy_R3",
90         1);
91     init_signal_spy("/TB_ejecucion_normal/uut/i_ID/i_pID/
92         i_RegisterBank/regs(4)", "/TB_ejecucion_normal/spy_R4",
93         1);
94     init_signal_spy("/TB_ejecucion_normal/uut/i_ID/i_pID/
95         i_RegisterBank/regs(5)", "/TB_ejecucion_normal/spy_R5",
96         1);
97
98     init_signal_spy("/TB_ejecucion_normal/uut/i_MEM/i_pMEM/
99         i_MemData/mem(0)", "/TB_ejecucion_normal/spy_M0", 1);
100    init_signal_spy("/TB_ejecucion_normal/uut/i_MEM/i_pMEM/
101        i_MemData/mem(5)", "/TB_ejecucion_normal/spy_M5", 1);
102
103    init_signal_spy("/TB_ejecucion_normal/uut/IF_out_pc_reg", "
104        /TB_ejecucion_normal/spy_PC", 1);
105    init_signal_spy("/TB_ejecucion_normal/uut/IF_out_inst_reg"
106        , "/TB_ejecucion_normal/spy_INSTR", 1);
107
108    wait;
109 end process spy_init;
110
111 — Test de funcionamiento
112 spy_proc: process
113 begin
114     — hold reset state for 100 ns.
115     rst <= '0';
116     wait for 100 ns;
117     rst <= '1';
118
119     assert spy_PC = std_logic_vector(to_unsigned(0, 32))
120         report "ERROR: Inicio" severity ERROR;
121     wait for clk_period*5; — Espera para que tengan efecto los
122         cambios (Procesador de cinco ciclos)
123 —LDR R2, R0, #5
124     assert spy_PC = std_logic_vector(to_unsigned(20, 32))
125         report "ERROR: PC 20" severity ERROR;

```

```

106         assert spy_R2 = std_logic_vector(to_unsigned(5, 32))
           report "ERROR: LDR R2, R0, #5" severity ERROR;
107     wait for clk_period;
108 --MOV R1, #25
109         assert spy_PC = std_logic_vector(to_unsigned(24, 32))
           report "ERROR: PC 24" severity ERROR;
110         assert spy_R1 = std_logic_vector(to_unsigned(25, 32))
           report "ERROR: MOV R1, #25" severity ERROR;
111     wait for clk_period;
112 --MOV R3, #0
113         assert spy_PC = std_logic_vector(to_unsigned(28, 32))
           report "ERROR: PC 28" severity ERROR;
114         assert spy_R3 = std_logic_vector(to_unsigned(0, 32))
           report "ERROR: MOV R3, #0" severity ERROR;
115     wait for clk_period;
116 --MOV R5, #1
117         assert spy_PC = std_logic_vector(to_unsigned(32, 32))
           report "ERROR: PC 32" severity ERROR;
118         assert spy_R5 = std_logic_vector(to_unsigned(1, 32))
           report "ERROR: MOV R5, #1" severity ERROR;
119     wait for clk_period;
120 --MOV R4, R2
121         assert spy_PC = std_logic_vector(to_unsigned(36, 32))
           report "ERROR: PC 36" severity ERROR;
122         assert spy_R4 = std_logic_vector(to_unsigned(5, 32))
           report "ERROR: MOV R4, R2" severity ERROR;
123 --NOP
124 --NOP
125 --NOP
126 --CMP R4, R0
127 --BEQ # 24
128 --NOP
129 --NOP
130 --NOP
131     wait for clk_period*10;
132 --ADD R3, R3, R1
133         assert spy_PC = std_logic_vector(to_unsigned(32, 32))
           report "ERROR: PC 72" severity ERROR;
134         assert spy_R3 = std_logic_vector(to_unsigned(25, 32))
           report "ERROR: ADD R3, R3, R1(25)" severity ERROR;
135     wait for clk_period;
136 --SUB R4, R4, R5
137         assert spy_PC = std_logic_vector(to_unsigned(36, 32))
           report "ERROR: PC 76" severity ERROR;
138         assert spy_R4 = std_logic_vector(to_unsigned(4, 32))
           report "ERROR: SUB R4, R4, R5(4)" severity ERROR;
139 --CMP R4, R0
140 --BEQ # 24
141 --NOP
142 --NOP
143 --NOP
144     wait for clk_period*10;
145 --ADD R3, R3, R1

```



```

146     assert spy_PC = std_logic_vector(to_unsigned(32, 32))
147         report "ERROR: PC 116" severity ERROR;
148     assert spy_R3 = std_logic_vector(to_unsigned(50, 32))
149         report "ERROR: ADD R3, R3, R1(50)" severity ERROR;
150     wait for clk_period;
151 —SUB R4, R4, R5
152     assert spy_PC = std_logic_vector(to_unsigned(36, 32))
153         report "ERROR: PC 120" severity ERROR;
154     assert spy_R4 = std_logic_vector(to_unsigned(3, 32))
155         report "ERROR: SUB R4, R4, R5(3)" severity ERROR;
156 —CMP R4, R0
157 —BEQ # 24
158 —NOP
159 —NOP
160 —NOP
161     wait for clk_period*10;
162 —ADD R3, R3, R1
163     assert spy_PC = std_logic_vector(to_unsigned(32, 32))
164         report "ERROR: PC 160" severity ERROR;
165     assert spy_R3 = std_logic_vector(to_unsigned(75, 32))
166         report "ERROR: ADD R3, R3, R1(75)" severity ERROR;
167     wait for clk_period;
168 —SUB R4, R4, R5
169     assert spy_PC = std_logic_vector(to_unsigned(36, 32))
170         report "ERROR: PC 164" severity ERROR;
171     assert spy_R4 = std_logic_vector(to_unsigned(2, 32))
172         report "ERROR: SUB R4, R4, R5(2)" severity ERROR;
173 —CMP R4, R0
174 —BEQ # 24
175 —NOP
176 —NOP
177 —NOP
178     wait for clk_period*10;
179 —ADD R3, R3, R1
180     assert spy_PC = std_logic_vector(to_unsigned(32, 32))
181         report "ERROR: PC 204" severity ERROR;
182     assert spy_R3 = std_logic_vector(to_unsigned(100, 32))
183         report "ERROR: ADD R3, R3, R1(100)" severity ERROR;
184     wait for clk_period;
185 —SUB R4, R4, R5
186     assert spy_PC = std_logic_vector(to_unsigned(36, 32))
187         report "ERROR: PC 208" severity ERROR;
188     assert spy_R4 = std_logic_vector(to_unsigned(1, 32))
189         report "ERROR: SUB R4, R4, R5(1)" severity ERROR;
190     wait for clk_period;
191 —CMP R4, R0
192 —BEQ # 24
193 —NOP
194 —NOP
195 —NOP
196     wait for clk_period*9;
197 —ADD R3, R3, R1
198     assert spy_PC = std_logic_vector(to_unsigned(32, 32))
199         report "ERROR: PC 248" severity ERROR;

```

