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 TEXIS 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

Resumen

En este proyecto, en primer lugar, se implementará la Unidad Central de Procesamiento (CPU) la cual será capaz de ejecutar un subconjunto de instrucciones del repertorio THUMB-2 propio de algunas arquitecturas ARM.

Los sistemas electrónicos, y los procesadores en particular, son susceptibles de sufrir fallos que interfieren en los procesos que lleven a cabo modificando los datos internos, y en consecuencia se produzcan resultados no deseados. Para evitar que esto suceda se aplicarán técnicas de tolerancia a fallos sobre la CPU.

Para finalizar, se comprobará el funcionamiento del procesador antes y después de añadir los mecanismos de tolerancia a fallos, y en éste ultimo caso se validará que se consigue aumentar el grado de fiabilidad de la CPU.

Abstract

In the first place a Central Processing Unit (CPU) will be designed and implemented. This CPU will be able to execute a sub-set of the THUMB-2 instruction set from the ARM architectures.

Electronic systems, and microprocessors in particular, are susceptible to faults that interfere in the normal execution of the CPU, this faults modify the internal data and produces unwanted results. To avoid this happens, fault tolerant techniques will be applied on the CPU.

Finally, the correct operation of the microprocessor will be tested before and after applying the fault tolerance techniques, for the latter case the fault sensitivity will be tested for an increase in the fiability of the system.

Índice

Índice de figuras

Índice de Tablas

Capítulo 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)» [12].

1.1. Motivación

Hoy en día, el uso de la tecnología y la informática se extienden a nivel mundial, con aplicación en aumento a 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.

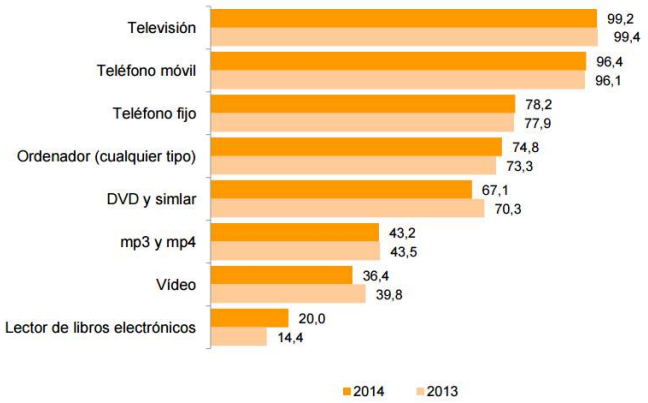


Figura 1.1: Equipamiento de los hogares.[16]

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) [16], 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, contienen con componentes micro-electrónicos. Poseen un microprocesador que es el «cerebro» y responsable de dirigir el sistema ejecutando los programas.

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 o 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.

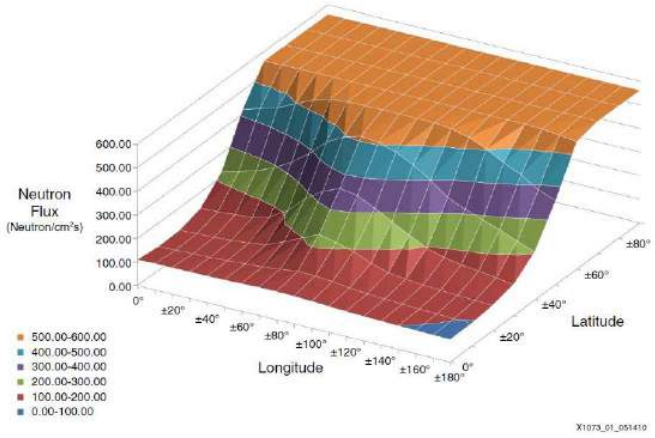


Figura 1.2: Flujo de neutrones a 40.000 pies de altitud [15].

Los fallos, en dispositivos médicos, pueden provocar consecuencias fa-

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

tales 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 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 [17].

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.

1.2. 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.

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.

1.3. 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 divido en tres 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. El juego de instrucciones de 32 bits empleado es un subconjunto de instrucciones de la arquitectura ARM. Simulando unos pequeños programas se comprueba que el procesador es capaz de decodificar y ejecutar las nuevas instrucciones.

2 **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.

3 **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.

1.4. Estructura del documento

Capitulo 1 *Introducción*:

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

Capitulo 2 *Arquitectura de un procesador*:

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

Capitulo 3 *Fallos y tolerancia*:

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

Capitulo 4 *Trabajo realizado*:

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

Capitulo 5 *Aplicando tolerancia a fallos*:

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

Capitulo 6 *Resultados*:

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

Arquitectura de un 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.

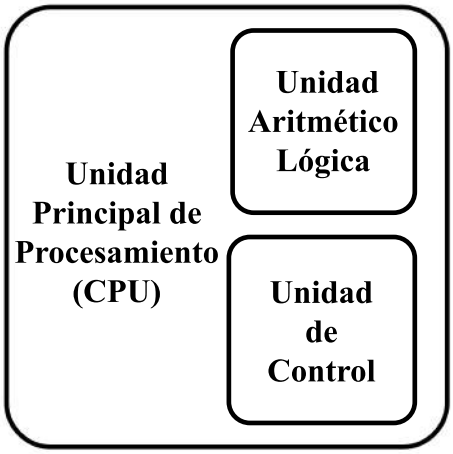


Figura 2.1: Procesador

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. Realizar accesos a memoria.

5. Almacenar los resultados en el banco de registros.

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 necesarios para realizar operaciones aritméticas y lógicas.

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.

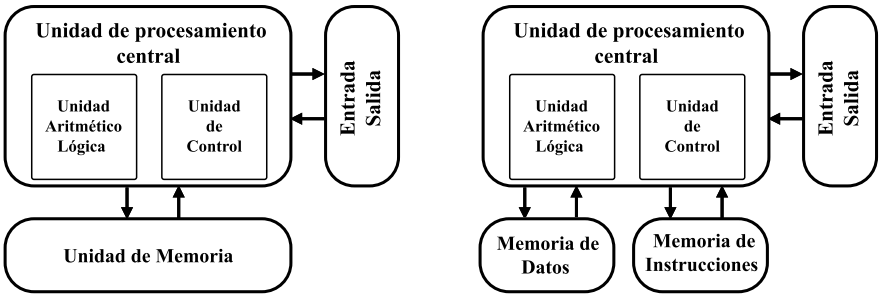


Figura 2.2: Arquitectura Von Neumann y Arquitectura Harvard

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.

	Ciclo de reloj								
Número de instrucción	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:

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

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:

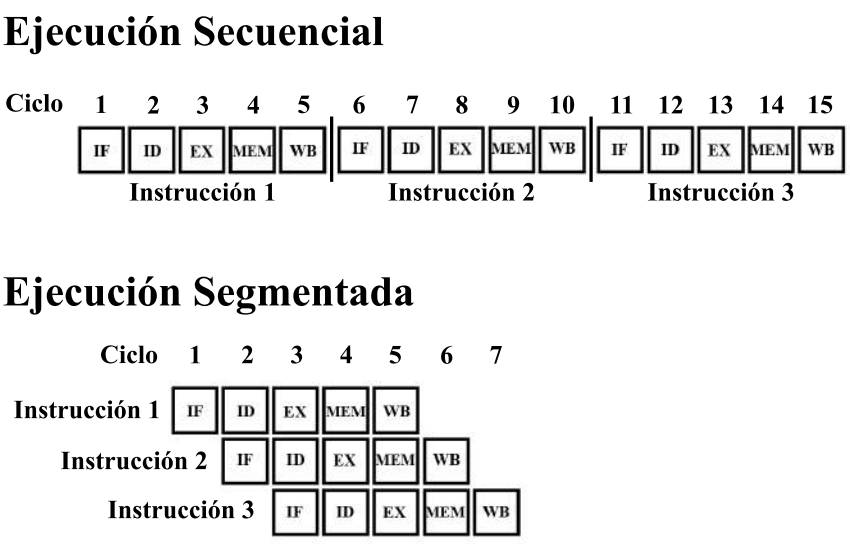


Figura 2.3: Ejecución secuencial comparada con ejecución segmentada

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 para la ejecución de estas tres instrucciones 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: [13]

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).* **No** es un riesgo ya que no se modifican datos.
3. *Riesgos de control.* Surgen a consecuencia de las instrucciones que afectan al registro del contador de programa (PC).

2.2. Procesador 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 [24].

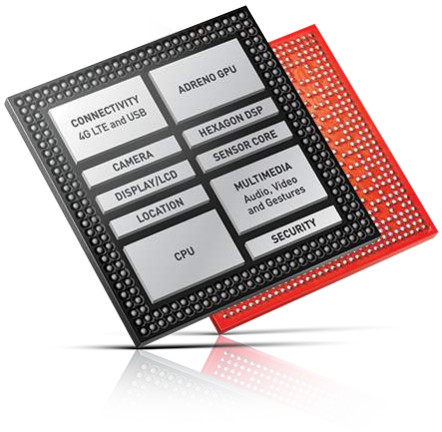


Figura 2.4: Procesador Qualcomm Snapdragon 810. [26]

Por ejemplo, uno de los procesadores de ultima generación de la empresa Qualcomm, el «Qualcomm Snapdragon 810» esta 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 [26]. Figura 2.4.

