Universidad Carlos III de Madrid

Departamento de Tecnología Electrónica

Proyecto de Fin de Carrera de Ingeniería Industrial

Diseño de un módulo I-IP para la detección de errores transitorios en sistemas embebidos

Autor: Antonio José Sánchez Clemente

Director: Luis Entrena Arrontes

19 de Enero de 2011

ÍNDICE

1 INTRODUCCIÓN 1

1.1 INTRODUCCIÓN 1
1.2 OBJETIVOS 2
1.3 FASES DE DESARROLLO 2
1.4 MEDIOS EMPLEADOS 3
1.5 ESTRUCTURA DE LA MEMORIA 3

2 ESTADO DEL ARTE 4

2.1 PROTOCOLO DE COMUNICACIONES AMBA 4
2.1.1 Bus AMBA AHB. 5
2.1.1.1 Señales del bus AHB 6
2.1.1.2 Interconexión del bus 7
2.1.1.3 Introducción a las operaciones del bus 8
2.1.1.4 Transferencia básica 9
2.1.1.5 Tipos de transferencia 11
2.1.1.6 Decodificación de direcciones 11
2.1.1.7 Respuesta del esclavo 12
2.1.1.8 Arbitraje del bus 13
2.1.2 Bus AMBA APB 14
2.1.2.1 Señales del bus APB 15
2.1.2.2 Funcionamiento del bus APB 15
2.2 MICROPROCESADOR SINTETIZABLE LEON 3 17
2.2.1 Arquitectura del microprocesador 17
2.2.1.1 Integer unit 19
2.2.1.2 Memoria caché 21
2.2.2 Arquitectura del sistema 21
2.2.3 GRLIB 22
2.2.3.1 Organización de GRLIB 23
2.2.4 Proceso de implementación 24
2.2.4.1 Configuración 25
2.2.4.2 Simulación y síntesis 26
2.2.4.3 Implementación del software 27
2.2.4.4 Integración del software y hardware 27
2.3 TÉCNICAS DE DETECCIÓN DE ERRORES PARA SISTEMAS EMBEBIDOS 27
2.3.1 Técnicas de detección Software 28
2.3.2 Técnicas de detección Hardware 28
2.3.3 Técnicas de detección híbridas 29

3 DISEÑO DEL MÓDULO DE DETECCIÓN DE FALLOS 31

3.1 FUNCIONALIDAD DEL MÓDULO DE DETECCIÓN 31
3.1.1 Genéricos del módulo de detección 34
3.1.2 Interfaz 35
3.1.3 Banco de registros 37
3.1.4 Control 39
3.1.5 Espía 42
3.2 ORGANIZACIÓN DEL DISEÑO 43
3.3 INSERCIÓN DEL MÓDULO DE DETECCIÓN EN EL SISTEMA 43

4 RESULTADOS 47

4.1 VALIDACIÓN DE LA FUNCIONALIDAD DEL DISEÑO 47
4.1.1 Simulación del módulo de detección sin interfaz 47
4.1.2 Simulación de la interfaz 54
4.1.3 Simulación del módulo de detección al completo 57
4.2 RESULTADOS DE SÍNTESIS 59
4.3 TEST DE ERRORES 61
4.3.1 Configuración del test 61

4.3.2 Procedimiento de ensayo 62
4.3.3 Resultados y clasificación de errores 63

5 CONCLUSIONES Y TRABAJOS FUTUROS 67

BIBLIOGRAFÍA 69

ANEXO A. PRESUPUESTO 70

A.1. INTRODUCCIÓN 70
A.2. FASES DEL PROYECTO 70
A.3. DESGLOSE DE COSTES 70

ANEXO B. CÓDIGO FUENTE DE LOS ARCHIVOS DEL PROYECTO 72

B.1. GENÉRICOS.VHD 72
B.2. SPYSLV.VHD 73
B.3. INTERFAZ.VHD 76
B.4. REGBANK.VHD 79
B.5. CONTROL.VHD 81
B.6. SPY.VHD 84
B.7. SPY.IN 85
B.8. SPY.IN.H 86
B.9. SPY.IN.VHD 86
B.10. SPY.IN.HELP 86

ÍNDICE DE FIGURAS

Figura 1: Arquitectura de un sistema basado en AMBA [1] 5
Figura 2: Interconexión del bus [1] 8
Figura 3: Transferencia simple [1] 9
Figura 4: Transferencia simple extendida [1] 10
Figura 5: Transferencia múltiple [1] 10
Figura 6: Decodificador de direcciones [1] 11
Figura 7: Respuesta de tipo *error* [1] 13
Figura 8: Diagrama de estados del bus APB [1] 16
Figura 9: Escritura en el bus APB [1] 16
Figura 10: Lectura en el bus APB [1] 17
Figura 11: Diagrama de bloques del microprocesador LEON3 [3] 18
Figura 12: Integer unit [3] 19
Figura 13: Registros *breakpoint* [3] 20
Figura 14: Arquitectura del sistema embebido [4] 21
Figura 15: Registros de configuración AHB [4] 23
Figura 16: Registros de configuración APB [4] 23
Figura 17: Proceso de implementación de un sistema basado en el LEON3 25
Figura 18: Xconfig 26
Figura 19: Flujograma de operación del módulo 33
Figura 20: Diagrama de bloques del módulo IP 34
Figura 21: Bloque de interfaz 35
Figura 22: Dirección válida 36
Figura 23: Flujograma de operación de la interfaz 37
Figura 24: Banco de registros 37
Figura 25: Registros de resultado del banco de registros 39
Figura 26: Registros de configuración del banco de registros 39
Figura 27: Bloque de control 40
Figura 28: Diagrama de estados del módulo IP 41
Figura 29: Bloque espía 42
Figura 30: Módulo de detección integrado en Xconfig 45
Figura 31: Menú de configuración del módulo de detección 45
Figura 32: Simulación del módulo aislado - reset y configuración 48
Figura 33: Simulación del módulo aislado - configuración e inicio de la 1ª rutina 49
Figura 34: Simulación del módulo aislado - fin de la 1ª rutina 49
Figura 35: Simulación del módulo aislado - inicio de la 2ª rutina 50
Figura 36: Simulación del módulo aislado - fin de la 2ª rutina 50
Figura 37: Simulación del módulo aislado - repetición correcta de la 1ª rutina 51
Figura 38: Simulación del módulo aislado - repetición correcta de la 2ª rutina 51
Figura 39: Simulación del módulo aislado - fallo por resultados distintos 52
Figura 40: Simulación del módulo aislado - repetición errónea de la 2ª rutina 52
Figura 41: Simulación del módulo aislado - desbordamiento del *watchdog* 53
Figura 42: Simulación del módulo aislado - protección contra escritura 53
Figura 43: Simulación del módulo aislado - protección contra lectura 54
Figura 44: Simulación de la interfaz - comunicación con la interfaz I 55
Figura 45: Simulación de la interfaz - comunicación con la interfaz II 55
Figura 46: Simulación de la interfaz - comunicación con la memoria I 56
Figura 47: Simulación de la interfaz - comunicación con la memoria II 56
Figura 48: Simulación de la interfaz - escritura en dirección no válida 56
Figura 49: Simulación del módulo de detección competo - configuración e inicio de rutina 57
Figura 50: Simulación del módulo de detección completo - interrupción por error de escritura 58
Figura 51: Simulación del módulo de detección completo - ejecución de rutina correcta 58
Figura 52: Simulación del módulo de detección completo - interrupción por error de ejecución 59
Figura A. 1: Diagrama de Gantt 70

ÍNDICE DE TABLAS

Tabla 1: Señales del bus AHB [1] 6
Tabla 2: Señales de arbitraje del bus AHB [1] 7
Tabla 3: Tipos de transferencia [1] 11
Tabla 4: Tipos de respuesta [1] 12
Tabla 5: Señales del bus APB [1] 15
Tabla 6: Ocupación de la FPGA 60
Tabla 7: Frecuencias máximas de funcionamiento 60
Tabla 8: Resultados del test de errores 64
Tabla 9: Proporción de errores relevantes 64
Tabla 10: Clasificación de errores no detectados 66
Tabla A. 1: Resumen de costes 71

1 INTRODUCCIÓN

1.1 Introducción

La fiabilidad de la electrónica digital es un campo de creciente importancia hoy día. Debido a los avances en la fabricación de circuitos electrónicos digitales, hace unos 15 años se daba por sentado que la electrónica digital era fiable y robusta, y la preocupación por la fiabilidad estaba relegada a ciertas aplicaciones específicas. Sin embargo esta percepción está cambiando actualmente. La causa de ello radica en la disminución progresiva del tamaño de la electrónica digital y a la aplicación cada vez más extensa de la misma a todo tipo de ámbitos y aplicaciones, como por ejemplo la domótica o la automoción.

Hay una serie de ámbitos en los que la fiabilidad de la electrónica cobra una especial importancia. En primer lugar, cuando la vida o la integridad de las personas están en peligro, como por ejemplo en medicina o en automoción. Aparte también se le da importancia a la fiabilidad cuando los fallos llevan asociados un coste muy elevado o cuando la electrónica se emplea en entornos hostiles, como es el caso de la industria aeroespacial. El hecho de que cada vez más se emplee la electrónica digital para el control y la supervisión de todo tipo de sistemas significa que cada vez mayor número de aplicaciones entran dentro de alguna de estas categorías.

Existen varios tipos de fallos que pueden afectar a un circuito electrónico digital:

- Transitorio (*soft error*). Cambio aleatorio y puntual de alguna de las señales del circuito debido a la incidencia de radiación electromagnética o la colisión de partículas de alta energía, con la energía suficiente como para cambiar el estado del circuito.

- Permanente (*hard error*). Fallo perpetuo debido a la rotura o deterioro de un componente electrónico del circuito.

- Intermitente. Fallo que se produce de forma periódica en el mismo punto de algún cierto componente. Por lo general es indicativo de un fallo permanente en el futuro próximo.

Este proyecto se centra en el estudio de los fallos transitorios. Estos fallos constituyen un problema de importancia en aumento, especialmente los debidos a la radiación. Con la disminución del tamaño de la electrónica digital, aparte de abaratar su coste se consigue que la energía de activación de los componentes sea menor. Esto hace que su sensibilidad sea mayor y que por tanto sea necesaria menor energía tanto para modificar su estado como para que se produzcan fallos. Hasta tal punto es así que hoy día muchos dispositivos electrónicos, y en especial las memorias, presentan niveles inaceptables de sensibilidad a la radiación incluso a nivel terrestre.

Es enormemente difícil y extremadamente costoso evitar que se produzcan errores en un sistema cualquiera, así que la forma de aumentar la fiabilidad pasa por el desarrollo de sistemas tolerantes a fallos. Estos sistemas son capaces, en mayor o menor medida, de proseguir con su funcionamiento habitual aún en presencia de errores. Existen varias formas de implementar un sistema tolerante a fallos:

- La solución trivial: replicar el sistema al completo y decidir la respuesta válida mediante un sistema de votación. Es una solución muy intuitiva, pero excesivamente cara.

- Replicación a nivel de componente. También es una solución cara.
- Fortalecimiento a nivel de sistema. Esta solución consiste en la modificación del hardware y/o del software para aportar al sistema robustez frente a fallos. Se trata de una buena solución si tiene un coste moderado y se obtiene una fiabilidad aceptable.

El enfoque que se adopta en este proyecto es el fortalecimiento a nivel de sistema, por ser la solución más económica. En concreto, se opta por una solución híbrida que combina modificaciones de software con la inserción de un I-IP (*Infrastructure IP*). Los I-IP son módulos IP que añaden propiedades no funcionales, como por ejemplo fiabilidad, al sistema al cual se conectan.

El presente proyecto de fin de carrera forma parte del proyecto OPTIMISE (OPTimisation of MItigations for Soft, firm and hard Errors), una iniciativa europea con el objetivo de desarrollar sistemas y aplicaciones que mejoren la fiabilidad de la electrónica digital. El Grupo de Diseño Microelectrónico y Aplicaciones del Departamento de Tecnología Electrónica de la Universidad Carlos III de Madrid participa en dicho proyecto, centrándose en la mitigación de errores transitorios mediante el empleo de módulos IP.

1.2 Objetivos

El objetivo del presente proyecto es diseñar un módulo IP que se conecte al bus de comunicaciones de un sistema embebido de forma no intrusiva y que, con las adecuadas modificaciones del software que se ejecuta, sea capaz de detectar errores transitorios en el sistema de forma no intrusiva. La solución propuesta debe ser eficiente, es decir, debe ser capaz de detectar una proporción de errores aceptable. Además, debe afectar lo menos posible a las prestaciones del procesador y debe ser económico, tanto en coste como en espacio. Dentro del proyecto se incluye también la validación del módulo y de sus capacidades de detección.

En concreto, la finalidad del módulo IP es supervisar las comunicaciones con la memoria u otros periféricos que funcionen como memorias. Dentro del proyecto OPTIMISE se han desarrollado módulos complementarios para otros componentes.

Para la consecución de estos objetivos se toma como vehículo de pruebas el LEON3, un microprocesador desarrollado por Gaisler Research. Esta elección se debe a que el LEON3 es un microprocesador sintetizable, es decir, que su funcionalidad está descrita en una serie de diseños desarrollados en un lenguaje de descripción de hardware, en concreto VHDL. Además estos diseños están disponibles en código fuente bajo licencia gratuita. Por último, se trata de un microprocesador de 32 bits con aplicaciones reales en la industria aeroespacial hoy día.

1.3 Fases de desarrollo

Para la realización del presente proyecto se parte de conocimientos previos en las asignaturas de Fundamentos de Electrónica digital, Sistemas Electrónicos Digitales, Informática Industrial, y Circuitos Integrados y Microelectrónica. A continuación se describen las distintas fases del desarrollo del proyecto.

En primer lugar, se ha estudiado la arquitectura del microprocesador LEON3 y las herramientas que lleva asociadas.

A continuación se ha diseñado el módulo de detección de fallos propiamente dicho, describiendo su funcionalidad en el lenguaje de descripción de hardware VHDL. Esto incluye sucesivas fases de simulación con banco de pruebas y posterior rediseño hasta obtener la versión final.

Por último se han efectuado las pruebas de validación. Por un lado se ha realizado una síntesis del código en VHDL para conocer los requerimientos de espacio y velocidad de funcionamiento, y por otro se ha efectuado un test de errores para comprobar la capacidad de detección de fallos del módulo diseñado.

1.4 Medios empleados

En la ejecución de este proyecto ha sido necesario emplear un ordenador con sistema operativo Linux. Además se ha hecho uso del programa Modelsim, de Mentor Graphics, para simulación; y el programa Quartus II, de Altera, para síntesis. También se ha empleado la librería de módulos GRLIB, desarrollada por Gaisler Research.

1.5 Estructura de la memoria

A continuación se resume brevemente el contenido de los capítulos en los que se divide esta memoria.

En el capítulo 1 (*Introducción*) se describe el contexto en el que se enmarca el presente proyecto, la problemática que se pretende resolver y los medios empleados para ello, así como un breve resumen del proceso de ejecución del proyecto al completo.

En el capítulo 2 (*Estado del arte*) se exponen de forma resumida las bases y herramientas sobre las que se asienta el presente proyecto: el protocolo de comunicaciones AMBA, el microprocesador LEON3 y las técnicas de detección de errores que tienen relación con el presente proyecto.

El capítulo 3 (*Diseño del módulo de detección de fallos*) contiene una descripción detallada del dispositivo diseñado así como del proceso de inserción del mismo en el LEON3.

En el capítulo 4 (*Resultados*) se explican las distintas pruebas realizadas para determinar las características del módulo IP diseñado y se muestran los resultados obtenidos en dichas pruebas.

En el capítulo 5 (*Conclusiones*) se infieren una serie de conclusiones lógicas a partir de los resultados experimentales obtenidos.

2 ESTADO DEL ARTE

En este capítulo se describen los fundamentos en los que se basa este proyecto. En primer lugar se aborda el protocolo de comunicaciones AMBA, ya que el LEON3 se ha diseñado sobre esta especificación. A continuación se introduce el microprocesador LEON3 y la forma de trabajar con él. Por último se muestran algunas técnicas relevantes de detección de errores, desde el punto de vista del presente proyecto.

2.1 Protocolo de comunicaciones AMBA

La especificación AMBA (*Advanced Microcontroller Bus Architecture*) define un estándar de comunicaciones on-chip para sistemas embebidos. Desarrollado por ARM, este protocolo independiente de la tecnología ha sido concebido para facilitar el desarrollo de nuevos microcontroladores para sistemas embebidos, a la vez que se minimiza el espacio necesario para dar soporte a las comunicaciones tanto dentro como fuera del chip y se estimula el diseño modular. Se trata de un protocolo ampliamente utilizado hoy día.

El contenido de todo este apartado se ha obtenido de la versión 2.0 de la especificación AMBA [1], donde se exponen en detalle todas las características de este bus.

El protocolo AMBA define tres buses de comunicaciones distintos:

- El bus AHB (*Advanced High-performance Bus*) es un bus de altas prestaciones, tal y como indica su nombre, para módulos de elevada frecuencia de funcionamiento. Actúa como el bus principal del sistema, dando soporte a la conexión eficiente de procesadores, memorias internas e interfaces de memorias externas, entre otros dispositivos.

- El bus ASB (*Advanced System Bus*) es un bus alternativo al AHB que puede emplearse como bus principal del sistema siempre y cuando no se requiera de las altas prestaciones del AHB.

- El bus APB (*Advanced Peripheral Bus*) es un bus secundario para la conexión de periféricos lentos o de bajo ancho de banda, especialmente diseñado para minimizar el consumo de energía. Además la interfaz es más simple respecto a los otros dos buses. Se comunica con el bus principal por medio de un módulo denominado puente (*APB bridge*).

La Figura 1 muestra un esquema típico de un sistema basado en el protocolo de comunicaciones AMBA. Se emplea un bus, que puede ser el AHB o el ASB, como bus principal del sistema, donde se conecta el procesador, la memoria interna y otros periféricos como las interfaces con memorias o con otros protocolos de comunicación. Además se emplea el bus APB para la conexión de periféricos de bajo ancho de banda, conectado al bus principal del sistema mediante el *APB bridge*.

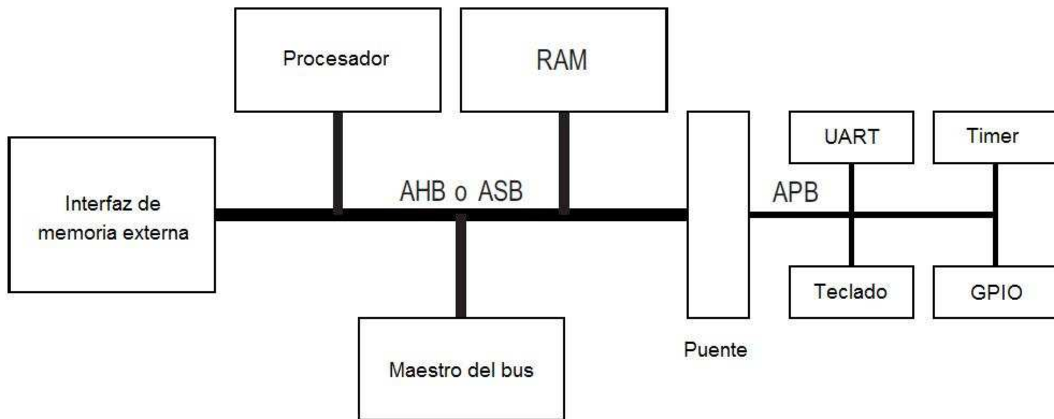

Figura 1: Arquitectura de un sistema basado en AMBA [1]

El estándar AMBA no define las características eléctricas de las líneas que constituyen los buses, ni especifica restricciones temporales más allá del comportamiento ciclo a ciclo. Esto otorga una gran flexibilidad en el diseño de sistemas embebidos.

A continuación se expone el funcionamiento general de los buses AHB y APB. El bus ASB carece de importancia para el presente proyecto por lo que no se aborda en este documento. Se remite a la bibliografía a aquellos lectores interesados en sus características.

2.1.1 Bus AMBA AHB.

El bus AHB es un bus diseñado para soportar los requerimientos de los diseños de altas prestaciones y elevada frecuencia de funcionamiento, incluyendo entre otras características:

- Transferencias en modo ráfaga (*burst*).
- Transferencias partidas (*split*).
- Operaciones de transferencia segmentadas.
- Operaciones de un único ciclo de reloj.
- Ausencia de componentes tri-estado.
- Configuración para buses de datos anchos, de hasta 64 o 128 bits.

Los elementos típicos en el bus AHB son:

- Maestro AHB, capaz de iniciar operaciones de lectura y escritura proporcionando la dirección y las señales de control de la transferencia. El bus AHB soporta un esquema de comunicación maestro-esclavo con múltiples maestros, pero sólo uno de los maestros puede realizar transferencias en cada momento, es decir, no puede haber dos maestros utilizando el bus simultáneamente. Ejemplos de maestros son el propio microprocesador, la interfaz de errores o la DMA (*Direct Memory Access*).
- Esclavo AHB, que responde a las operaciones de lectura y escritura en un rango de direcciones dado. El esclavo genera unas señales que indican al maestro activo el

estado de la transferencia. Ejemplos de esclavos son las memorias internas, las interfaces de memorias externas o el *APB bridge.*

- Árbitro AHB, encargado de determinar cuál es el maestro que puede usar el bus en cada momento. El algoritmo de arbitraje puede implementarse según las necesidades de la aplicación. Debe haber uno y sólo uno en cada diseño.

- Decodificador AHB, que decodifica la dirección que envía el maestro activo para seleccionar el esclavo involucrado en la transferencia. Se requiere un único decodificador centralizado en todas las implementaciones.

En los siguientes sub-apartados se abordan distintos aspectos del bus de comunicaciones AHB. En primer lugar se enumeran las señales presentes en el bus y se indica la forma de conectar los distintos elementos del mismo. A continuación se resume el funcionamiento general del bus y se explica cómo se desarrolla una transferencia básica. Después se listan los diferentes modos de transferencia, y se detalla el sistema de decodificación de direcciones. Por último se señalan las posibilidades que tienen los esclavos para responder a las transferencias, y se explica cómo se efectúa el arbitraje del bus.

2.1.1.1 Señales del bus AHB

Nombre	Origen	Descripción
HCLK	Reloj	Reloj del bus. Todos los cambios en el bus se coordinan mediante el flanco de subida del reloj.
HRESETn	Controlador del reset	Señal de reset del sistema, activa por nivel bajo.
HADDR[31:0]	Maestro	Bus de direcciones de 32 bits.
HTRANS[1:0]	Maestro	Tipo de transferencia de la transferencia en curso.
HWRITE	Maestro	Indica el sentido de la transferencia. Nivel alto significa escritura y nivel bajo, lectura (siempre desde el punto de vista del maestro).
HSIZE[2:0]	Maestro	Indica el tamaño de la transferencia.
HBURST[2:0]	Maestro	Indica si la transferencia en curso es en modo ráfaga, y en caso afirmativo de qué tipo de ráfaga se trata.
HPROT[3:0]	Maestro	Señales de control de protección.
HWDATA[31:0]	Maestro	Bus de datos de las operaciones de escritura, de 32 bits.
HSELx	Decodificador	Selección de esclavo.
HRDATA[31:0]	Esclavo	Bus de datos para las operaciones de lectura, de 32 bits.
HREADY	Esclavo	A nivel alto indica que ha concluido una transferencia. Para los esclavos es a la vez una señal de salida y de entrada.
HRESP[1:0]	Esclavo	Respuesta del esclavo.

Tabla 1: Señales del bus AHB [1]

La Tabla 1 muestra el listado de señales que conforman el bus AHB. Todas las señales llevan como prefijo la letra 'H', para diferenciarlas de otras señales similares pertenecientes a otros buses de la especificación AMBA. Algunas de estas señales llevan como sufijo la letra 'x', lo que indica que existe una señal específica e independiente para cada uno de los módulos. Las

señales son por defecto activas a nivel alto, salvo las que tienen como sufijo la letra 'n' que son activas por nivel bajo.

Existe además un conjunto de señales necesarias para el arbitraje del bus en los sistemas con más de un maestro AHB. Estas señales se muestran en la Tabla 2.

Nombre	Origen	Descripción
HBUSREQx	Maestro	Petición de uso del bus por parte del maestro x.
HLOCKx	Maestro	Cuando está activo indica que el maestro requiere que un conjunto de transferencias se realicen de forma indivisible.
HGRANTx	Árbitro	Indica qué maestro es el siguiente en acceder al bus. Un maestro obtiene acceso al bus cuando HREADY y HGRANT están activos.
HMASTER[3:0]	Árbitro	Indica el maestro activo. Empleado por los esclavos con soporte para transferencias partidas (*split*), para conocer qué maestro está realizando una transferencia.
HMASTLOCK	Árbitro	Indica si el maestro activo realiza un conjunto de transferencias de forma indivisible.
HSPLITx[15:0]	Esclavo	Indica a qué maestros se les permite reintentar una transferencia partida con el esclavo x. Cada esclavo tiene una señal de 16 bits independiente.

Tabla 2: Señales de arbitraje del bus AHB [1]

2.1.1.2 Interconexión del bus

Los módulos presentes en el bus AHB se conectan entre sí mediante multiplexores centrales, siguiendo el esquema de la Figura 2. Con esta disposición cada uno de los maestros puede generar en todo momento la dirección y las señales de control de la transferencia que quiere efectuar, y el árbitro decide cuál es el maestro cuyas señales llegan hasta los esclavos. Por otro lado es necesario un decodificador que seleccione el esclavo con el que se comunica el maestro activo, de modo que sean las señales de dicho esclavo las que reciban los maestros.

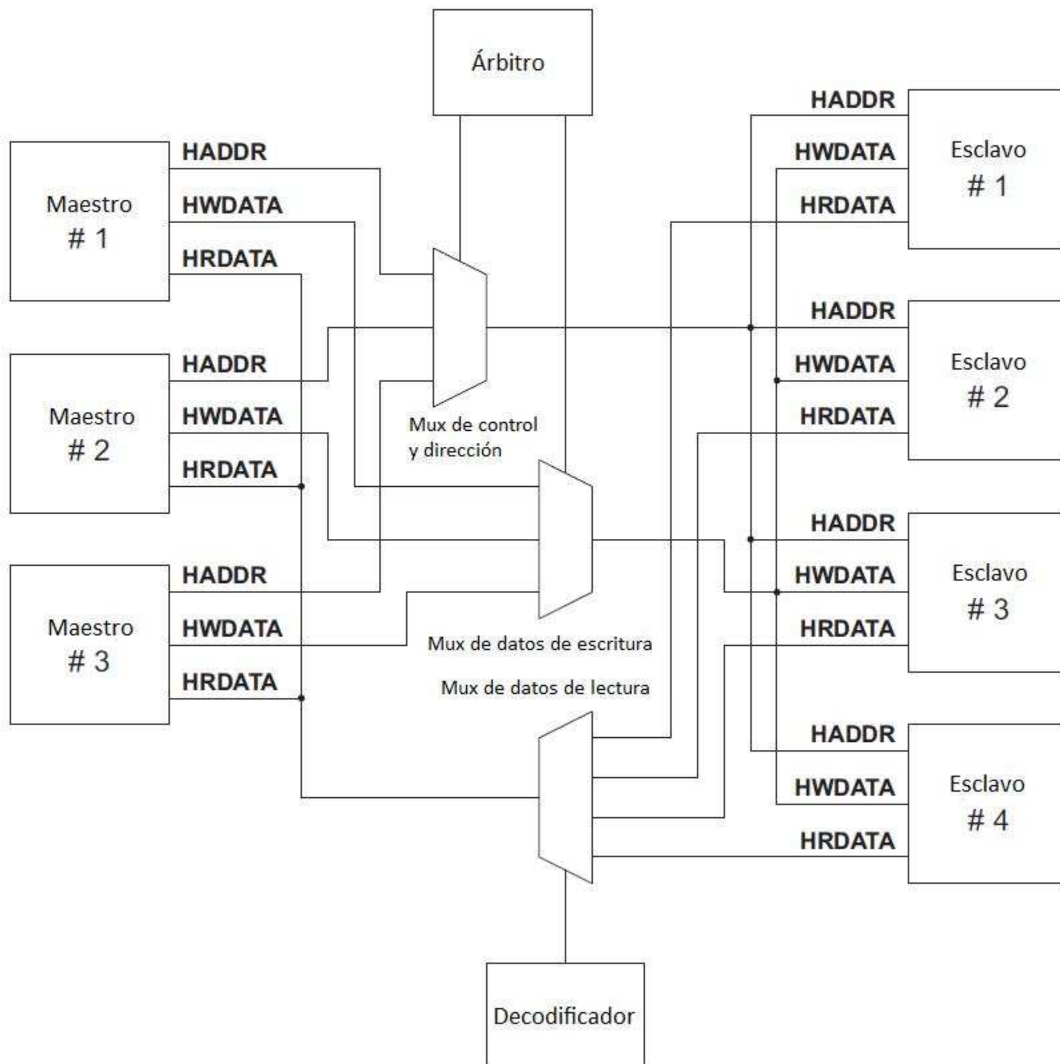

Figura 2: Interconexión del bus [1]

2.1.1.3 Introducción a las operaciones del bus

Antes de que se inicie una transferencia entre un maestro y un esclavo, el maestro debe solicitar el uso del bus al árbitro mediante una señal de petición, HBUSREQx. Tras ello el árbitro debe concedérselo activando la señal HGRANTx correspondiente al maestro en cuestión. La transferencia dará comienzo cuando termine la operación anterior, lo que se indica con la activación de la señal HREADY.

Las transferencias en el bus AHB están segmentadas, así cada transferencia consta de dos fases: en primer lugar se envían las señales de control de la transferencia incluyendo la dirección, y a continuación se envía el dato.

El maestro inicia una transferencia estableciendo la dirección y las señales de control de la misma. Dichas señales indican el sentido de la transferencia, el ancho de datos y si la transferencia forma parte de una ráfaga. Esta información está disponible en el bus un único ciclo de reloj y por tanto todos los esclavos deben leer estos datos en cuanto aparecen en el bus.