```

187         assert spy_R3 = std_logic_vector(to_unsigned(125, 32))
           report "ERROR: ADD R3, R3, R1(125)" severity ERROR;
188     wait for clk_period;
189 --SUB R4, R4, R5
190         assert spy_PC = std_logic_vector(to_unsigned(36, 32))
           report "ERROR: PC 252" severity ERROR;
191         assert spy_R4 = std_logic_vector(to_unsigned(0, 32))
           report "ERROR: SUB R4, R4, R5(0)" severity ERROR;
192 --B # -32
193 --NOP
194 --NOP
195 --NOP
196 --CMP R4, R0
197 --BEQ # 24
198 --NOP
199 --NOP
200 --NOP
201 --STR R3, R0, #0
202     wait for clk_period*12;
203         assert spy_PC = std_logic_vector(to_unsigned(96, 32))
           report "ERROR: PC 296" severity ERROR;
204         assert spy_M0 = std_logic_vector(to_unsigned(125, 32))
           report "ERROR: STR R3, R0, #0" severity ERROR;
205     wait;
206 end process;
207 END;

```

Apéndice B

Código de simulación con fallos

Este código se compone de 4 partes:

- **Declaración de componentes y señales:** Se declaran los componentes y las señales necesarias para controlar la simulación. Esto incluye las señales de entrada y salida del procesador, junto a las señales espía¹ que mostrarán los valores de señales internas del sistema.
- **Proceso de conexión:** Se establece la conexión entre las señales espía y las señales internas del sistema.
- **Proceso de control:** Se realizan las comprobaciones para asegurar el correcto funcionamiento del procesador.
- **Introducción de fallos:** Se fuerzan señales internas del procesador a valores incorrectos.

B.1. Testbench para ejecución con inserción de fallos en CPU estándar

```
1
2 — Company: Universidad Complutense de Madrid
3 — Engineer: Andres Gamboa Melendez
4 —
5 — Module Name: TB_ejecucion_fallos – Testbench
6 — Project Name: ARM compatible micro-processor
7 — Target Devices: Nexys4
8 — Tool versions: Xilinx ISE Webpack 14.4
9 — Description: Prueba de control para la ejecucion del
10 —               programa multiplicacion basado en bucle de sumas.
11 —
12 — VHDL Test Bench Created by ISE for module: cpu
```

¹Estas señales solo se pueden utilizar con el software «ModelSim» de Altera.

```

13  --
14  -----
15  library modelsim_lib;
16  use modelsim_lib.util.all;
17
18  LIBRARY ieee;
19  USE ieee.std_logic_1164.ALL;
20
21  -- Uncomment the following library declaration if using
22  -- arithmetic functions with Signed or Unsigned values
23  USE ieee.numeric_std.ALL;
24
25  ENTITY TB_ejecucion_fallos IS
26  END TB_ejecucion_fallos;
27
28  ARCHITECTURE behavior OF TB_ejecucion_fallos IS
29
30      -- Component Declaration for the Unit Under Test (UUT)
31      COMPONENT cpu
32      PORT(
33          clk : IN    std_logic;
34          rst : IN    std_logic;
35          led : OUT   std_logic_vector(15 downto 0)
36      );
37      END COMPONENT;
38
39
40      -- Inputs
41      signal clk : std_logic := '0';
42      signal rst : std_logic := '0';
43
44      -- Outputs
45      signal led : std_logic_vector(15 downto 0);
46
47      -- Spy signals
48      signal spy_R0, spy_R1, spy_R2, spy_R3, spy_R4, spy_R5 :
49          std_logic_vector(31 downto 0);
50      signal spy_M0, spy_M5 : std_logic_vector(31 downto 0);
51      signal spy_PC         : std_logic_vector(31 downto 0);
52      signal spy_INSTR       : std_logic_vector(31 downto 0);
53
54      -- Fault signals
55      -- Fallo 1
56      signal fspy_MEMbus_reg : std_logic_vector(31 downto 0);
57      -- Fallo 2
58      signal fspy_M3 : std_logic_vector(31 downto 0);
59      signal fspy_MEM_control_reg : std_logic_vector(5 downto 0);
60      -- Fallo 3
61      signal fspy_busB_reg : std_logic_vector(31 downto 0);
62
63      -- Clock period definitions
64      constant clk_period : time := 10 ns;
65
66  BEGIN

```

```

66
67 — Instantiate the Unit Under Test (UUT)
68 uut: cpu PORT MAP (
69     clk => clk ,
70     rst => rst ,
71     led => led
72 );
73
74 — Clock process definitions
75 clk_process : process
76 begin
77     clk <= '0';
78     wait for clk_period/2;
79     clk <= '1';
80     wait for clk_period/2;
81 end process;
82
83 — Configurar señales espías
84 spy_init : process
85 begin
86     — iniciar señales espía
87     init_signal_spy ("/TB_ejecucion_fallos/uut/i_ID/i_pID/
88         i_RegisterBank/regs(0)", "/TB_ejecucion_fallos/spy_R0",
89         1);
90     init_signal_spy ("/TB_ejecucion_fallos/uut/i_ID/i_pID/
91         i_RegisterBank/regs(1)", "/TB_ejecucion_fallos/spy_R1",
92         1);
93     init_signal_spy ("/TB_ejecucion_fallos/uut/i_ID/i_pID/
94         i_RegisterBank/regs(2)", "/TB_ejecucion_fallos/spy_R2",
95         1);
96     init_signal_spy ("/TB_ejecucion_fallos/uut/i_ID/i_pID/
97         i_RegisterBank/regs(3)", "/TB_ejecucion_fallos/spy_R3",
98         1);
99     init_signal_spy ("/TB_ejecucion_fallos/uut/i_ID/i_pID/
100        i_RegisterBank/regs(4)", "/TB_ejecucion_fallos/spy_R4",
101        1);
102     init_signal_spy ("/TB_ejecucion_fallos/uut/i_ID/i_pID/
103        i_RegisterBank/regs(5)", "/TB_ejecucion_fallos/spy_R5",
104        1);
105
106     init_signal_spy ("/TB_ejecucion_fallos/uut/i_MEM/i_pMEM/
107        i_MemData/mem(0)", "/TB_ejecucion_fallos/spy_M0", 1);
108     init_signal_spy ("/TB_ejecucion_fallos/uut/i_MEM/i_pMEM/
109        i_MemData/mem(5)", "/TB_ejecucion_fallos/spy_M5", 1);
110
111     init_signal_spy ("/TB_ejecucion_fallos/uut/IF_out_pc_reg",
112        "/TB_ejecucion_fallos/spy_PC", 1);
113     init_signal_spy ("/TB_ejecucion_fallos/uut/IF_out_inst_reg",
114        "/TB_ejecucion_fallos/spy_INSTR", 1);
115
116 — Iniciar señales para mostrar fallos
117 — Fallo 1
118 init_signal_spy ("/TB_ejecucion_fallos/uut/
119     MEM_out_MEMbus_reg", "/TB_ejecucion_fallos/

```