Además de requerir poco espacio, los diseños de la arquitectura ARM consiguen minimizar el consumo de energía, haciéndolo apropiado para sistemas móviles empotrados[1] 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 [3] 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.

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

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.
- 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 entre otros de uso específico como son los registros de coma flotante. De estos registros de uso general sólo son visibles 16, a los cuales puede acceder cualquier instrucción. Los otros registros se utilizan para acelerar el procesado. 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[2] o Jazelle[3].

- **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.

[2]Conjunto de instrucciones de 16 bits utilizado para reducir el tamaño del código.
[3]Repertorio de instrucciones de tipo «bytecode Java».

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 [1], el set Thumb/Thumb-2[4] [4] y

[4]Extensión de Thumb con instrucciones de 32 bits.

las instrucciones Jazelle [1].

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. [2]

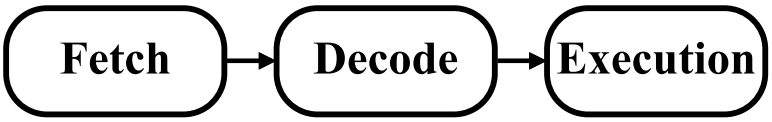


Figura 2.5: Segmentación ARM

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

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. [2]

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 [27]. Figura 2.6.

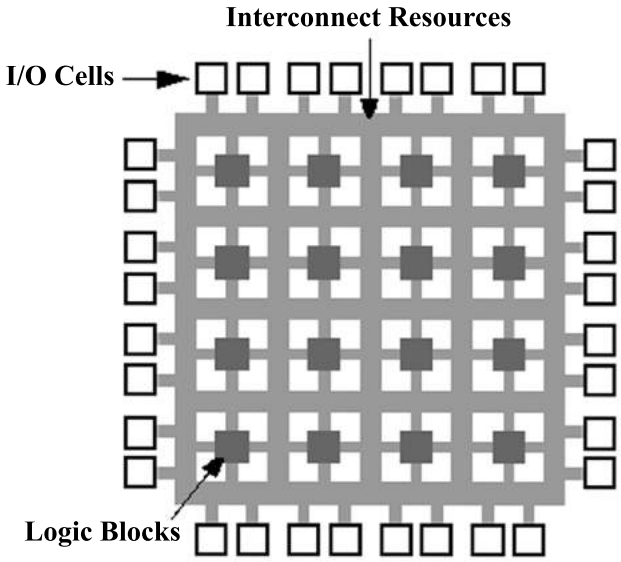


Figura 2.6: Arquitectura de una FPGA [5].

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.

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 [23].

Las ventajas de este tipo de dispositivos residen en su gran versatilidad, flexibilidad, su alta frecuencia de trabajo, su capacidad de procesamiento en

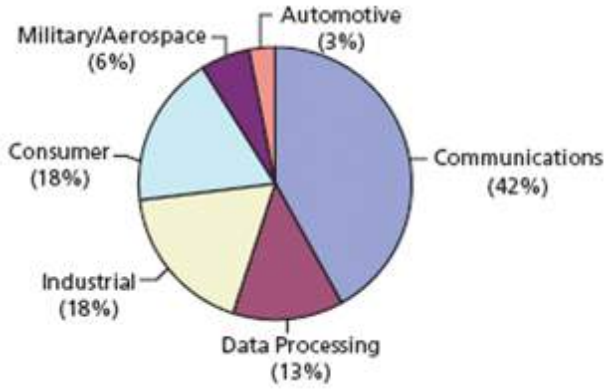


Figura 2.7: «Distribución de las aplicaciones de las FPGAs en el año 2008.» [10]

paralelo, y a su bajo precio en comparación con los «Circuitos Integrados para Aplicaciones Específicas (ASICs)»[5].

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. Placa de desarrollo

Para el desarrollo de este proyecto se ha utilizado la placa de prototipado «Nexys 4» [6], 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).

La FPGA Artix-7 (XC7A100T-1CSG324C) [32], 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 lineas de reloj.
- Velocidad interna de reloj superior a 450MHz.
- 240 bloques DSP.
- Conversor analógico-digital integrado (XADC).

[5]Circuitos integrados hechos a la medida para un uso particular.

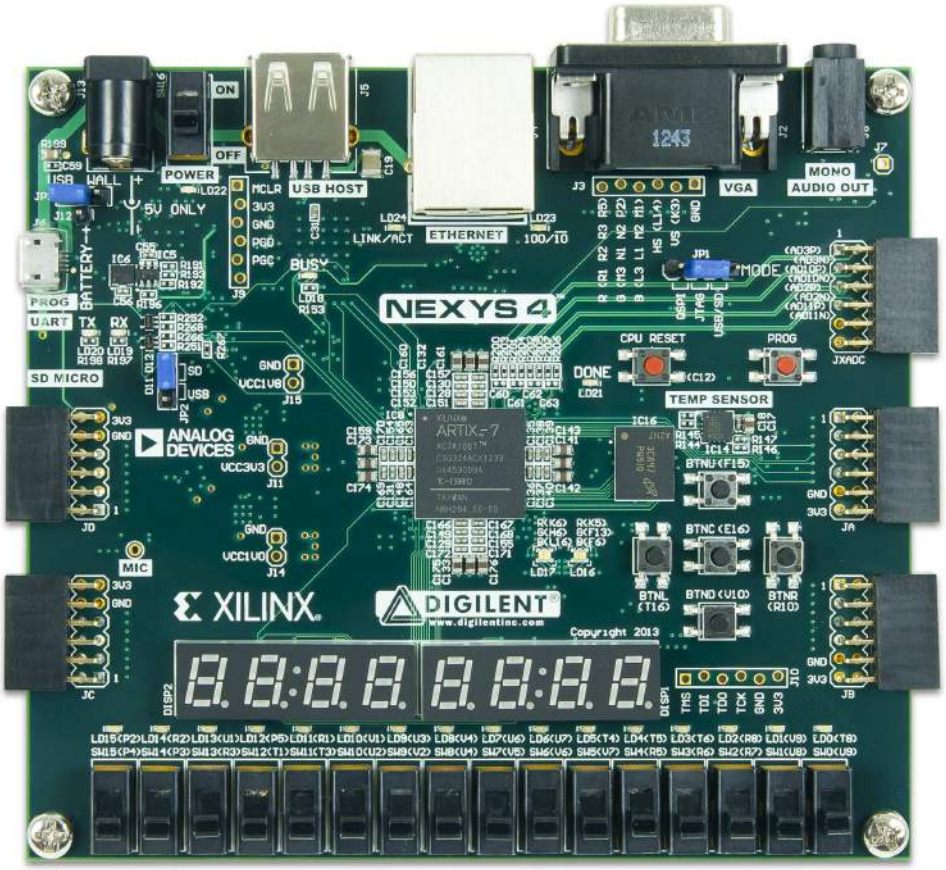


Figura 2.8: Placa de prototipado Nexys 4

Capítulo 3

Fallos y tolerancia

3.1. 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

Entre otros casos podemos destacar el choque de una partícula de energía contra un elemento del circuito integrado. Figura 3.1. En general, los fallos se conocen como *«Single-Event Effects (SEEs)»*.

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 un origen galáctico o solar.
- Los **protones y neutrones de alta energía**: Bien sean de origen

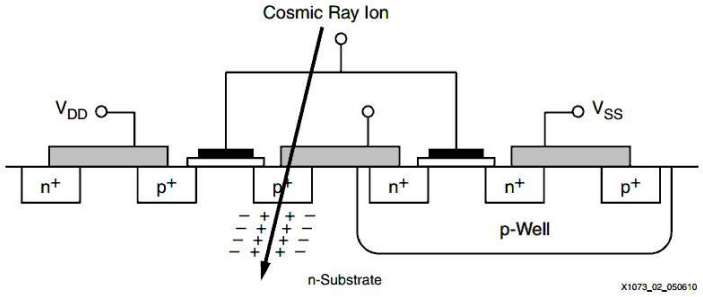


Figura 3.1: Single Event Upset en una FPGA [15].

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.