Para la transmisión de los datos se emplean dos buses de datos, uno para escritura y otro para lectura. El bus de datos de escritura transmite datos de los maestros a los esclavos, mientras que el bus de lectura se emplea para las transferencias de los esclavos a los maestros. A diferencia de las señales de control, se puede disponer de los datos en el bus tanto tiempo como sea necesario manteniendo a nivel bajo la señal HREADY.

Durante la transferencia el esclavo indica el estado de la misma mediante la señal HRESP. El estado puede reflejar el desarrollo normal de la transferencia (*okay*), un fallo (*error*) o que la transferencia en curso debe interrumpirse y completarse en otro momento (*retry* y *split*). Estos estados se explican con detalle en el apartado 2.1.1.7.

En funcionamiento normal a un maestro se le permite completar un conjunto de transferencias en ráfaga antes de ceder el control del bus a otro maestro. Sin embargo, es posible que el árbitro decida interrumpir una ráfaga de transferencias para evitar retardos excesivos, en cuyo caso el maestro debe solicitar de nuevo acceso al bus para finalizar dicha ráfaga.

2.1.1.4 Transferencia básica

Cada transferencia en el bus AHB consta de dos fases: una fase de dirección que dura un único ciclo de reloj, y una fase de datos que puede extenderse durante varios ciclos de reloj por medio de la señal HREADY.

La transferencia más simple de todas es aquella en la cual la fase de datos dura un único ciclo de reloj, como se muestra en la Figura 3. En esta transferencia el maestro indica la dirección y las señales de control tras el primer flanco de subida del reloj, señales que el esclavo registra en el siguiente flanco. A continuación el esclavo envía por el bus la respuesta apropiada, que el maestro recibe en el tercer flanco de subida del reloj.

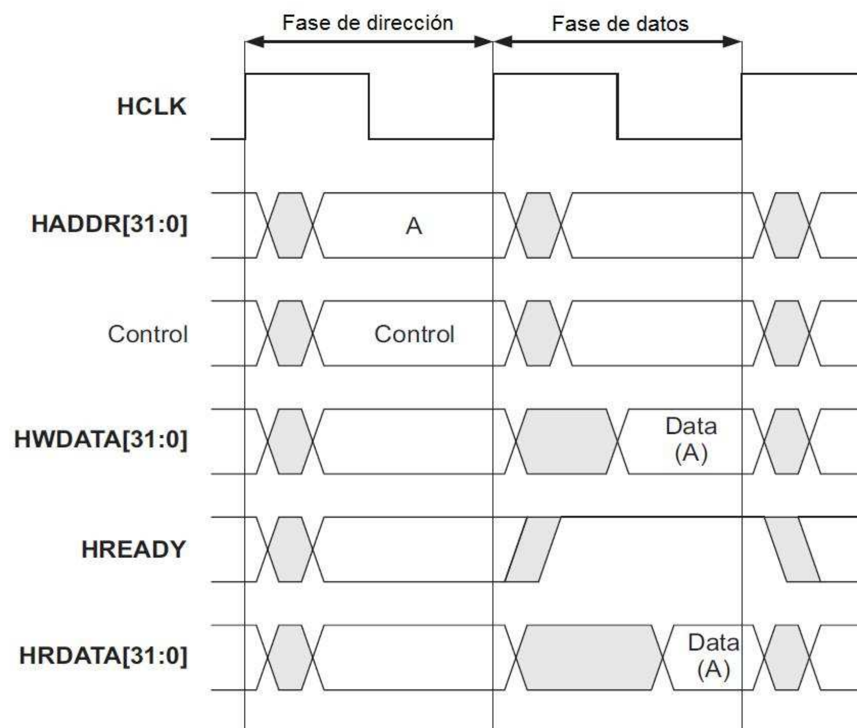

Figura 3: Transferencia simple [1]

El esclavo puede prolongar la fase de datos de la transferencia manteniendo a nivel bajo la señal HREADY, como se puede apreciar en la Figura 4, lo que permite disponer de tiempo adicional para completar la transferencia. Para las operaciones de escritura el maestro debe mantener el dato estable en el bus durante toda la fase de datos, mientras que en las operaciones de lectura el esclavo no tiene que proporcionar un dato válido hasta que la transferencia esté a punto de finalizar.

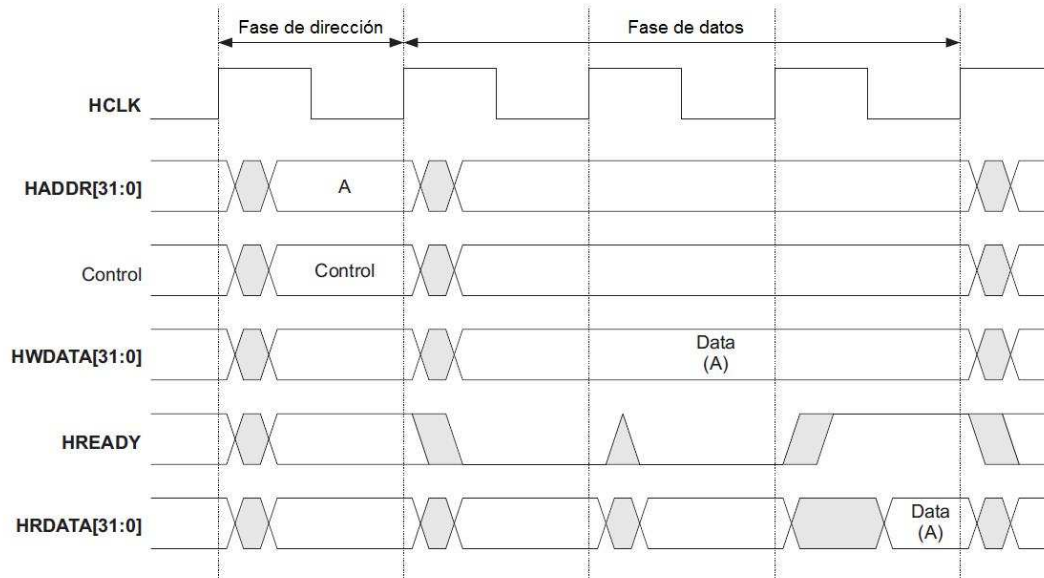

Figura 4: Transferencia simple extendida [1]

La fase de dirección y la de datos tienen lugar en diferentes ciclos de reloj. De hecho, la fase de dirección de una transferencia coincide con la fase de datos de la transferencia previa. Esta superposición de las fases es fundamental para el funcionamiento segmentado del bus y permite altas prestaciones a la vez que proporciona un tiempo de respuesta adecuado a los esclavos. De este modo, cuando se extiende la fase de datos de una transferencia se da el efecto colateral de prolongar la fase de dirección de la siguiente transferencia, tal y como puede verse en la Figura 5. A pesar de ello los esclavos leen las señales de control únicamente cuando se activa la señal HREADY.

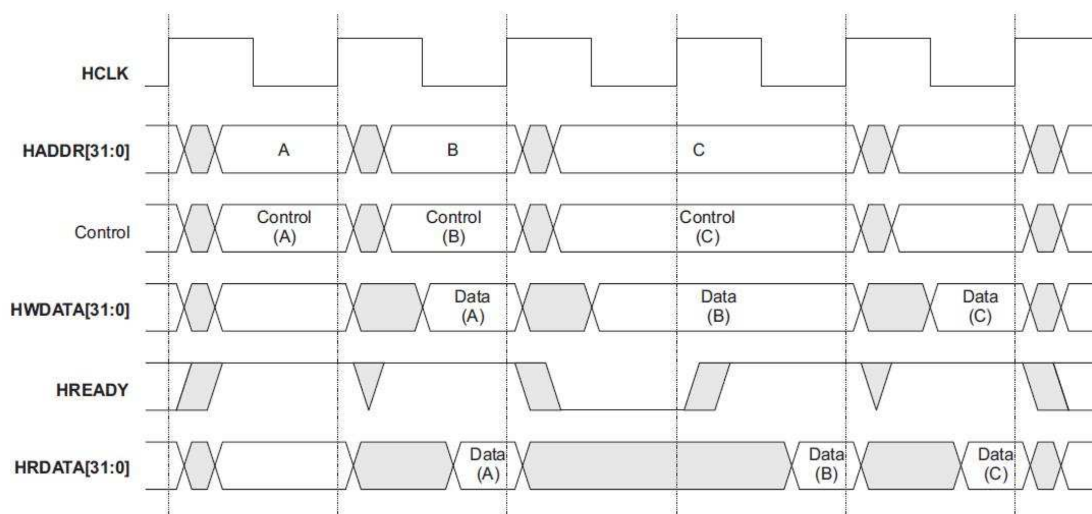

Figura 5: Transferencia múltiple [1]

2.1.1.5 Tipos de transferencia

Existen cuatros tipos de transferencia, cada uno de los cuales se corresponde con un valor de la señal HTRANS, tal y como muestra la Tabla 3.

Tipo de transferencia	**HTRANS[1:0]**	**Descripción**
Ocioso (*IDLE*)	00	No se requiere transferencia. Los esclavos deben responder con el estado *okay* e ignorar la transferencia.
Ocupado (*BUSY*)	01	Indica que, en medio de una ráfaga de transferencias, la siguiente transferencia no puede comenzar inmediatamente. En este caso la dirección y el resto de señales de control de la transferencia deben referirse a la próxima transferencia de la ráfaga. Los esclavos deben ignorar la operación y responder con el estado *okay*.
No secuencial (*NONSEQ*)	10	Transferencia simple o primera transferencia de una ráfaga. La dirección y las señales de control no guardan relación con la transferencia previa.
Secuencial (*SEQ*)	11	Se emplea para todas las transferencias de una ráfaga, a excepción de la primera. Las señales de control tienen el mismo valor que en la transferencia anterior.

Tabla 3: Tipos de transferencia [1]

2.1.1.6 Decodificación de direcciones

Se emplea un decodificador central para generar las señales de selección de esclavo, HSELx, a partir de los bits más significativos de la dirección. Se prefieren esquemas simples de decodificación que permitan una alta velocidad del bus. La Figura 6 muestra un sistema típico de decodificación de direcciones.

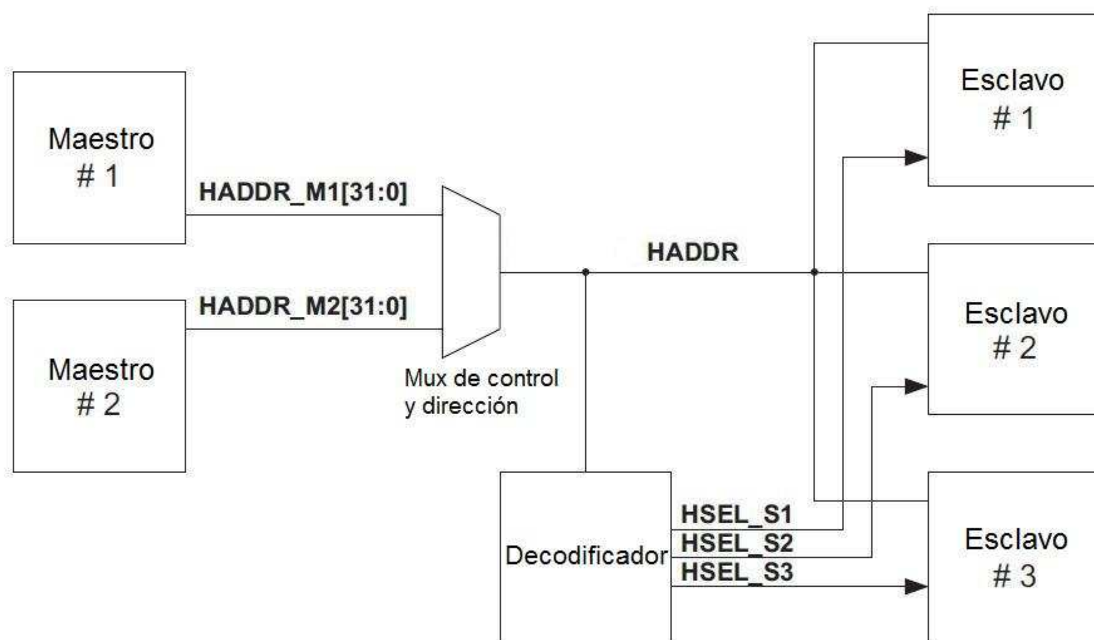

Figura 6: Decodificador de direcciones [1]

Los esclavos deben leer la dirección y las señales de control, incluyendo su correspondiente HSELx, únicamente cuando se activa la señal HREADY. Esto indica el fin de la transferencia en curso.

El espacio de memoria asignado a cada esclavo es como mínimo de un KB. Por este motivo los maestros no deben realizar transferencias en ráfaga más allá del límite de un KB.

Cuando en un diseño no se llena por completo el mapa de memoria se añade un esclavo adicional, el cual permite responder a los accesos a direcciones de memoria inexistentes. Este esclavo por defecto está implementado típicamente como parte del propio decodificador. Dicho esclavo debe responder a las transferencias de tipo *secuencial* y *no secuencial* con un *error*, mientras que para las transferencias de tipo *ocioso* y *ocupado* debe devolver una respuesta *okay*.

2.1.1.7 Respuesta del esclavo

Cuando se accede a un esclavo éste se encarga de indicar el estado de la transferencia mediante las señales HREADY y HRESP. La especificación AMBA no permite a los maestros cancelar la transferencia una vez iniciada, siendo el esclavo el que decide cuándo concluye la transferencia y el que debe indicar si la transferencia se completa con éxito o si hay algún error.

La señal HREADY se emplea para extender la fase de datos de una transferencia mientras esté en nivel bajo. Es recomendable que exista un límite al número de ciclos que puede prolongarse una transferencia mediante este procedimiento, de modo que pueda calcularse el tiempo de acceso al bus.

La señal HRESP indica el tipo de respuesta del esclavo. Existen cuatro tipos de respuesta, cada uno de los cuales está asociado a un valor de la señal HRESP, tal y como indica la Tabla 4.

Tipo de respuesta	HRESP [1:0]	Descripción
Okay	00	La transferencia progresa adecuadamente. Con HREADY a nivel bajo permite extender la fase de datos de la transferencia, mientras que si HREADY está activo indica que la transferencia ha finalizado con éxito.
Error	01	Se ha producido un error, la transferencia no puede completarse con éxito.
Retry	10	No se puede completar la transferencia. El maestro debe solicitar de nuevo acceso al bus para repetir dicha transferencia.
Split	11	No se puede completar la transferencia. Se deja libre el bus para que otro maestro pueda utilizarlo. El maestro debe solicitar de nuevo acceso al bus para repetir dicha transferencia.

Tabla 4: Tipos de respuesta [1]

Debido a la segmentación de las transferencias del bus, las respuestas de tipo *error*, *retry* y *split* deben mantenerse en el bus durante dos ciclos consecutivos. En el primer ciclo de la respuesta la señal HREADY se mantiene a nivel bajo y en el segundo ciclo se activa dicha señal, poniendo fin a la transferencia en curso. Si fuera necesario insertar más ciclos para dar una respuesta de *error*, *retry* o *split*, previamente se mantiene HREADY a nivel bajo con una respuesta *okay*. La Figura 7 muestra un ejemplo de respuesta de *error* en tres ciclos de reloj.

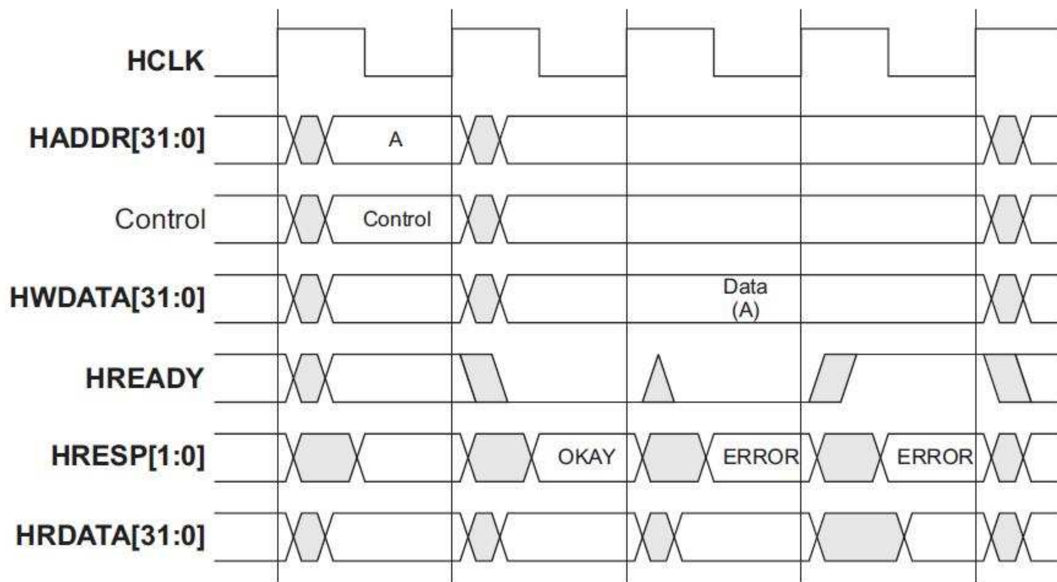

Figura 7: Respuesta de tipo *error* [1]

La respuesta de *error* se emplea para indicar alguna condición de error en la transferencia. Típicamente se debe a un conflicto con algún tipo de protección, como en el caso de un intento de escritura en una zona de memoria de lectura exclusivamente. Ante una respuesta de este tipo se puede optar por cancelar las siguientes transferencias o por realizarlas como estaba previsto.

Las respuestas *retry* y *split* permiten liberar el bus cuando un esclavo no puede atender inmediatamente a una transferencia, que debe repetirse en otro momento. Por lo general esta técnica es utilizada por aquellos esclavos que poseen una gran latencia de acceso, para evitar la ocupación del bus durante largos periodos de tiempo. A diferencia de la respuesta de *error*, en estos casos la siguiente transferencia debe cancelarse, ya que no se puede efectuar hasta que la transferencia actual se haya completado.

La diferencia entre las respuestas *retry* y *split* radica en el esquema de prioridades que impone el árbitro tras una de estas respuestas:

- En una respuesta *retry* no se modifican las prioridades, por lo que únicamente los maestros con mayor prioridad que el maestro activo pueden obtener acceso al bus.

- En el modo *split* cambia el esquema de prioridades, asignando momentáneamente al maestro activo la prioridad más baja. De este modo cualquier otro maestro puede acceder al bus. Para completar una transferencia *split* es necesario que el esclavo indique al árbitro del bus que puede atender dicha transferencia.

La respuesta *split* requiere una mayor complejidad tanto del esclavo como del árbitro del bus, pero tiene la ventaja de liberar completamente el bus para su uso por otros maestros.

2.1.1.8 Arbitraje del bus

El árbitro del bus debe garantizar que tan sólo un maestro tiene acceso al bus en cada instante. Para ello recibe las peticiones de los maestros y decide cuál es el maestro con mayor prioridad de entre los que solicitan utilizar el bus. Asimismo debe atender las peticiones de los esclavos que están en condiciones de finalizar una transferencia *split*.

Un maestro puede solicitar acceso al bus en cualquier momento activando su señal HBUSREQx. El árbitro registra la petición en el siguiente flanco de subida del reloj y emplea un algoritmo interno de prioridades para decidir cuál es el siguiente maestro en acceder al bus.

Cuando un maestro realiza una ráfaga de transferencias de longitud predeterminada no es necesario solicitar acceso al bus durante toda la ráfaga: el árbitro puede deducir a partir de las señales de control de cuántas transferencias consta la ráfaga. Sin embargo para ráfagas de longitud no especificada el maestro debe solicitar acceso al bus constantemente, ya que el árbitro no puede predecir el final de la ráfaga.

El árbitro indica mediante la señal HGRANTx cuál es el siguiente maestro en acceder al bus. Cuando la transferencia en curso termina, con lo que se activa la señal HREADY, dicho maestro obtiene acceso al bus y la señal HMASTER cambia su valor para indicar el nuevo maestro activo.

Debido a la segmentación de las transferencias hay un retardo entre la propiedad del bus de direcciones y la del bus de datos. Cuando un maestro obtiene acceso al bus, gana inmediatamente el control del bus de direcciones, y con ello el de todas las señales de control. Pero el bus de datos todavía contiene el dato de la transferencia anterior, y por tanto aún es gobernado por el anterior maestro activo hasta que dicha transferencia finalice.

Por lo general, si un maestro realiza una ráfaga de transferencias el árbitro no dará paso a otro maestro hasta que la ráfaga finalice. Sin embargo, el árbitro puede interrumpir una ráfaga para prevenir excesivos tiempos de acceso al bus. En tal caso el maestro que realiza la ráfaga pierde el acceso al bus y debe volver a solicitarlo para terminar la ráfaga, modificando las señales de control HTRANS y HBURST para que tengan unos valores adecuados a la nueva situación.

Si un maestro necesita realizar una serie de transferencias de forma indivisible, debe activar la señal HLOCKx para indicar al árbitro que ningún otro maestro puede acceder al bus hasta que el maestro activo termine sus transferencias. Tras una secuencia indivisible de transferencias el árbitro espera una transferencia adicional antes de ceder el uso del bus a otro maestro, para asegurar que la última transferencia de la serie se completa satisfactoriamente. Durante un grupo de transferencias indivisible el árbitro activa la señal HMASTLOCK para informar a todos los esclavos de ello.

Todo diseño debe incluir un maestro por defecto, que obtiene acceso al bus cuando ningún otro maestro puede hacer uso del bus, por ejemplo cuando todos los demás maestros están esperando para completar transferencias *split*. Este maestro no puede efectuar transferencias, debe asignar en todo momento el estado ocioso (*idle*) a la señal HTRANS.

2.1.2 Bus AMBA APB

El bus AMBA APB es un bus secundario, encapsulado como un esclavo del bus principal del sistema. El bus APB está optimizado para un consumo mínimo de energía y una interfaz sencilla.

La función del bus APB es servir de interfaz para todos aquellos periféricos de bajo ancho de banda y que no requieran de altas prestaciones ni de una interfaz de comunicaciones segmentada.

El bus APB implementa una comunicación maestro-esclavo con un único maestro, denominado *APB bridge*, que a su vez es esclavo del bus principal del sistema. El *APB bridge* se

encarga de traducir las señales del bus principal en las del APB y viceversa. Además incluye un segundo nivel de decodificación para generar la señal de selección de esclavo para los periféricos del bus APB. Los demás módulos conectados al APB son esclavos, cuyas interfaces deben ser acordes con las siguientes especificaciones:

- Transferencias no segmentadas.
- Consumo nulo de energía mientras el bus está inactivo.
- Funcionamiento no gobernado por el reloj del bus.

A continuación se enumeran las señales de las que consta el bus APB, y luego se describe cómo se realizan las transferencias en el bus.

2.1.2.1 Señales del bus APB

La Tabla 5 muestra todas las señales que contiene el bus APB, junto con una breve descripción de cada una. Algunas de estas señales, como el reloj del bus o el reset, pueden conectarse directamente con la señal equivalente del bus principal del sistema. Todas las señales llevan el prefijo 'P' para indicar que pertenecen al bus APB. Al igual que en el bus AHB, el sufijo 'x' indica que existe una señal independiente para cada módulo, mientras que el sufijo 'n' indica que la señal se activa a nivel bajo.

Nombre	**Descripción**
PCLK	Reloj del bus. Los cambios de las señales del bus se coordinan mediante el flanco de subida del reloj.
PRESETn	Señal de reset del bus, activa por nivel bajo.
PADDR[31:0]	Bus de direcciones, de hasta 32 bits.
PSELx	Selección del esclavo activo.
PENABLE	Señal de habilitación. Coordina las transferencias del bus.
PWRITE	Indica el sentido de la transferencia vista desde el *APB bridge*. Nivel alto significa escritura y nivel bajo, lectura.
PRDATA[31:0]	Bus de datos de lectura, de hasta 32 bits.
PWDATA[31:0]	Bus de datos de escritura, de hasta 32 bits.

Tabla 5: Señales del bus APB [1]

2.1.2.2 Funcionamiento del bus APB

El funcionamiento del bus APB puede representarse mediante el diagrama de estados de la Figura 8. A continuación se describen los diferentes estados de los que consta:

- Ocioso (*Idle*). Estado de inactividad del bus, donde no se realizan transferencias. En el momento en que se necesita efectuar una transferencia se pasa al estado *setup*.
- Configuración (*Setup*). En esta fase se selecciona el esclavo al que se quiere acceder activando la señal PSELx apropiada. Además se indica la dirección y el sentido de la transferencia con las señales PADDR y PWRITE, respectivamente. Si se trata de una operación de escritura, además se debe introducir el dato correspondiente en el bus

de datos de escritura. Se permanece en este estado durante un único ciclo de reloj, tras lo cual se pasa al estado *enable*.

- Habilitación (*Enable*). En este estado se completa la transferencia. La señal PENABLE está activa y además todas las señales fijadas en el estado anterior deben conservar estable su valor. Desde esta fase se pasa de forma automática al estado *idle* si no hay más transferencias pendientes, o al estado *setup* si se requiere realizar otra transferencia.

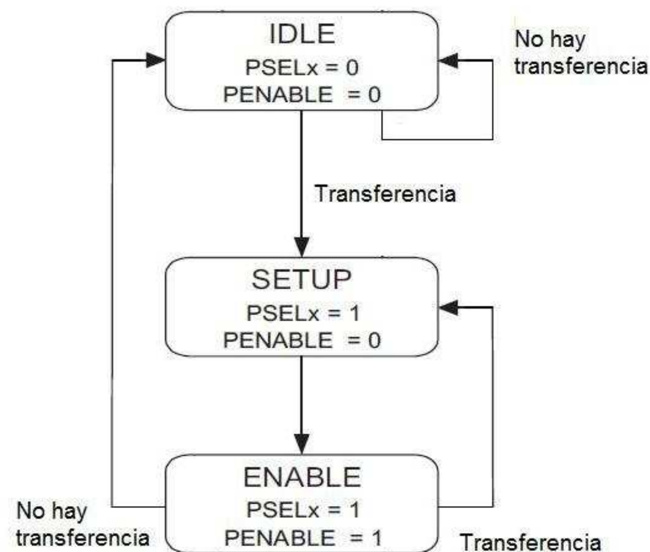

Figura 8: Diagrama de estados del bus APB [1]

Los cronogramas de una operación de escritura y de otra de lectura se muestran en las Figuras 9 y 10, respectivamente.

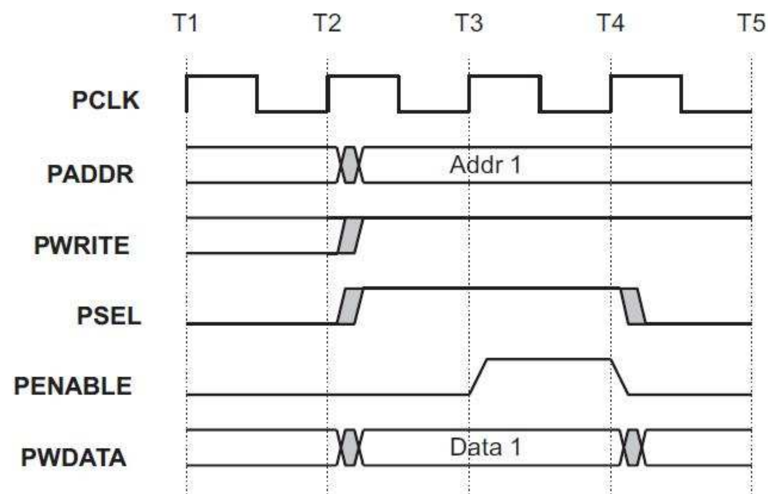

Figura 9: Escritura en el bus APB [1]

En una transferencia en el bus APB, tras el primer flanco de reloj (T2 en las Figuras 9 y 10) se activa la señal PSEL correspondiente al esclavo con el que se realiza la comunicación, entrando en el estado *setup*. Además se indica la dirección de la transferencia con la señal PADDR y si se

trata de una lectura o una escritura estableciendo el valor de la señal PWRITE a 0 o a 1 respectivamente. Tras el siguiente flanco de reloj (T3) se activa la señal PENABLE mientras el resto de señales permanecen con el mismo valor, lo que da lugar al estado *enable*. La transferencia debe completarse cuando se llega al tercer ciclo de reloj (T4), dejando entonces el bus libre para otras transferencias. Para reducir el consumo de energía las señales PADDR y PWRITE no cambian su valor tras finalizar una transferencia hasta que se inicia una nueva transferencia.

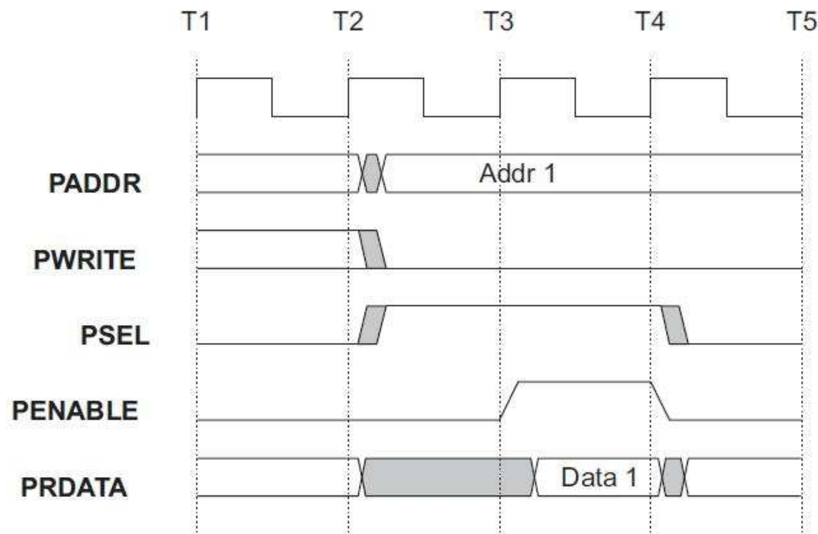

Figura 10: Lectura en el bus APB [1]

En una operación de escritura el dato debe estar disponible en el bus durante los dos ciclos de la transferencia, mientras que en una operación de lectura el esclavo proporciona el dato durante el segundo ciclo.