```

    fspy_MEMbus_reg", 1);
103   — Fallo 2
104   init_signal_spy("/TB_ejecucion_fallos/unt/
    EXE_out_MEM_control_reg", "/TB_ejecucion_fallos/
    fspy_MEM_control_reg", 1);
105   init_signal_spy("/TB_ejecucion_fallos/unt/i_MEM/i_pMEM/
    i_MemData/mem(3)", "/TB_ejecucion_fallos/fspy_M3", 1);
106   — Fallo 3
107   init_signal_spy("/TB_ejecucion_fallos/unt/ID_out_busB_reg"
    , "/TB_ejecucion_fallos/fspy_busB_reg", 1);
108
109   wait;
110   end process spy_init;
111
112
113   — Test de funcionamiento
114   spy_proc: process
115   begin
116     — hold reset state for 100 ns.
117     rst <= '0';
118     wait for 100 ns;
119     rst <= '1';
120
121     assert spy_PC = std_logic_vector(to_unsigned(0, 32))
        report "ERROR: Inicio" severity ERROR;
122     wait for clk_period*5; — Espera para que tengan efecto los
        cambios (Procesador de cinco ciclos)
123   —LDR R2, R0, #5
124     assert spy_PC = std_logic_vector(to_unsigned(20, 32))
        report "ERROR: PC 20" severity ERROR;
125     assert spy_R2 = std_logic_vector(to_unsigned(5, 32))
        report "ERROR: LDR R2, R0, #5" severity ERROR;
126     wait for clk_period;
127   —MOV R1, #25
128     assert spy_PC = std_logic_vector(to_unsigned(24, 32))
        report "ERROR: PC 24" severity ERROR;
129     assert spy_R1 = std_logic_vector(to_unsigned(25, 32))
        report "ERROR: MOV R1, #25" severity ERROR;
130     wait for clk_period;
131   —MOV R3, #0
132     assert spy_PC = std_logic_vector(to_unsigned(28, 32))
        report "ERROR: PC 28" severity ERROR;
133     assert spy_R3 = std_logic_vector(to_unsigned(0, 32))
        report "ERROR: MOV R3, #0" severity ERROR;
134     wait for clk_period;
135   —MOV R5, #1
136     assert spy_PC = std_logic_vector(to_unsigned(32, 32))
        report "ERROR: PC 32" severity ERROR;
137     assert spy_R5 = std_logic_vector(to_unsigned(1, 32))
        report "ERROR: MOV R5, #1" severity ERROR;
138     wait for clk_period;
139   —MOV R4, R2
140     assert spy_PC = std_logic_vector(to_unsigned(36, 32))
        report "ERROR: PC 36" severity ERROR;

```

```

141     assert spy_R4 = std_logic_vector(to_unsigned(5, 32))
        report "ERROR: MOV R4, R2" severity ERROR;
142 —NOP
143 —NOP
144 —NOP
145 —CMP R4, R0
146 —BEQ # 24
147 —NOP
148 —NOP
149 —NOP
150     wait for clk_period*10;
151 —ADD R3, R3, R1
152     assert spy_PC = std_logic_vector(to_unsigned(32, 32))
        report "ERROR: PC 72" severity ERROR;
153     assert spy_R3 = std_logic_vector(to_unsigned(25, 32))
        report "ERROR: ADD R3, R3, R1(25)" severity ERROR;
154     wait for clk_period;
155 —SUB R4, R4, R5
156     assert spy_PC = std_logic_vector(to_unsigned(36, 32))
        report "ERROR: PC 76" severity ERROR;
157     assert spy_R4 = std_logic_vector(to_unsigned(4, 32))
        report "ERROR: SUB R4, R4, R5(4)" severity ERROR;
158 —CMP R4, R0
159 —BEQ # 24
160 —NOP
161 —NOP
162 —NOP
163     wait for clk_period*10;
164 —ADD R3, R3, R1
165     assert spy_PC = std_logic_vector(to_unsigned(32, 32))
        report "ERROR: PC 116" severity ERROR;
166     assert spy_R3 = std_logic_vector(to_unsigned(50, 32))
        report "ERROR: ADD R3, R3, R1(50)" severity ERROR;
167     wait for clk_period;
168 —SUB R4, R4, R5
169     assert spy_PC = std_logic_vector(to_unsigned(36, 32))
        report "ERROR: PC 120" severity ERROR;
170     assert spy_R4 = std_logic_vector(to_unsigned(3, 32))
        report "ERROR: SUB R4, R4, R5(3)" severity ERROR;
171 —CMP R4, R0
172 —BEQ # 24
173 —NOP
174 —NOP
175 —NOP
176     wait for clk_period*10;
177 —ADD R3, R3, R1
178     assert spy_PC = std_logic_vector(to_unsigned(32, 32))
        report "ERROR: PC 160" severity ERROR;
179     assert spy_R3 = std_logic_vector(to_unsigned(75, 32))
        report "ERROR: ADD R3, R3, R1(75)" severity ERROR;
180     wait for clk_period;
181 —SUB R4, R4, R5
182     assert spy_PC = std_logic_vector(to_unsigned(36, 32))
        report "ERROR: PC 164" severity ERROR;

```

```

183         assert spy_R4 = std_logic_vector(to_unsigned(2, 32))
           report "ERROR: SUB R4, R4, R5(2)" severity ERROR;
184 —CMP R4, R0
185 —BEQ # 24
186 —NOP
187 —NOP
188 —NOP
189     wait for clk_period*10;
190 —ADD R3, R3, R1
191     assert spy_PC = std_logic_vector(to_unsigned(32, 32))
           report "ERROR: PC 204" severity ERROR;
192     assert spy_R3 = std_logic_vector(to_unsigned(100, 32))
           report "ERROR: ADD R3, R3, R1(100)" severity ERROR;
193     wait for clk_period;
194 —SUB R4, R4, R5
195     assert spy_PC = std_logic_vector(to_unsigned(36, 32))
           report "ERROR: PC 208" severity ERROR;
196     assert spy_R4 = std_logic_vector(to_unsigned(1, 32))
           report "ERROR: SUB R4, R4, R5(1)" severity ERROR;
197     wait for clk_period;
198 —CMP R4, R0
199 —BEQ # 24
200 —NOP
201 —NOP
202 —NOP
203     wait for clk_period*9;
204 —ADD R3, R3, R1
205     assert spy_PC = std_logic_vector(to_unsigned(32, 32))
           report "ERROR: PC 248" severity ERROR;
206     assert spy_R3 = std_logic_vector(to_unsigned(125, 32))
           report "ERROR: ADD R3, R3, R1(125)" severity ERROR;
207     wait for clk_period;
208 —SUB R4, R4, R5
209     assert spy_PC = std_logic_vector(to_unsigned(36, 32))
           report "ERROR: PC 252" severity ERROR;
210     assert spy_R4 = std_logic_vector(to_unsigned(0, 32))
           report "ERROR: SUB R4, R4, R5(0)" severity ERROR;
211 —B # -32
212 —NOP
213 —NOP
214 —NOP
215 —CMP R4, R0
216 —BEQ # 24
217 —NOP
218 —NOP
219 —NOP
220 —STR R3, R0, #0
221     wait for clk_period*12;
222     assert spy_PC = std_logic_vector(to_unsigned(96, 32))
           report "ERROR: PC 296" severity ERROR;
223     assert spy_M0 = std_logic_vector(to_unsigned(125, 32))
           report "ERROR: STR R3, R0, #0" severity ERROR;
224     wait;
225 end process;

```