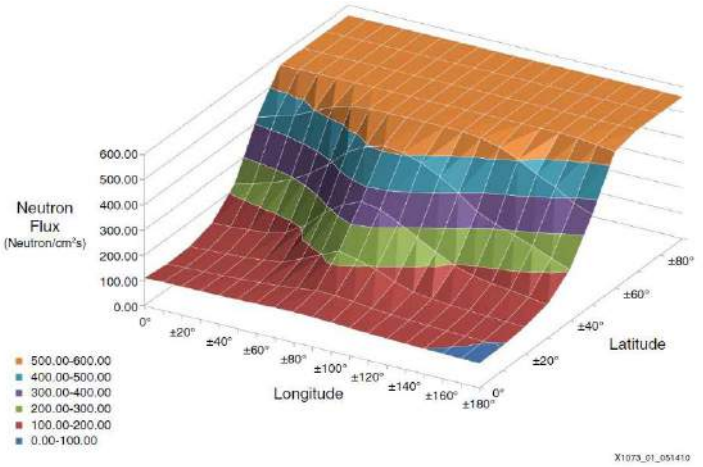


Figura 3.2: Flujo de neutrones a 40.000 pies de altitud [15].

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 numero 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[25].

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 [19]:

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

  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.

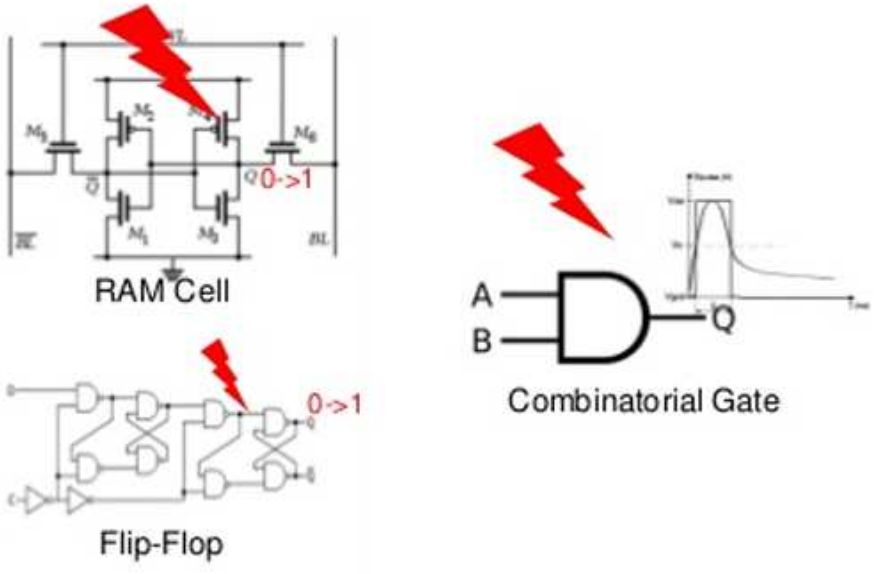
- **Single-Event Functional Interrupt (SEFI)**

  Fallos que producen una perdida 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.



(a) Single-Event Upset (SEU) [18] (b) Single-Event Transient (SET) [18]

Figura 3.3: Fallos Transitorios

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 en el diseño que impiden el correcto funcionamiento del módulo o circuito que lo sufre. [19]

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.

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 sitema 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».

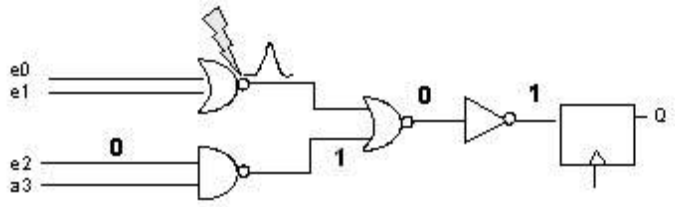


Figura 3.4: Fallo enmascarado por una puerta lógica[21].

- **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.

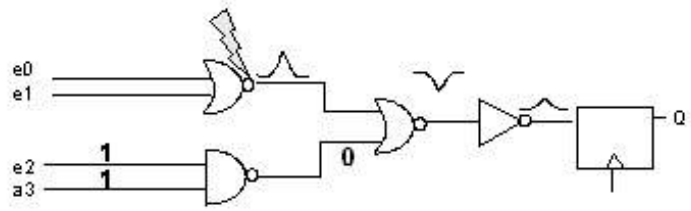


Figura 3.5: Fallo enmascarado eléctricamente[21].

- **Enmascarado temporal**

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

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

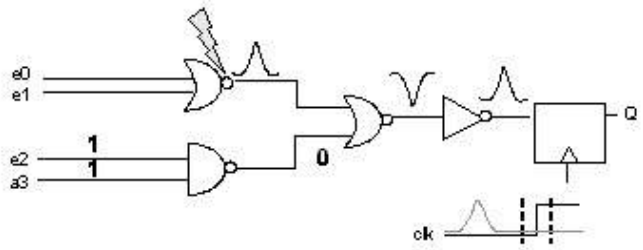


Figura 3.6: Fallo enmascarado por ventana de tiempo[21].

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 perdida o daño de una memoria no implique la perdida 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 [30]. 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». eUna vez se detecta un error se recurre al último «checkpoint» en lugar de tener que reiniciar el programa completo [20].

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, paradógicamente, 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 [22].

3.2.3.1. Redundancia modular

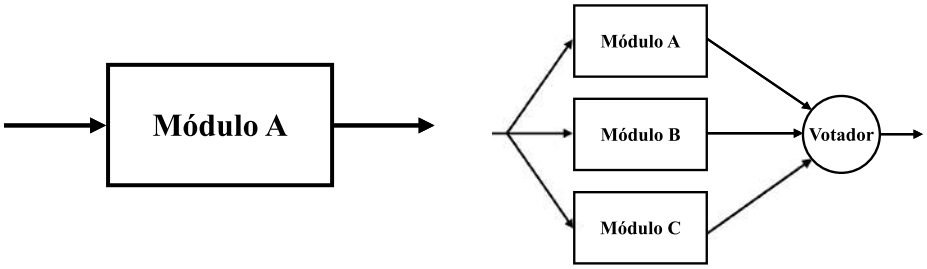
La redundancia modular consiste en replicar N veces el bloque al que se desea aplicar la tolerancia, siendo N un número impar, y a través de una

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.



(a) Módulo original (b) Módulo con TMR

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.

3.3. 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 cotroladores 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 [29]. 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 circuito, evitando tener que modificar la estructura interna del microprocesador, se replica el sistema de comprobación, que normalmente tendrá una implementación más simple que el propio microprocesador. [31]

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 simultanea. 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 proposito general[28].

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» [9].

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: [8]

- **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. [9]

Por último, la implementación de Intel en el «Intel Itanium» incluye una combinación de códigos de correción de errores y códigos de paridad en las memoria caché y TLB. [9]

Capítulo 4

Procesador

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 realizada para 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 Mentor Graphics para las simulaciones. Con ello se ha conseguido probar el diseño y la implementación de forma rápida y realizar las correcciones necesarias.

4.1. Arquitectura del procesador

El microprocesador se ha diseñado para ejecutar instrucciones del repertorio ARM y se ha segmentado en 5 etapas. A continuación se describen sus características completas.

4.1.1. Estructura

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

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

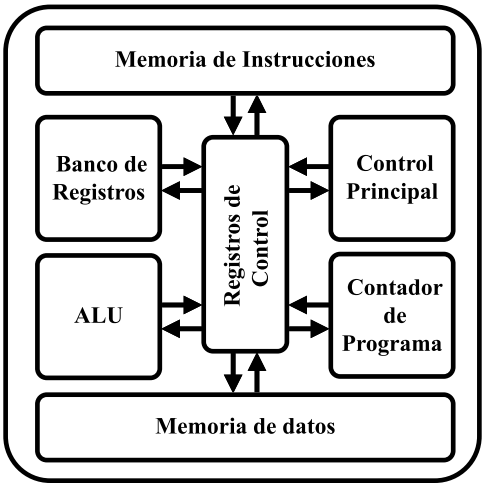


Figura 4.1: Estructura del sistema diseñado (CPU + Memorias).

tro 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.

- **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.

Al sistema se le han incorporado dos memorias junto al procesador para el almacenamiento de instrucciones y datos:

- **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.

4.1.2. 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.

4.1.2.1. 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 Rn + imm12", es decir, la dirección base se obtiene del registro Rn, y se suma un inmediato de 12 bits extraído de la instrucción.

4.1.2.2. 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 Rn es R15 ("1111") entonces la operación ejecutada será la operación "MOVE". Si el registro Rn 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.

4.1.2.3. 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.

4.1.3. Segmentación

El microprocesador se 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 transmitirlas 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.

4.1.4. Memoria

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

- **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.