2.2 Microprocesador sintetizable LEON 3

El microprocesador LEON3, desarrollado por Gaisler Research, es un microprocesador de 32 bits diseñado para sistemas embebidos. Actualmente se emplea en aplicaciones de la industria aeroespacial. Es un microprocesador sintetizable, lo cual quiere decir que se suministra en código fuente. Está constituido por un conjunto de diseños en un lenguaje de descripción de hardware, en este caso VHDL, lo que le otorga múltiples posibilidades de configuración.

El código VHDL del microprocesador forma parte de una librería de módulos denominada GRLIB, la cual engloba los diseños de distintos fabricantes para el desarrollo de sistemas embebidos. Esta librería está disponible en código fuente bajo una licencia gratuita.

En los siguientes apartados se introducen distintos aspectos del microprocesador LEON3. En primer lugar se describe la arquitectura tanto del propio microprocesador como del sistema embebido en su conjunto. A continuación se describe la librería GRLIB. Por último se explica el proceso que se debe seguir para implementar el LEON3 en una aplicación cualquiera.

2.2.1 Arquitectura del microprocesador

El LEON3 es un microprocesador de 32 bits basado en la arquitectura SPARC V8 [2], diseñado para aplicaciones embebidas.

Toda la información que se expone a continuación se ha obtenido de [3].

La Figura 11 muestra el diagrama de bloques del microprocesador. Entre otras características, tiene segmentación de instrucciones en 7 etapas y arquitectura Harvard, es decir, emplea dos memorias caché diferentes, una para datos y otra para programas. Además cuenta con multiplicador y divisor hardware y soporte para depuración en el propio chip. En los siguientes párrafos se realiza un repaso general a los distintos bloques de los que consta el LEON3.

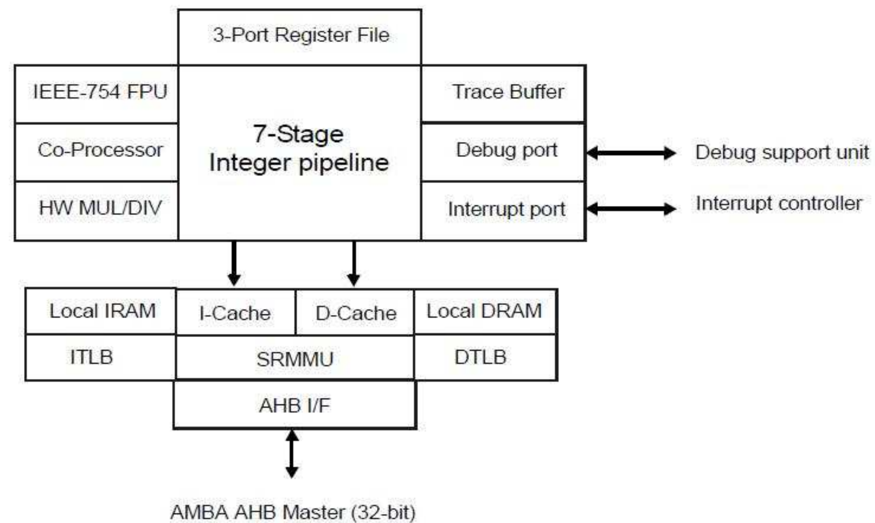

Figura 11: Diagrama de bloques del microprocesador LEON3 [3]

La *integer unit* (IU) implementa el estándar SPARC V8 al completo, incluyendo instrucciones de multiplicación y división por hardware. El número de ventanas de registro es configurable entre los valores 2 y 32. La segmentación de instrucciones consta de 7 fases. Además la *integer unit* dispone de interfaces para una unidad de punto flotante (FPU, *Floating Point Unit*) y para un co-procesador definido por el usuario. La FPU y el co-procesador trabajan en paralelo con la *integer unit*, así que estos bloques no se interrumpen entre sí a menos que exista una dependencia de datos.

El microprocesador LEON3 tiene memorias caché separadas para datos y para programas. Ambas memorias son ampliamente configurables, pudiendo establecer entre 1 y 4 bloques en cada memoria y entre 1 y 256 KB por bloque.

Opcionalmente se puede habilitar una unidad de gestión de memoria (MMU, *Memory Management Unit*) acorde a la especificación SPARC V8. La MMU proporciona correspondencia entre múltiples espacios de direcciones virtuales de 32 bits y la memoria física de 36 bits.

El microprocesador LEON3 permite la depuración en el propio chip. Para facilitar la depuración por software, se pueden habilitar hasta cuatro registros denominados *watchpoint*, cada uno de los cuales puede originar un *breakpoint* en un rango de direcciones de la caché de datos o de instrucciones. Cuando se conecta la unidad de depuración, que es completamente opcional, estos registros pueden usarse para entrar en modo de depuración. La unidad de depuración tiene acceso a todos los registros del procesador y a las dos memorias caché, así como al buffer de traza, un buffer circular que almacena las últimas instrucciones ejecutadas.

El sistema de memorias caché implementa la interfaz como maestro al bus AMBA AHB, según la versión 2.0 de la especificación AMBA. Además cuenta con una interfaz de interrupciones que consta de 15 interrupciones asíncronas.

El LEON3 ha sido diseñado para su uso en sistemas multi-procesador. Cada uno de los microprocesadores LEON3 en un sistema multi-procesador se trata como un maestro distinto del bus AHB. El LEON3 implementa mecanismos que garantizan la coherencia de datos en sistemas de memoria compartida.

En los siguientes sub-apartados se explican con más detalle algunas partes del microprocesador. En concreto se exponen la *integer unit* y la memoria caché.

2.2.1.1 Integer unit

La *integer unit* implementa el conjunto de instrucciones del estándar SPARC V8, a excepción de las instrucciones de punto flotante, de las cuales se encarga la FPU. La ejecución de instrucciones se realiza a través de una segmentación en 7 pasos. La Figura 12 muestra un diagrama de la *integer unit*, indicando además las fases de la segmentación.

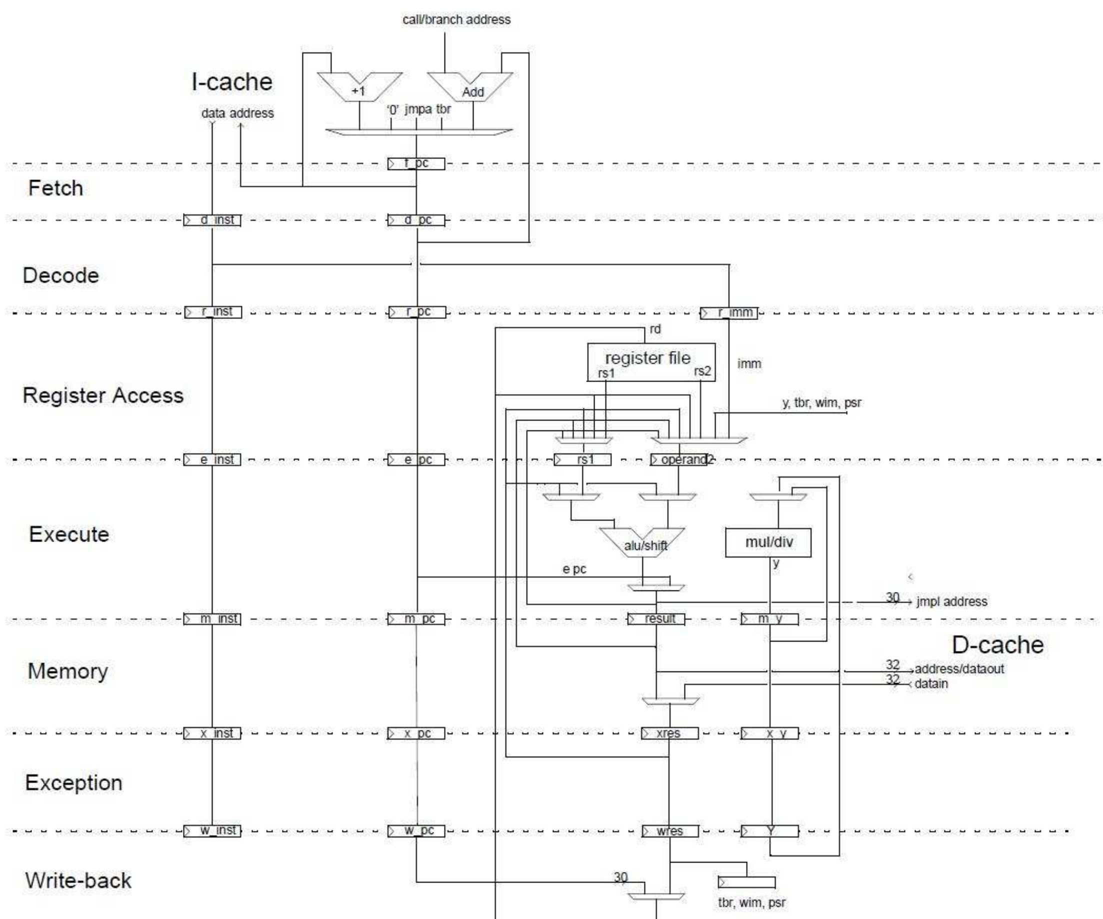

Figura 12: Integer unit [3]

Las fases de la segmentación de instrucciones son:

- Búsqueda de instrucción (*Fetch*). La instrucción se busca en la caché de programas, o en su defecto se envía la petición al controlador de memoria. La instrucción se registra en el interior de la IU.

- Decodificación (*Decode*). Se decodifica la instrucción. Si se trata de una instrucción de llamada o un salto, en esta fase se genera la dirección de destino.

- Acceso a registros (*Register access*). Se toman los datos necesarios del banco de registros. Si alguno de los operandos no se ha escrito todavía en el banco de registros por tratarse del resultado de alguna de las instrucciones previas, se toma en su lugar del registro correspondiente de la propia IU.

- Ejecución (*Execute*). En este paso se realizan las operaciones de tipo aritmético o lógico. Para operaciones de manejo de memoria se genera la dirección correspondiente.

- Acceso a memoria (*Memory*). Se accede a la memoria caché de datos en las operaciones de carga y almacenamiento.

- Excepción (*Exception*). Se resuelven las excepciones e interrupciones. Para lecturas de la caché, el dato se alinea adecuadamente.

- Escritura (*Write*). El resultado de cualquier operación de tipo aritmético, lógico o de caché se almacena en el banco de registros.

Por otra parte, la IU puede configurarse para incluir hasta cuatro registros de *breakpoint*. Cada *breakpoint* consta de dos registros, uno para la dirección y otro para la máscara, tal y como se muestra en la Figura 13. Los bits IF, DL y DS indican si el *breakpoint* se produce en la búsqueda de instrucciones, en la carga de datos o en el almacenamiento de datos, respectivamente.

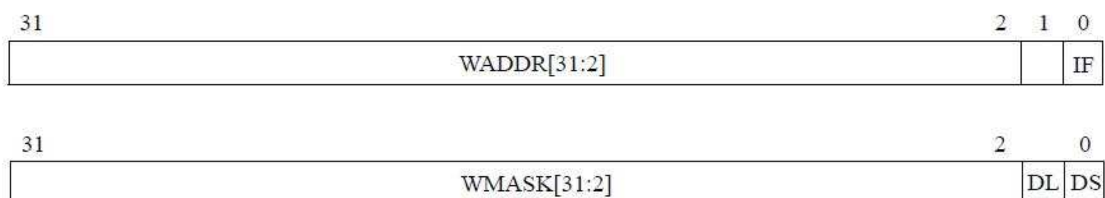

Figura 13: Registros *breakpoint* [3]

La IU dispone además de un buffer de traza, que es un buffer circular que almacena las últimas instrucciones ejecutadas. La operación del buffer de traza se controla mediante la interfaz de depuración y no afecta al funcionamiento del procesador. El tamaño del buffer de traza se puede configurar entre los valores 1 y 64 KB, y tiene un ancho de 128 bits. El buffer almacena el código de operación, la dirección y el resultado de la operación, el dato y la dirección de memoria en las operaciones de carga y almacenamiento, información relativa a las excepciones y una etiqueta de tiempo de 30 bits.

El LEON3 adopta el modelo de excepciones de SPARC. Además, implementa la opción SVT (*Single-Vector Trapping*) que permite reducir el tamaño del código para aplicaciones embebidas. Cuando se habilita el SVT, todas las excepciones que se producen saltan a la

misma dirección del código, en concreto al reset. En tal caso el código de tratamiento de excepciones debe identificar de qué excepción se trata.

2.2.1.2 Memoria caché

El LEON3 dispone de memorias caché separadas para instrucciones y para datos, cada una de las cuales se puede configurar independientemente. Cada caché puede disponer de entre 1 y 4 bloques de tamaño configurable entre 1 y 256 KB. En configuraciones de varios bloques pueden emplearse tres algoritmos de reemplazo distintos: LRU (*Least Recently Used*), LRR (*Least Recently Replaced*) o pseudo-aleatorio. El algoritmo LRR sólo puede emplearse en cachés de 2 bloques.

La cacheabilidad de los datos, es decir, si éstos pueden o no almacenarse en la memoria caché, está determinada por la configuración *plug & play* del bus AHB (ver apartado 2.2.3). El mapeo de memoria de cada uno de los esclavos del bus AHB indica si una zona de memoria en concreto es cacheable por la caché de datos o por la de programa. Esta técnica garantiza que la cacheabilidad de todo el espacio de direcciones es coherente con la configuración actual del bus AHB.

2.2.2 Arquitectura del sistema

Los sistemas embebidos con el microprocesador LEON3 basados en la especificación AMBA siguen un esquema similar al que se muestra en la Figura 14.

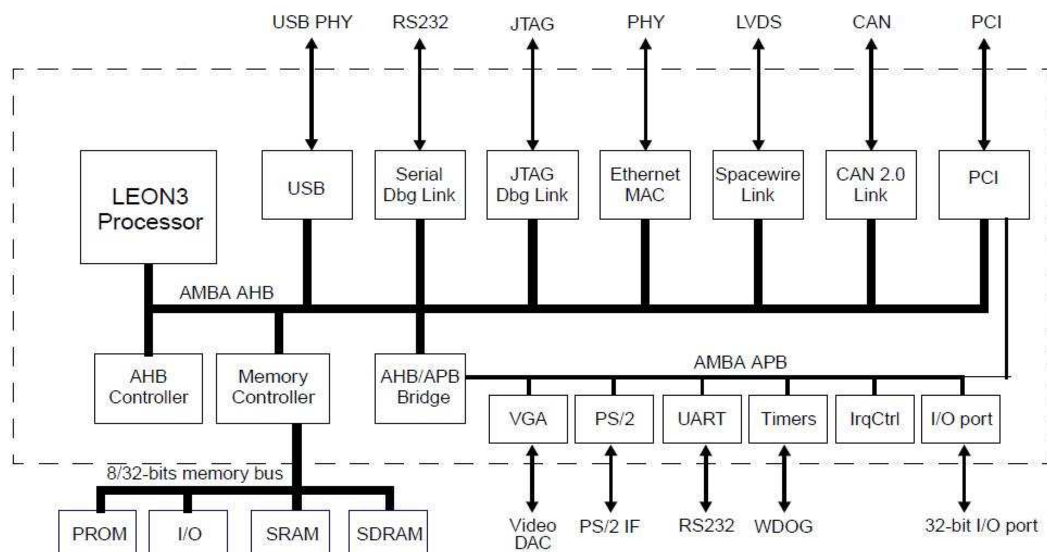

Figura 14: Arquitectura del sistema embebido [4]

El sistema consta de un bus de comunicaciones principal, el AHB, que conecta el microprocesador con el controlador de memoria y otros periféricos rápidos, típicamente las interfaces con otros estándares de comunicación como USB, CAN o Ethernet. El controlador de memoria puede, a su vez, comunicarse con distintos tipos de memorias, incluyendo memorias RAM dinámicas. Además hay un segundo bus de comunicaciones, el APB, para la conexión de periféricos lentos, como son los temporizadores, la Uart o los puertos de propósito general. Los buses AHB y APB están conectados por medio de un módulo denominado puente (*APB bridge*), que es al mismo tiempo esclavo del bus AHB y maestro del bus APB.

2.2.3 GRLIB

La librería GRLIB es un conjunto integrado de módulos IP reutilizables, diseñados para el desarrollo de sistemas embebidos. Los módulos IP se han concebido para el protocolo de comunicaciones AMBA y emplean un método coherente de configuración, simulación y síntesis. Es una librería independiente de vendedor alguno, ofreciendo soporte para distintas herramientas de desarrollo y diversas tecnologías de implementación física.

Toda la información de este apartado se ha obtenido de [4].

GRLIB está organizado en torno a librerías VHDL, cada una de las cuales se asocia a un distribuidor de IP's o *IP vendor*, lo que evita conflictos de nombres entre los distintos módulos IP. Cada una de estas librerías contiene los códigos escritos en VHDL de cada uno de los módulos IP de ese distribuidor y típicamente algunos paquetes, diseños que permiten definir funciones y declarar componentes entre otras cosas. Esta estructura permite añadir y eliminar librerías sin afectar al resto ni modificar los archivos globales.

La librería GRLIB se ha construido asumiendo que la mayoría de los módulos IP se conectan entre sí por medio de un bus interno del sistema. En concreto se emplean los buses AHB y APB de la especificación AMBA 2.0 como buses internos, aunque con algunas modificaciones que se detallan a continuación.

Según el estándar AMBA, el bus AHB cuenta con un decodificador de direcciones centralizado. Esto significa que insertar o eliminar módulos en el sistema no es tan simple como conectar las señales al bus, siendo necesario modificar el decodificador central. En lugar de ello, con el fin de evitar la dependencia respecto de un módulo en particular, se ha implementado una decodificación de direcciones distribuida entre los módulos IP y el controlador del bus.

GRLIB se apoya en *plug & play* para efectuar la decodificación distribuida de direcciones. *Plug & play* hace referencia a la capacidad de detectar la configuración hardware mediante software. Este hecho simplifica el desarrollo de aplicaciones, que de esta forma no se diseñan para una configuración específica de hardware. La información necesaria en GRLIB para la capacidad *plug & play* consta de: un identificador unívoco del módulo IP, el direccionamiento de memoria en el bus, y el vector de interrupciones empleado. Esta información se envía como un valor constante al controlador del bus, que se ubica en un área sólo de lectura en lo más alto del espacio de direcciones del bus. Cualquier maestro del bus AHB puede leer esta información. Para proporcionar dicha información, en cada uno de los módulos del bus AHB se definen unos registros de configuración, que constan de 8 registros de 32 bits cada uno, como muestra la Figura 15. Uno de ellos, denominado palabra de configuración (*configuration word*), contiene los identificadores del distribuidor y del dispositivo, el número de versión e información sobre las interrupciones. A continuación se dejan tres registros para uso definido por el usuario. Los cuatro registros restantes constituyen el denominado BAR (*bank address registers*), que definen el mapeo de memoria, además de indicar si la información presente en ese rango de direcciones se puede almacenar en alguna de las memorias caché. En el caso de los módulos del bus APB, los registros de configuración se componen de la palabra de configuración y un registro para el direccionamiento, como se muestra en la Figura 16.

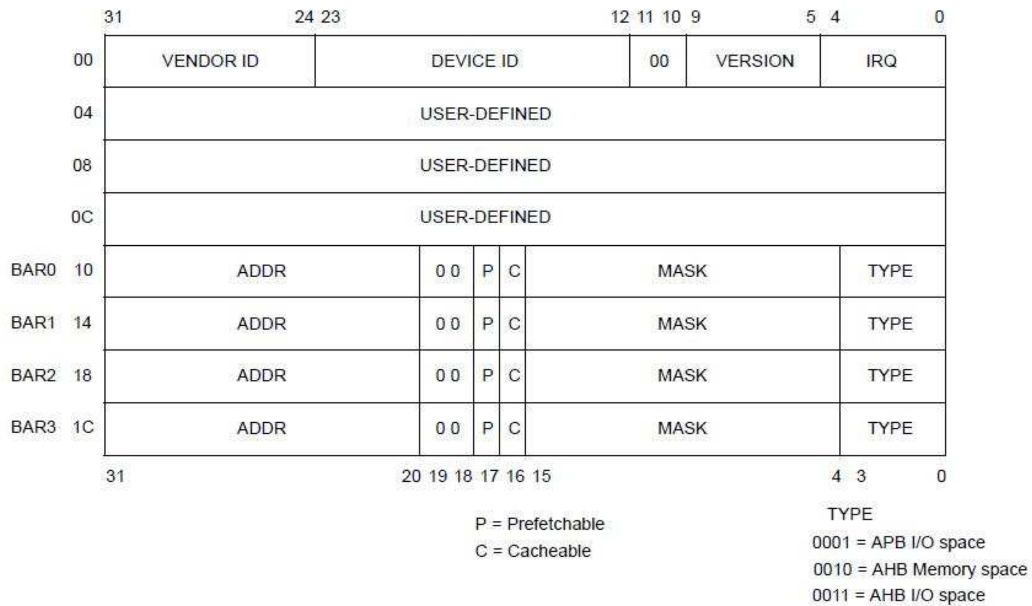

Figura 15: Registros de configuración AHB [4]

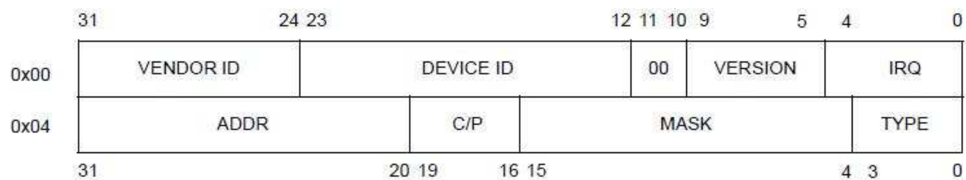

Figura 16: Registros de configuración APB [4]

Por otro lado, la librería GRLIB implementa un esquema de tratamiento de interrupciones unificado, añadiendo un bus de interrupciones de 32 bits a los buses APB y AHB. Cualquier módulo conectado al bus puede activar cualquier interrupción. El controlador de interrupciones observa el vector de interrupciones combinado de todos los módulos y genera las interrupciones apropiadas para los procesadores. De esta forma se pueden generar interrupciones independientemente del procesador y el controlador de interrupciones que se empleen. Además este esquema permite que se compartan interrupciones entre varios módulos y que se resuelvan por software.

2.2.3.1 Organización de GRLIB

GRLIB se distribuye como un archivo comprimido en formato tar que contiene los siguientes directorios:

- *Bin*. Contiene diversos scripts y archivos para las herramientas asociadas con GRLIB.
- *Boards*. Contiene los ficheros de soporte para distintos modelos de dispositivos lógicos programables.
- *Designs*. Almacena las plantillas de los distintos diseños disponibles. Por diseño aquí se entiende un sistema embebido al completo basado en el microprocesador LEON3.
- *Doc*. Documentación de GRLIB.
- *Lib*. Librerías VHDL de los distintos distribuidores de IP's.

- *Netlists*. Archivos que describen la conectividad de diseños específicos de ciertos distribuidores de IP's.
- *Software*. Utilidades software y diversos programas de prueba (*test benches*).
- *Verification*. Programas de prueba definidos por el usuario.

Una buena parte del trabajo con GRLIB se realiza desde los directorios de los diseños preestablecidos que hay dentro de la carpeta *designs*. En cada uno de estos directorios hay una serie de archivos en los que se basa el diseño, que se listan a continuación:

- *Config.vhd*. Contiene los parámetros de configuración del diseño.
- *Leon3mp.vhd*. Representa la entidad de más alto nivel del diseño, e instancia todos los módulos IP que se requieren en base a la configuración del sistema.
- *Testbench.vhd*. Banco de pruebas del diseño, que emula el dispositivo lógico programable.

2.2.4 Proceso de implementación

Para llegar a insertar el LEON3 en un dispositivo lógico programable y hacer que ejecute un programa a partir de la librería GRLIB es necesario seguir una serie de pasos y disponer de las herramientas adecuadas. Es conveniente señalar que existen otras posibilidades para implementar físicamente un diseño descrito en lenguaje de descripción de hardware, la que aquí se describe es la técnica empleada en el presente proyecto.

GRLIB hace uso de la herramienta *make* de generación de códigos para crear los archivos necesarios para diferentes aplicaciones a lo largo del proceso de implementación, por lo que es necesario trabajar en un entorno Unix [4].

El proceso de implementación del LEON3 en un sistema real se muestra en forma de flujograma en la Figura 17. El proceso consta de las siguientes fases:

- Configuración, donde se seleccionan los módulos a incluir en el sistema y se editan las características de cada uno de ellos. Además se pueden decidir ciertos aspectos relativos a la síntesis del diseño como la tecnología y el modelo de FPGA a utilizar.
- Simulación. Esta fase permite probar el diseño escogido en la fase anterior antes de volcarlo en un dispositivo físico.
- Síntesis y enrutado, que consiste en la reprogramación de un dispositivo lógico programable para que adopte la funcionalidad del diseño requerido.
- Programación de la aplicación a ejecutar por parte del microprocesador en un lenguaje de alto nivel.
- Depuración del programa mediante un simulador del microprocesador.
- Generación de la memoria PROM de arranque. El código del programa se compila mediante un compilador cruzado para su inserción en el dispositivo físico.

- Ejecución de aplicaciones en el sistema físico. Se inserta la PROM de arranque en la memoria del microprocesador y se hace funcionar el sistema. Opcionalmente se puede supervisar el funcionamiento del sistema mediante un programa de monitorización.

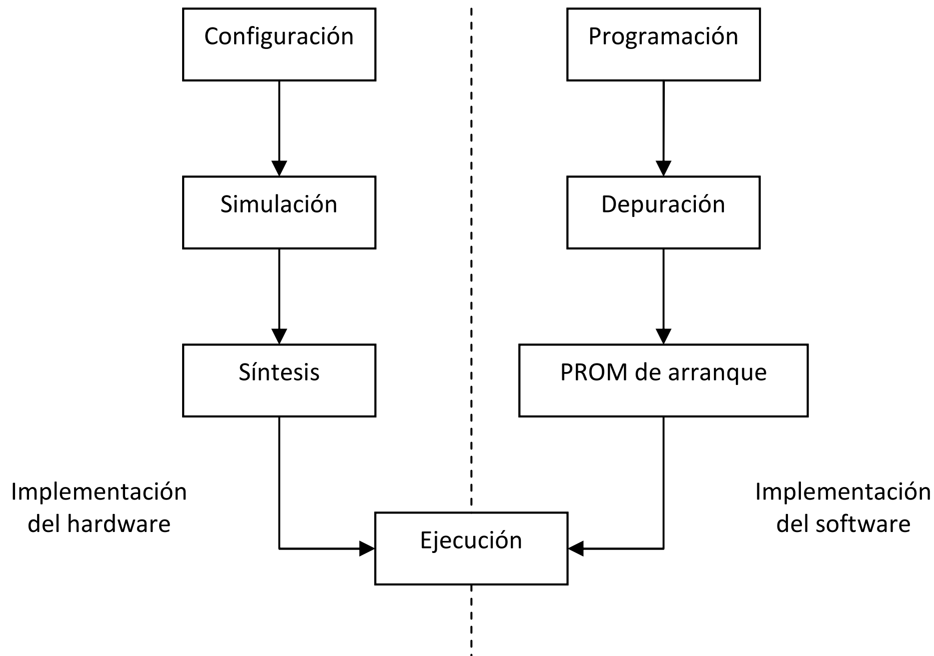

Figura 17: Proceso de implementación de un sistema basado en el LEON3

Como puede observarse el proceso puede dividirse en dos ramas independientes: por un lado se implementa la arquitectura hardware y por otro se desarrolla el software que se ejecutará en el sistema. En el presente proyecto únicamente se ha abordado la implementación hardware, aunque sin llegar a utilizar dispositivo lógico programable alguno.

En los siguientes sub-apartados se explica con mayor detalle cada una de las fases del proceso de implementación completo y se comentan las herramientas informáticas que llevan asociadas.

2.2.4.1 Configuración

El primer paso de la implementación del hardware es decidir la configuración del sistema. Esto incluye seleccionar los módulos que se van a incluir y escoger para cada uno de ellos las características más adecuadas de ente las opciones disponibles.

La configuración de todo el hardware se realiza mediante la herramienta Xconfig, implementada en la propia librería GRLIB. Para ejecutar esta aplicación se debe introducir en un terminal el comando "make xconfig" desde el directorio correspondiente a uno de los diseños incluidos en la librería GRLIB [4]. Esto hace que se despliegue la ventana principal de la aplicación, que se muestra en la Figura 18.

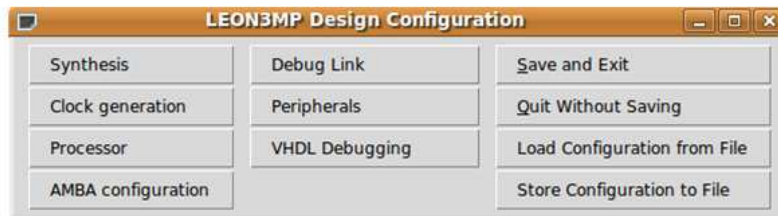

Figura 18: Xconfig

Pulsando en los botones de las dos columnas de la izquierda se despliegan sub-menús donde se pueden modificar las opciones de configuración. Además se pueden seleccionar opciones relativas a la síntesis del diseño. Los botones de la derecha sirven para guardar y cargar la configuración, y para cerrar la aplicación.

Al concluir la configuración se genera un archivo denominado *config.vhd* que contiene una serie de parámetros cuyo valor se corresponde con la configuración seleccionada. Este archivo se ubica en el directorio del diseño.

2.2.4.2 Simulación y síntesis

El diseño puede simularse con un banco de pruebas que emula la placa donde se insertará posteriormente. El banco de pruebas incluye memorias PROM y SDRAM externas, precargadas con un programa de test. Dicho programa se ejecuta en el propio microprocesador LEON3, y por defecto prueba ciertas funcionalidades en el diseño [4].