```

226
227
228   — Inserción de fallos
229   fault_proc: process
230   begin
231       — Fallo 1: Forzar Data read de memoria (bit 0) a 0 (Data
          read != 5 => Data read = 4)
232       signal_force("/TB_ejecucion_fallos/uut/MEM_out_MEMbus_reg
          (0)", "0", 140 ns, freeze, 145 ns, 1); — f_MEMbus
233
234       — Fallo 2: Forzar Data read de memoria (bit 0) a 0 (Data
          read != 5 => Data read = 4)
235       signal_force("/TB_ejecucion_fallos/uut/
          EXE_out_MEM_control_reg(0)", "1", 270 ns, freeze, 275
          ns, 1);
236
237       — Fallo 3: Forzar Data read de memoria (bit 0) a 0 (Data
          read != 5 => Data read = 4)
238       signal_force("/TB_ejecucion_fallos/uut/ID_out_busB_reg(0) "
          , "1", 530 ns, freeze, 535 ns, 1);
239
240       wait;
241   end process;
242
243 END;

```

B.2. Testbench para ejecución con inserción de fallos en CPU tolerante a fallos

```

1  ---
2  -- Company: Universidad Complutense de Madrid
3  -- Engineer: Andres Gamboa Melendez
4  ---
5  -- Module Name: TB_ejecucion_fallos - Testbench
6  -- Project Name: ARM compatible micro-processor
7  -- Target Devices: Nexys4
8  -- Tool versions: Xilinx ISE Webpack 14.4
9  -- Description: Prueba de control para la ejecucion del
10 --                programa multiplicacion basado en bucle de sumas
11 --
12 -- VHDL Test Bench Created by ISE for module: cpu
13 --
14 ---
15 library modelsim_lib;
16 use modelsim_lib.util.all;
17
18 LIBRARY ieee;
19 USE ieee.std_logic_1164.ALL;
20
21 -- Uncomment the following library declaration if using
22 -- arithmetic functions with Signed or Unsigned values
23 USE ieee.numeric_std.ALL;
24
25 ENTITY TB_ejecucion_fallos IS
26 END TB_ejecucion_fallos;
27
28 ARCHITECTURE behavior OF TB_ejecucion_fallos IS
29
30     -- Component Declaration for the Unit Under Test (UUT)
31
32     COMPONENT cpu
33     PORT(
34         clk : IN  std_logic;
35         rst : IN  std_logic;
36         led : OUT std_logic_vector(15 downto 0)
37     );
38     END COMPONENT;
39
40
41     -- Inputs
42     signal clk : std_logic := '0';
43     signal rst : std_logic := '0';
44
45     -- Outputs
46     signal led : std_logic_vector(15 downto 0);
47
48     -- Spy signals

```

```

49     signal spy_R0, spy_R1, spy_R2, spy_R3, spy_R4, spy_R5 :
        std_logic_vector(31 downto 0);
50     signal spy_M0, spy_M5 : std_logic_vector(31 downto 0);
51     signal spy_PC          : std_logic_vector(31 downto 0);
52     signal spy_INSTR       : std_logic_vector(31 downto 0);
53
54     -- Fault signals
55     type TF_regs_32 is array (0 to 2) of STD_LOGIC_VECTOR (31
        downto 0);
56     type TF_regs_6 is array (0 to 2) of STD_LOGIC_VECTOR (5
        downto 0);
57     -- Fallo 1
58     signal fspy_MEMbus_regs : TF_regs_32;
59     -- Fallo 2
60     signal fspy_M3 : STD_LOGIC_VECTOR (31 downto 0);
61     signal fspy_MEM_control_regs : TF_regs_6;
62     -- Fallo 3
63     signal fspy_busB_regs : TF_regs_32;
64
65
66     -- signal : std_logic_vector(31 downto 0);
67
68     -- Clock period definitions
69     constant clk_period : time := 10 ns;
70
71 BEGIN
72
73     -- Instantiate the Unit Under Test (UUT)
74     uut: cpu PORT MAP (
75         clk => clk ,
76         rst => rst ,
77         led => led
78     );
79
80     -- Clock process definitions
81     clk_process : process
82     begin
83         clk <= '0';
84         wait for clk_period/2;
85         clk <= '1';
86         wait for clk_period/2;
87     end process;
88
89     -- Configurar señales espías
90     spy_init : process
91     begin
92         -- iniciar señales espía
93         init_signal_spy("/TB_ejecucion_fallos/uut/i_ID/i_pID/
            i_RegisterBank/regs(0)", "/TB_ejecucion_fallos/spy_R0",
            1);
94         init_signal_spy("/TB_ejecucion_fallos/uut/i_ID/i_pID/
            i_RegisterBank/regs(1)", "/TB_ejecucion_fallos/spy_R1",
            1);

```

```

95     init_signal_spy("/TB_ejecucion_fallos/ uut/i_ID/i_pID/
        i_RegisterBank/regs(2)", "/TB_ejecucion_fallos/spy_R2",
        1);
96     init_signal_spy("/TB_ejecucion_fallos/ uut/i_ID/i_pID/
        i_RegisterBank/regs(3)", "/TB_ejecucion_fallos/spy_R3",
        1);
97     init_signal_spy("/TB_ejecucion_fallos/ uut/i_ID/i_pID/
        i_RegisterBank/regs(4)", "/TB_ejecucion_fallos/spy_R4",
        1);
98     init_signal_spy("/TB_ejecucion_fallos/ uut/i_ID/i_pID/
        i_RegisterBank/regs(5)", "/TB_ejecucion_fallos/spy_R5",
        1);
99     init_signal_spy("/TB_ejecucion_fallos/ uut/i_MEM/i_pMEM/
        i_MemData/mem(0)", "/TB_ejecucion_fallos/spy_M0", 1);
100    init_signal_spy("/TB_ejecucion_fallos/ uut/i_MEM/i_pMEM/
        i_MemData/mem(5)", "/TB_ejecucion_fallos/spy_M5", 1);
101    init_signal_spy("/TB_ejecucion_fallos/ uut/IF_out_pc_reg", "
        /TB_ejecucion_fallos/spy_PC", 1);
102    init_signal_spy("/TB_ejecucion_fallos/ uut/IF_out_inst_reg"
        , "/TB_ejecucion_fallos/spy_INSTR", 1);
103
104    — Iniciar señales para mostrar fallos
105    — Fallo 1
106    init_signal_spy("/TB_ejecucion_fallos/ uut/r_MEM_out_MEMbus
        /regs", "/TB_ejecucion_fallos/fspy_MEMbus_regs", 1);
107    — Fallo 2
108    init_signal_spy("/TB_ejecucion_fallos/ uut/
        r_EXE_out_MEM_control/regs", "/TB_ejecucion_fallos/
        fspy_MEM_control_regs", 1);
109    init_signal_spy("/TB_ejecucion_fallos/ uut/i_MEM/i_pMEM/
        i_MemData/mem(3)", "/TB_ejecucion_fallos/fspy_M3", 1);
110    — Fallo 3
111    init_signal_spy("/TB_ejecucion_fallos/ uut/
        r_ID_out_busB_reg/regs", "/TB_ejecucion_fallos/
        fspy_busB_regs", 1);
112    wait;
113    end process spy_init;
114
115
116    — Test de funcionamiento
117    spy_proc: process
118    begin
119        — hold reset state for 100 ns.
120        rst <= '0';
121        wait for 100 ns;
122        rst <= '1';
123
124        assert spy_PC = std_logic_vector(to_unsigned(0, 32))
            report "ERROR: Inicio" severity ERROR;
125        wait for clk_period*5; — Espera para que tengan efecto los
            cambios (Procesador de cinco ciclos)
126    —LDR R2, R0, #5
127        assert spy_PC = std_logic_vector(to_unsigned(20, 32))
            report "ERROR: PC 20" severity ERROR;

```