4.2. Implementación

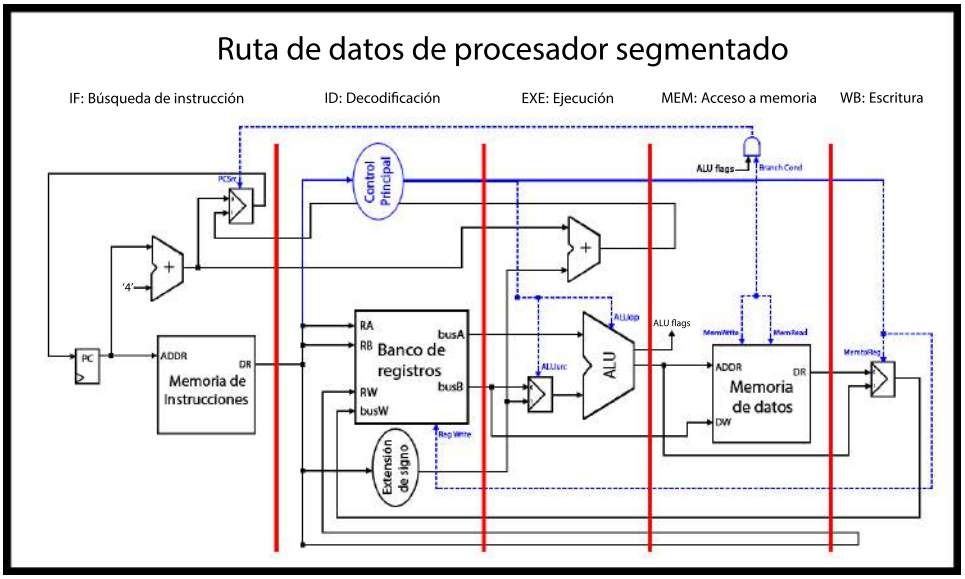


Figura 4.2: Diseño completo del procesador segmentado.

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 lineas continuas se muestra la ruta de datos y los componentes.

En lineas discontinuas se destacan las señales de control generadas en el módulo de control principal. Las líneas verticales representan los registros de control y las divisiones entre etapas.

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

4.2.1. 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».

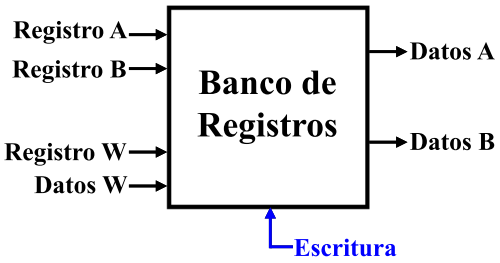


Figura 4.3: Banco de registros.

Sus señales externas son (figura 4.3):

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

4.2.2. 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.

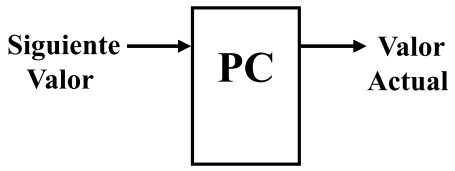


Figura 4.4: Contador de programa.

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.

4.2.3. 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.

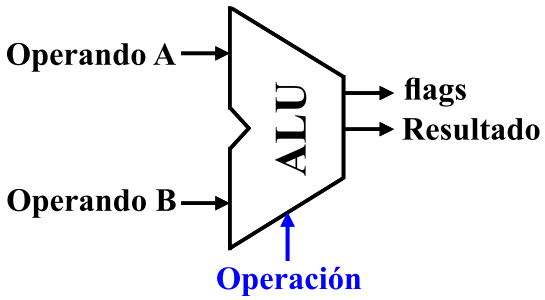


Figura 4.5: Unidad aritmético-lógica.

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 condiciones[1].
  - **Resultado:** Valor de aplicar la operación a los operandos.

4.2.4. 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.

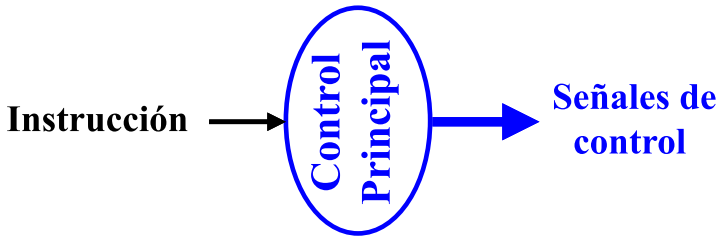


Figura 4.6: Control principal.

La instrucción es la una 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.

[1] Si 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 escritura[2].
   - **MemRead:** Indica al módulo «memoria de datos» si debe acceder a memoria en modo lectura[2].

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.

4.2.5. Registros de control

Los registros de control (figura 4.7) son todos aquellos que almacenan la información entre las etapas, se representan como líneas verticales, que dividen el procesador en sus etapas, en la figura 4.2.

Figura 4.7: Registros de control

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.

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

4.2.6. 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.

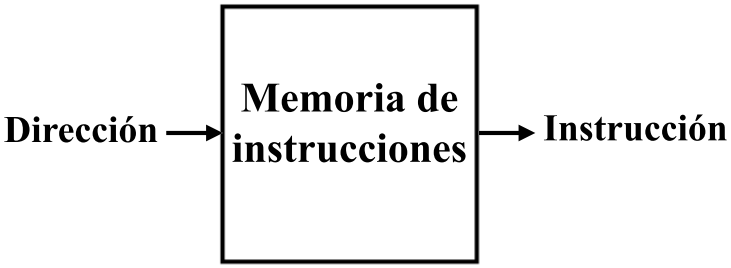


Figura 4.8: Memoria de Instrucciones.

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.

4.2.7. 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.

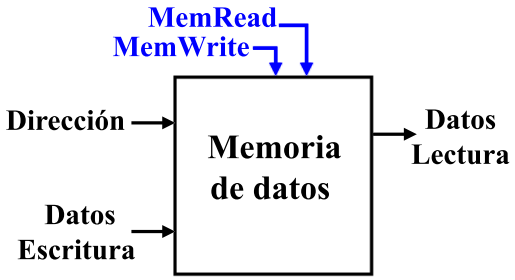


Figura 4.9: Memoria de datos.

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 memoria[3].
  - **MemRead:** Indica que los datos deben leerse de memoria[3].
- **Señales de salida**
  - **Datos Lectura:** Datos que se han leído de la dirección indicada de memoria.

4.3. 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.

4.3.1. 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)

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

	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)

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

- **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.

4.3.2. 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.

4.3.2.1. 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.

	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)

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

- **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.

Operación	Código	Restricciones
ADD	1 0 0 0	
AND	0 0 0 0	
CMP	1 1 0 1	(Rd==«1111»,S==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

4.3.2.2. 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».

	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)

	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)

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.

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

4.3.3. 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:

	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)

- **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 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

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.

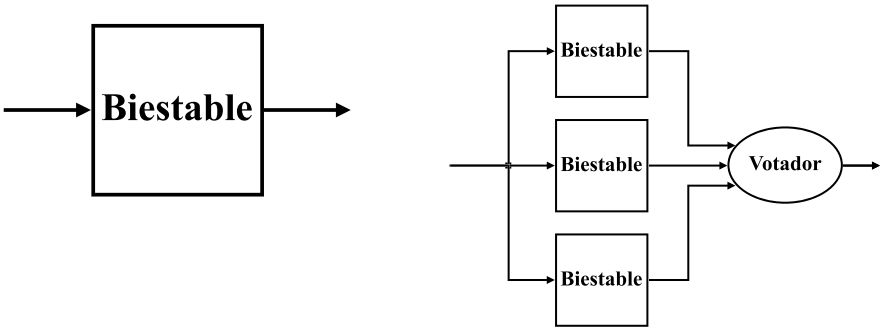
5.1. Redundancia modular

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.

5.2. El Votador

El votador es un circuito combinacional, 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 para otorgar la capacidad de tolerar los fallos transitorios de tipo SEU al sistema.



(a) Biestable original. (b) Conjunto de biestables con votador.

Figura 5.1: Sustitución de biestable.

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 enmascararse 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.

$$Z = (A * B) + (B * C) + (A * C) \tag{5.1}$$

Para exponer cómo funciona el votador se muestran las figuras 5.3a y

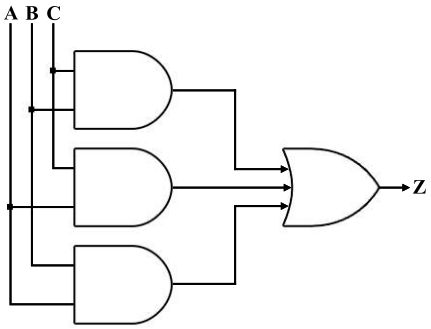
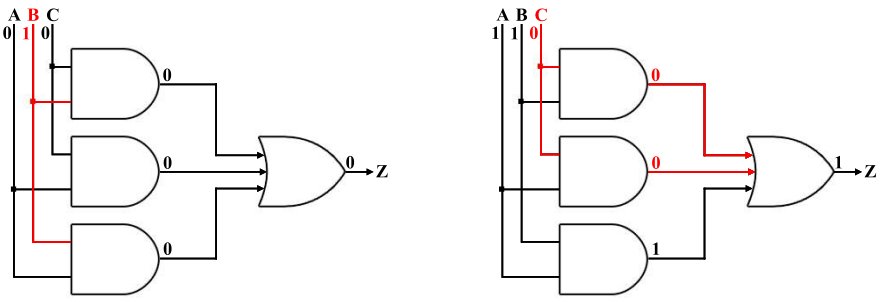


Figura 5.2: Diseño de votador con puertas lógicas

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»$(0 + 0 + 1 = 1)$.