Para esta fase del proceso de implementación del hardware se necesita un simulador de circuitos electrónicos digitales. GRLIB admite el empleo de cualquiera de los siguientes simuladores: GHDL (GNU VHDL), Aldec Active-HDL, Aldec Riviera, Mentor Modelsim, Cadence NcSim, y Synphony EDA-Sonata. Ejecutar el comando "make scripts" desde el directorio de uno de los diseños permite generar los archivos de proyecto y de compilación para todos estos programas. Alternativamente, existen comandos específicos para cada uno de los simuladores [4].

El programa de test ejecutado por el test bench consta de dos archivos: un archivo de imagen de la memoria prom de arranque (*prom.S*) y el programa de test propiamente dicho (*systest.c*). Ambos archivos se encuentran en el directorio del diseño, y pueden modificarse. Para volver a compilar estos dos archivos tras su modificación se emplea el comando "make soft", lo cual requiere el uso del compilador cruzado BCC [4].

Por otra parte, en la fase de síntesis y enrutado se pasa de los archivos que definen la funcionalidad del diseño a un dispositivo físico que implementa dicho diseño. Los programas de síntesis que soporta GRLIB son: Actel Designer/Libero, Altera Quartus, Cadence RTLC, Lattice ispLEVER, Mentor Leonardo Precision, Synopsis DC, Synplify, y Xilinx ISE/XST. El comando "make scripts" también se emplea para generar los archivos de proyecto y de compilación para todas las herramientas de síntesis. De forma análoga a la fase de simulación, también existen comandos específicos para manejar cada uno de estos programas. Además, algunos de estos programas permiten la simulación en un test bench de los archivos generados en la síntesis [4].

2.2.4.3 Implementación del software

El desarrollo de programas para el LEON3 consta de varias fases: programación, depuración y generación de la PROM de arranque.

La programación se realiza en lenguaje C o en C++. Para pasar del código de alto nivel a un lenguaje que el microprocesador LEON3 pueda manejar se requiere emplear el compilador cruzado BCC (*Bare-C Cross Compiler*). El comando que se emplea para compilar con BCC es "sparc-elf-gcc". El compilador BCC admite todas las opciones del compilador estándar de C, GCC, además de una serie de opciones relativas a las características del sistema para el que se va a compilar, enumeradas en [5].

Para la depuración de programas se emplea la herramienta TSIM, un simulador de la arquitectura SPARC capaz de emular los sistemas basados en el microprocesador LEON3. EL programa TSIM se inicia introduciendo por terminal el comando "tsim-leon" o "tsim-leon3", seguidos de una serie de opciones. TSIM puede ejecutarse solo o conectado a los depuradores GDB (GNU debugger) o Insight, incluidos ambos en el compilador cruzado BCC. Cuando TSIM se ejecuta solo, su manejo se realiza mediante líneas de comandos. Para más información consultar el manual de TSIM [6].

Por último es necesario generar una imagen de la memoria PROM de arranque que contiene el programa depurado. La PROM de arranque se genera mediante la utilidad *mkprom*, incluida en el compilador cruzado BCC. Tras compilar con BCC el programa depurado, la imagen PROM se genera mediante el comando "sparc-elf-mkprom". En este paso se deben indicar como opciones del comando de *mkprom* todos los parámetros dependientes del diseño, tales como tamaño de memorias, la frecuencia del reloj del sistema o la tasa de baudios de la UART. Es necesario que todos estos parámetros coincidan con la configuración del sistema físico, o de lo contrario se creará una imagen defectuosa [5].

2.2.4.4 Integración del software y hardware

El último paso del proceso completo de implementación es aunar los desarrollos hardware y software, cargando el programa desarrollado en la memoria del sistema físico. Esto se realiza con el software GRMON, desarrollado por Gaisler Research. GRMON se conecta a la placa donde se ha implementado el hardware por medio de RS232, JTAG, USB o ethernet. Esta aplicación permite escribir la memoria PROM de arranque del sistema físico en base al archivo de imagen PROM generado con *mkprom*. Además puede supervisar el funcionamiento del sistema físico, lo que permite una depuración en el propio chip. GRMON se puede ejecutar junto a los depuradores GDB o Insight. Véase [5] para más detalles.

2.3 Técnicas de detección de errores para sistemas embebidos

En electrónica digital, la detección de errores se basa en la redundancia, elementos adicionales que no aportan información nueva pero que son necesarios para verificar la validez de ciertos datos relevantes para la aplicación en cuestión.

A lo largo de los años se han desarrollado múltiples formas de implementar esta redundancia para la supervisión y control de aplicaciones diversas. Estas técnicas se pueden agrupar según la importancia relativa entre software y hardware en dicha implementación. Así, se tienen técnicas de detección software, hardware e híbridas. A continuación se exponen las peculiaridades de cada uno de estos grupos junto con algunos ejemplos característicos.

2.3.1 Técnicas de detección Software

Las técnicas de detección software se fundamentan en la modificación del código del programa que se ejecuta, duplicando instrucciones y añadiendo instrucciones de control.

Las técnicas más antiguas se basan en la replicación del programa y posterior votación entre las distintas ejecuciones [7] las cuales, aun siendo efectivas, no están estandarizadas. Por lo tanto queda en manos de los programadores la tarea de implementar la replicación y la votación de la forma más adecuada para el programa.

Posteriormente se han desarrollado técnicas que simplemente añaden algunas instrucciones de control para detectar errores, como en [8]. Algunas de estas técnicas simplifican la labor de los programadores al poder aplicarse directamente al software. Sin embargo algunas de estas técnicas no son aplicables con generalidad, sino que están destinadas a robustecer ciertos tipos específicos de aplicaciones.

Más recientemente se han desarrollado técnicas aplicables directamente al código fuente de un programa, e incluso algunas de ellas pueden emplearse en una gran variedad de aplicaciones. Estas técnicas han sido desarrolladas para detectar errores que afecten al flujo del programa, recibiendo por ello la denominación de técnicas de supervisión del flujo de control (*control-flow checking*). Este grupo de técnicas se fundamenta en la división del código de un programa en *bloques*, secuencias de instrucciones que, en ausencia de errores, se ejecutan de forma consecutiva de principio a fin. De este modo, tan sólo la última instrucción de cada bloque puede ser un salto o una llamada a rutina, y únicamente la primera instrucción del bloque puede ser el destino de un salto o el inicio de una rutina. Las conexiones entre bloques quedan perfectamente definidas y por tanto pueden detectarse saltos no acordes con el flujo del programa. Ejemplos de estas técnicas se describen en [8] y [9].

Por último, existe una serie de técnicas dirigidas a detectar fallos sobre los datos de programa, basados en la replicación de variables. Se emplean controles de consistencia entre las variables originales y sus réplicas para determinar la presencia de errores. Dos ejemplos de este tipo de técnicas se exponen en [10] y [11].

Estas técnicas no requieren modificaciones del hardware para su implementación y por tanto pueden emplearse a muy bajo coste. Por otra parte algunas de estas técnicas son muy efectivas en la detección de fallos que afecten a los datos del programa o al flujo de control. Sin embargo su principal inconveniente es la excesiva sobrecarga de instrucciones que originan, lo cual impide su aplicación a sistemas con tiempo de respuesta crítico.

2.3.2 Técnicas de detección Hardware

El enfoque de detección hardware se basa en la inserción de un módulo adicional en el sistema, denominado *watchdog processor* [12], que supervisa el comportamiento del procesador principal. Un *watchdog processor* puede realizar tres tipos de controles: control de acceso de memoria, de consistencia y de flujo de control.

- El control de acceso a memoria (*Memory-accesses check*) permite detectar accesos a memoria inesperados por parte del procesador principal. Como ejemplo, en la solución propuesta en [13] el *watchdog processor* conoce las zonas de la memoria de datos y de código a las que se puede acceder en cada momento, y genera una señal de error en caso de detectar un acceso no permitido.

- El control de consistencia (*Consistency check*) permite comprobar si el valor de una variable es razonable según el conocimiento que se posee del programa a ejecutar. Así el *watchdog processor* valida cada dato que escribe o lee el procesador principal mediante comprobaciones de rango o explotando el conocimiento sobre las relaciones entre variables [14].

- La comprobación del flujo de control (*Control-flow check*) permite verificar que los saltos ejecutados en el programa son consistentes con el diagrama de flujo de dicho programa. Según la rigurosidad del control se tienen dos tipos distintos de *watchdog processor*, activo y pasivo. El *watchdog processor* activo ejecuta el programa del procesador principal a la misma vez que éste, y comprueba que ambos programas siguen la misma evolución [15]. Por su parte el *watchdog processor* pasivo no ejecuta programa alguno, en su lugar genera una firma a partir de la información presente en el bus de comunicaciones del procesador principal [16].

Este conjunto de técnicas apenas requiere modificaciones del programa a ejecutar, únicamente es necesario añadir instrucciones para comunicarse con el *watchdog processor*, por lo que garantiza una sobrecarga de instrucciones mínima. Pero por otro lado la inserción de un módulo adicional supone un coste extra en componentes y en área.

2.3.3 Técnicas de detección híbridas

Las técnicas de detección híbridas, de desarrollo relativamente reciente, constituyen un punto intermedio entre las técnicas software y las técnicas hardware. Este enfoque se basa en la modificación del programa a ejecutar de forma similar a las técnicas puramente software, pero los esfuerzos computacionales, como la generación de firmas o los test de validez, son ejecutados por un módulo externo, denominado I-IP.

Un ejemplo de este tipo técnicas es el que se expone en [17]. En él se efectúan la supervisión del flujo de control y el control de datos mediante una implementación híbrida basada en técnicas software.

La supervisión del flujo de programa se efectúa mediante la división del programa en bloques, de forma similar a las técnicas software equivalentes. Aplicando un conjunto de reglas al código de alto nivel del programa se identifican los bloques de los que consta y se agregan dos instrucciones nuevas a cada uno de esos bloques, IIPset() e IIPtest(), cuya función es la siguiente:

- IIPset(B) indica que el programa va a entrar en el bloque B.

- IIPtest(B) indica que el bloque B pertenece al conjunto de predecesores del siguiente bloque.

Previamente al inicio de un bloque se debe insertar una función IIPtest() por cada uno de los predecesores de ese bloque, y a continuación se debe añadir la función IIPset() correspondiente al propio bloque. Cada una de estas instrucciones se traduce en una operación de escritura sobre un registro del I-IP. Cuando se escribe un valor en el registro correspondiente a la instrucción IIPtest() se compara este valor con una firma almacenada en el I-IP, y en función de ello se activa o desactiva un flag interno. Cuando se escribe un valor en el registro asociado a la función IIPset() se comprueba el valor del flag y en función de ello se activa la señal de error o no. En caso negativo la firma se actualiza con el valor del registro asociado a la instrucción IIPset().

Para detectar los fallos que afectan a los datos del programa deben aplicarse unas transformaciones al código del programa por las que todas las variables definidas se replican y todas las instrucciones de lectura y/o escritura de variables se duplican, empleando las variables replicadas en lugar de las originales. El I-IP se encarga de observar el bus de comunicaciones a la búsqueda de las operaciones de lectura de dichas variables. Además debe identificar el acceso a una variable y a su réplica y comprobar que ambas tienen el mismo valor. Para simplificar el diseño se asume que el segmento de memoria asignado a los datos de programa se divide en dos partes, una para las variables originales y otra para sus réplicas; además se supone que siempre se accede a una variable original antes que a su respectiva réplica. De esta forma es sencillo determinar si una variable es el original o la copia y además se genera un error si aparece la réplica en primer lugar.

El I-IP contiene una memoria CAM que almacena la dirección y el valor de cada una de las variables originales cuando se leen. Más tarde, cuando se accede a alguna de las réplicas, se localiza en la memoria CAM su correspondiente variable original, se comprueba si ambas tienen el mismo valor y se borra dicha entrada de la memoria. Este modo de proceder permite realizar un control adicional del flujo de programa, ya que el acceso a una variable y a su réplica se producen en el mismo bloque de instrucciones. Por tanto, en ausencia de fallos que afecten al flujo, la memoria CAM debe estar vacía al final de cada uno de los bloques de instrucciones.

Desde el punto de vista del hardware, esta solución puede implementarse de forma sencilla en sistemas embebidos ya que apenas requiere de cambios en el hardware, aparte de insertar el propio I-IP. Aunque si se pretende que la técnica sea aplicable con generalidad el I-IP no debe incluir información alguna sobre el código de la aplicación a supervisar. Por otro lado se reduce la sobrecarga de instrucciones respecto de los enfoques software puros.

3 DISEÑO DEL MÓDULO DE DETECCIÓN DE FALLOS

En este capítulo se describen todos los aspectos relevantes que han involucrado el diseño del módulo de detección de fallos, desde la funcionalidad deseada hasta la inserción del módulo en la librería de componentes GRLIB.

La finalidad del módulo de detección es supervisar las comunicaciones a través del bus principal del sistema para detectar los errores transitorios que se produzcan. La solución propuesta debe ser capaz de reconocer una amplia proporción de errores. Si bien éste es el objetivo principal, es deseable que el módulo de detección sea lo más económico posible, tanto en coste como en espacio. Asimismo, el módulo de detección debería afectar lo menos posible a las prestaciones del sistema al cual se conecte.

El módulo de detección se ha diseñado como un esclavo del bus AMBA AHB [1], cuya misión es supervisar las comunicaciones de un maestro a un esclavo específicos, aunque con modificaciones mínimas del diseño se podría emplear para espiar las comunicaciones en sentido contrario. Para ello debe ser configurado de forma adecuada y es necesario que cada uno de los bloques de comunicación, denominados *rutinas*, se envíe dos veces, aunque no es imprescindible que tengan lugar de forma consecutiva. En cada una de dichas ejecuciones se genera una firma a partir de los datos que se transmiten por el bus de comunicaciones, determinando la existencia de error mediante comparación de los dos resultados obtenidos. Adicionalmente se ha incluido un temporizador *watchdog* que genera un error si un bloque de comunicaciones tarda en enviarse más de lo indicado en la configuración, de modo que se prevenga un posible bloqueo del sistema.

En el diseño del módulo de detección se ha asumido que el esclavo que se observa no efectúa transferencias partidas [1], ya sea porque no las implementa o debido a que es lo suficientemente rápido como para no hacer uso de ellas.

En los siguientes apartados se describe en profundidad el diseño del módulo de detección. En primer lugar se explica en detalle la funcionalidad del módulo y de cada uno de los bloques que lo componen. A continuación se expone la organización de los archivos que componen el diseño. Por último, se describen los pasos que se han seguido para añadir el módulo a la librería GRLIB.

3.1 Funcionalidad del módulo de detección

En su forma más simple, el proceso completo de utilización del módulo se compone de: configuración del módulo previa a la síntesis, programación del módulo ya sintetizado para la supervisión de bloques de comunicación (*rutinas*), y por último el control de las dos ejecuciones de la rutina que son necesarias. En los siguientes párrafos se describen cada una de estas fases.

Durante la fase de configuración del LEON3 mediante el programa Xconfig (véase 2.2.4) debe indicarse el número de bits necesario para indexar el banco de registros del módulo de detección, que determina de forma indirecta el número de rutinas que el módulo será capaz de gestionar simultáneamente. Por otra parte debe indicarse el maestro objetivo y el esclavo objetivo, es decir, el maestro y el esclavo que se van a observar. Además se selecciona el rango de direcciones que se le asignará al módulo de detección en el bus AHB por medio de los 12 bits más significativos del bus de direcciones del bus AHB. Por último se indican las interrupciones que se activarán en caso de producirse un error de escritura o de ejecución. Un error de escritura indica que se ha intentado escribir en una dirección del banco de registros

ilegal, debido a que está protegida contra escritura; mientras que un error de ejecución significa que el módulo de detección ha detectado un error en la comunicación entre el maestro y el esclavo seleccionados. Como resultado de estas opciones la aplicación Xconfig genera un archivo en lenguaje VHDL que contiene una serie de parámetros cuyo valor se adecúa a la configuración seleccionada (ver apartado 2.2.4.1). Estos parámetros se emplean en el diseño como *genéricos*, parámetros que definen la configuración del hardware subyacente (véase 3.1.1).

La Figura 19 muestra de forma gráfica el funcionamiento del módulo de detección para el tratamiento de una rutina. Una vez que se ha implementado el sistema, en primer lugar debe enviarse al módulo de detección la configuración de la rutina. Tras ello se puede proceder a la supervisión de la misma.

La configuración de rutinas consiste en indicar al módulo de detección la información necesaria para supervisar cada una de dichas rutinas. Esta información consta de las direcciones de inicio y de fin de la rutina, y del número máximo de ciclos de reloj en el que debe completarse la ejecución de dicha rutina. El formato de dirección necesario para pasar esta información al módulo de detección se describe entre los apartados 3.1.2 y 3.1.3. Es importante señalar que este módulo de detección no es capaz comprobar la coherencia entre las direcciones y los módulos IP, quedando en manos de los programadores la tarea de verificar que las direcciones escritas en el banco de registros se corresponden con el esclavo que se va a espiar. Si alguno de los datos que se envían al módulo de detección de fallos no puede almacenarse, debido a que está habilitada la protección contra escritura, se genera un error de escritura.

Una vez concluida la configuración del módulo de detección, éste vigila el bus de comunicaciones esperando que se produzca una transferencia del maestro al esclavo deseados y en una de las direcciones previamente almacenadas como direcciones de inicio. En ese momento se inicia la supervisión de una de las rutinas y se genera una firma (ver apartado 3.1.5) en base a las transferencias que vayan del maestro al esclavo elegidos. Al mismo tiempo se pone en funcionamiento un temporizador *watchdog* (véase 3.1.4) que activa el error de ejecución en caso de que alcance el número máximo de ciclos de reloj indicado en el banco de registros. Si el maestro realiza una escritura en la dirección de fin de la rutina en curso antes de que desborde el *watchdog*, se considera que dicha rutina ha finalizado. Cuando termina la rutina se toma la firma resultante, la cual se almacena en el banco de registros si se trata de la primera ejecución de la rutina. En caso contrario se compara con la firma resultante de la primera rutina, activando el error de ejecución si las firmas son distintas entre sí. Tras este proceso el módulo estaría de nuevo preparado para entrar en una nueva rutina, aunque por motivos de diseño no es posible entrar en una rutina inmediatamente después de finalizar la anterior, debe transcurrir al menos un ciclo de reloj entre ambos eventos (ver apartado 3.1.4). Esto significa que, por ejemplo, un bloque de transferencias en modo ráfaga [1] no se puede dividir en varias rutinas.

Cuando se desea supervisar la ejecución de varias rutinas a la vez se pueden configurar todas al inicio, pero también existe la posibilidad de configurar algunas rutinas entre las ejecuciones de otras, o incluso durante las propias ejecuciones de otras rutinas. Por otra parte sólo es posible supervisar la ejecución de una rutina en cada momento, pero no es necesario que las dos ejecuciones de una misma rutina se realicen de forma consecutiva e inmediata. Pueden intercalarse ejecuciones de distintas rutinas.

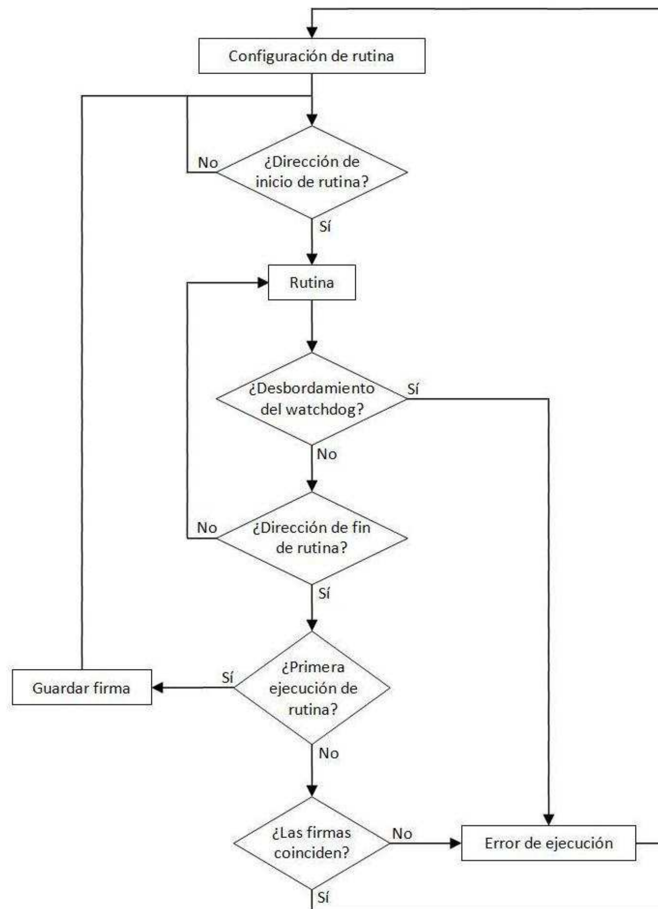

Figura 19: Flujograma de operación del módulo

El diseño se divide en cuatro sub-módulos, tal y como muestra la Figura 20:

- *Interfaz*, que es la interfaz de comunicaciones con el bus AHB.
- *Regbank*, banco de registros que almacena la configuración de cada una de las rutinas así como el resultado de la primera ejecución de cada rutina.
- *Control*, máquina de estados que determina el funcionamiento del módulo.
- *Espía*, bloque encargado de generar la firma a partir del bus de datos del bus AHB.

El funcionamiento detallado de cada uno de estos bloques se expone a continuación, no sin antes hacer un inciso sobre los parámetros de configuración, denominados *genéricos*, que emplea el módulo de detección.

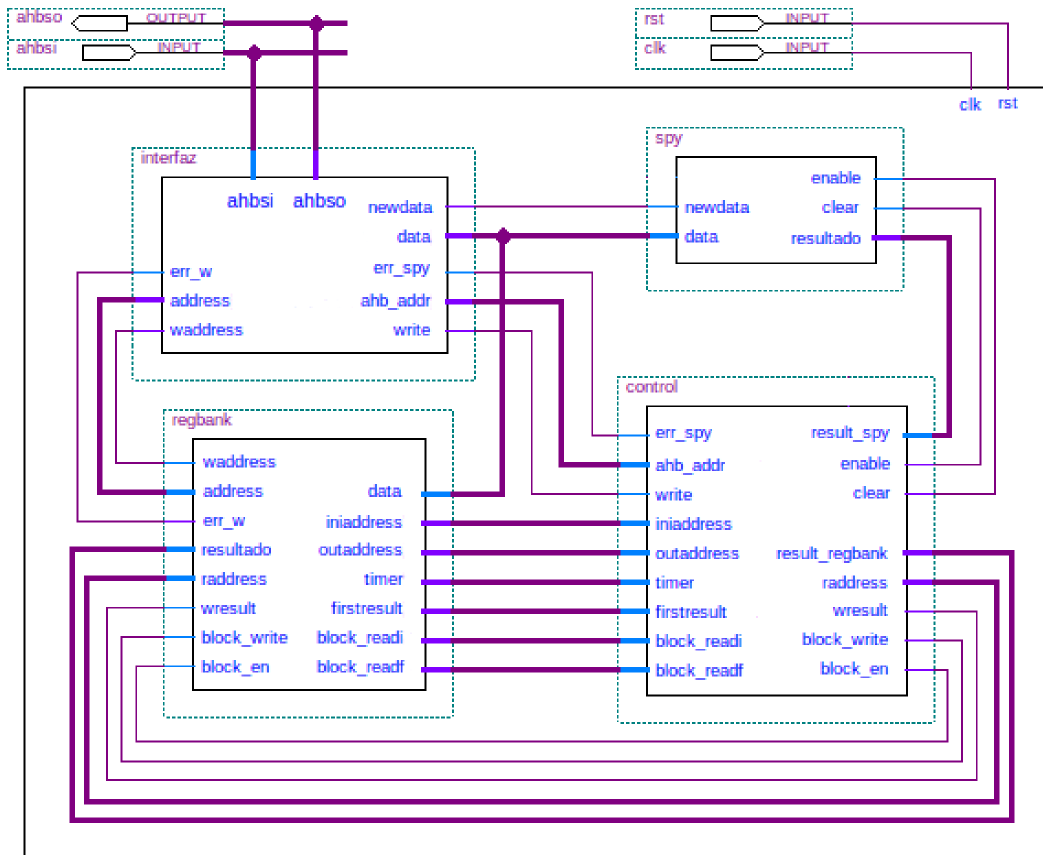

Figura 20: Diagrama de bloques del módulo IP

3.1.1 Genéricos del módulo de detección

Los genéricos son una serie de parámetros cuyo valor es constante dentro de una implementación del diseño, pero que pueden variar su valor de una implementación a otra. Proporcionan flexibilidad en el diseño, permitiendo modificar de forma sencilla la configuración del hardware del diseño.

En el presente proyecto los valores de los genéricos se asignan durante la configuración del sistema mediante la aplicación Xconfig (ver apartado 2.2.4.1). A continuación se describe cada uno de los genéricos empleados en el módulo de detección:

- *Hindex*. Posición que ocupa el módulo de detección como esclavo del bus AHB.
- *Spyaddr*. Los 12 bits más significativos, en formato hexadecimal, de las direcciones del bus AHB que corresponden al módulo de detección, lo que indica de forma indirecta el rango de direcciones asignado al módulo de detección.
- *Hirq_w*. Número de la interrupción asignada a error de escritura.
- *Hirq_spy*. Número de la interrupción asignada a error de ejecución.
- *Nummaster*. Posición que ocupa el maestro a espiar como maestro del bus AHB.
- *Numslave*. Posición que ocupa el esclavo a espiar como esclavo del bus AHB.

- *Addrsize*. Número de bits necesario para indexar el banco de registros del módulo de detección, lo cual indica de forma indirecta el número máximo de rutinas que pueden tratarse simultáneamente (2 elevado a este valor). Para direccionar por completo el banco de registros se necesitan 2 bits adicionales.

Por otra parte hay un grupo de genéricos que no tienen relevancia en la funcionalidad del módulo de detección y cuya misión es permitir la inserción en la librería GRLIB, dichos parámetros se tratan en el apartado 3.3.

3.1.2 Interfaz

La Figura 21 muestra las señales de entrada y salida del bloque de interfaz, las cuales se detallan a continuación.

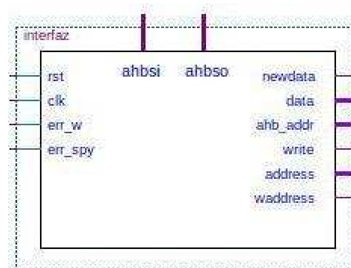

Figura 21: Bloque de interfaz

Señales de entrada:

- *Rst*. Reset asíncrono, activo por nivel bajo.
- *Clk*. Señal de reloj.
- *Ahbsi*. Entrada de esclavo del bus AHB.
- *Err_spy*. Indica que se ha detectado un error de ejecución.
- *Err_w*. Indica que se ha producido un error en la escritura de datos en el banco de registros.

Señales de salida:

- *Ahbso*. Salida de esclavo del bus AHB.
- *Newdata*. Indica que hay un nuevo dato en el bus AHB.
- *Data*. Bus de datos del bus AHB.
- *Ahb_addr*. Bus de direcciones del bus AHB.
- *Write*. Indica que se produce una transferencia del maestro al esclavo seleccionados.
- *Address*. Dirección para escritura de los registros de configuración del banco de registros.
- *Waddress*. Habilitar escritura de dirección en el banco de registros.

La interfaz es el bloque dedicado a gestionar las comunicaciones con el bus AHB. El funcionamiento de la interfaz se describe de forma gráfica en la Figura 23, con excepción de la generación de interrupciones. La interfaz vigila constantemente el bus de comunicaciones a la espera de una transferencia entre el maestro objetivo y el esclavo objetivo, o bien una comunicación con el propio módulo de detección de fallos. Si se trata de una transferencia de tipo *secuencial* o *no secuencial* (véase 2.1.1.5) del maestro objetivo al esclavo objetivo, entonces la interfaz indica al resto de bloques del módulo de detección que se ha producido una transferencia susceptible de ser vigilada. En tal caso, se activa la señal *write* en la fase de dirección de dicha transferencia, que indica al bloque de control que se trata de una operación de escritura entre el maestro y el esclavo deseados, y la señal *newdata* durante el primer ciclo de reloj de la fase de datos de esa transferencia (ver apartado 2.1.1.4), que indica al bloque espía que hay un nuevo dato presente en el bus para que lo capture.

Por otra parte, si se produce una comunicación con el propio módulo de detección, se deben considerar cuidadosamente las características de la transferencia. El espacio de direcciones asignado al módulo de detección se ha concebido como un área de escritura exclusivamente, así que cualquier intento de lectura del módulo de detección generará una respuesta de *error* en dos ciclos por parte de la interfaz (véase 2.1.1.7). En caso contrario, se estudia el tipo de transferencia en cuestión. Si se trata de una transferencia de tipo *secuencial* o *no secuencial* (ver apartado 2.1.1.5) se habilita la escritura en el banco de registros, siempre que se realice sobre una dirección del banco de registros que exista realmente. Para que una dirección se considere válida debe construirse de la siguiente forma:

- Los dos bits menos significativos del bus de direcciones del bus AHB se ignoran, ya que el LEON3 es un microprocesador de 32 bits y por tanto realiza los incrementos de dirección de 4 en 4.

- A partir del bit 2 de la dirección, un cierto número de bits igual al valor del genérico *addrsize* más 2 constituyen la señal de direccionamiento del banco de registros (*address*). Dentro de este grupo de bits, los dos más significativos indican el tipo de registro al que se accede (dirección de inicio, dirección de fin, o número máximo de ciclos), mientras que el resto representan el índice de rutina. Se remite al apartado 3.1.3 para conocer la codificación de dicha señal.

- Los 12 bits más significativos de la dirección deben corresponderse con el valor del genérico *spyaddr*.

- El resto de bits deben tomar el valor '0'.

La Figura 22 muestra el modelo de dirección válida descrito previamente.

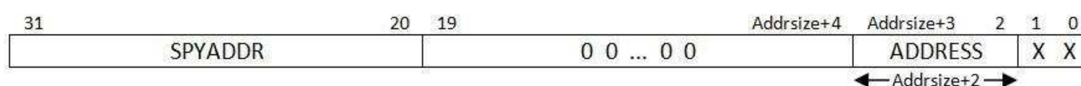

Figura 22: Dirección válida

En el caso de producirse un intento de escritura en una dirección no válida se genera una respuesta de *error* de dos ciclos de reloj. En el resto de casos, incluyendo aquellas transferencias en las que no participa el módulo de detección, la interfaz proporciona una respuesta de tipo *okay* (véase 2.1.1.7).