```

128     assert spy_R2 = std_logic_vector(to_unsigned(5, 32))
        report "ERROR: LDR R2, R0, #5" severity ERROR;
129     wait for clk_period;
130 —MOV R1, #25
131     assert spy_PC = std_logic_vector(to_unsigned(24, 32))
        report "ERROR: PC 24" severity ERROR;
132     assert spy_R1 = std_logic_vector(to_unsigned(25, 32))
        report "ERROR: MOV R1, #25" severity ERROR;
133     wait for clk_period;
134 —MOV R3, #0
135     assert spy_PC = std_logic_vector(to_unsigned(28, 32))
        report "ERROR: PC 28" severity ERROR;
136     assert spy_R3 = std_logic_vector(to_unsigned(0, 32))
        report "ERROR: MOV R3, #0" severity ERROR;
137     wait for clk_period;
138 —MOV R5, #1
139     assert spy_PC = std_logic_vector(to_unsigned(32, 32))
        report "ERROR: PC 32" severity ERROR;
140     assert spy_R5 = std_logic_vector(to_unsigned(1, 32))
        report "ERROR: MOV R5, #1" severity ERROR;
141     wait for clk_period;
142 —MOV R4, R2
143     assert spy_PC = std_logic_vector(to_unsigned(36, 32))
        report "ERROR: PC 36" severity ERROR;
144     assert spy_R4 = std_logic_vector(to_unsigned(5, 32))
        report "ERROR: MOV R4, R2" severity ERROR;
145 —NOP
146 —NOP
147 —NOP
148 —CMP R4, R0
149 —BEQ # 24
150 —NOP
151 —NOP
152 —NOP
153     wait for clk_period*10;
154 —ADD R3, R3, R1
155     assert spy_PC = std_logic_vector(to_unsigned(32, 32))
        report "ERROR: PC 72" severity ERROR;
156     assert spy_R3 = std_logic_vector(to_unsigned(25, 32))
        report "ERROR: ADD R3, R3, R1(25)" severity ERROR;
157     wait for clk_period;
158 —SUB R4, R4, R5
159     assert spy_PC = std_logic_vector(to_unsigned(36, 32))
        report "ERROR: PC 76" severity ERROR;
160     assert spy_R4 = std_logic_vector(to_unsigned(4, 32))
        report "ERROR: SUB R4, R4, R5(4)" severity ERROR;
161 —CMP R4, R0
162 —BEQ # 24
163 —NOP
164 —NOP
165 —NOP
166     wait for clk_period*10;
167 —ADD R3, R3, R1

```

```

168     assert spy_PC = std_logic_vector(to_unsigned(32, 32))
169     report "ERROR: PC 116" severity ERROR;
170     assert spy_R3 = std_logic_vector(to_unsigned(50, 32))
171     report "ERROR: ADD R3, R3, R1(50)" severity ERROR;
172     wait for clk_period;
173   --SUB R4, R4, R5
174     assert spy_PC = std_logic_vector(to_unsigned(36, 32))
175     report "ERROR: PC 120" severity ERROR;
176     assert spy_R4 = std_logic_vector(to_unsigned(3, 32))
177     report "ERROR: SUB R4, R4, R5(3)" severity ERROR;
178   --CMP R4, R0
179   --BEQ # 24
180   --NOP
181   --NOP
182   --NOP
183     wait for clk_period*10;
184   --ADD R3, R3, R1
185     assert spy_PC = std_logic_vector(to_unsigned(32, 32))
186     report "ERROR: PC 160" severity ERROR;
187     assert spy_R3 = std_logic_vector(to_unsigned(75, 32))
188     report "ERROR: ADD R3, R3, R1(75)" severity ERROR;
189     wait for clk_period;
190   --SUB R4, R4, R5
191     assert spy_PC = std_logic_vector(to_unsigned(36, 32))
192     report "ERROR: PC 164" severity ERROR;
193     assert spy_R4 = std_logic_vector(to_unsigned(2, 32))
194     report "ERROR: SUB R4, R4, R5(2)" severity ERROR;
195   --CMP R4, R0
196   --BEQ # 24
197   --NOP
198   --NOP
199   --NOP
200     wait for clk_period*10;
201   --ADD R3, R3, R1
202     assert spy_PC = std_logic_vector(to_unsigned(32, 32))
203     report "ERROR: PC 204" severity ERROR;
204     assert spy_R3 = std_logic_vector(to_unsigned(100, 32))
205     report "ERROR: ADD R3, R3, R1(100)" severity ERROR;
206     wait for clk_period;
207   --SUB R4, R4, R5
208     assert spy_PC = std_logic_vector(to_unsigned(36, 32))
209     report "ERROR: PC 208" severity ERROR;
210     assert spy_R4 = std_logic_vector(to_unsigned(1, 32))
211     report "ERROR: SUB R4, R4, R5(1)" severity ERROR;
212     wait for clk_period;
213   --CMP R4, R0
214   --BEQ # 24
215   --NOP
216   --NOP
217   --NOP
218     wait for clk_period*9;
219   --ADD R3, R3, R1
220     assert spy_PC = std_logic_vector(to_unsigned(32, 32))
221     report "ERROR: PC 248" severity ERROR;

```