(a) Fallo en entrada B (0 ->1) (b) Fallo en entrada C (1 ->0)

Figura 5.3: Ejemplos de fallos en entradas.

5.2.1. Implementación

La implementación de este módulo se divide en dos secciones:

- **Registros triplicados**

  Al aplicar la 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));
```

5.3. 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».

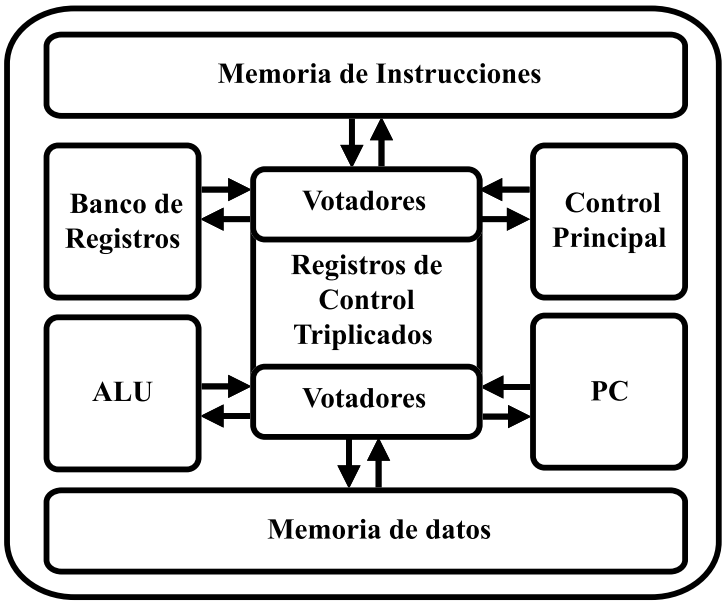


Figura 5.4: Sistema con TMR en los registros de control.

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

Resultados

La herramienta software «ISE Design Suite 14.4» de Xilinx es capaz de sintetizar e implementar el diseño VHDL. Se utilizará esta herramienta para obtener información sobre los recursos utilizados por ambas implementaciones del microprocesador.

Se utilizará esta herramienta para comprobar qué recursos son necesarios 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".

6.1. 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.

6.2. 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.

6.3. Validación del 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.

6.3.1. Recursos empleados

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:

6.3.1.1. 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%
```

Informe 6.1: Recursos utilizados por el procesador estándar

6.3.1.2. 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:   6.435ns{1}   (Maximum frequency: 155.400MHz
       )
```

Informe 6.2: Restricciones de tiempo del procesador estándar

6.3.2. Ejecución de control

Se procede a realizar la primera prueba, o prueba de control, que consiste en ejecutar el programa explicado en la sección 6.1 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.

6.3.3. 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:

```
# ** Error: ERROR: LDR R2, R0, #5
#    Time: 150 ns  Iteration: 0  Instance: /tb_ejecucion_fallos
# ** Error: ERROR: MOV R4, R2
#    Time: 190 ns  Iteration: 0  Instance: /tb_ejecucion_fallos
# ** Error: ERROR: SUB R4, R4, R5(4)
#    Time: 300 ns  Iteration: 0  Instance: /tb_ejecucion_fallos
# ** Error: ERROR: SUB R4, R4, R5(3)
#    Time: 410 ns  Iteration: 0  Instance: /tb_ejecucion_fallos
# ** Error: ERROR: SUB R4, R4, R5(2)
#    Time: 520 ns  Iteration: 0  Instance: /tb_ejecucion_fallos
# ** Error: ERROR:PC 204
#    Time: 620 ns  Iteration: 0  Instance: /tb_ejecucion_fallos
# ** Error: ERROR: ADD R3, R3, R1(100)
#    Time: 620 ns  Iteration: 0  Instance: /tb_ejecucion_fallos
# ** Error: ERROR: PC 208
#    Time: 630 ns  Iteration: 0  Instance: /tb_ejecucion_fallos
# ** Error: ERROR: PC 248
#    Time: 730 ns  Iteration: 0  Instance: /tb_ejecucion_fallos
# ** Error: ERROR: ADD R3, R3, R1(125)
#    Time: 730 ns  Iteration: 0  Instance: /tb_ejecucion_fallos
# ** Error: ERROR: PC 252
#    Time: 740 ns  Iteration: 0  Instance: /tb_ejecucion_fallos
# ** Error: ERROR: SUB R4, R4, R5(0)
#    Time: 740 ns  Iteration: 0  Instance: /tb_ejecucion_fallos
# ** Error: ERROR: PC 296
#    Time: 860 ns  Iteration: 0  Instance: /tb_ejecucion_fallos
# ** Error: ERROR: STR R3, R0, #0
#    Time: 860 ns  Iteration: 0  Instance: /tb_ejecucion_fallos
```

Informe 6.3: Errores detectados tras la inserción de fallos.

6.4. Validación del 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.

6.4.1. Recursos empleados

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

```
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
       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
       63,400    2%
12     Number used as logic:                    1,464 out of
         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
         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%
```

Informe 6.4: Recursos utilizados por el procesador tolerante a fallos

6.4.1.2. 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
          )
```

Informe 6.5: Restricciones de tiempo del procesador tolerante a fallos

6.4.2. 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.

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

Capítulo 7

Análisis de los resultados

7.1. Comparativa de procesadores

El procesador estándar requiere un total de 1,952 registros para ser utilizados como biestables, lo que supone aproximadamente el 1,5 % del total disponible (informe 6.1). De los registros 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 (informe 6.1).

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 (informe 6.2).

El procesador tolerante a fallos se deriva del procesador estándar aplicando la técnica TMR sobre los registros de control entre etapas. Al incluir lógica adicional, 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 (informe 6.4). 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 (informe 6.4). 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 (informe 6.5), lo que significa una reducción de la frecuencia máxima de funcionamiento en casi 10 MHz.

7.2. Simulaciones

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

7.2.1. Procesador estándar

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

7.2.1.1. 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.

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.

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.

7.2.1.2. Prueba con fallos

Al volver a ejecutar el mismo código de prueba, esta vez insertando fallos (Apéndice C, sección C.2). Por texto (informe 6.3) se informa que los errores comenzaron al ejecutar la instrucción «$LDR\ R2, R0, \#5$» y se han propagado hasta el final de la ejecución del programa. En la gráfica 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 (MEM-bus_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.

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.

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.

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.

7.2.2. Procesador tolerante a fallos

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

7.2.2.1. 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 del procesador se conservan al introducir la tolerancia a fallos.

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 .

7.2.2.2. 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.

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.

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 que siempre está activo.

Capítulo 8

Conclusiones

8.1. Conclusiones

Se ha diseñado una Unidad Central de Proceso (CPU) segmentada en 5 etapas, capaz de ejecutar un conjunto reducido de instrucciones conocido como THUMB-2, concretamente es un subconjunto de la arquitectura ARM. Esta CPU ejecuta correctamente instrucciones aritmético-lógicas, instrucciones de acceso a memoria e instrucciones de bifurcación o salto.

Una vez obtenido el procesador con un repertorio de instrucciones capaz de ejecutar programas sencillos, se ha aumentado el grado de fiabilidad de la CPU diseñada aplicando la técnica de tolerancia a fallos «Triple Modular Redundancy» (TMR). Se han triplicado los registros que almacenan información entre las etapas de la CPU segmentada e insertado votadores de mayoría para obtener los valores correctos.

Para finalizar se ha demostrado, a base de simulaciones sobre el procesador diseñado en este proyecto, que al aplicar técnicas como la TMR en la CPU se han conseguido tolerar fallos que habrían alterado el funcionamiento del programa. El mecanismo de tolerancia aplicado ha permitido tolerar un fallo en cada elemento triplicado.

Capítulo 9

Conclusions

9.1. Conclusions

A Central Processing Unit (CPU) has been developed. This CPU has a 5 stage pipeline and is capable of executing a reduce instruction set knwon as THUMB-2, this is a sub-set of ARM architecture instruction set. This CPU correctly executes arithmetic and logic instructions, memory access instructions and branch instructions.

Once the microprocessor was capable of running simple programs, it's dependecy level was enhanced by applying fault tolerance techniques, specifically the «Triple Modular Redundancy» (TMR) technique. The pipeline registers have been tripled and majority voters have been inserted to obtain the correct values.

Finally, it has been displayed, based on simulations performed on the CPU developed, that applying fault tolerance techniques as TMR it is possible to tolerate faults that would cause malfunction on the system. The fault tolerance mechanism applied allows to mask one fault in each tripled element.

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 espía[1] 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.