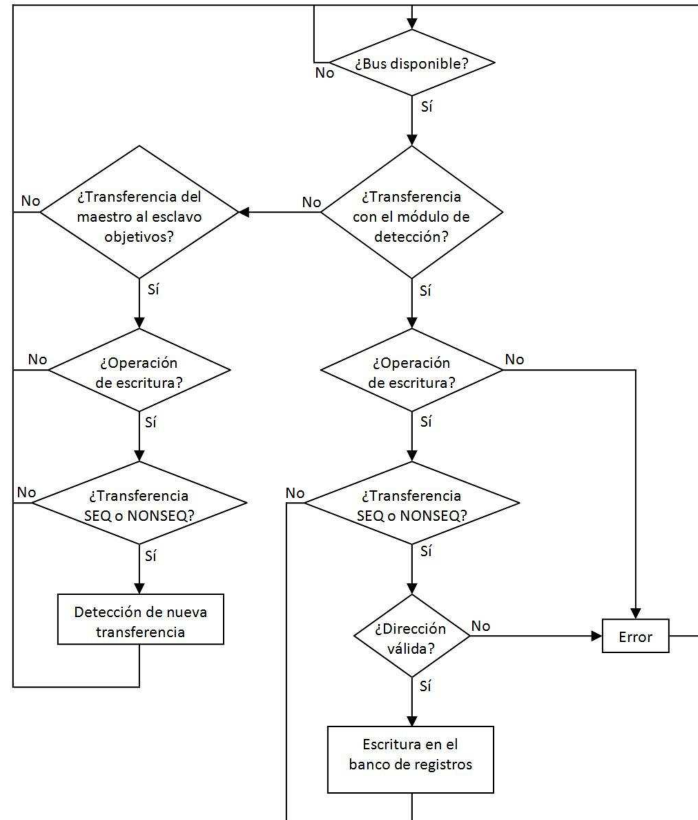

Figura 23: Flujograma de operación de la interfaz

Por otro lado, la interfaz también se encarga de generar las interrupciones asociadas a los errores de escritura y de ejecución cuando lo indican el banco de registros y el bloque de control por medio de las señales *err_w* y *err_spy*, respectivamente.

3.1.3 Banco de registros

La Figura 24 muestra la interfaz del banco de registros. A continuación se detalla la función de cada una de las señales de las que consta.

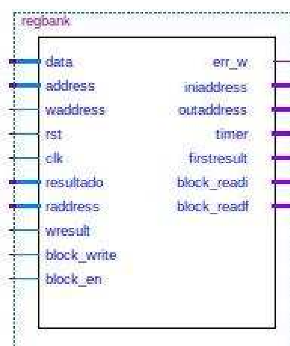

Figura 24: Banco de registros

Señales de entrada:

- *Rst*. Reset asíncrono, activo por nivel bajo.
- *Clk*. Señal de reloj.
- *Data*. Bus de datos del bus AHB.
- *Address*. Dirección para escritura de los registros de configuración del banco de registros.
- *Waddress*. Habilitar escritura de dirección en el banco de registros.
- *Resultado*. Firma resultante de la primera ejecución de una rutina para almacenar en registro.
- *Raddress*. Dirección de los registros de resultado del banco de registros, y dirección para lectura de los registros de configuración. Se corresponde con el índice de la rutina en ejecución.
- *Wresult*. Habilitar escritura de resultado en el banco de registros.
- *Block_write*. Modificación de la protección contra escritura.
- *Block_en*. Habilitar modificación de la protección contra escritura.

Señales de salida:

- *Err_w*. Indica que se ha producido un error en la escritura de datos en el banco de registros.
- *Iniaddress*. Direcciones de inicio de todas las rutinas.
- *Outaddress*. Dirección de fin de la rutina actual.
- *Timer*. Número máximo de ciclos de reloj permitidos para la rutina actual.
- *Firstresult*. Resultado de la primera ejecución de la rutina actual.
- *Block_readi*. Estado de la protección contra lectura de las direcciones de inicio.
- *Block_readf*. Estado de la protección contra lectura de las direcciones de fin.

Este bloque es el encargado de almacenar toda la información relevante para cada una de las rutinas, el cual se compone de dos bancos de registros en parte independientes. Uno contiene la configuración de las rutinas, cuya escritura está gobernada por la interfaz, mientras que el otro almacena los resultados de la primera ejecución de cada rutina, cuya escritura depende del sub-módulo de control. La salida de ambos bancos de registros está gobernada por el bloque de control.

La parte de resultados del banco de registros está formada por dos clases de registros, como muestra la Figura 25. Por un lado está el registro de resultado propiamente dicho, denominado *regresult*, y por otro lado un bit de protección contra escritura para cada una de las rutinas, denominado *block_w*. Ambos elementos se direccionan mediante la misma señal

(*raddress*), aunque cuentan con señales independientes de habilitación. La protección contra escritura de una rutina cualquiera se activa cuando se inicia la primera ejecución de dicha rutina y permanece así hasta que concluye su segunda ejecución o se detecta un error de ejecución. Dicha protección impide que se modifique la configuración de la rutina entre estos dos eventos.

Bw 0	REGRESULT 0 [31:0]
Bw 1	REGRESULT 1 [31:0]
...	...
Bw n	REGRESULT n [31:0]

Bw : Block_w (protección contra escritura)

Figura 25: Registros de resultado del banco de registros

Los registros de configuración del banco de registros se componen de la dirección de inicio y de fin de la rutina, denominados *reginicio* y *regfin* respectivamente, del número de ciclos de reloj permitidos, denominado *regtime*, y de la protección contra lectura de las direcciones de inicio y de fin, denominados *block_ri* y *block_rf* respectivamente, como puede verse en la Figura 26. Los registros *reginicio*, *regfin* y *regtime* comparten la misma señal de habilitación (*waddress*) y de direccionamiento (*address*). Si el bit más significativo de *address* vale '1' se accede a un registro *regtime*, mientras que si está a '0' se accede a una dirección de inicio o de fin, dependiendo de si el siguiente bit de *address* vale '1' o '0' respectivamente. Los demás bits de *address* indican la posición del banco de registros a la que se accede. La protección contra lectura se activa al efectuar un reset del módulo y se desactiva al escribir un dato en el registro correspondiente. La protección contra lectura impide al bloque de control comparar la dirección de inicio de la rutina en cuestión con la dirección del bus. Por tanto para que el módulo pueda espiar una rutina en concreto debe haberse escrito tanto la dirección de inicio como la de fin. Esto se hace así para evitar que el módulo de detección comience a observar la ejecución de una rutina sólo porque en el bus de direcciones ha aparecido el valor al que se inicializan los registros del banco de registros. Es necesario señalar además que los registros *regtime* de todas las rutinas se inicializan al valor 0. Por tanto se debe escribir otro valor en este registro de modo que no se active el error de ejecución nada más entrar en la rutina.

Address	"1X..."		"01..."		"00..."
"...000"	REGTIME 0 [31:0]	Bri 0	REGINICIO 0 [31:0]	Brf 0	REGFIN 0 [31:0]
"...001"	REGTIME 1 [31:0]	Bri 1	REGINICIO 1 [31:0]	Brf 1	REGFIN 1 [31:0]
...	...	...	...	...	...
"...111"	REGTIME n [31:0]	Bri n	REGINICIO n [31:0]	Brf n	REGFIN n [31:0]

Bri : Block_ri (protección contra lectura de la dirección de inicio)
Brf : Block_rf (protección contra lectura de la dirección de fin)

Figura 26: Registros de configuración del banco de registros

El bloque de control puede leer los datos almacenados en el banco de registros empleando la señal *raddress* como direccionamiento, ya que su valor se corresponde con el índice de la rutina en curso, tal y como se explica en el siguiente apartado.

3.1.4 Control

La Figura 27 muestra las señales del bloque de control que le permiten comunicarse con los demás módulos. Cada una de estas señales se explica a continuación.

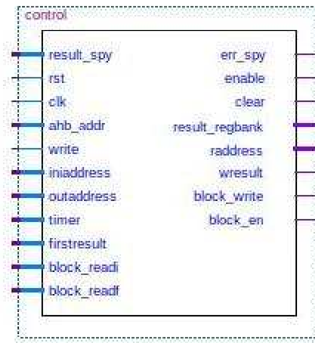

Figura 27: Bloque de control

Señales de entrada:

- *Rst*. Reset asíncrono, activo por nivel bajo.
- *Clk*. Señal de reloj.
- *Result_spy*. Firma que se recibe del bloque espía al completar una rutina.
- *Ahb_addr*. Bus de direcciones del bus AHB.
- *Write*. Indica que se produce una transferencia del maestro al esclavo seleccionados.
- *Iniaddress*. Direcciones de inicio de todas las rutinas.
- *Outaddress*. Dirección de fin de la rutina actual.
- *Timer*. Número máximo de ciclos de reloj permitidos para la rutina actual.
- *Firstresult*. Resultado de la primera ejecución de la rutina actual.
- *Block_readi*. Estado de la protección contra lectura de las direcciones de inicio.
- *Block_readf*. Estado de la protección contra lectura de las direcciones de fin.

Señales de salida:

- *Err_spy*. Indica que se ha detectado un error de ejecución.
- *Enable*. Habilitación del módulo espía.
- *Clear*. Señal de borrado síncrono de la firma.
- *Result_regbank*. Firma que se almacena en el banco de registros al finalizar la primera ejecución de una rutina.
- *Raddress*. Dirección de los registros de resultado del banco de registros, y dirección para lectura de los registros de configuración. Se corresponde con el índice de la rutina en ejecución.
- *Wresult*. Habilitar escritura de resultado en el banco de registros.

- *Block_write*. Modificación de la protección contra escritura.
- *Block_en*. Habilitar modificación de la protección contra escritura.

El bloque de control es el encargado de dirigir el funcionamiento del módulo. Diseñado como una máquina de estados, implementa el diagrama de estados que se muestra en la Figura 28, y que consta de los siguientes estados:

Figura 28: Diagrama de estados del módulo IP

- *Espera*. Estado de inactividad del módulo donde únicamente está operativa la interfaz de comunicaciones. En este estado se compara la dirección del bus AHB con todas las direcciones de inicio almacenadas en el banco de registros que no tengan protección contra lectura, es decir, que sus correspondientes señales *block_readi* y *block_readf* estén a ‘0’, para lo cual es necesario haber escrito tanto la dirección de inicio como la de fin de la rutina en cuestión. Si se produce una coincidencia y además se trata de una transferencia del maestro al esclavo (*write* toma el valor '1'), se almacena la posición que ocupa dicha rutina en el banco de registros en un registro llamado *pos*, se activa la protección contra escritura para esa rutina y se pasa al estado *rutina*. El valor del registro *pos*, que almacena el índice de la rutina en curso, se asigna a la salida *raddress* para el direccionamiento del banco de registros. Por otra parte se asigna a un registro interno denominado *watchdog* el valor 0, que hace las veces de temporizador y cuyo funcionamiento se detalla en el siguiente estado.

- *Rutina*. Ejecución de una rutina. En este estado se envía la señal *enable* al bloque espía para habilitar la captura de datos del bus de comunicaciones, y al mismo tiempo se activa el temporizador *watchdog*, cuyo valor aumenta en uno cada ciclo de reloj. Además se compara constantemente el valor del temporizador con la señal *timer*, que indica el número máximo de ciclos de reloj en los que debe completarse la rutina que se está ejecutando. En el momento en que el valor del *watchdog* sea mayor o igual que el del *timer* se retorna al estado de *espera*, se activa la señal *err_spy* indicando un error de ejecución y se envía la señal *clear* al bloque espía para borrar la firma

generada. Por contra, si antes de que desborde el *watchdog* se escribe un dato en el bus en la dirección correspondiente a la dirección de fin de la rutina actual y además la transferencia se realiza del maestro al esclavo deseados (*write* toma el valor '1'), se pasa al estado *fin de rutina*.

- *Fin de rutina*. Estado correspondiente al último ciclo de reloj de la ejecución de una rutina, necesario para capturar el dato correspondiente a la dirección de fin, ya que debido a las especificaciones del bus AHB el dato aparece en el bus un ciclo de reloj después de que lo haga la dirección correspondiente [1]. Por tanto en este estado también se habilita el espía. Desde este estado se pasa de forma inmediata a uno de los dos siguientes en función del valor de un registro interno, denominado *check*, que indica para cada una de las rutinas si ya se ha realizado la primera ejecución o no.

- *Iteración 1*. Fin de la primera ejecución de una rutina. En este estado se toma la firma del espía y se almacena en el banco de registros. Además se envía la señal *clear* para borrar el registro que contiene la firma y se pone a '1' el bit del registro *check* correspondiente a la rutina que se ha ejecutado, indicando que se ha efectuado la primera ejecución de la rutina. De este estado se pasa inmediatamente al estado de *espera*.

- *Iteración 2*. Fin de la segunda ejecución de una rutina. Aquí se toma el valor de la firma de la primera ejecución del banco de registros y el de la segunda ejecución del espía y se comparan entre sí, generando una señal de error en caso de haber diferencias. Además en este estado se desactiva la protección contra escritura de la rutina ejecutada, se borra el registro del espía mediante la señal *clear* y se invierte el bit del registro *check* correspondiente a la rutina que se ha ejecutado. Se pasa de forma inmediata al estado *espera*.

3.1.5 Espía

En la Figura 29 se pueden ver las señales de las que consta la interfaz del bloque espía, cada una de las cuales se comenta a continuación.

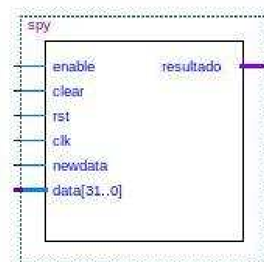

Figura 29: Bloque espía

Señales de entrada:

- *Rst*. Reset asíncrono, activo por nivel bajo.

- *Clk*. Señal de reloj.

- *Enable*. Habilitación del módulo espía.

- *Clear*. Señal de borrado síncrono de la firma.

- *Newdata*. Indica que hay un nuevo dato en el bus AHB.
- *Data*. Bus de datos del bus AHB.

Señales de salida:

- *Resultado*. Firma generada.

Este bloque es el encargado de generar la firma, un número pseudo-aleatorio obtenido a partir de una secuencia de valores semilla mediante el empleo de un registro LFSR (Linear Feedback Shift Register), de forma que ante ligeras variaciones de la secuencia de números de entrada el resultado es muy distinto. En el presente diseño esa secuencia de valores proviene del bus de datos del bus AHB.

El registro que contiene la firma cuenta con una señal de habilitación, *enable*, activada por el bloque de control mientras se está ejecutando una rutina. Para evitar que el registro capture varias veces el mismo dato, la interfaz envía la señal *newdata* cuando detecta que hay un nuevo dato en el bus correspondiente a una transferencia entre el maestro objetivo y el esclavo objetivo, de forma que para actualizar la firma es necesario que ambas señales estén activas.

Al finalizar la rutina en curso el bloque de control toma el valor de la firma y envía una señal de borrado síncrono, *clear*, para inicializar la firma para una nueva ejecución.

3.2 Organización del diseño

Cada uno de los bloques que constituyen el módulo de detección de fallos (interfaz, banco de registros, control y espía) se ha descrito en lenguaje VHDL en un diseño independiente. Los nombres de estos diseños son, respectivamente, *interfaz.vhd*, *regbank.vhd*, *control.vhd* y *spy.vhd*. Todos estos bloques se engloban en una entidad de nivel superior denominada *spyslv*, que representa al propio módulo de detección, y definida en el diseño *spyslv.vhd*.

Aparte, se ha escrito un archivo adicional, denominado *genericos.vhd*. Este diseño se trata de un paquete, un archivo de definiciones que incluye la declaración de la entidad de más alto nivel del diseño, así como declaraciones de tipos de variables empleadas en el código del diseño.

Los códigos de todos estos diseños se incluyen en el ANEXO B al final de esta memoria.

3.3 Inserción del módulo de detección en el sistema

Tras escribir los diseños que definen la funcionalidad del módulo de detección de fallos, se ha seguido una serie de pasos para insertar dicho módulo en la librería GRLIB, con el fin de poder añadir de forma sencilla e intuitiva el módulo de detección a un sistema basado en el microprocesador LEON3.

En primer lugar ha sido necesario añadir en los diseños del módulo de detección de fallos algunos parámetros genéricos adicionales (ver apartado 3.1.1) que le proporcionan un identificador único dentro de la biblioteca de componentes GRLIB. Dichos genéricos son:

- *Venid*. Identificador del distribuidor, es decir, de la organización que ha desarrollado el módulo.

- *Devid*. Identificador del módulo dentro de los diferentes diseños del distribuidor.
- *Version*. Versión del módulo.

Además, se han definido dentro del archivo que describe la interfaz los registros de configuración necesarios para soportar la capacidad *plug & play* de GRLIB (véase 2.2.3), a partir de los valores de los genéricos del módulo de detección (ver anexo B.3).

El siguiente paso ha consistido en introducir los diseños que constituyen el módulo de detección de fallos en la librería GRLIB. Para ello, se ha creado en el directorio *lib* de GRLIB, que contiene las librerías en VHDL de todos los distribuidores, una carpeta para almacenar los módulos diseñados en relación a este proyecto, denominada *sp*. El nombre de esta carpeta se ha registrado en un fichero de texto, de nombre *libs.txt*, presente en ese mismo directorio. De este modo se consigue que dicha carpeta sea visible para todos los diseños de GRLIB. A continuación se ha creado dentro del directorio *sp* otra carpeta, *spy*, donde almacenar todos los diseños que conforman el módulo de detección. De forma análoga al paso anterior se ha creado dentro del directorio *sp* un fichero *dirs.txt* que contiene el nombre de la carpeta *spy*. Por último, dentro del directorio *spy*, se han creado otros dos archivos de texto, *vhdlsim.txt* y *vhdlsyn.txt*, en los que se incluyen, respectivamente, los archivos que se utilizan en la simulación y en la síntesis.

A continuación se ha integrado el módulo de detección en la herramienta de configuración Xconfig. Para ello se han incluido cuatro archivos nuevos dentro de la carpeta *spy*, que se enumeran a continuación. En el ANEXO B se muestra el código fuente de todos estos archivos.

- *Spy.in*. Descripción del menú de configuración. Define las distintas variables que registran las opciones de configuración y les asigna un valor por defecto.
- *Spy.in.h*. Archivo de definición para las variables de configuración. Permite entre otras cosas traducir variables de configuración en cadenas de caracteres, así como asignar valores a las variables de configuración en caso de que alguna de dichas variables no esté definida con una configuración dada.
- *Spy.in.vhd*. Define una serie de constantes que toman sus valores de las variables de configuración. Estas constantes se agregan al archivo *config.vhd* cuando se ejecuta la herramienta de configuración.
- *Spy.in.help*. Contiene los mensajes de ayuda para cada una de las opciones de configuración de la aplicación Xconfig.

La configuración del módulo de detección definida en estos cuatro archivos debe hacerse visible en el diseño que se vaya a utilizar. Para ello se ha añadido la siguiente línea en el archivo *config.in*, ubicado en el directorio del propio diseño:

```
source lib/sp/spy/spy.in
```

Para finalizar la integración del módulo en el menú de configuración se ha introducido el comando "make scripts" por terminal desde el directorio correspondiente a uno de los diseños de la librería GRLIB. De este modo se compilan los archivos mencionados anteriormente, con lo que la configuración del módulo de detección de fallos aparece en el menú de la herramienta Xconfig, como muestran las Figuras 30 y 31.

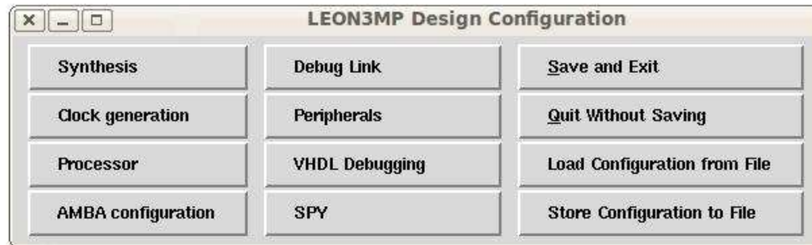

Figura 30: Módulo de detección integrado en Xconfig

Figura 31: Menú de configuración del módulo de detección

Como puede observarse en la Figura 31, el maestro y el esclavo objetivos se seleccionan por su nombre. Esta elección se traduce en una cadena de caracteres por medio del archivo *spy.in.h*. Sin embargo, es necesario transformar estos datos en valores numéricos que el módulo de detección sea capaz de manejar. Para ello se han incluido unas tablas de referencia en el paquete del módulo (*genericos.vhd*) que permiten traducir las cadenas de caracteres generadas en la configuración a los índices correspondientes a las posiciones del bus AHB que ocupan los módulos seleccionados. Sin embargo, las posiciones de los diversos maestros y esclavos del bus pueden variar de un diseño a otro, por lo que es recomendable revisar dichas posiciones al cambiar de diseño, y actualizar las tablas en consecuencia. Las tablas que se muestran dentro del código de *genericos.vhd* en el anexo de este documento se corresponden con el diseño leon3-gr-xc3s-1500. Este paquete debe estar declarado en el archivo *config.vhd* del diseño en el que se vaya a utilizar el módulo de detección, para lo cual se han añadido las siguientes líneas al archivo *config.vhd.in*, ubicado en el mismo directorio del diseño:

```
library sp
use sp.genericos.all
```

De este modo, al ejecutar la herramienta de configuración Xconfig estas líneas se añaden al archivo *config.vhd* de forma automática.

Después se deben registrar los identificadores de la nueva librería y de los módulos IP que ésta contiene. Esto se ha conseguido añadiendo al archivo *devices.vhd*, ubicado en el directorio *lib/grlib/amba*, las siguientes líneas de código:

```
-- Vendor codes
constant VENDOR_SP: amba_vendor_type := 16#03#;

-- Spy id's
constant SP_SPY : amba_device_type := 16#001#;
constant sp_device_table : device_table_type := (
  SP_SPY => "Spy                            ",
  others   => "Unknown Device                 ");
constant SP_DESC : vendor_description :=   "Spy                     ";
constant sp_lib : vendor_library_type :=(
  vendorid           => VENDOR_SP,
  vendordesc         => SP_DESC,
  device_table       => sp_device_table
);
```

Es necesario señalar que el identificador de la nueva librería no debe coincidir con el de cualquier otra librería presente en GRLIB.

El último paso es instanciar el módulo en la entidad de más alto nivel del diseño. Para ello se ha declarado el paquete *genéricos.vhd* en el archivo *leon3mp.vhd* correspondiente al diseño que se va a utilizar. Además, en dicho archivo se han añadido las siguientes líneas:

```
----------------------------------------------------------------------
---  SPY -------------------------------------------------------------
----------------------------------------------------------------------

SPYgen0: if CFG_SPY_EN = 1 generate
SPY0    : spyslv generic map( hindex => 4, spyaddr => CFG_SPY_ADDR, nummaster =>
       CFG_SPY_NUMMASTER, numslave => CFG_SPY_NUMSLAVE, addrsize => CFG_SPY_ADDRSIZE,
       hirq_w => CFG_SPY_HIRQ_W, hirq_spy => CFG_SPY_HIRQ_SPY)
       port map (rstn,clkm,ahbsi,ahbso(4));
end generate;
```

Como puede comprobarse, se ha asignado la posición número 4 del bus AHB al módulo de detección de fallos.

En el presente proyecto se han empleado los diseños leon3-gr-xc3s-1500 y leon3-altera-ep3sl150. Las modificaciones indicadas anteriormente se han realizado para estos dos diseños.

4 RESULTADOS

Una vez concluido el diseño del módulo de detección de fallos se ha procedido a realizar un conjunto de pruebas sobre el mismo, que pueden clasificarse en tres grupos:

- Validación de funcionalidad, que consiste en la simulación del módulo de detección de fallos, tanto por separado como conectado a un sistema embebido, para verificar su funcionalidad. Se corresponde con la fase de depuración del diseño.

- Síntesis del diseño, con el objetivo de determinar los requisitos de espacio y frecuencia de funcionamiento del módulo de detección en relación al microprocesador LEON3.

- Test de errores, donde se comprueba la capacidad de detección del módulo de detección de fallos mediante la inserción de fallos en la simulación del propio módulo de detección conectado al microprocesador LEON3.

A continuación se explica en profundidad cada una de estas pruebas y se muestran los resultados obtenidos.

4.1 Validación de la funcionalidad del diseño

La finalidad del este conjunto de pruebas es comprobar que la funcionalidad del diseño se corresponde con lo esperado, y en caso contrario se procede a modificar el diseño. Esto equivale a la fase de depuración del diseño. Para realizar la simulación de los diseños utilizados en esta fase se ha empleado el simulador Modelsim.

La depuración del diseño se ha dividido en tres fases. Por un lado se ha probado la funcionalidad del módulo de detección sin la interfaz, es decir, completamente aislado. Por otro se ha verificado el funcionamiento de la interfaz, por separado, conectada al bus AHB. Por último, se ha realizado una comprobación del módulo de detección al completo conectado al bus AHB. En este apartado se muestran únicamente los resultados de la última versión del módulo de detección, plenamente funcional, como prueba de que el módulo opera según la descripción dada en el apartado 3.1.

4.1.1 Simulación del módulo de detección sin interfaz

En esta primera fase de la depuración se ha comprobado la funcionalidad del diseño sin conectar al bus AHB. Esto se ha realizado antes de insertar el módulo de detección en la librería GRLIB.

Para realizar esta simulación ha sido necesaria cierta preparación previa. En primer lugar se ha modificado la entidad de más alto nivel del diseño, que originalmente englobaba la interfaz, el banco de registros, el bloque de control y el bloque espía, para que no incluya la interfaz (ver apartado 3.1). Las señales que comunicaban la interfaz con el resto de bloques pasan a ser los puertos de entrada y salida de la entidad de más alto nivel. Además se establece el número de rutinas soportadas en 8, lo que equivale a asignar al genérico *addrsize* el valor 3 (ver apartado 3.1.1). Por otra parte se ha escrito un diseño en VHDL que contiene un banco de pruebas, es decir, una evolución temporal del conjunto de señales de entrada del diseño con las que se simula el propio diseño. Se han realizado tres simulaciones del diseño con tres versiones distintas del banco de pruebas: una con el funcionamiento normal (Figuras 32 a 38), otra para

probar errores de ejecución (Figuras 39 a 41) y otra para probar las protecciones contra lectura y escritura del banco de registros (Figuras 42 y 43).

Las tres versiones del banco de pruebas comienzan con un reset del módulo de detección y la posterior configuración de las rutinas. El reset, activo por nivel bajo, hace que se borren todos los datos almacenados en el banco de registros (*reginicio*, *regfin*, *regtime* y *regresult*) y que se active la protección contra lectura de todas las rutinas (*block_ri* y *block_rf*), como se muestra en la Figura 32. Además, desactiva la protección contra escritura de todas las rutinas (*block_w*), establece el estado del bloque de control en *espera*, y asigna el valor 0 al resto de registros del módulo (*check*, *watchdog*, *pos* y *result*), como puede verse en la Figura 33.

El módulo de detección se configura para observar dos rutinas diferentes, que se asignan a las posiciones 3 y 7 del banco de registros. La primera rutina comienza en la dirección hexadecimal 00FF0020 y termina en 00FF0034, mientras que las direcciones de inicio y de fin para la segunda rutina son, respectivamente, 00FF0044 y 00FF0050. Por otro lado se establece un límite en el tiempo de ejecución de 50 ciclos para la primera rutina y 20 para la segunda. En la Figura 32 se muestra cómo se registran las direcciones de inicio y de fin (*reginicio* y *regfin*), mientras que la escritura del registro *regtime* se puede ver en la Figura 33. Cuando la señal *waddress* se activa, uno de los registros del banco de registros, en función del valor de *address*, se actualiza con el valor presente en el bus de datos (*data*). Puede comprobarse además, en la Figura 32, cómo cada vez que escribe una dirección en el banco de registros se desactiva la protección contra lectura asociada a ese registro.