```

209     assert spy_R3 = std_logic_vector(to_unsigned(125, 32))
        report "ERROR: ADD R3, R3, R1(125)" severity ERROR;
210     wait for clk_period;
211 —SUB R4, R4, R5
212     assert spy_PC = std_logic_vector(to_unsigned(36, 32))
        report "ERROR: PC 252" severity ERROR;
213     assert spy_R4 = std_logic_vector(to_unsigned(0, 32))
        report "ERROR: SUB R4, R4, R5(0)" severity ERROR;
214 —B # -32
215 —NOP
216 —NOP
217 —NOP
218 —CMP R4, R0
219 —BEQ # 24
220 —NOP
221 —NOP
222 —NOP
223 —STR R3, R0, #0
224     wait for clk_period*12;
225     assert spy_PC = std_logic_vector(to_unsigned(96, 32))
        report "ERROR: PC 296" severity ERROR;
226     assert spy_M0 = std_logic_vector(to_unsigned(125, 32))
        report "ERROR: STR R3, R0, #0" severity ERROR;
227     wait;
228 end process;
229
230 — Inserción de fallos
231 fault_proc: process
232 begin
233     — Fallo 1: Forzar Data read de memoria (bit 0) a 0 (Data
        read != 5 => Data read = 4)
234     signal_force("/TB_ejecucion_fallos/uut/r_MEM_out_MEMbus/
        regs(0)(0)", "0", 140 ns, freeze, 145 ns, 1); —
        f_MEMbus
235     — Fallo 2: Forzar Data read de memoria (bit 0) a 0 (Data
        read != 5 => Data read = 4)
236     signal_force("/TB_ejecucion_fallos/uut/
        r_EXE_out_MEM_control/regs(0)(0)", "1", 270 ns, freeze,
        275 ns, 1);
237     — Fallo 3: Forzar Data read de memoria (bit 0) a 0 (Data
        read != 5 => Data read = 4)
238     signal_force("/TB_ejecucion_fallos/uut/r_ID_out_busB_reg/
        regs(0)(0)", "1", 530 ns, freeze, 535 ns, 1);
239     wait;
240 end process;
241
242 END;
```

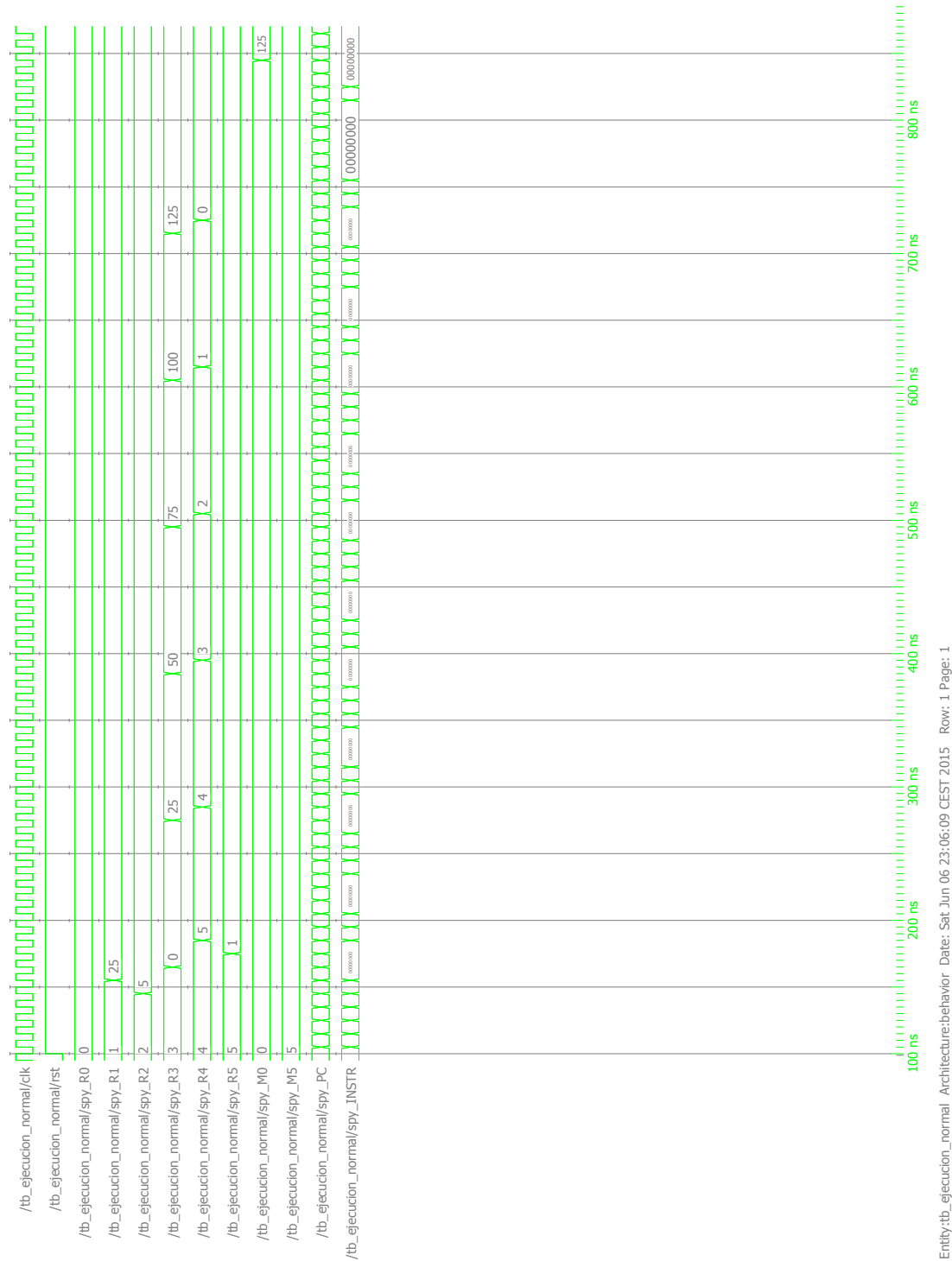

Apéndice C

Simulaciones

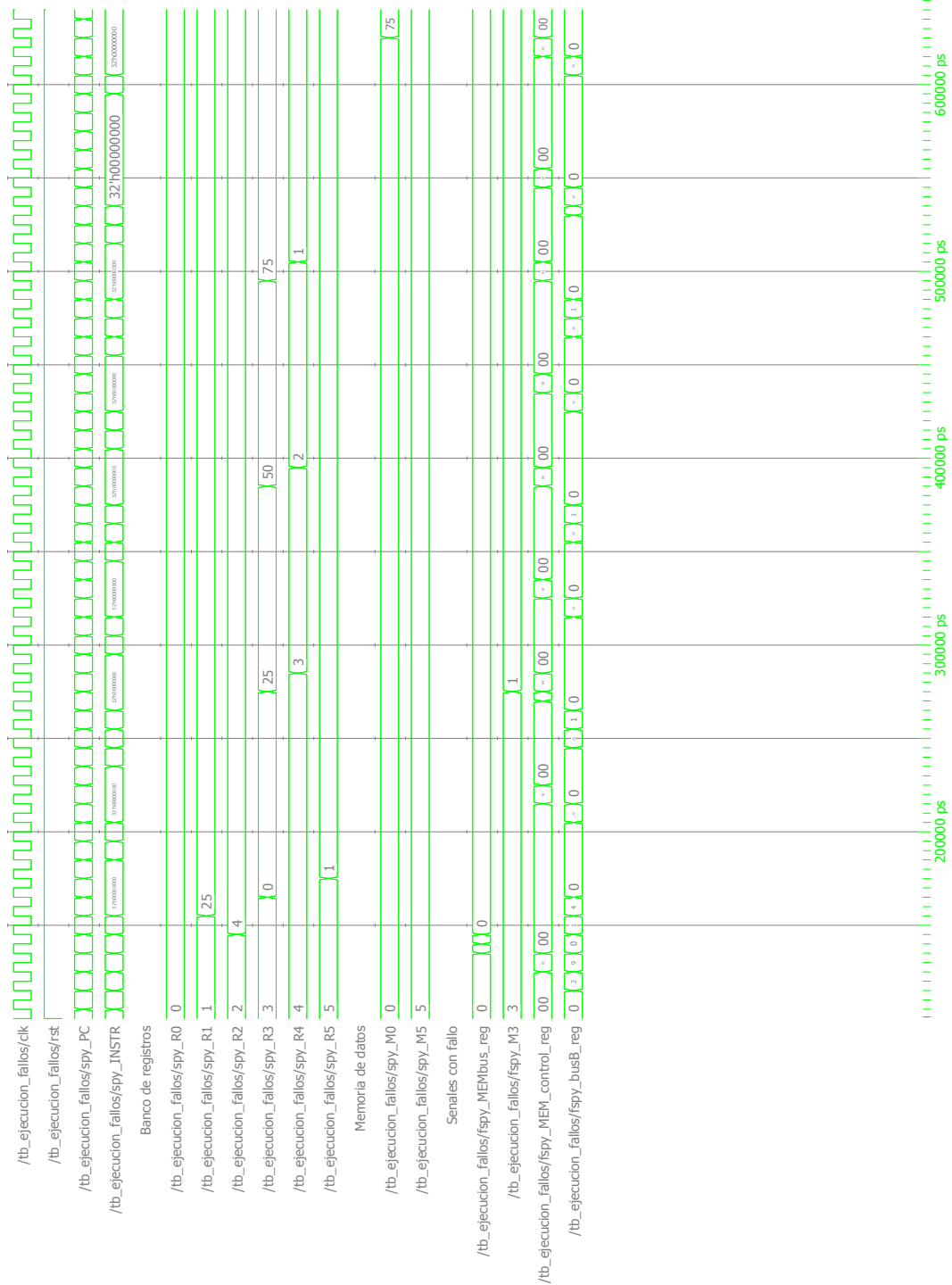
A continuación se incluyen las simulaciones realizadas sobre los procesadores en el siguiente orden:

1. Prueba de control sobre procesador estándar.
2. Prueba con fallos sobre procesador estándar.
3. Prueba de control sobre procesador tolerante a fallos.
4. Prueba con fallos sobre procesador tolerante a fallos.

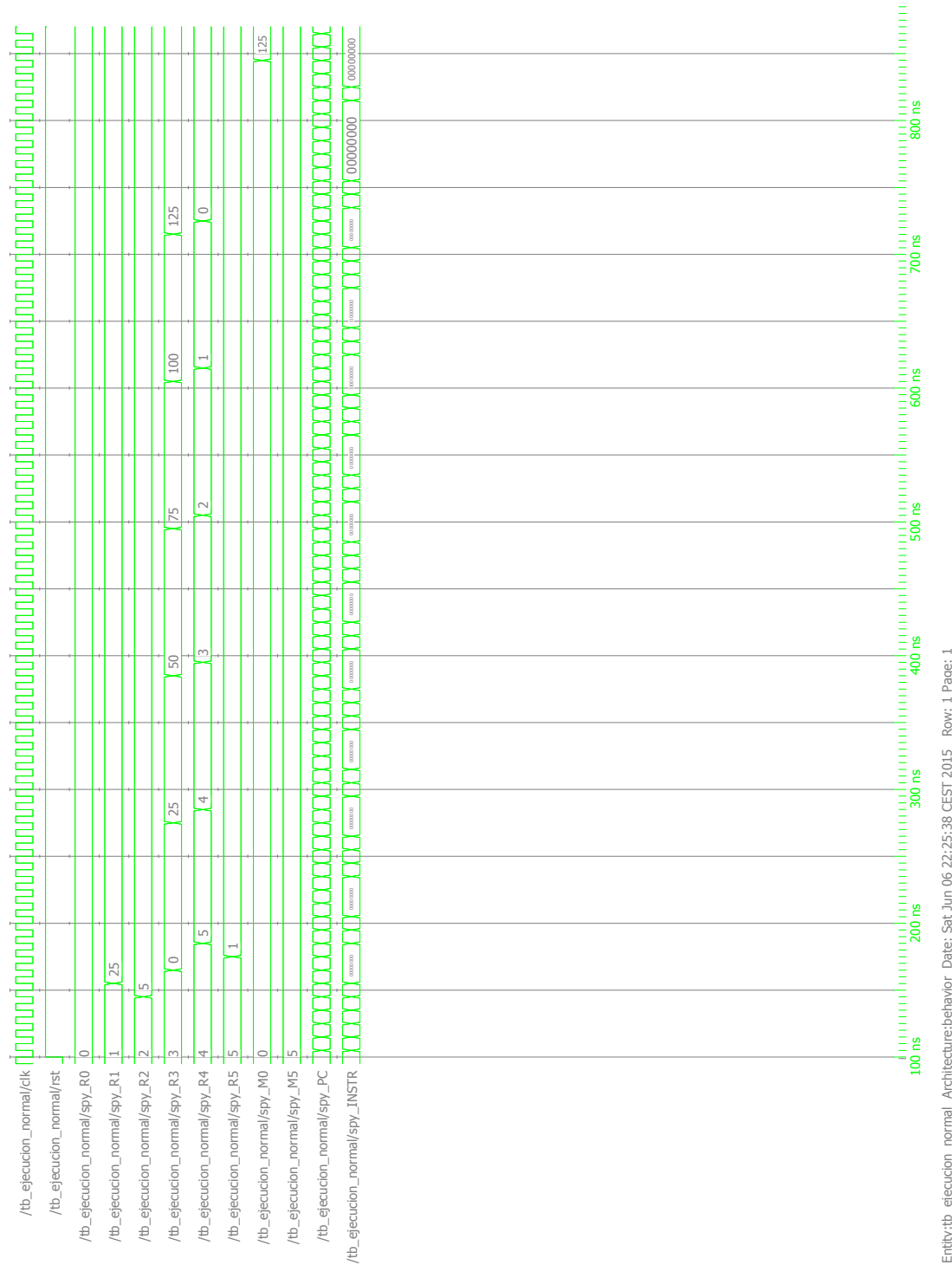
C.1. Procesador estándar sin fallos



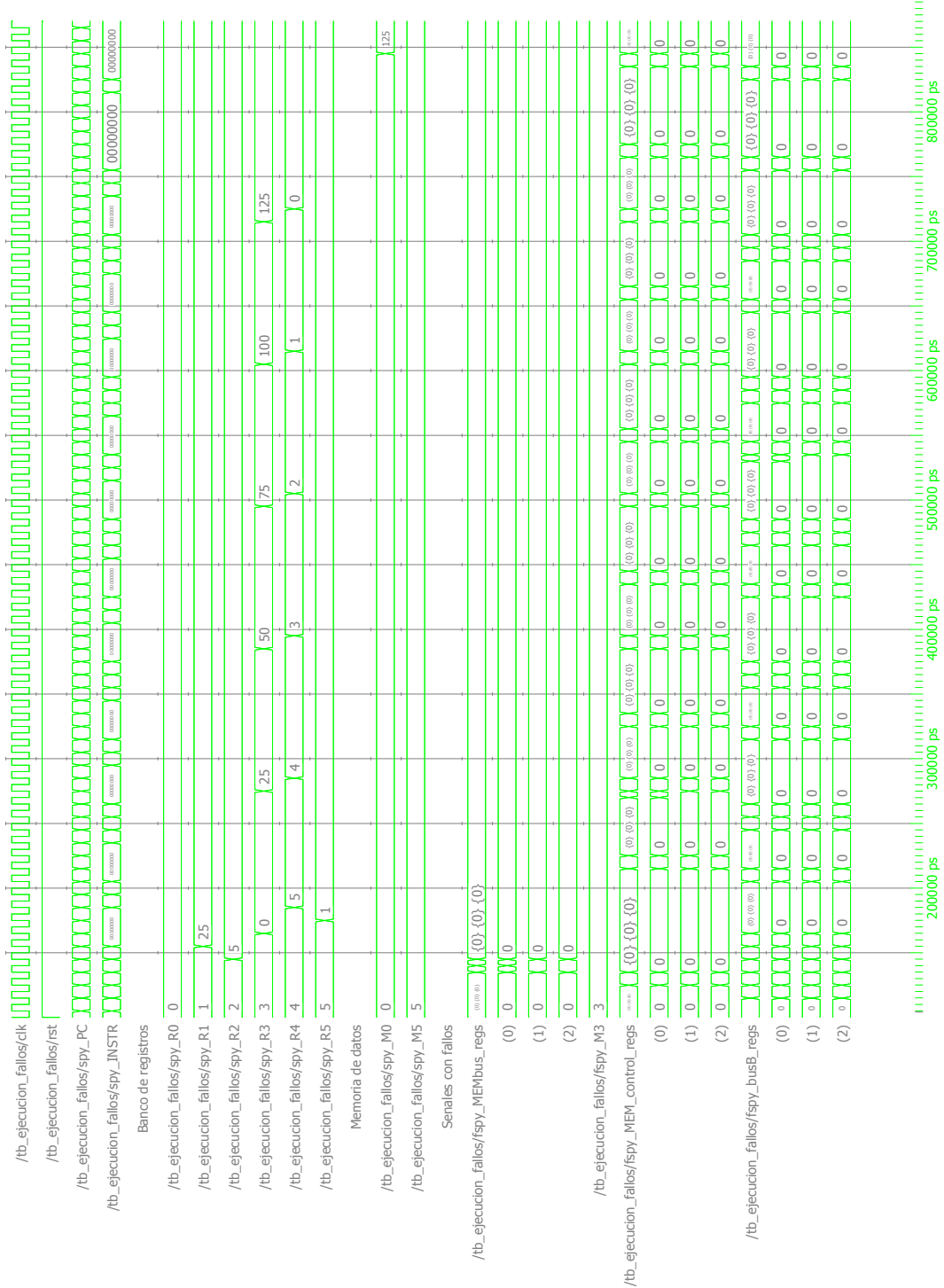
C.2. Procesador estándar con fallos



C.3. Procesador tolerante a fallos sin fallos



C.4. Procesador tolerante a fallos con fallos



Bibliografía

- [1] ARM. ARM Instruction Set. In *ARM7TDMI-S Data Sheet*, chapter 4 ARM Inst.
- [2] ARM. *ARM7TDMI-S*. Number Rev 3. 2000.
- [3] ARM. ARM Architecture Reference Manual, 2007.
- [4] ARM. *ARM Architecture Reference Manual Thumb-2 Supplement*. 2011.
- [5] S. Brown and J. Rose. Architecture of FPGAs and CPLDs: A tutorial. *IEEE Design and Test of Computers*, 13, 1996.
- [6] Digilent. Nexys4 FPGA Board Reference Manual, 2013.
- [7] D. Dye. Partial reconfiguration of Xilinx FPGAs using ISE Design Suite, 2012.
- [8] A. Gaisler. *LEON3 / LEON3-FT CompanionCore Data Sheet*. Number March. 2010.
- [9] J. Gaisler. A portable and fault-tolerant microprocessor based on the SPARC V8 architecture. *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 409–415, 2002.
- [10] J. C. González Salas. *Filtro adaptativo tolerante a fallos*. PhD thesis, 2014.
- [11] S. Habinc. *Functional Triple Modular Redundancy (FTMR)*. 2002.
- [12] J. L. Hennessy and D. A. Patterson. *Arquitectura de Computadores: Un enfoque cuantitativo*. McGraw Hill Editorial, 1993.
- [13] J. L. Hennessy and D. a. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. 2006.
- [14] A. C. Hu and S. Zain. NSEU Mitigation in Avionics Applications, 2010.
- [15] I.N.E. Encuesta sobre equipamiento y uso de tecnologías de información y comunicación en los hogares, 2014.