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

A.1. Testbench para ejecución de control

```
1  [...]
2
3     -- Test de funcionamiento
4     spy_proc: process
5     begin
6
7  [...]
8
9        assert spy_PC = std_logic_vector(to_unsigned(0, 32))
             report "ERROR: Inicio" severity ERROR;
10    wait for clk_period*5; -- Espera para que tengan efecto los
          cambios (Procesador de cinco ciclos)
11 --LDR R2, R0, #5
12       assert spy_PC = std_logic_vector(to_unsigned(20, 32))
             report "ERROR: PC 20" severity ERROR;
13       assert spy_R2 = std_logic_vector(to_unsigned(5, 32))
             report "ERROR: LDR R2, R0, #5" severity ERROR;
14    wait for clk_period;
15 --MOV R1, #25
16       assert spy_PC = std_logic_vector(to_unsigned(24, 32))
             report "ERROR: PC 24" severity ERROR;
17       assert spy_R1 = std_logic_vector(to_unsigned(25, 32))
             report "ERROR: MOV R1, #25" severity ERROR;
18    wait for clk_period;
19 --MOV R3, #0
20       assert spy_PC = std_logic_vector(to_unsigned(28, 32))
             report "ERROR: PC 28" severity ERROR;
21       assert spy_R3 = std_logic_vector(to_unsigned(0, 32))
             report "ERROR: MOV R3, #0" severity ERROR;
22    wait for clk_period;
23 --MOV R5, #1
24       assert spy_PC = std_logic_vector(to_unsigned(32, 32))
             report "ERROR: PC 32" severity ERROR;
25       assert spy_R5 = std_logic_vector(to_unsigned(1, 32))
             report "ERROR: MOV R5, #1" severity ERROR;
26    wait for clk_period;
27
28 [...]
29
30 --STR R3, R0, #0
31    wait for clk_period*12;
32       assert spy_PC = std_logic_vector(to_unsigned(96, 32))
             report "ERROR: PC 296" severity ERROR;
33       assert spy_M0 = std_logic_vector(to_unsigned(125, 32))
             report "ERROR: STR R3, R0, #0" severity ERROR;
34    wait;
35    end process;
36
37 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 espía[1] 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.

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

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

```
1  [...]
2
3     -- Test de funcionamiento
4     spy_proc: process
5     begin
6
7  [...]
8
9        assert spy_PC = std_logic_vector(to_unsigned(0, 32))
             report "ERROR: Inicio" severity ERROR;
10    wait for clk_period*5; -- Espera para que tengan efecto los
          cambios (Procesador de cinco ciclos)
11 --LDR R2, R0, #5
12       assert spy_PC = std_logic_vector(to_unsigned(20, 32))
             report "ERROR: PC 20" severity ERROR;
13       assert spy_R2 = std_logic_vector(to_unsigned(5, 32))
             report "ERROR: LDR R2, R0, #5" severity ERROR;
14    wait for clk_period;
15 --MOV R1, #25
16       assert spy_PC = std_logic_vector(to_unsigned(24, 32))
             report "ERROR: PC 24" severity ERROR;
17       assert spy_R1 = std_logic_vector(to_unsigned(25, 32))
             report "ERROR: MOV R1, #25" severity ERROR;
18    wait for clk_period;
19 --MOV R3, #0
20       assert spy_PC = std_logic_vector(to_unsigned(28, 32))
             report "ERROR: PC 28" severity ERROR;
21       assert spy_R3 = std_logic_vector(to_unsigned(0, 32))
             report "ERROR: MOV R3, #0" severity ERROR;
22    wait for clk_period;
23 --MOV R5, #1
24       assert spy_PC = std_logic_vector(to_unsigned(32, 32))
             report "ERROR: PC 32" severity ERROR;
25       assert spy_R5 = std_logic_vector(to_unsigned(1, 32))
             report "ERROR: MOV R5, #1" severity ERROR;
26    wait for clk_period;
27
28 [...]
29
30 --STR R3, R0, #0
31    wait for clk_period*12;
32       assert spy_PC = std_logic_vector(to_unsigned(96, 32))
             report "ERROR: PC 296" severity ERROR;
33       assert spy_M0 = std_logic_vector(to_unsigned(125, 32))
             report "ERROR: STR R3, R0, #0" severity ERROR;
34    wait;
35    end process;
36 [...]
```

```
37 [...]
38
39    -- Inserción de fallos
40    fault_proc: process
41    begin
42       -- Fallo 1: Forzar Data read de memoria (bit 0) a 0 (Data
              read != 5 => Data read = 4)
43       signal_force("/TB_ejecucion_fallos/uut/MEM_out_MEMbus_reg
              (0)", "0", 140 ns, freeze, 145 ns, 1); -- f_MEMbus
44
45       -- Fallo 2: Forzar Data read de memoria (bit 0) a 0 (Data
              read != 5 => Data read = 4)
46       signal_force("/TB_ejecucion_fallos/uut/
              EXE_out_MEM_control_reg(0)", "1", 270 ns, freeze, 275
              ns, 1);
47
48       -- Fallo 3: Forzar Data read de memoria (bit 0) a 0 (Data
              read != 5 => Data read = 4)
49       signal_force("/TB_ejecucion_fallos/uut/ID_out_busB_reg(0)"
              , "1",  530 ns, freeze, 535 ns, 1);
50
51     wait;
52    end process;
53
54 END;
```

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

```
55 [...]
56
57    -- Test de funcionamiento
58    spy_proc: process
59    begin
60
61 [...]
62
63        assert spy_PC = std_logic_vector(to_unsigned(0, 32))
              report "ERROR: Inicio" severity ERROR;
64    wait for clk_period*5; -- Espera para que tengan efecto los
          cambios (Procesador de cinco ciclos)
65 --LDR R2, R0, #5
66        assert spy_PC = std_logic_vector(to_unsigned(20, 32))
              report "ERROR: PC 20" severity ERROR;
67        assert spy_R2 = std_logic_vector(to_unsigned(5, 32))
              report "ERROR: LDR R2, R0, #5" severity ERROR;
68    wait for clk_period;
69 --MOV R1, #25
70        assert spy_PC = std_logic_vector(to_unsigned(24, 32))
              report "ERROR: PC 24" severity ERROR;
71        assert spy_R1 = std_logic_vector(to_unsigned(25, 32))
              report "ERROR: MOV R1, #25" severity ERROR;
72    wait for clk_period;
73 --MOV R3, #0
74        assert spy_PC = std_logic_vector(to_unsigned(28, 32))
              report "ERROR: PC 28" severity ERROR;
75        assert spy_R3 = std_logic_vector(to_unsigned(0, 32))
              report "ERROR: MOV R3, #0" severity ERROR;
76    wait for clk_period;
77 --MOV R5, #1
78        assert spy_PC = std_logic_vector(to_unsigned(32, 32))
              report "ERROR: PC 32" severity ERROR;
79        assert spy_R5 = std_logic_vector(to_unsigned(1, 32))
              report "ERROR: MOV R5, #1" severity ERROR;
80    wait for clk_period;
81
82 [...]
83
84 --STR R3, R0, #0
85    wait for clk_period*12;
86        assert spy_PC = std_logic_vector(to_unsigned(96, 32))
              report "ERROR: PC 296" severity ERROR;
87        assert spy_M0 = std_logic_vector(to_unsigned(125, 32))
              report "ERROR: STR R3, R0, #0" severity ERROR;
88    wait;
89    end process;
90 [...]
```