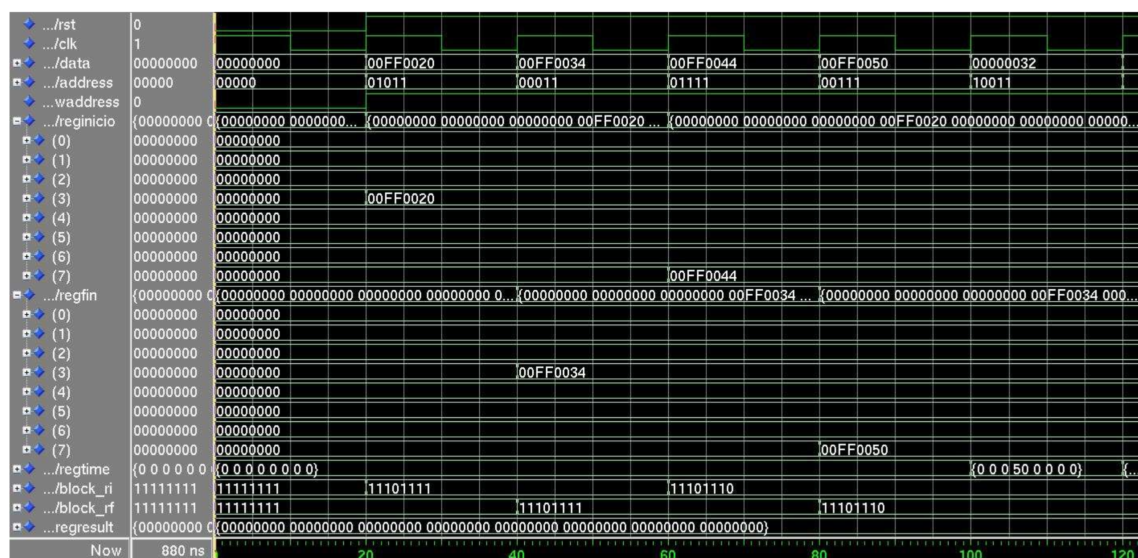

Figura 32: Simulación del módulo aislado - reset y configuración

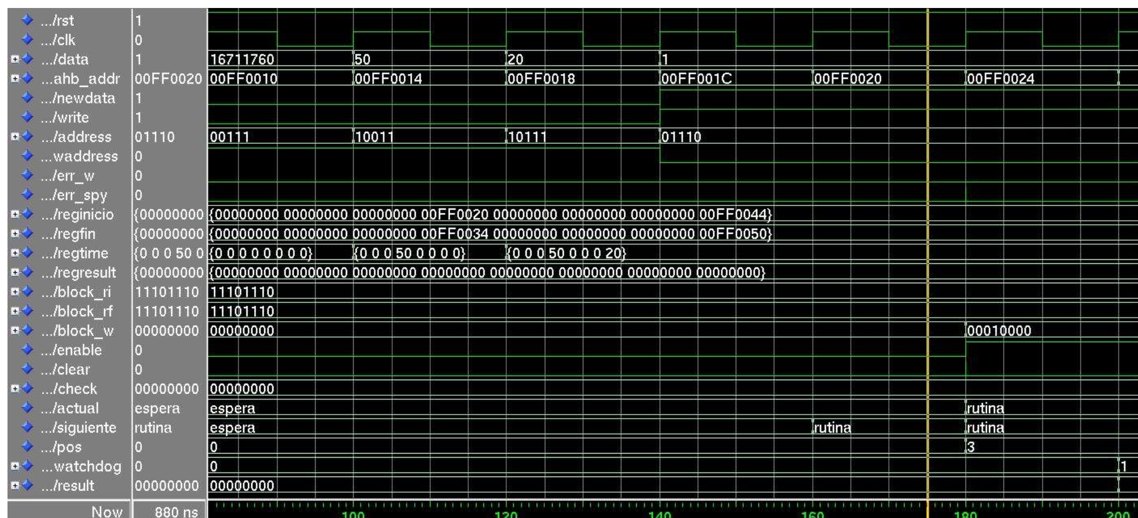

Figura 33: Simulación del módulo aislado - configuración e inicio de la 1ª rutina

En la primera versión del banco de pruebas se ejecutan dos veces cada una de las dos rutinas, de forma que ambas ejecuciones son idénticas. El inicio de la primera de las rutinas se muestra en la Figura 33. Cuando la dirección del bus (*ahb_addr*) coincide con alguna de las almacenadas en *reginicio* y además la señal *write* está activa, se pasa al estado *rutina* y se almacena en el registro *pos* el índice correspondiente a la rutina que acaba de comenzar, 3 en este caso. Además, se activa la protección contra escritura de la rutina en curso. Una vez ha comenzado una rutina, se pone en marcha el temporizador *watchdog* y se habilita el espía, de modo que cada vez que se activa la señal *newdata* se actualiza la firma en el registro *result* (ver Figura 34). Cuando en el bus de direcciones aparece la dirección de fin de la rutina en curso y además la señal *write* está activa, se procede a finalizar la rutina. En primer lugar se pasa al estado *fin_rutina* en el que se actualiza por última vez la firma con el dato presente en el bus. A continuación, como se trata de la primera ejecución de esta rutina (el bit número 3 del registro *check* está a '0') se pasa al estado *iteración_1*. En dicho estado se toma la firma y se almacena en la posición 3 del banco de registros, se borra el registro del resultado, y además se invierte el bit número 3 del registro *check* (ver Figura 34). Por último se vuelve al estado de *espera*, donde se inicializa el *watchdog*.

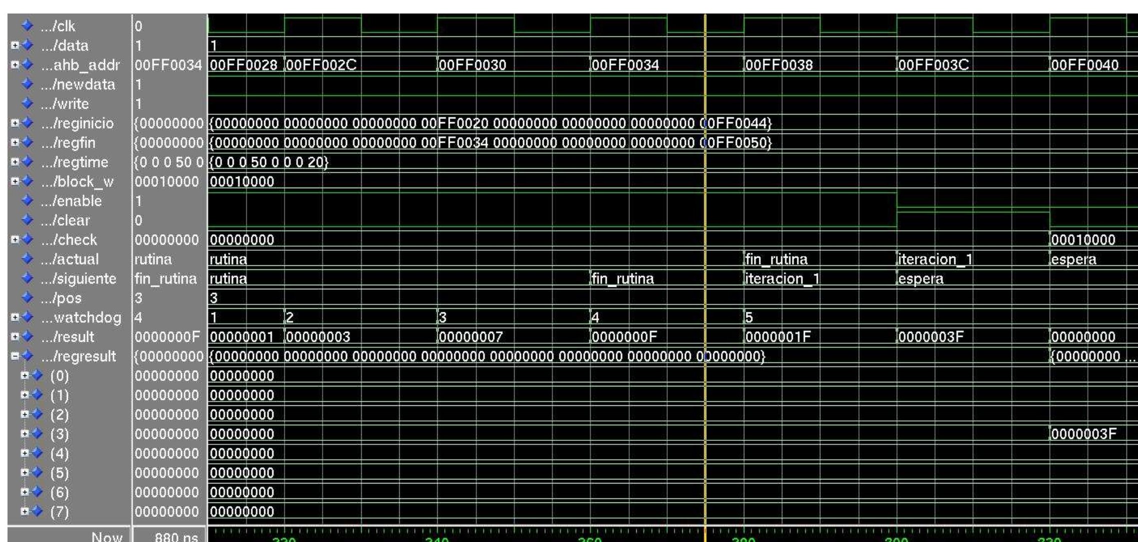

Figura 34: Simulación del módulo aislado - fin de la 1ª rutina

De forma análoga se produce la primera ejecución de la segunda rutina, que ocupa la posición 7 del banco de registros (ver Figuras 35 y 36). Como puede comprobarse no es necesario que las dos ejecuciones de una misma rutina tengan lugar de forma consecutiva.

Figura 35: Simulación del módulo aislado - inicio de la 2ª rutina

Figura 36: Simulación del módulo aislado - fin de la 2ª rutina

A continuación se ejecuta de nuevo la primera rutina (ver Figura 37). En este caso, tras el estado *fin_rutina* se pasa al estado *iteración_2*, donde se compara el resultado de la ejecución con el almacenado en el banco de registros. Como ambos resultados coinciden no se activa el error de ejecución (*err_spy*). Además, en este estado se invierte el bit del registro check, se desactiva la protección contra escritura correspondientes a la rutina ejecutada (3 en este caso) y se borra el registro que contiene la firma para una ejecución posterior.

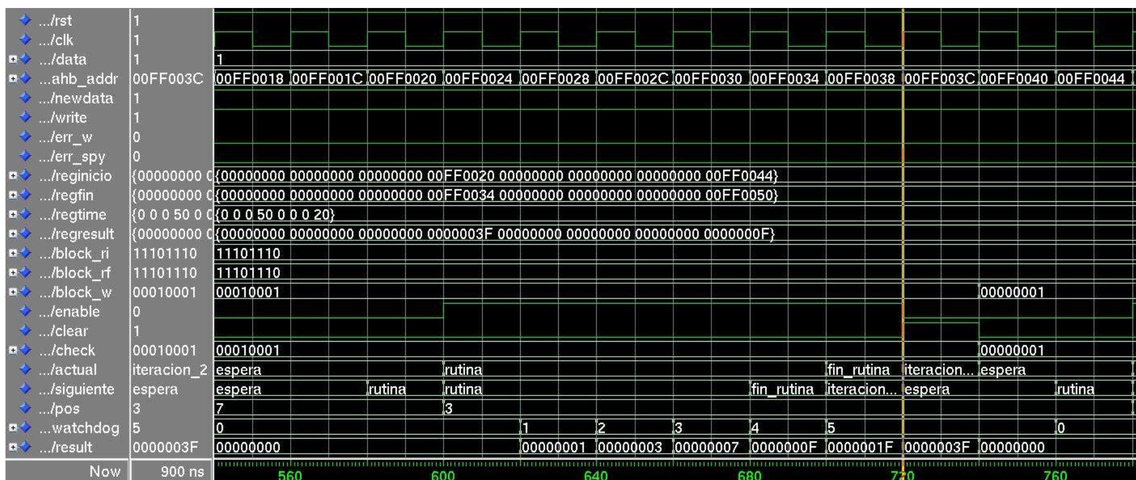

Figura 37: Simulación del módulo aislado - repetición correcta de la 1ª rutina

Por último, tiene lugar la segunda ejecución de la segunda rutina (ver Figura 38), de la misma forma que para la primera rutina.

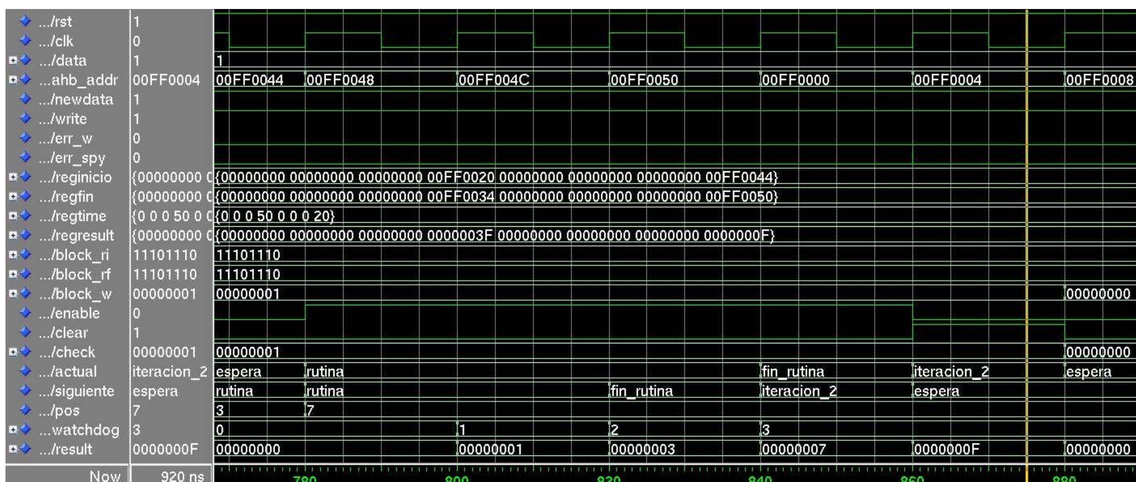

Figura 38: Simulación del módulo aislado - repetición correcta de la 2ª rutina

En la segunda versión del banco de pruebas se comprueba que el módulo de detección es capaz de detectar los errores de ejecución que se producen. El reset, la configuración y la primera ejecución de cada una de las dos rutinas se realizan de forma idéntica a la primera versión (Figuras 32 a 36). Sin embargo, la segunda ejecución de ambas rutinas es diferente en esta simulación. En el caso de la primera rutina, el fallo se produce debido a que el resultado de las ejecuciones varía de una a otra. Como puede verse en la Figura 39, la secuencia de datos presente en el bus de datos (*data*) difiere respecto a la primera ejecución (Figura 34), por lo que la firma generada (*result*) es distinta al finalizar la rutina. Por tanto, se activa el error de ejecución (*err_spy*) al término de la rutina.

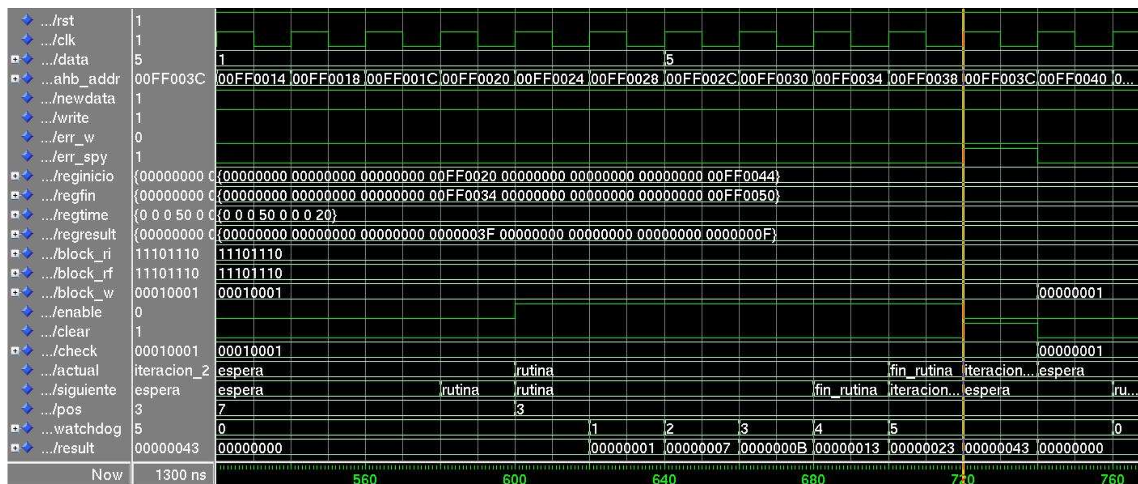

Figura 39: Simulación del módulo aislado - fallo por resultados distintos

La segunda rutina falla al exceder el número máximo de ciclos de reloj permitidos para su ejecución. La Figura 40 muestra cómo, al haberse iniciado la segunda ejecución de esta rutina, el módulo de detección no abandona el estado *rutina* cuando en el bus de direcciones (*ahb_addr*) aparece la dirección de fin de la rutina actual. Esto se debe a que la señal *write* está desactivada en ese momento, lo que se interpreta como una transferencia del esclavo al maestro, y por tanto se ignora. Es interesante comprobar, además, que mientras la señal *newdata* está desactivada, lo que se interpreta como que no hay un dato nuevo en el bus, la firma (*result*) no se actualiza. La ejecución de la rutina continúa en la Figura 41, hasta que el temporizador *watchdog* alcanza el valor almacenado en el banco de registros (*regtime*) correspondiente a la rutina en curso. En este momento se activa el error de ejecución (*err_spy*), se borra el registro que genera la firma (*result*) y se retorna al estado de *espera*. Además se desactiva la protección contra escritura y se pone a '0' el bit del registro *check* correspondientes a la rutina ejecutada.

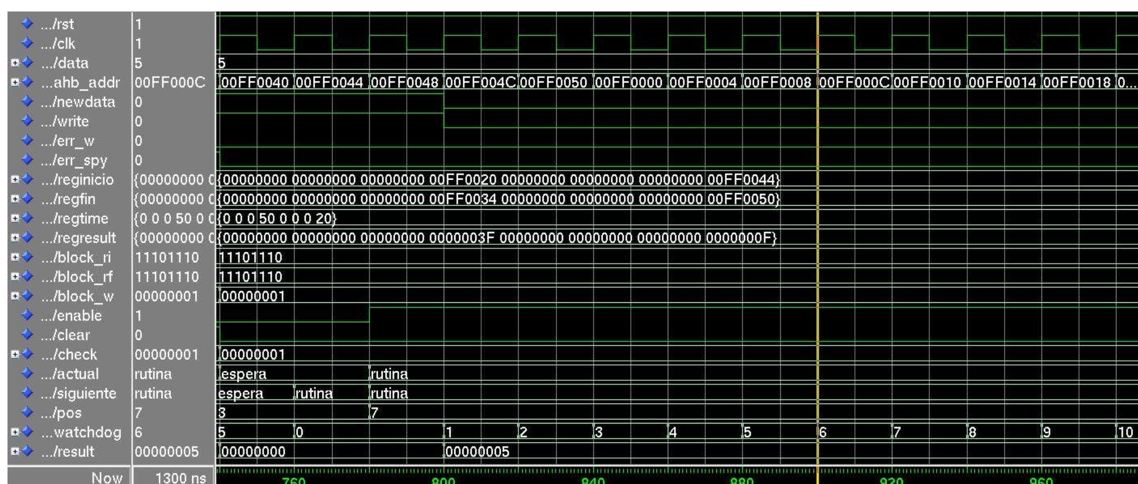

Figura 40: Simulación del módulo aislado - repetición errónea de la 2ª rutina

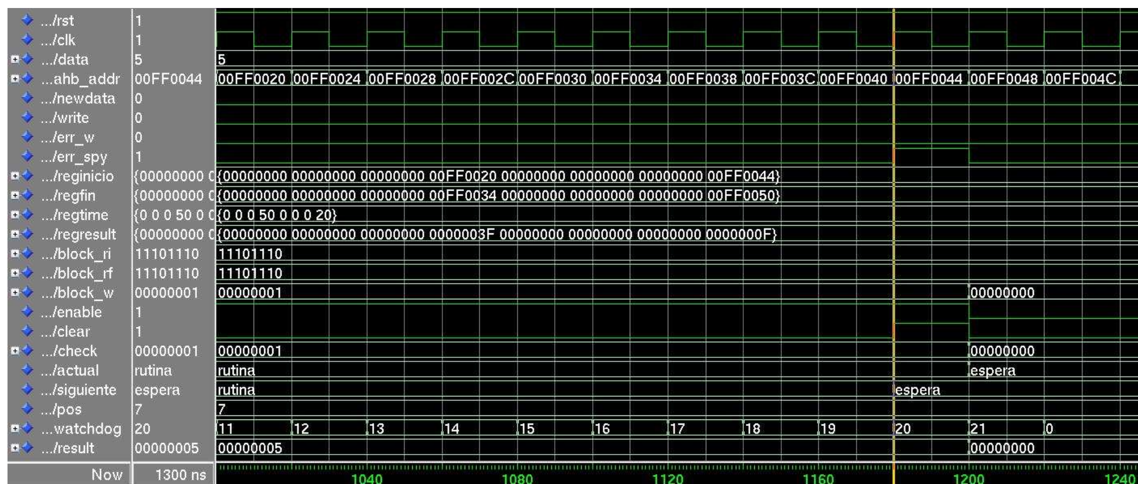

Figura 41: Simulación del módulo aislado - desbordamiento del *watchdog*

La tercera y última versión del banco de pruebas permite verificar el funcionamiento de las protecciones contra escritura y contra lectura. En la Figura 42 se observan varios intentos de escritura en el banco de registros una vez que el módulo de detección ha iniciado su labor de detección de errores. El primer intento, en la dirección de inicio en la posición número 6, tiene éxito ya que se trata de una región del banco de registros que no está protegida contra escritura (el bit número 6 del registro *block_w* está desactivado). El segundo intento, en la dirección de fin con el índice 0, no tiene efecto ya que en ese mismo momento se está modificando la protección contra escritura. En este caso no se activa la protección contra escritura (*err_w*) ya que se trata de una situación imposible en la práctica: el bus sólo permite comunicar a un maestro con un esclavo, por lo que si el microprocesador se comunica con el módulo de detección no puede comunicarse con el esclavo objetivo y viceversa. Los otros tres intentos de escritura se producen en las posiciones 3 y 7 del banco de registros, zonas protegidas contra escritura, por lo que se activa el error de escritura en los tres casos. La protección contra escritura impide que se modifique la configuración de una rutina desde que se inicia su primera ejecución hasta que finaliza la segunda o se detecta un error de ejecución.

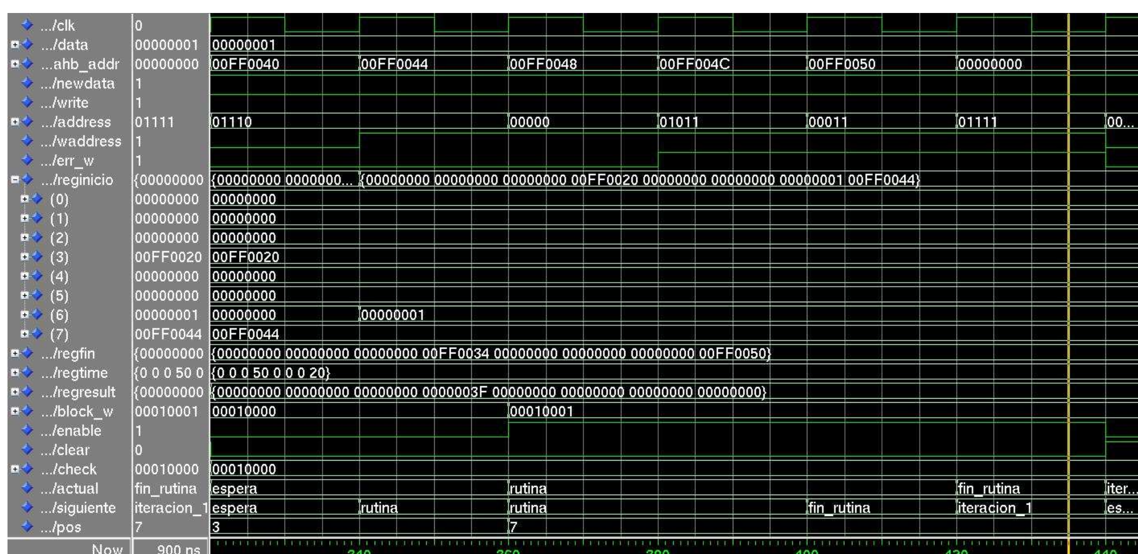

Figura 42: Simulación del módulo aislado - protección contra escritura

La protección contra lectura sirve para que tan sólo aquellas posiciones del banco de registros que han sido adecuadamente configuradas puedan emplearse por parte del bloque de control. De este modo, como puede verse en la Figura 43, no se inicia la ejecución de una rutina a pesar de que el estado de *espera* está activo, la señal *write* vale '1' y la dirección del

bus coincide con hasta 5 de las almacenadas en el banco de registros, porque todas esas posiciones están protegidas contra lectura.

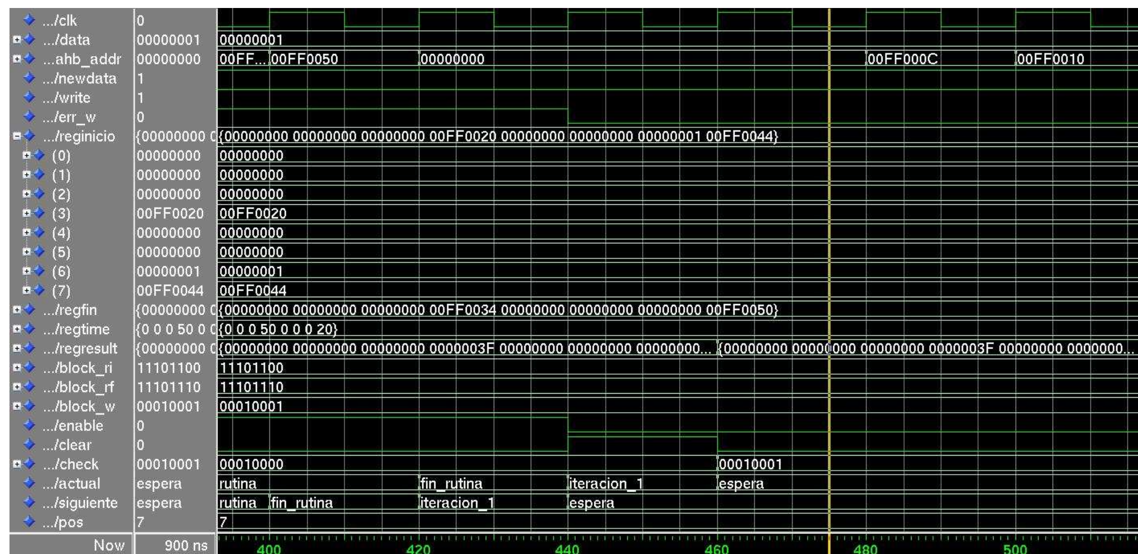

Figura 43: Simulación del módulo aislado - protección contra lectura

4.1.2 Simulación de la interfaz

El segundo paso en la depuración del módulo de detección ha consistido en la simulación de la interfaz conectada al bus AHB. Para ello ha sido necesario realizar algunas modificaciones en el diseño de la propia interfaz, e incluir la interfaz modificada como un módulo independiente en la biblioteca de componentes GRLIB.

Para simular la interfaz por separado, se han eliminado los puertos de entrada y salida de la misma que comunicaban con el resto de bloques del módulo de detección. Dichos puertos de entrada y salida pasan a ser señales internas de la propia interfaz. Esta nueva interfaz ha sido bautizada como *interfazb* para diferenciarla de la original.

Para insertar la nueva interfaz en GRLIB se ha seguido un procedimiento similar al descrito en 3.3. En primer lugar, se ha creado una nueva carpeta dentro del directorio *lib/sp* para almacenar los diseños asociados a la nueva interfaz. Dicha carpeta, denominada *interfaz*, se ha registrado en el archivo *dirs.txt* presente en ese mismo directorio. En la carpeta *interfaz* se ha incluido el diseño que define la interfaz, *interfazb.vhd*, y un paquete de declaraciones análogo al del módulo de detección, *genericosb.vhd*. En esa misma carpeta se han creado los archivos *vhdlsim.txt* y *vhdlsyn.txt*, con el mismo propósito que se ha descrito en 3.3. En este caso no se han añadido archivos para la herramienta de configuración Xconfig, en su lugar se ha empleado la configuración del módulo de detección de fallos. Por otra parte, se ha incluido en el diseño *devices.vhd* un identificador para la nueva interfaz como se indica en 3.3. Además, se ha cambiado la declaración del paquete del módulo de detección por el de la interfaz en los archivos *config.vhd.in* y *leon3mp.vhd* presentes en el diseño leon3-gr-xc3s-1500. Por último, en la entidad de más alto nivel del diseño, se ha modificado la instancia del módulo de detección por la de la interfaz, la cual se muestra a continuación:

```
----------------------------------------------------------------------
---  INTERFAZ --------------------------------------------------------
----------------------------------------------------------------------

INTERFAZgen0 : if CFG_SPY_EN = 1 generate
```

```
INTERFAZ0 : interfazb generic map (hindex => 4, spyaddr => CFG_SPY_ADDR, nummaster =>
         CFG_SPY_NUMMASTER, numslave => CFG_SPY_NUMSLAVE, addrsize =>
         CFG_SPY_ADDRSIZE, hirq_w => CFG_SPY_HIRQ_W, hirq_spy => CFG_SPY_HIRQ_SPY)
         port map (rstn,clkm,ahbsi,ahbso(4));
end generate;
```

En cuanto a la configuración de la interfaz, el número de rutinas soportadas se establece en 8 (*Addrsize* toma el valor 3), el maestro objetivo es el microprocesador LEON3, el esclavo objetivo es el controlador de memoria, y el rango de direcciones asignado a la interfaz es el comprendido entre los valores hexadecimales B0000000 y B00FFFFF.

El programa de prueba cargado en la memoria del sistema consiste en una serie de escrituras tanto en la interfaz como en la memoria. En las Figuras 44 y 45 se muestran varias operaciones de escritura en la interfaz. Como puede verse, tras cada una de estas operaciones se habilita la escritura en el banco de registros (*waddress*) en la dirección (*address*) indicada por los bits del 2 al 6 del bus de direcciones del bus AHB (*ahbsi.haddr*). Además se comprueba que, ante una escritura en una dirección válida del banco de registros, la respuesta de la interfaz es de tipo *okay* (señal *ahbso(4).hresp* con valor "00"). Por otra parte, en estas figuras se aprecia que la interfaz toma inmediatamente los valores de los buses de dirección y de datos (*ahbsi.haddr* y *ahbsi.hwdata*) para el resto de bloques del módulo de detección de fallos (*ahb_addr* y *data* respectivamente).

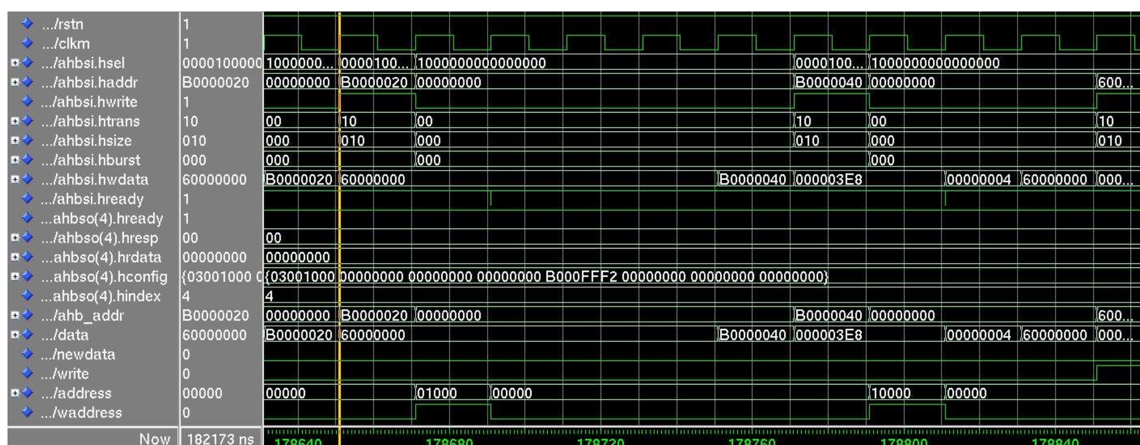

Figura 44: Simulación de la interfaz - comunicación con la interfaz I

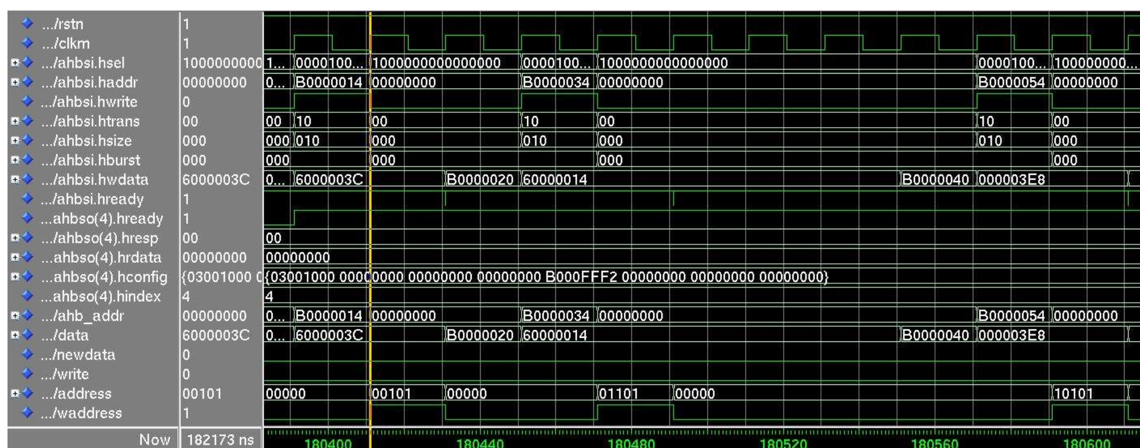

Figura 45: Simulación de la interfaz - comunicación con la interfaz II

Después se han realizado una serie de escrituras en la memoria, como muestran las Figuras 46 y 47. Puede apreciarse que, al producirse una transferencia del maestro objetivo al esclavo objetivo se activa inmediatamente la señal *write*, que indica al bloque de control una

transferencia del maestro al esclavo seleccionados, y como un ciclo más tarde se activa la señal *newdata*, que indica al bloque espía que hay un nuevo dato en el bus.