- [16] A. O. Investigation. *ATSB TRANSPORT SAFETY REPORT Aviation Occurrence Investigation AO-2008-070 Final*. Number October. 2008.
- [17] Jedec. Measurement and Reporting of Alpha Particle and Terrestrial Cosmic Ray Induced Soft Error in Semiconductor Devices, 2006.
- [18] A. Kadav, M. J. Renzelmann, and M. M. Swift. Fine-grained fault tolerance using device checkpoints. *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems - ASPLOS '13*, page 473, 2013.
- [19] H. Kirmann. *Fault Tolerant Computing in Industrial Automation*. 2005.
- [20] I. Kuon, R. Tessier, and J. Rose. *FPGA Architecture: Survey and Challenges*, volume 2. 2007.
- [21] I. M. Martín, J. E. Celador Hernández, and C. T. Bustillos. *Simulador arm en el ámbito docente*. PhD thesis, Universidad Complutense de Madrid, 2012.
- [22] K. W. Melis. *Reconstruction of High-energy Neutrino-induced Particle Showers in KM3NeT*. PhD thesis, Faculteit der Natuurwetenschappen, 2014.
- [23] Qualcomm. Qualcomm Snapdragon 810 Processor.
- [24] J. Rose, A. E. Gamal, and A. Sangiovanni-Vincentelli. Architecture of Field-Programmable Gate Arrays. *Proceeding of the IEEE*, 81(7):1013–1029, 1993.
- [25] E. Rotenberg. AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. *Digest of Papers. Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing (Cat. No.99CB36352)*, 1999.
- [26] D. J. Sorin and S. Ozev. Fault Tolerant Microprocessors for Space Missions.
- [27] J. M. Torrecillas. Tecnología RAID - Tolerancia a Fallos.
- [28] C. Weaver and T. Austin. A fault tolerant approach to microprocessor design. *Proceedings of the International Conference on Dependable Systems and Networks*, (July):411–420, 2001.
- [29] Xilinx. Xilinx Artix-7 Fpgas: a New Performance Standard for Power-Limited, Cost-Sensitive Markets, 2015.