```
91  [...]
92
93     -- Inserción de fallos
94     fault_proc: process
95     begin
96        -- Fallo 1: Forzar Data read de memoria (bit 0) a 0 (Data
              read != 5 => Data read = 4)
97        signal_force("/TB_ejecucion_fallos/uut/r_MEM_out_MEMbus/
              regs(0)(0)", "0", 140 ns, freeze, 145 ns, 1); --
              f_MEMbus
98
99        -- Fallo 2: Forzar Data read de memoria (bit 0) a 0 (Data
              read != 5 => Data read = 4)
100       signal_force("/TB_ejecucion_fallos/uut/
              r_EXE_out_MEM_control/regs(0)(0)", "1", 270 ns, freeze,
               275 ns, 1);
101
102       -- Fallo 3: Forzar Data read de memoria (bit 0) a 0 (Data
              read != 5 => Data read = 4)
103       signal_force("/TB_ejecucion_fallos/uut/r_ID_out_busB_reg/
              regs(0)(0)", "1",  530 ns, freeze, 535 ns, 1);
104    wait;
105    end process;
106
107 END;
```

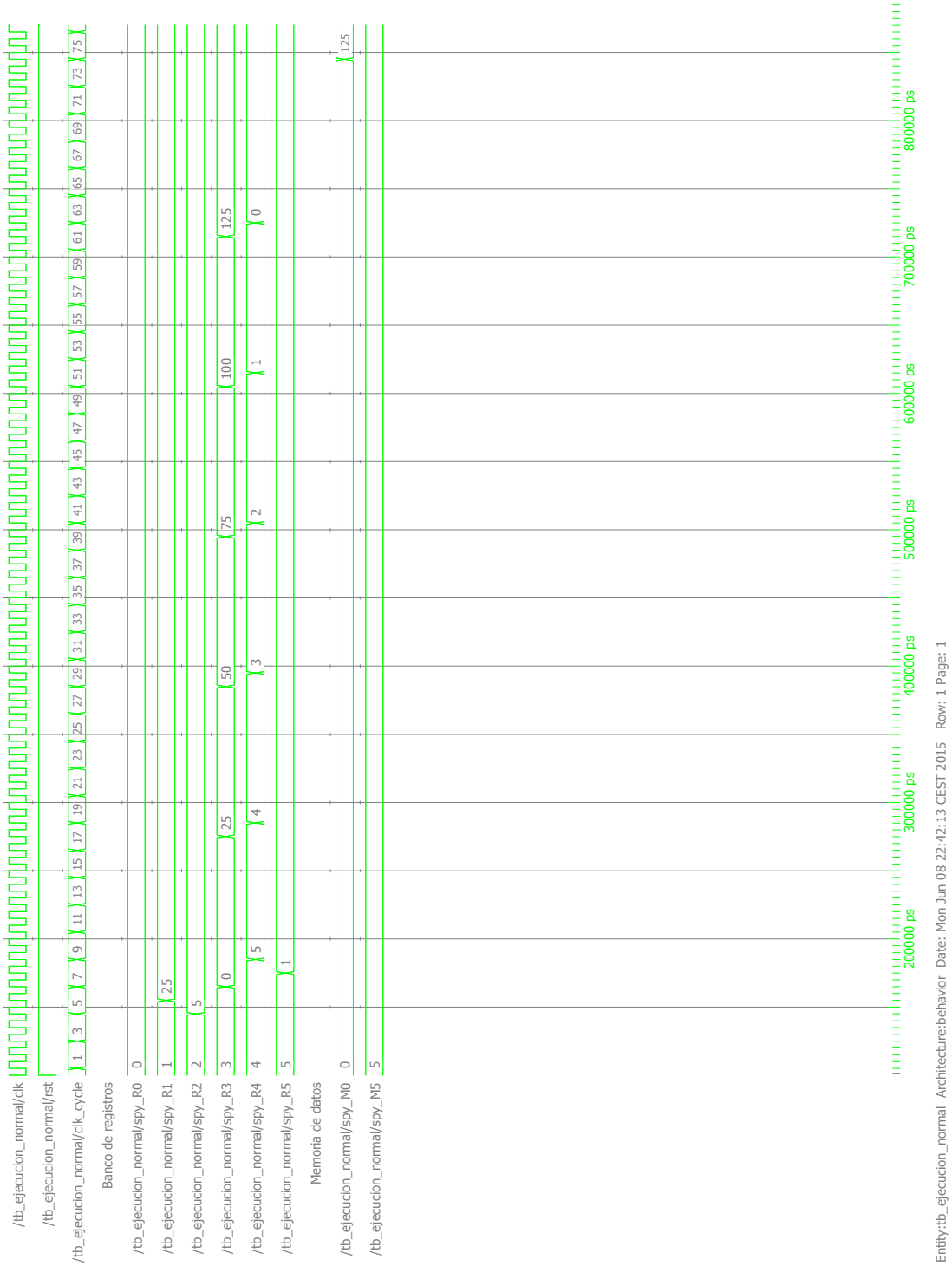
Apéndice C

Simulaciones

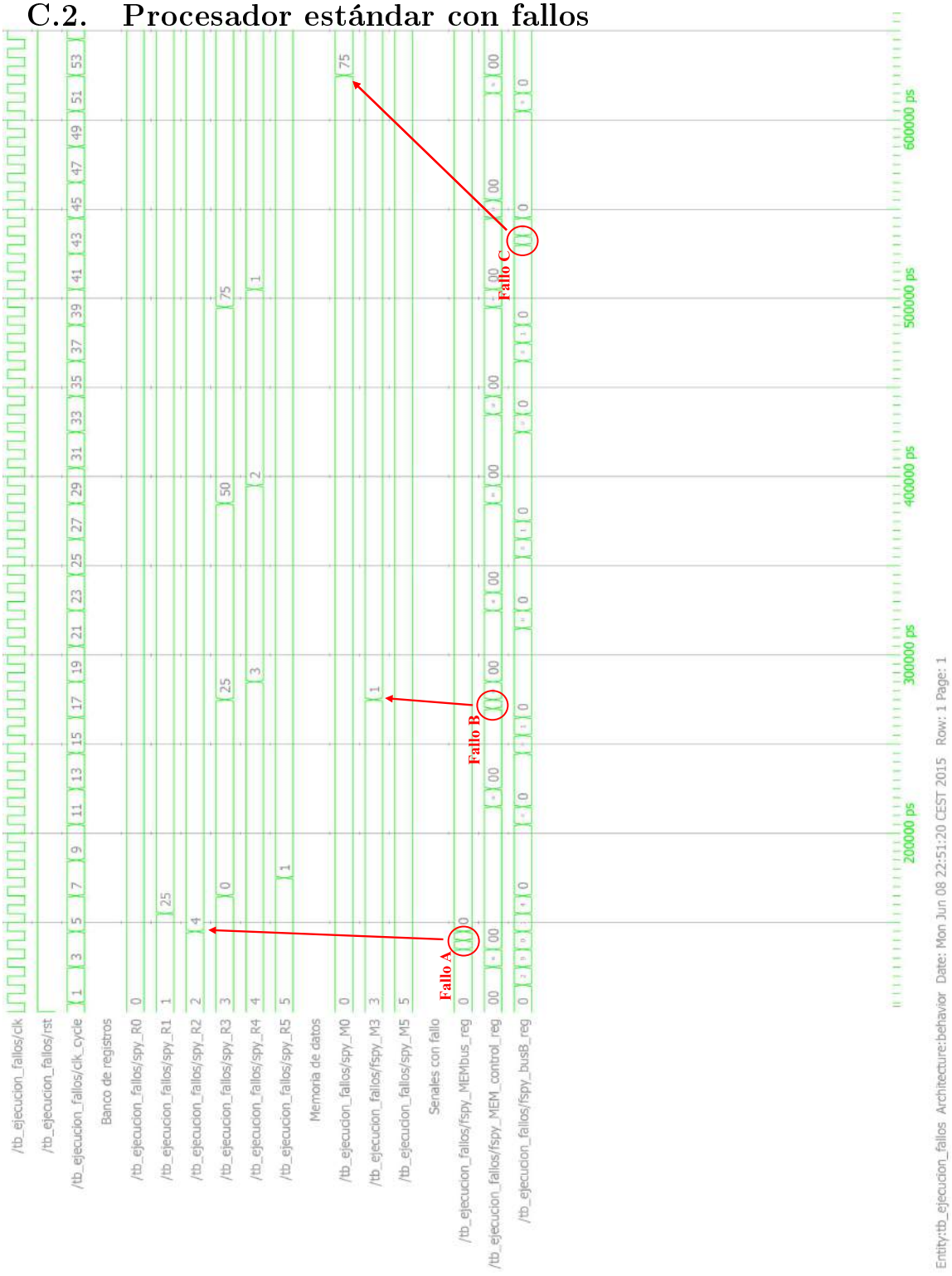
A continuación se incluyen las simulaciones realizadas sobre los procesadores en el siguiente orden:

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.

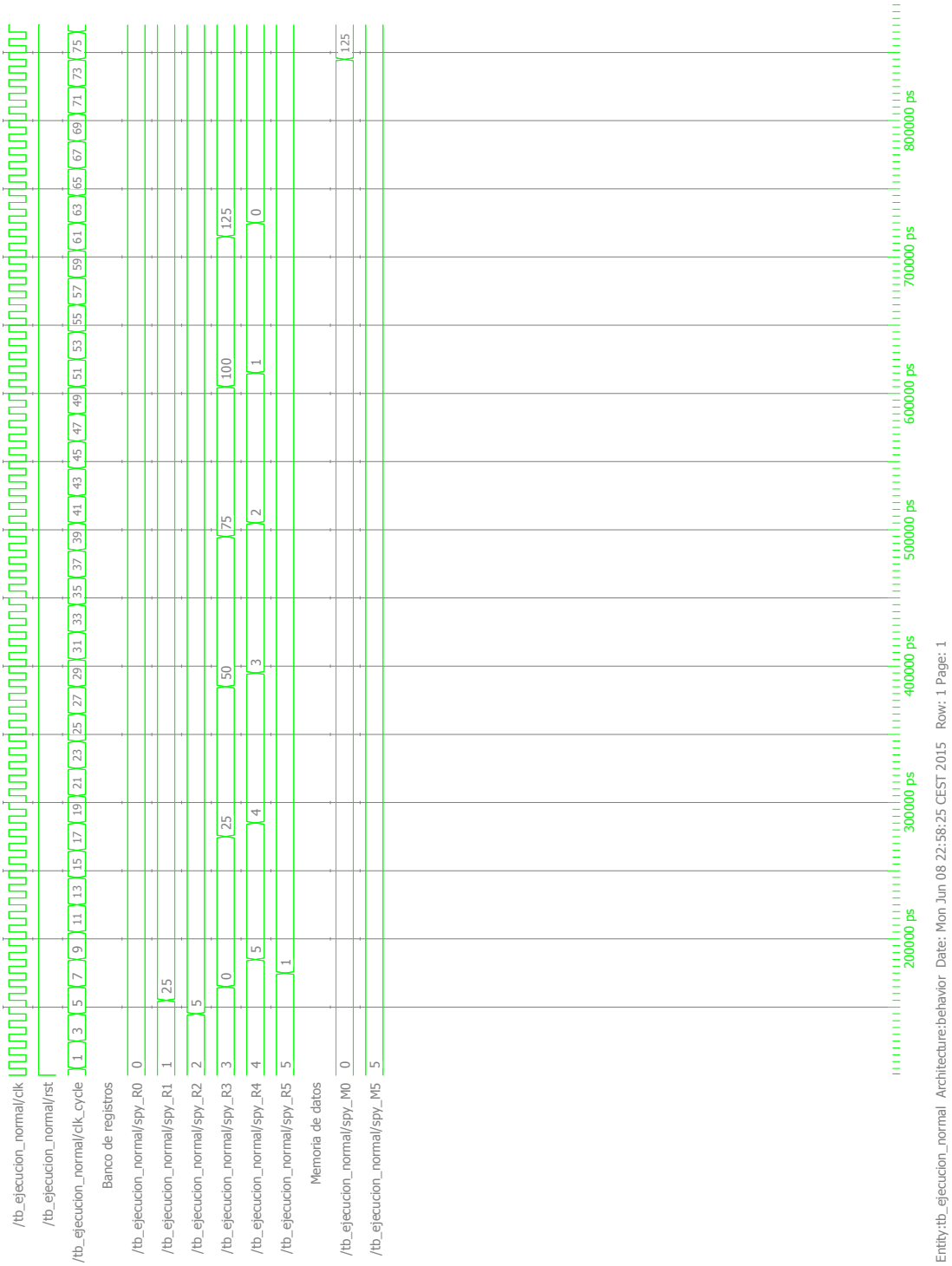
C.1. Procesador estándar sin fallos



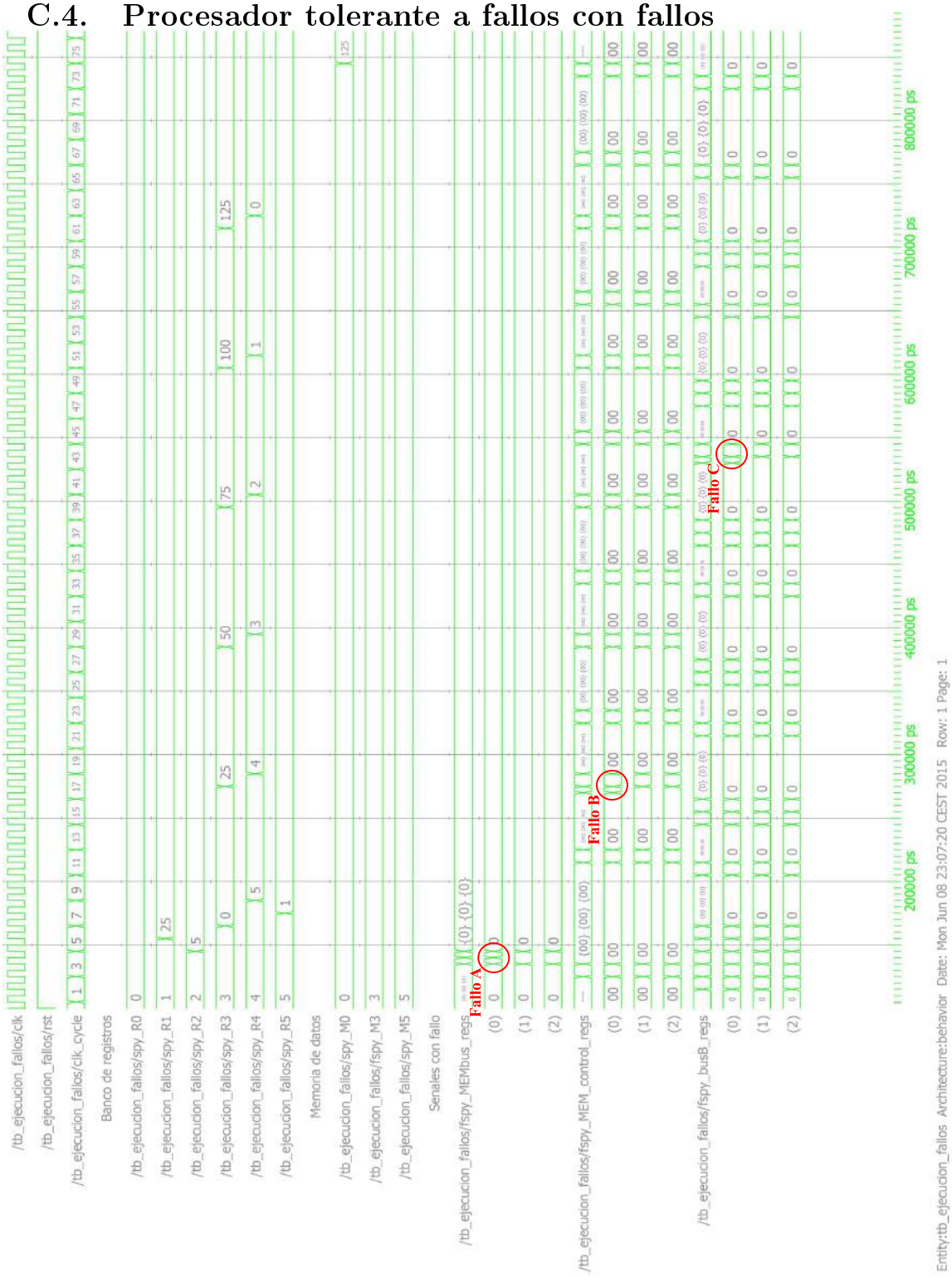
C.2. Procesador estándar con fallos



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 Xilix 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] GENERA. http://www.generatecnologias.es/aplicaciones_fpga.html.

[11] J. C. González Salas. *Filtro adaptativo tolerante a fallos*. PhD thesis, 2014.

[12] S. Habinc. *Functional Triple Modular Redundancy (FTMR)*. 2002.

[13] J. L. Hennessy and D. A. Patterson. *Arquitectura de Computadores: Un enfoque cuantitativo*. Mcgraw Hill Editorial, 1993.