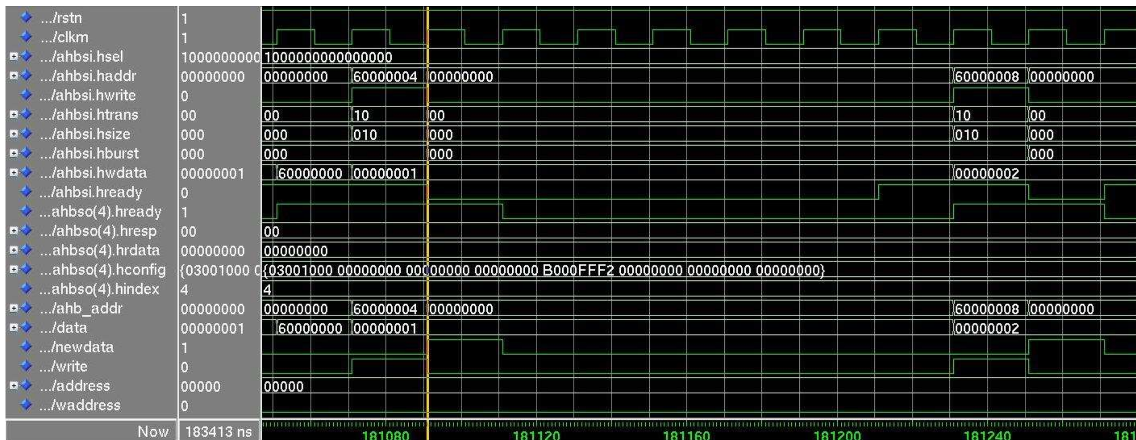

Figura 46: Simulación de la interfaz - comunicación con la memoria I

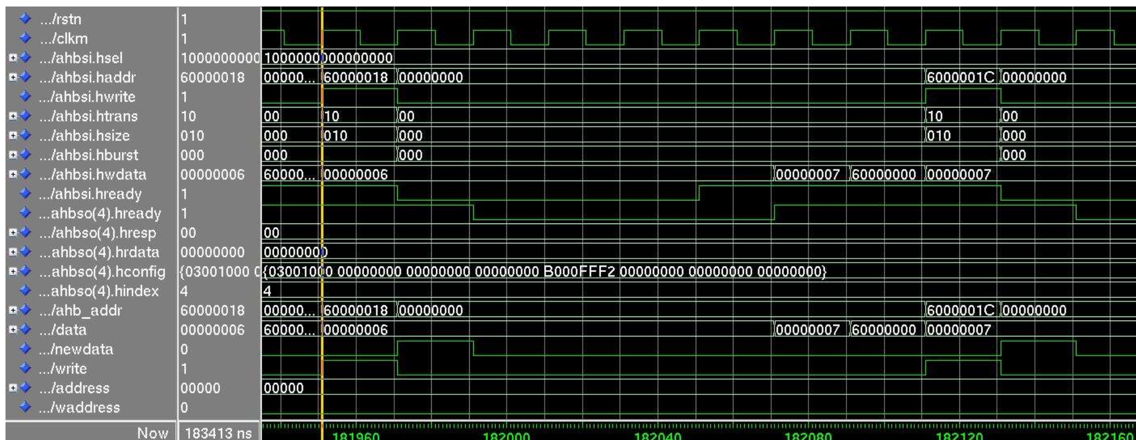

Figura 47: Simulación de la interfaz - comunicación con la memoria II

En último lugar, se ha realizado un intento de escritura en una zona no válida del espacio de direcciones de la interfaz. Como se muestra en la Figura 48, se genera una respuesta de tipo *error* (*ahbso(4).hresp* con valor "01") de dos ciclos de reloj. En el primer ciclo de la respuesta de error la señal *ahbso(4).hready* está desactivada, que se activa en el segundo ciclo de la respuesta.

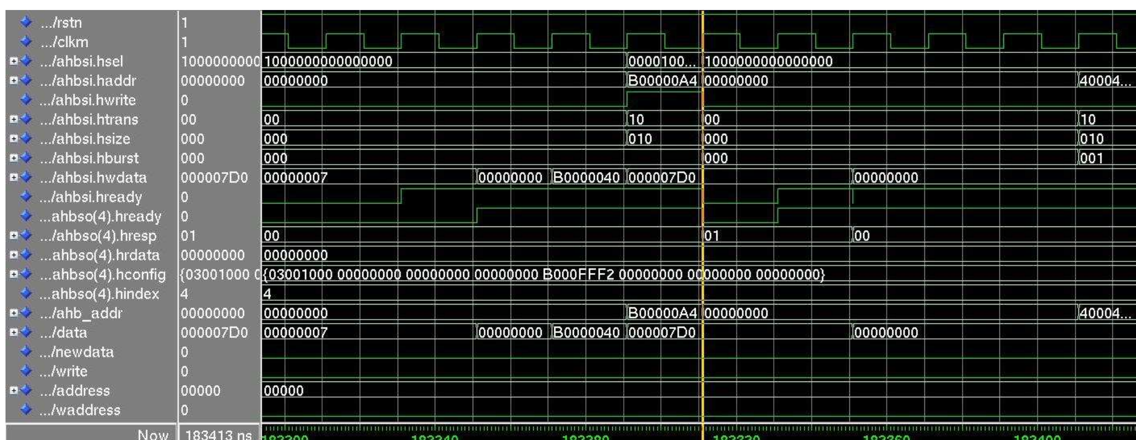

Figura 48: Simulación de la interfaz - escritura en dirección no válida

4.1.3 Simulación del módulo de detección al completo

En último lugar se ha realizado una simulación del módulo de detección completo conectado al bus AHB. Lo interesante de esta prueba es comprobar que la comunicación entre la interfaz y el resto de bloques del módulo de detección se realiza adecuadamente, y que las interrupciones por error de ejecución y de escritura se activan cuando corresponde.

La configuración del módulo de detección para esta fase es la siguiente:

- Capacidad para ocho rutinas independientes, es decir, se ha asignado valor 3 al genérico *Addrsize*.
- El maestro objetivo y el esclavo objetivo son, respectivamente, el microprocesador LEON3 y el controlador de memoria.
- Las interrupciones asignadas a los errores de ejecución y de escritura son la número 10 y la 9, respectivamente.
- El espacio de direcciones asignado es el comprendido entre los valores hexadecimales B0000000 y B00FFFFF.

El programa cargado en la memoria del microprocesador LEON3 realiza en primer lugar la configuración del módulo de detección para la supervisión de dos rutinas, ubicadas en las posiciones 2 y 5 del banco de registros. Las direcciones de inicio y de fin para la rutina número 2 son, respectivamente, 60000000 y 60000028, en valor hexadecimal. Por su parte, para la rutina en la posición 5 dichas direcciones son la 60000030 y la 60000050, respectivamente. El tiempo máximo de ejecución permitido es de 300 ciclos para la rutina número 2, y de 200 ciclos para la 5. En la Figura 49 puede comprobarse que la configuración de las rutinas es acorde con lo expuesto anteriormente. Además se muestra la escritura de las direcciones de inicio en el banco de registros (*reginicio*), a la vez que se desactiva la protección contra lectura correspondiente (*block_ri*).

Figura 49: Simulación del módulo de detección competo - configuración e inicio de rutina

Tras la configuración de las rutinas se realiza la primera ejecución de cada una de ellas. La Figura 49 muestra el inicio de la rutina en la posición número 2, con la activación del *watchdog* y de la protección contra escritura correspondiente (*block_w*).

A continuación se realiza una escritura en una dirección del banco de registros protegida contra escritura, en concreto en la dirección de inicio de la rutina número 2. Como se muestra en la Figura 50, esto provoca que, en vez de escribir en el banco de registros, se active el bit número 9 del vector de interrupciones (*ahbso(4).hirq*). Este bit se activa durante un único ciclo de reloj. Sin embargo, el controlador de interrupciones del sistema (*irqctrl0*) detecta la petición de interrupción, y activa el bit número 9 del registro *ipend*, indicando que hay una interrupción pendiente de resolver. Los bits del registro *ipend* permanecen activos hasta que son borrados por software tras atender la interrupción correspondiente, por lo que queda un registro de la petición. El programa de prueba empleado en la simulación no realiza el tratamiento de interrupciones, así que el bit de interrupción pendiente permanece activo hasta el final del programa.

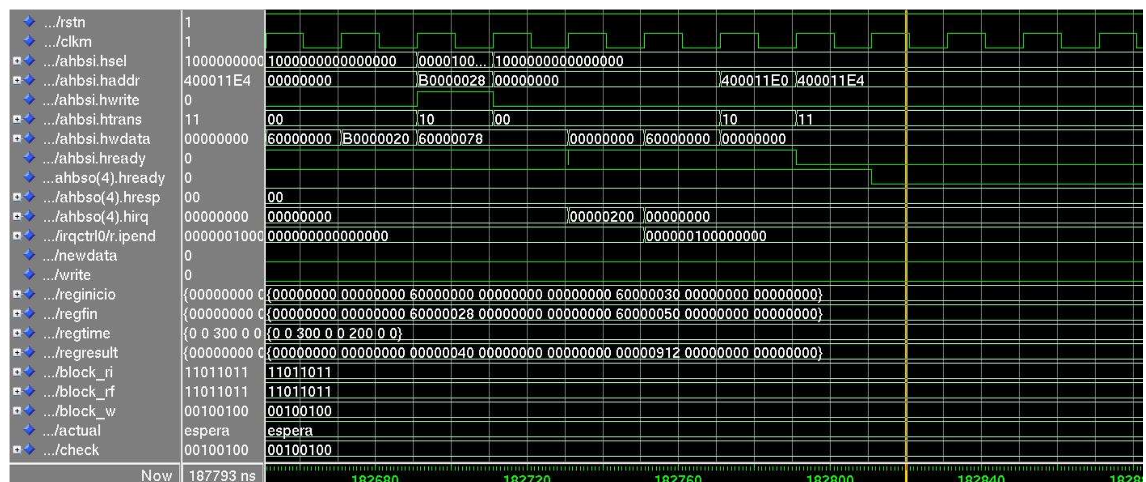

Figura 50: Simulación del módulo de detección completo - interrupción por error de escritura

En último lugar se repiten las ejecuciones de ambas rutinas. En la rutina número 2 no se detecta ningún error, como demuestra la Figura 51, ya que el resultado de la primera ejecución, almacenado en el banco de registros (*regresult*), coincide con el de la segunda ejecución (*result*). Por tanto no se activa interrupción alguna.

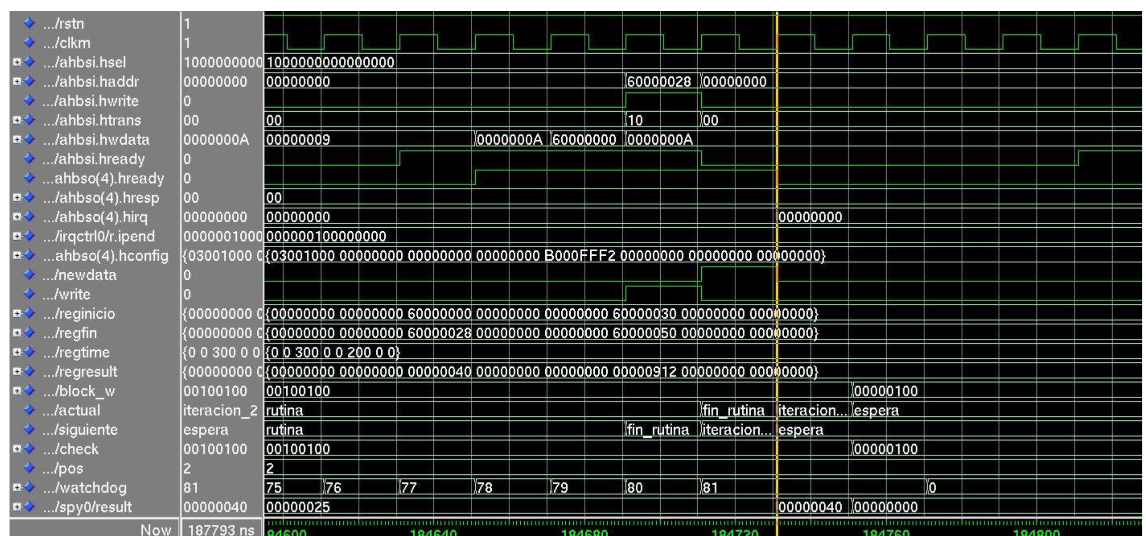

Figura 51: Simulación del módulo de detección completo - ejecución de rutina correcta

En el caso de la rutina número 5 las dos ejecuciones han sido distintas entre sí, como puede verse en la Figura 52 comparando las señales *regresult* y *result*. Esto ocasiona la activación del bit número 10 del vector de interrupciones del módulo de detección (*ahbso(4).hirq*) durante

un ciclo de reloj. Y al igual que ocurría con la interrupción por error de escritura, se activa el bit 10 del registro *ipend* del controlador de interrupciones hasta que se trate la interrupción correspondiente.

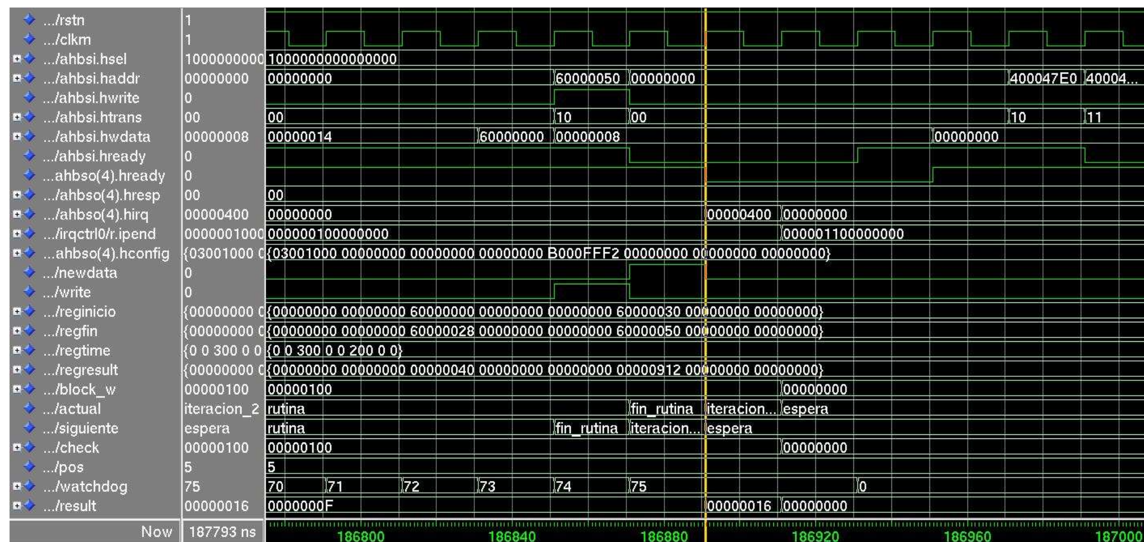

Figura 52: Simulación del módulo de detección completo - interrupción por error de ejecución

4.2 Resultados de síntesis

La síntesis del diseño permite conocer el área ocupada por el mismo y su frecuencia máxima de funcionamiento al introducirlo en un dispositivo lógico programable. En el caso del presente proyecto lo interesante es comprobar los requisitos de funcionamiento del módulo de detección en relación a los del LEON3, por lo que a tal efecto se han realizado tres síntesis distintas: una con el microprocesador LEON3 por separado, y otras dos con el módulo de detección conectado al LEON3. En todos estos casos se ha empleado la misma configuración del LEON3.

El software empleado para la síntesis del diseño es Quartus II de Altera. Esto se debe a que se ha escogido como vehículo de pruebas para la síntesis la familia de dispositivos de la marca Altera, lo que ha obligado a escoger una configuración del LEON3 compatible con esa familia de dispositivos. En concreto se ha elegido para la síntesis el diseño leon3-altera-ep3sl150. La síntesis se ha realizado para una placa de la familia Stratix III, quedando en manos del sintetizador la elección del modelo concreto de FPGA. De entre las opciones de síntesis se ha elegido un modo equilibrado, el cual busca un equilibrio entre el área ocupada por el diseño y la rapidez del mismo.

Se ha seleccionado una configuración básica del sistema embebido basado en el microprocesador LEON3, con los siguientes elementos:

- Reloj de frecuencia base de 50 MHz, con un factor de multiplicación del reloj de x30, y un factor de división de x10.
- Número de procesadores LEON3: 1.
- Sin unidad de punto flotante, en su lugar la *Integer Unit* tiene habilitadas instrucciones de multiplicación y división.
- Caché de datos, de dos sets de 4 KB cada uno, y de programa, de dos sets de 8 KB cada uno, separadas. Algoritmo de reemplazo LRU.

- Controlador de memoria del LEON2 con soporte para datos de 16 bits.
- Controlador de memoria DDR2 para SDRAM de 16 bits.
- UART, GPIO.
- Dos temporizadores con interrupciones separadas.
- Controlador de interrupciones del LEON3 con interrupciones secundarias habilitadas.

Por otra parte, se han probado dos configuraciones diferentes del módulo de detección, una con capacidad para cuatro rutinas independientes y otra para dos. Es decir, se han asignado los valores 2 y 1 respectivamente al genérico *Addrsize*.

Los resultados de síntesis se componen de ocupación y de análisis de tiempos. En lo que se refiere a ocupación los datos relevantes son:

- Número de ALUT's combinacionales. Una ALUT es un elemento propio de las FPGA's de la familia Altera que consiste en la agrupación de dos LUT's, tablas de búsqueda o *Look-Up Tables*, con el objetivo de aprovechar señales comunes.
- Número de registros.
- Número de bloques DSP, módulos aritméticos.
- Número de bloques M9K, memorias de 9 KB.

La información relativa a la ocupación se muestra en la Tabla 6. Además de indicar la cantidad empleada de cada uno de los componentes mencionados para cada una de las configuraciones descritas, se incluye el incremento porcentual que supone añadir el módulo de detección en relación al sistema basado en el microprocesador LEON3.

Diseño	ALUT's	Registros	Bloques DSP	Bloques M9K
LEON3 sin módulo de detección	5514	3434	4	52
LEON3 con módulo de detección con soporte para 4 rutinas	5952 (+7,9 %)	4050 (+17,9 %)	4 (+0 %)	52 (+0 %)
LEON3 con módulo de detección con soporte para 2 rutinas	5829 (+5,7 %)	3770 (+9,8%)	4 (+0 %)	52 (+0 %)

Tabla 6: Ocupación de la FPGA

Respecto al análisis de tiempos, interesa conocer la frecuencia máxima de funcionamiento. El analizador de tiempos del Quartus II proporciona la frecuencia máxima del camino más lento posible. En la Tabla 7 se muestran los resultados del análisis de tiempos.

Diseño	Frecuencia máx.
LEON3 sin módulo de detección	169,89 MHz
LEON3 con módulo de detección con soporte para 4 rutinas	149,77 MHz
LEON3 con módulo de detección con soporte para 2 rutinas	159,92 MHz

Tabla 7: Frecuencias máximas de funcionamiento

Nótese que el banco de registros del módulo de detección de fallos se ha implementado con biestables. Para un número de rutinas elevado podría implementarse en su lugar con memorias, lo que resultaría en un menor impacto en el número de ALUT's y registros, así como en las prestaciones.

4.3 Test de errores

El test de errores es el procedimiento por el que se determina la capacidad de respuesta ante fallos de un diseño mediante la inyección sistemática de fallos en el mismo. En el caso de circuitos muy simples el test puede abarcar todas las posibles combinaciones de entradas y estados. Pero en los circuitos complejos, con cientos de entradas y un gran número de posibles evoluciones en el tiempo, como son los circuitos de aplicación práctica hoy día, no se pueden abarcar todas las posibilidades, por lo que es necesario escoger un número significativo de pruebas.

En el presente proyecto el objetivo del test es conocer la capacidad de detección de errores del módulo de detección diseñado. Para ello se ha simulado mediante Modelsim el módulo de detección conectado al LEON3 y se han insertado fallos al microprocesador mientras éste ejecutaba un programa preestablecido. A continuación se describen las decisiones efectuadas y el procedimiento empleado en el test de errores. Por último, se muestran los resultados obtenidos del ensayo.

4.3.1 Configuración del test

Para la simulación del microprocesador LEON3 se ha escogido el diseño leon3-gr-xc3s-1500 con la configuración que se detalla a continuación. El diseño se ha escogido por ser una implementación tipo recomendada en la documentación de la librería GRLIB [4].

- Reloj de frecuencia base de 50 MHz.
- Número de procesadores LEON3: 1.
- Sin unidad de punto flotante, en su lugar la *Integer Unit* tiene habilitadas instrucciones de multiplicación y división.
- Caché de datos, de un set de 4 KB, y de programa, de dos sets de 4 KB cada uno, separadas. Algoritmo de reemplazo LRU.
- Unidad de gestión de memoria, MMU, habilitada.
- Unidad de depuración, DSU, habilitada.
- Bus AMBA AHB sin soporte para transferencias partidas (*split*).
- Puertos JTAG y RS232 para depuración.
- Controlador de memoria del LEON2 con soporte para datos de 8 bits.
- UART, GPIO.
- Dos temporizadores con interrupciones separadas.
- Controlador de interrupciones del LEON3 con interrupciones secundarias habilitadas.

Por su parte, el módulo de detección se ha configurado con capacidad para ocho rutinas independientes, es decir, se ha asignado valor 3 al genérico *Addrsize*. Además se ha designado como maestro objetivo el propio microprocesador y como esclavo objetivo el controlador de memoria. La interrupción asignada al error de ejecución es la número 10, y la 9 corresponde al error de escritura. Por último, el espacio de direcciones asignado al módulo de detección es el comprendido entre los valores hexadecimales B0000000 y B00FFFFF.

El programa escogido para su ejecución por parte del microprocesador durante el test es el algoritmo de la burbuja. Este programa consiste en la ordenación de un vector de números mediante un recurrente rastreo y posterior permutación de aquellos números en posiciones adyacentes que no cumplan la regla de ordenación establecida. En el presente proyecto, este algoritmo se aplica sobre un vector de 10 elementos con valores del 1 al 10 que deben ordenarse de forma ascendente y que inicialmente están ordenados de forma descendente, lo que constituye el caso más desfavorable posible. Los valores desordenados se introducen inicialmente en la memoria, y cada vez que se lee o modifica uno de estos valores debe accederse a memoria. Sobre este algoritmo básico se han añadido una serie de modificaciones que hacen posible la detección de fallos por parte del módulo de detección, las cuales se listan a continuación:

- Antes de ejecutar el algoritmo el módulo de detección debe recibir la configuración de la rutina. La dirección de inicio se corresponde con el primer elemento del vector, pero la de fin no se asocia con el elemento número 10 del vector, sino con el 11. Esto se debe a que se accede varias veces a cada uno de los 10 elementos del vector antes de finalizar el algoritmo, por lo que es necesario establecer una dirección distinta como dirección de fin. El valor al que desborda el watchdog se fija en 3000, habiendo determinado por simulación previa que constituye tiempo más que suficiente para ejecutar el algoritmo.

- Tras completar el algoritmo de ordenación se ha incluido una escritura adicional en la memoria, en la dirección correspondiente al elemento número 11 del vector de números. Esta escritura permite que el módulo de detección detecte el final del algoritmo.

- Inmediatamente después se ejecuta un bucle que lee todos los datos del vector y los manda por el puerto de salida de propósito general (GPIO). Este bucle se ha incluido para facilitar la labor de verificación a lo largo de las pruebas.

- Finalmente, se incluye un bucle que permite ejecutar la escritura inicial de datos, el algoritmo de ordenación y la salida por puertos dos veces.

4.3.2 Procedimiento de ensayo

Durante el test de errores se han insertado un total de 1000 fallos en el sistema LEON3. Se ha optado por un número de errores inferior a lo que es habitual en estas prácticas para realizar manualmente la inserción de fallos y la posterior verificación.

El modelo de error empleado, denominado *bit-flip*, asume que los fallos son provocados por modificaciones imprevistas y no permanentes de los valores que almacenan los biestables del circuito. Este tipo de errores modelan el fenómeno conocido como SEU (*Single Event Upset*), producido por partículas de alta energía que inciden sobre los biestables del circuito y provocan un cambio del valor almacenado; de ahí el nombre de *bit-flip*. Se supone que estos

fenómenos tienen lugar con poca frecuencia, por lo que se asume que sólo se dará como máximo uno de estos eventos por cada ejecución del programa.

Los fallos se han insertado en el propio microprocesador, ya que el módulo de detección observa las transferencias del maestro al esclavo. En concreto se han considerado los registros de la *integer unit*, más todos aquellos registros del banco de registros del microprocesador cuyo valor cambia al menos una vez durante la ejecución del programa, determinados por simulación previa.

Se ha definido una ventana de tiempos dentro de la cual se han insertado los fallos, que abarca la secuencia de operaciones de lectura y escritura para la ordenación de los datos, para cada una de las dos ejecuciones del algoritmo de ordenación. Es decir, en esta ventana de tiempos no se incluye ni la configuración del módulo ni la salida de datos por el puerto de propósito general.

La selección del biestable y del instante de tiempo donde se inserta el fallo para cada una de las pruebas se ha realizado de forma aleatoria con ayuda del programa KFinfasim, desarrollado por el grupo de Diseño Microelectrónico y Aplicaciones del departamento de Tecnología Electrónica de la Universidad Carlos III de Madrid.

Para simplificar la tarea de verificación se ha incluido en el código del módulo de detección una sentencia *assert*. El comando *assert* es una instrucción no sintetizable del código VHDL cuya misión es mostrar un mensaje por la consola del simulador, en este caso Modelsim, cuando se cumple una cierta condición, en este caso la detección de un error de ejecución. Por otra parte se ha realizado una primera simulación sin errores y se ha capturado la evolución temporal del bus de direcciones y del bus de datos de escritura del bus AHB, mediante la utilidad *dataset snapshot* de Modelsim, para compararlos con cada una de las pruebas, empleando para ello la utilidad *waveform compare* de Modelsim.

La metodología del test se describe a continuación:

- En primer lugar se hace avanzar la simulación del sistema hasta el instante de tiempo donde se debe insertar el fallo.
- Después se selecciona el bit en el que se debe insertar el fallo y se fuerza su valor de modo que éste cambia. Esta función está implementada en el propio Modelsim.
- A continuación se ejecuta el resto de la simulación.
- Por último se identifica el tipo de error con ayuda de la sentencia *assert* y de la utilidad *waveform compare*, que compara de los valores del bus AHB respecto a la simulación sin errores. Sin embargo estos dos indicadores no son definitivos. En caso de duda se pasa a estudiar en profundidad el cronograma de la simulación.

4.3.3 Resultados y clasificación de errores

Cada uno de los errores insertados durante el test se ha clasificado de acuerdo a la siguiente lista:

- *Sin efecto*. Error que no afecta al conjunto de transferencias.

- *Detectado*. Error que modifica la secuencia de operaciones de escritura y que ha sido detectado por el módulo de detección de fallos, incluso si el resultado final es el esperado.

- *No detectado*. Error que afecta al bloque de transferencias, pero que no es detectado por el módulo de detección.

- *Error fatal*. Fallo como consecuencia del empleo de un programa de simulación de circuitos digitales. La simulación no puede progresar debido a una violación de las normas del lenguaje VHDL. Típicamente se trata de una variable entera con rango preestablecido a la que se asigna un valor fuera de dicho rango.

Los resultados del test de errores se muestran en la Tabla 8, donde se indica el número total de cada uno de los tipos de fallos y la proporción que representan respecto del total de fallos insertados.

	#	%
Sin efecto	818	81,8
Detectado	160	16,0
No detectado	17	1,7
Error fatal	5	0,5

Tabla 8: Resultados del test de errores

En esta clasificación de fallos hay una serie de errores que son relevantes, es decir, que permiten determinar la eficiencia del módulo de detección. Estos errores son los detectados, los no detectados y los errores fatales. Por tanto es interesante conocer la frecuencia de aparición de cada uno de estos tipos de errores en relación al conjunto de errores relevantes, como se muestra en la Tabla 9.

Detectado	87,91 %
No detectado	9,34 %
Error fatal	2,75 %

Tabla 9: Proporción de errores relevantes

En el caso de los errores fatales, como no ha sido posible completar la simulación, se considera el caso más desfavorable posible. Es decir, los errores fatales tienen el mismo tratamiento que los no detectados a efectos de determinar la eficiencia del módulo de detección.

La inserción de errores y la clasificación de los mismos se han realizado manualmente. Esta metodología requiere invertir un tiempo y esfuerzo superiores a los necesarios si se hubiera empleado un sistema más automatizado. De hecho, el número de fallos insertados en el sistema es inferior a lo habitual en un test de errores. Sin embargo, este procedimiento permite conocer en profundidad las causas que originan los fallos, ya sean detectados o no. Este conocimiento resulta muy valioso para la optimización de cualquier técnica de detección de errores transitorios que se emplee en un sistema basado en el LEON3, o incluso en otros microprocesadores con arquitectura SPARC.