[14] J. L. Hennessy and D. a. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. 2006.

[15] A. C. Hu and S. Zain. NSEU Mitigation in Avionics Applications, 2010.

[16] I.N.E. Encuesta sobre equipamiento y uso de tecnologías de información y comunicación en los hogares, 2014.

[17] A. O. Investigation. *ATSB TRANSPORT SAFETY REPORT Aviation Occurrence Investigation AO-2008-070 Final.* Number October. 2008.

[18] IROC. Do I Really Need to Worry About Soft Errors?, 2013.

[19] Jedec. Measurement and Reporting of Alpha Particle and Terrestrial Cosmic Ray Induced Soft Error in Semiconductor Devices, 2006.

[20] 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.

[21] F. L. Kastensmidt. http://www.inf.ufrgs.br/~fglima/res.htm.

[22] H. Kirrmann. *Fault Tolerant Computing in Industrial Automation.* 2005.

[23] I. Kuon, R. Tessier, and J. Rose. *FPGA Architecture: Survey and Challenges*, volume 2. 2007.

[24] I. M. Martín, J. E. Celador Hernández, and C. T. Bustillos. *Simulador arm en el ámbito docente.* PhD thesis, Universidad Complutense de Madrid, 2012.

[25] K. W. Melis. *Reconstruction of High-energy Neutrino-induced Particle Showers in KM3NeT.* PhD thesis, Faculteit der Natuurwetenschappen, 2014.

[26] Qualcomm. Qualcomm Snapdragon 810 Processor.

[27] J. Rose, A. E. Gamal, and A. Sangiovanni-Vincentelli. Architecture of Field-Programmable Gate Arrays. *Proceeding of the IEEE*, 81(7):1013–1029, 1993.

[28] 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.

[29] D. J. Sorin and S. Ozev. Fault Tolerant Microprocessors for Space Missions.

[30] J. M. Torrecillas. Tecnología RAID - Tolerancia a Fallos.

[31] 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.

[32] Xilinx. Xilinx Artix-7 Fpgas: a New Performance Standard for Power-Limited, Cost-Sensitive Markets, 2015.