Dentro de la categoría de errores detectados, las causas más comunes de fallo son las siguientes:

- Pérdida de secuencia, lo que causa el desbordamiento del *watchdog*. Esto puede suceder por diferentes motivos, entre los cuales se citan:
  - Error transitorio sobre el contador de programa.
  - Alteración de la dirección de retorno de un procedimiento.
  - Intento de escritura en una dirección que no está asignada a ningún esclavo.
- Fallos que afecten a los registros de la ALU, ya sea sobre alguno de los operandos, sobre el resultado o un error que modifique la operación que se realiza. Dependiendo del dato afectado y de la operación que se haya ejecutado, puede producirse tanto una alteración de la firma como el desbordamiento del *watchdog*.
- Modificación de la dirección de escritura de un dato en memoria. Dentro de este caso se distinguen varios comportamientos:
  - La dirección afectada no corresponde al último dato de la rutina. Esto generará una firma incorrecta siempre que el dato en cuestión se vuelva a utilizar en el programa. En caso contrario se trataría de un error no detectado.
  - La dirección de fin de la rutina resulta alterada. En tal caso no se detecta el final de la rutina, lo que provoca el desbordamiento del *watchdog*.
- Error sobre alguno de los datos a almacenar en memoria. Esto origina una firma errónea.
- Alteraciones imprevistas en los registros de las variables empleadas en el flujo de control del algoritmo. Este hecho provoca habitualmente la pérdida de instrucciones, aunque en algunos casos podría suponer la repetición de instrucciones. De cualquier forma la firma resultante de la ejecución es diferente de la ejecución sin errores.

Entre los errores que no han sido detectados por el módulo de detección se han encontrado los siguientes casos:

- Fallos que afectan al registro que almacena el número de ejecuciones del algoritmo. Dentro de este fenómeno se han descubierto dos tendencias posibles, según cómo interprete el compilador la condición de salida del bucle que ejecuta dos veces el programa:
  - Una posible condición de fin es que el número de ejecuciones sea mayor o igual que 2. En tal caso un error que modifique dicho registro durante la primera ejecución provoca la terminación del programa sin que la segunda ejecución haya tenido lugar.
  - Otra posibilidad es que el bucle termine cuando el número de ejecuciones es exactamente igual a 2. Entonces un fallo transitorio sobre el registro del número de ejecuciones supone que el contador, de 32 bits, debe desbordar para alcanzar de nuevo el valor 2. Esto implica que el algoritmo se ejecutará un gran número de veces de forma correcta.
- Modificación de la dirección de escritura de un dato en memoria. Este tipo de eventos origina un error no detectado en el caso de que el dato escrito en una dirección

errónea no vuelva a emplearse en lo que queda de la ejecución del programa, siempre que no se trate del último dato de la rutina. En tal caso se genera la misma firma que en ausencia de error.

- Pérdida de secuencia fuera de la rutina, debida a un intento de escritura en una dirección que no está asignada a ningún esclavo. Es necesario señalar que todos los fallos se han insertado durante la ejecución de la rutina, pero en este caso sus efectos tienen lugar una vez finalizada dicha rutina.

La Tabla 10 muestra las frecuencias de aparición de cada una de las clases de error no detectado que se han encontrado durante la realización del test de errores.

	#	%
Fallo en el contador de ejecuciones del algoritmo	12	70,59
Dirección errónea	1	5,88
Pérdida de secuencia fuera de rutina	4	23,53

Tabla 10: Clasificación de errores no detectados

5 CONCLUSIONES Y TRABAJOS FUTUROS

En el presente proyecto se ha diseñado un módulo de detección de fallos para sistemas embebidos capaz de determinar la existencia de errores transitorios mediante la supervisión de las comunicaciones del bus interno del sistema. El módulo de detección ha sido concebido para supervisar las comunicaciones que vayan del microprocesador a la memoria, aunque su aplicación puede extenderse a la supervisión de comunicaciones entre un maestro cualquiera del bus y un esclavo cualquiera que funcione de forma similar a una memoria. Dicho módulo de detección se ha integrado en la biblioteca de componentes GRLIB, así como en la herramienta de configuración Xconfig, con el objetivo de facilitar su inserción en sistemas embebidos. Como caso de aplicación específico se ha implementado en un sistema basado en el microprocesador LEON3 y el protocolo de comunicaciones AMBA.

Como parte del proyecto se han realizado diversas pruebas de validación de las características del módulo de detección de fallos. En concreto se han efectuado simulaciones para verificar la funcionalidad del módulo de detección, un test de errores para cuantificar la capacidad de detección de errores, y una síntesis para conocer los requerimientos de área y velocidad de funcionamiento del módulo de detección.

El módulo diseñado presenta una eficiencia bastante aceptable. Es capaz de detectar en torno a un 88 % de los fallos relevantes que se producen. Es interesante destacar que la gran mayoría de errores no detectados son errores que quedan latentes y cuyos efectos se muestran una vez finalizada la rutina, por lo que están fuera de las capacidades de supervisión del módulo de detección de fallos. Como consecuencia de ello sería posible mejorar la fiabilidad simplemente configurando más rutinas para su supervisión.

El módulo de detección tiene unas dimensiones reducidas en comparación con el sistema LEON3. Respecto de una configuración del LEON3 muy básica, el módulo utiliza el 7,9 % de las ALUT's que necesita el LEON3, dando soporte a 4 rutinas. Esta cifra se reduce el 5,7 % al dar soporte sólo a 2 rutinas. Por otra parte el módulo de detección emplea el 17,9 % o el 9,8% de los registros que implementa el LEON3 con soporte para 4 o 2 rutinas, respectivamente. Estas cifras disminuirán aún más al implementar configuraciones más complejas del LEON3. Alternativamente, al aumentar el número de rutinas soportadas podría emplearse una memoria en vez de un grupo de registros para implementar el banco de registros.

El módulo de detección apenas afecta a las capacidades del LEON3 en lo que se refiere a frecuencia de funcionamiento. La frecuencia máxima de funcionamiento es superior a la especificada en la configuración (150 MHz) dando soporte a 2 rutinas, mientras que para 4 rutinas tan sólo se resiente un 0,15%.

Como posibles líneas de trabajo a seguir se citan:

- Optimización del diseño a partir del conocimiento adquirido sobre las causas de fallo. Se podría aumentar la eficiencia del módulo de detección en base al análisis realizado sobre los errores no detectados, mientras que las causas más probables de errores detectados permitirían optimizar el coste económico, el área ocupada y la velocidad de funcionamiento. Por ejemplo, dado que una buena parte de los errores no detectados se deben al contador de ejecuciones del algoritmo, en los que las opciones del compilador presentan una gran relevancia, sería interesante comprobar el efecto resultante de utilizar diferentes opciones de compilación en conjunto con modificaciones del diseño del módulo de detección de errores.

- Exportación del diseño a otras arquitecturas, que empleen diferentes microprocesadores o distintos buses de comunicaciones. En este último caso habría que modificar previamente la interfaz del diseño, como es lógico.

BIBLIOGRAFÍA

[1] ARM, *AMBA Specification (Rev 2.0)*, ARM IHI 0011A, 1999.

[2] SPARC International, *The SPARC Architecture Manual*, v. 8, Revision SAV080SI9308, 1992.

[3] J. Gaisler, E. Catovic, M. Isomäki, K. Glembo, y S. Habinc, "LEON3 - High-performance SPARC V8 32-bit processor", *GRLIB IP Core User´s Manual*, v. 1.0.20, págs. 533-557, Feb. 2009.

[4] J. Gaisler, S. Habinc, y E. Catovic, *GRLIB IP Library User´s Manual,* v. 1.0.20, 2008.

[5] J. Gaisler, *BCC - Bare-C Cross-Compiler User's Manual*, v. 1.0.29, Feb. 2007.

[6] Gaisler Research, *TSIM2 Simulator User's Manual*, v. 2.0.11, Dec. 2008.

[7] A. Avizienis, "The N-Version Approach to Fault-Tolerant Software", *IEEE Transactions on Software Engineering.*, vol. 1, nº 2, págs 1491-1501, Dic. 1985.

[8] Z. Alkhalifa, V.S.S. Nair, N. Krishnamurthy, y J.A. Abraham, "Design and Evaluation of System-Level Checks for On-Line Control Flow Error Detection", *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, nº 6, págs. 627-641, Jun. 1999.

[9] N. Oh, P.P. Shirvani, y E.J. McCluskey, "Control-Flow Checking by Software Signatures", *IEEE Transactions on Reliability*, vol.51, nº 2, págs 111-122, Dic. 2000.

[10] P. Cheynet, B. Nicolescu, R. Velazco, M. Rebaudengo, M. Sonza Reorda, y M. Violante, "Experimentally Evaluating an Automatic Approach for Generating Safety-Critical Software with Respect to Transient Errors", *IEEE Transactions on Nuclear Science*, vol. 47, nº 6, págs 2231-2236, Dic. 2000.

[11] N. Oh, S. Mitra, y E.J. McCluskey, "ED4I: Error Detection by Diverse Data and Duplicated Instructions", *IEEE Transactions on Computers*, nº 2, págs 180-199, Feb. 2002.

[12] A. Mahmood, y E.J. McCluskey, "Concurrent Error Detection Using Watchdog Processors - A Survey", *IEEE Transactions on Computers*, vol. 37, nº 2, págs. 160-174, Feb. 1988.

[13] M. Namjoo, y E.J. McCluskey, "Watchdog Processors and Capability Checking", *Proc. Int'l Symp. Fault-Tolerant Computing*, págs. 245-248, 1982.

[14] A. Mahmood, D.J. Lu, y E.J. McCluskey, "Concurrent Fault Detection Using a Watchdog Processor and Assertions", *Proc. IEEE Int'l Test Conf.*, págs. 622-628, 1983.

[15] M Namjoo, "CERBERUS-16: An architecture for a General Purpose Watchdog Processor", *Proc. IEEE Int'l Symp. Fault-Tolerant Computing*, págs. 216-219, 1983.

[16] K. Wilken, y J.P. Shen, "Continuous Signature Monitoring: Low-Cost Concurrent Detection of Processor Control Errors", *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 9, nº 6, págs. 629-641, Jun. 1990.

[17] P. Bernardi, L.M. Veiras Bolzani, M. Rebaudengo, M. Sonza Reorda, F.L. Vargas, y M. Violante "A New Hybrid Fault Detection Technique for Systems-on-a-Chip", *IEEE Transactions on Computers*, vol. 55, nº 2, págs. 185-198, Feb. 2006.

ANEXO A. PRESUPUESTO

A.1. Introducción

El objetivo del presente proyecto es diseñar un módulo capaz de detectar errores en sistemas embebidos mediante la supervisión del bus de comunicaciones del sistema. Este módulo debe ser lo más eficiente posible, es decir, debe detectar una proporción de errores tan alta como sea posible. A parte de ello el módulo diseñado debe ser económico tanto en coste como en espacio ocupado, debe afectar en la menor medida posible a las prestaciones del microprocesador al cual se conecta y debe poderse insertar en el sistema de forma no intrusiva.

A.2. Fases del proyecto

La ejecución del proyecto se divide en las fases que se detallan a continuación:

- Preparación previa. Se refiere al estudio del protocolo de comunicaciones AMBA, del microprocesador LEON3 y de las herramientas que llevan asociadas. Se estiman unas 240 horas de dedicación.

- Diseño y depuración del módulo de detección. En cada una de estas fases se emplean aproximadamente 200 horas.

- Pruebas. Por un lado se realiza una síntesis del diseño, con el objetivo de conocer los requerimientos de espacio y frecuencia de funcionamiento. Por otro, se efectúa un test de errores para comprobar la eficiencia del módulo de detección. El tiempo invertido en cada una de estas pruebas se estima en 24 y 120 horas, respectivamente.

La Figura A. 1 muestra el diagrama de Gantt del proyecto. La distribución de tiempos se muestra en horas.

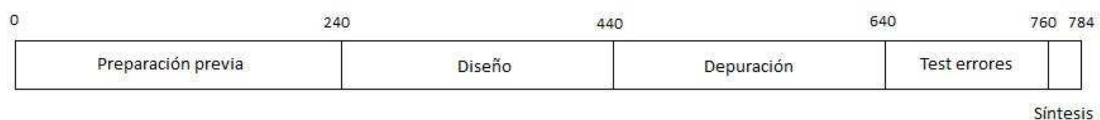

Figura A. 1: Diagrama de Gantt

A.3. Desglose de costes

Los costes directos asociados al proyecto constan de tres conceptos: personal, equipos y licencias software.

Para el cálculo de los costes de personal se considera un coste horario del empleado de 25 € por hora, una jornada laboral de 8 horas y 5 días laborables a la semana. El proyecto puede ser ejecutado íntegramente por un único empleado.

Se requiere de un puesto informático para la realización del proyecto. El equipo debe estar disponible durante toda la duración del proyecto. Dicho equipo se valora en 2000 €, y se le atribuye un período de amortización de 3 años.

Durante el desarrollo del proyecto es necesario utilizar los programas informáticos Modelsim (para la depuración y el test de errores) y Quartus (para la síntesis). El coste anual de las licencias de estos programas se consideran de 8000 € y 6000 € al año, respectivamente. Estos programas se emplean en diversos proyectos de la misma empresa, por lo que se imputa un coste proporcional al número de horas dedicados al presente proyecto.

Se estima que los costes indirectos del proyecto suponen un 10% de los costes directos. Por otro lado, la empresa aplica una tasa del 35% del coste total como beneficios.

La Tabla A. 1 muestra el coste imputado a cada uno de los conceptos antes mencionados, así como el coste total del proyecto.

Concepto	**Cantidad**
Costes de personal	19600 €
Amortización de equipos	251 €
Licencias software	1301 €
Costes indirectos	2507 €
Total	23267 €

Tabla A. 1: Resumen de costes

El presupuesto completo del presente proyecto se cifra en 31410 euros.

ANEXO B. CÓDIGO FUENTE DE LOS ARCHIVOS DEL PROYECTO

B.1. Genéricos.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
library grlib;
use grlib.amba.all;
use grlib.stdlib.all;
use grlib.devices.all;
library gaisler;
use gaisler.misc.all;

package genericos is
  type banco_reg is array(integer range <>) of std_logic_vector(31 downto 0);

-- maestros
  constant nmaster: integer := 2;

  constant leon3     : integer := 0;
  constant dma       : integer := 1;  -- comprobar posición del dma

  subtype master_name is string(1 to 10);

  type master_table_type is array (0 to nmaster-1) of master_name;

  constant master_table : master_table_type := (
  leon3  => "leon3     ",dma => "dma       "
);

--esclavos
  constant nslave : integer := 2;

  constant memory : integer := 0;
  constant apb : integer := 1; -- comprobar posición del APB bridge

  subtype slave_name is string(1 to 10);

  type slave_table_type is array (0 to nslave-1) of slave_name;

  constant slave_table : slave_table_type := (
  memory => "memory    ", apb => "apb       "
);

-- declaración de componentes

component spyslv is
  generic (
    hindex   : integer := 4;
    spyaddr  : integer := 16#B00#;
    hirq_w   : integer := 9;  -- número de interrupción para error de escritura
    hirq_spy : integer := 10; -- número de interrupción para error de ejecución
    venid    : integer := VENDOR_SP;
    devid    : integer := SP_SPY;
    version  : integer := 0;
    nummaster: integer := 0;  -- posición del maestro que se quiere espiar
    numslave : integer := 0;  -- posición del esclavo que se quiere espiar
    addrsize : integer := 2); -- numero de bits para direccionar el banco de
            registros-resultados, el numero de rutinas soportadas será 2**adrrsize
  port (
    rst   : in  std_ulogic;
    clk   : in  std_ulogic;
    ahbsi : in  ahb_slv_in_type;
```

```
    ahbso : out  ahb_slv_out_type
    );
end component;

end genericos;
```

B.2. Spyslv.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
library grlib;
use grlib.amba.all;
use grlib.stdlib.all;
use grlib.devices.all;
library gaisler;
use gaisler.misc.all;
library sp;
use sp.genericos.all;

entity spyslv is
  generic (
    hindex   : integer := 4;
    spyaddr  : integer := 16#B00#;
    hirq_w   : integer := 9;  -- número de interrupción para error de escritura
    hirq_spy : integer := 10; -- número de interrupción para error de ejecución
    venid    : integer := VENDOR_SP;
    devid    : integer := SP_SPY;
    version  : integer := 0;
    nummaster: integer := 0;  -- posición del maestro que se quiere espiar
    numslave : integer := 0;  -- posición del esclavo que se quiere espiar
    addrsize : integer := 2); -- número de bits para direccionar el banco de
                registros-resultados, el numero de rutinas soportadas será 2**adrrsize
  port (
    rst   : in  std_ulogic;
    clk   : in  std_ulogic;
    ahbsi : in  ahb_slv_in_type;
    ahbso : out  ahb_slv_out_type
    );
end;

architecture spyslv of spyslv is

-- declaración de componentes
component interfaz is
  generic (
    hindex   : integer := 4;
    spyaddr  : integer := 16#B00#;
    hirq_w   : integer := 9;   -- número de interrupción para error de escritura
    hirq_spy : integer := 10;  -- número de interrupción para error de ejecución
    venid    : integer := VENDOR_SP;
    devid    : integer := SP_SPY;
    version  : integer := 0;
    nummaster: integer := 0;  -- posición del maestro que se quiere espiar
    numslave : integer := 0;  -- posición del esclavo que se quiere espiar
    addrsize : integer := 2); -- número de bits para direccionar el banco de
                registros-resultados, el numero de rutinas soportadas será 2**adrrsize
  port (
    rst      : in  std_ulogic;
    clk      : in  std_ulogic;
    ahbsi    : in  ahb_slv_in_type;
    ahbso    : out ahb_slv_out_type;
    err_w    : in std_ulogic;                     -- error de escritura
    err_spy  : in std_ulogic;                     -- error de ejecución
    newdata  : out std_ulogic;                    -- nuevo dato en el bus
```

```
    data     : out std_logic_vector(31 downto 0); -- bus de datos
    ahb_addr : out std_logic_vector(31 downto 0);
                                        -- dirección del bus amba ahb para el control
    write    : out std_ulogic;                    -- escritura del maestro al esclavo
    address  : out unsigned(addrsize+1 downto 0); -- dirección del banco de registros
    waddress : out std_ulogic                   -- habilitar escritura de direcciones
    );
end component;
component regbank is
  generic (
    hindex    : integer := 4;
    nummaster : integer := 0;  -- posición del maestro que se quiere espiar
    numslave  : integer := 0;  -- posición del esclavo que se quiere espiar
    addrsize  : integer := 2); -- número de bits para direccionar el banco de
                registros-resultados, el numero de rutinas soportadas será 2**adrrsize
  port (
    data        : in std_logic_vector(31 downto 0);
                                                   -- bus de datos, proveniente de la
                                                                              interfaz
    address     : in unsigned(addrsize+1 downto 0);
                      -- dirección del banco de registros, proveniente de la interfaz
    waddress    : in std_ulogic;           -- habilitar escritura de direcciones
    rst         : in std_ulogic;
    clk         : in std_ulogic;
    resultado   : in std_logic_vector(31 downto 0); -- resultado
    raddress    : in integer range 0 to (2**addrsize)-1;
            -- dirección del banco de registros de resultado, proveniente del control
    wresult     : in std_ulogic;           -- habilitar escritura de resultado
    block_write : in std_ulogic;           -- protección contra escritura mientras se
                                            comprueba una rutina, indicado por control
    block_en    : in std_ulogic;
       -- habilitar modificación de protección contra escritura, indicado por control
    err_w       : out std_ulogic;          -- error de escritura
    iniaddress  : out banco_reg (0 to (2**addrsize)-1);
                                           -- dirección de inicio de todos los intervalos
    outaddress  : out std_logic_vector(31 downto 0);
                                                 -- dirección de fin del intervalo actual
    timer       : out std_logic_vector(31 downto 0);
                                            -- salida del watchdog-timer hacia el control
    firstresult : out std_logic_vector(31 downto 0);
                              -- resultado de la primera iteración del intervalo actual
    block_readi :    out std_ulogic_vector (0 to (2**addrsize)-1);
                                  -- protección contra lectura, dirección inicio => '1'
    block_readf :    out std_ulogic_vector (0 to (2**addrsize)-1)
                                  -- protección contra lectura, dirección fin    => '1'
    );
end component;
component control is
  generic (
    hindex    : integer := 4;
    nummaster : integer := 0;  -- posición del maestro que se quiere espiar
    numslave  : integer := 0;  -- posición del esclavo que se quiere espiar
    addrsize  : integer := 2); -- número máximo de rutinas soportadas
  port (
    result_spy    : in std_logic_vector(31 downto 0);
                                                   -- resultado proveniente del espía
    rst           : in  std_ulogic;
    clk           : in  std_ulogic;
    ahb_addr      : in std_logic_vector(31 downto 0);
                         -- dirección del bus amba ahb proveniente de la interfaz
    write         : in std_ulogic;         -- escritura del maestro al esclavo
    iniaddress    : in banco_reg (0 to (2**addrsize)-1);
                                           -- dirección de inicio de todos los intervalos
    outaddress    : in  std_logic_vector(31 downto 0);
                                                 -- dirección de fin del intervalo actual
    timer         : in std_logic_vector(31 downto 0);
```

```
                                    -- salida del watchdog-timer hacia el control
    firstresult   : in std_logic_vector(31 downto 0);
                            -- resultado de la primera iteración del intervalo actual
    block_readi   :  in std_ulogic_vector (0 to (2**addrsize)-1);
                                -- protección contra lectura, dirección inicio => '1'
    block_readf   :  in std_ulogic_vector (0 to (2**addrsize)-1);
                                -- protección contra lectura, dirección fin    => '1'
    err_spy       : out std_ulogic;        -- error de ejecución
    enable        : out std_ulogic;
       -- habilitación, en el rango de direcciones de algun bloque que se debe espiar
    clear         : out std_ulogic;        -- borra el resultado, indicado por
control
    result_regbank: out std_logic_vector(31 downto 0);
                                         -- resultado hacia el banco de registros
    raddress      : out integer range 0 to (2**addrsize)-1;
                                    -- dirección del banco de registros de resultado
    wresult       : out std_ulogic;        -- habilitar escritura de resultado
    block_write   : out std_ulogic;        -- protección contra escritura mientras se
                                           comprueba una rutina, indicado por control
    block_en      : out std_ulogic
       -- habilitar modificación de protección contra escritura, indicado por control
    );
end component;

component spy is
  generic (
    hindex   : integer := 4;
    nummaster: integer := 0;  -- posición del maestro que se quiere espiar
    numslave : integer := 0); -- posición del esclavo que se quiere espiar
  port (
    enable            : in std_ulogic;      -- habilitación, en el rango de
direcciones                                    de algún bloque que se debe espiar
    clear     : in std_ulogic;             -- borra el resultado, indicado por
control
    rst       : in std_ulogic;
    clk       : in std_ulogic;
    newdata   : in std_ulogic;             -- nuevo dato en el bus
    data      : in std_logic_vector(31 downto 0);
                                      -- bus de datos, proveniente de la interfaz
    resultado : out std_logic_vector(31 downto 0) -- se pasa el resultado al control
    );
end component;

-- declarción de señales
  signal ahb_addr      : std_logic_vector(31 downto 0);
                                    -- dirección del bus amba ahb para el control
  signal newdata       : std_ulogic;       -- nuevo dato en el bus
  signal data          : std_logic_vector(31 downto 0);  -- bus de datos
  signal address       : unsigned(addrsize+1 downto 0);
                                              -- dirección del banco de registros
  signal waddress      : std_ulogic;
  signal wresult       : std_ulogic;       -- habilitar escritura de resultado
  signal raddress      : integer range 0 to (2**addrsize)-1;
          -- dirección del banco de registros de resultado, proveniente del control
  signal iniaddress    : banco_reg (0 to (2**addrsize)-1);
                                   -- dirección de inicio de todos los intervalos
  signal outaddress    : std_logic_vector(31 downto 0);
                                         -- dirección de fin del intervalo actual
  signal timer         : std_logic_vector(31 downto 0);
                                    -- salida del watchdog-timer hacia el control
  signal firstresult   : std_logic_vector(31 downto 0);
                            -- resultado de la primera iteración del intervalo actual
  signal result_spy    : std_logic_vector(31 downto 0);
                                               -- resultado proveniente del espía
  signal result_regbank: std_logic_vector(31 downto 0);
                                         -- resultado hacia el banco de registros
```

```
  signal enable        : std_ulogic;
       -- habilitación, en el rango de direcciones de algun bloque que se debe espiar
  signal clear         : std_ulogic;       -- borra el resultado, indicado por
control
  signal block_write   : std_ulogic;       -- protección contra escritura mientras se
                                           comprueba una rutina, indicado por control
  signal block_en      : std_ulogic;
       -- habilitar modificación de protección contra escritura, indicado por control
  signal err_spy       : std_ulogic;       -- error de ejecución
  signal err_w         : std_ulogic;       -- error de escritura
  signal block_readi   : std_ulogic_vector ((2**addrsize)-1 downto 0);
  signal block_readf   : std_ulogic_vector ((2**addrsize)-1 downto 0);
  signal write         : std_ulogic;

begin

interfaz0: interfaz generic map(hindex,spyaddr,hirq_w,hirq_spy,venid,devid,version,
           nummaster,numslave,addrsize)
           port map(rst,clk,ahbsi,ahbso,err_w,err_spy,newdata,data,ahb_addr,write,
           address,waddress);

regbank0:  regbank generic map(hindex,nummaster,numslave,addrsize)
           port map(data,address,waddress,rst,clk,result_regbank,raddress,wresult,
           block_write,block_en,err_w,iniaddress,outaddress,timer,firstresult,
           block_readi,block_readf);

control0:  control generic map(hindex,nummaster,numslave,addrsize)
           port map(result_spy,rst,clk,ahb_addr,write,iniaddress,outaddress,timer,
           firstresult,block_readi,block_readf,err_spy,enable,clear,result_regbank,
           raddress,wresult,block_write,block_en);

spy0:      spy generic map(hindex,nummaster,numslave)
           port map(enable,clear,rst,clk,newdata,data,result_spy);

end spyslv;
```

B.3. Interfaz.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
library grlib;
use grlib.amba.all;
use grlib.stdlib.all;
use grlib.devices.all;
library gaisler;
use gaisler.misc.all;
library sp;
use sp.genericos.all;

entity interfaz is
  generic (
    hindex    : integer := 4;
    spyaddr   : integer := 16#B00#;
    hirq_w    : integer := 9;  -- número de interrupción para error de escritura
    hirq_spy  : integer := 10; -- número de interrupción para error de ejecución
    venid     : integer := VENDOR_SP;
    devid     : integer := SP_SPY;
    version   : integer := 0;
    nummaster : integer := 0;  -- posición del maestro que se quiere espiar
    numslave  : integer := 0;  -- posición del esclavo que se quiere espiar
    addrsize  : integer := 2); -- número de bits para direccionar el banco de
                registros-resultados, el número de rutinas soportadas será 2**adrrsize
  port (
    rst      : in std_ulogic;
```

```
    clk      : in std_ulogic;
    ahbsi    : in ahb_slv_in_type;
    ahbso    : out ahb_slv_out_type;
    err_w    : in std_ulogic;                     -- error de escritura
    err_spy  : in std_ulogic;                     -- error de ejecución
    newdata  : out std_ulogic;                    -- nuevo dato en el bus
    data     : out std_logic_vector(31 downto 0); -- bus de datos
    ahb_addr : out std_logic_vector(31 downto 0);
                                        -- dirección del bus amba ahb para el control
    write    : out std_ulogic;                    -- escritura del maestro al esclavo
    address  : out unsigned(addrsize+1 downto 0); -- dirección del banco de registros
    waddress : out std_ulogic                   -- habilitar escritura de direcciones
    );
end;

architecture interfaz of interfaz is

-- registro de configuración
  constant hconfig : ahb_config_type := (
    0 => ahb_device_reg ( venid, devid, 0, version, 0),
    4 => ahb_membar(spyaddr,'0','0',16#FFF#),
    others => zero32);

  constant zeros : std_logic_vector (19 downto addrsize+4) := (others => '0');

-- señales internas
  signal fallo      : std_ulogic;                    -- error de comunicación
  signal newtrans   : std_ulogic;                    -- nueva transferencia
  signal ready      : std_ulogic;                    -- ahbso.hready
  signal resp       : std_logic_vector (1 downto 0); -- ahbso.hresp
  signal s_address  : unsigned(addrsize+1 downto 0);
                                                 -- dirección del banco de registros
  signal s_waddress : std_ulogic;   -- habilitación de escritura en registro
  signal newtrans_c : std_ulogic;   -- escritura del maestro al esclavo seleccionados

begin

-- espionaje del bus - detección de nueva transferencia
  process (ahbsi)
  begin
    if ahbsi.hmaster = conv_std_logic_vector(nummaster,4) and ahbsi.hsel(numslave) =
    '1' then                        -- transferencia entre esclavo y maestro deseados
      if ahbsi.hwrite = '1' then  -- transferencia del maestro al esclavo
        case (ahbsi.htrans) is    -- modo de trasferencia
          when HTRANS_NONSEQ => newtrans_c <= '0';
            if ahbsi.hready = '1' then
              newtrans_c <= '1';
            end if;
          when HTRANS_SEQ    => newtrans_c <= '0';
            if ahbsi.hready = '1' then

              newtrans_c <= '1';
            end if;
          when others        => newtrans_c <= '0';
        end case;
      else    -- transferencia del esclavo al maestro
        newtrans_c <= '0';
      end if;
    else      -- transferencia entre módulos incorrectos
      newtrans_c <= '0';
    end if;
  end process;

  process(clk,rst)
  begin
    if rst = '0' then
```

```
      newtrans <= '0';
    elsif clk'event and clk = '1' then
      newtrans <= newtrans_c;
    end if;
  end process;

--interfaz de comunicaciones
  process (clk,rst)
  begin
    if rst = '0' then  -- se resetean todas las señales relacionadas con el bus amba
                          y el estado de transferencia
      s_address  <= (others => '0');
      s_waddress <= '0';
      fallo      <= '0';
      resp       <= HRESP_OKAY;
      ready      <= '0';
    elsif clk'event and clk = '1' then
      fallo      <= '0';
      ready      <= '0';
      resp       <= HRESP_OKAY;
      s_address  <= (others => '0');
      s_waddress <= '0';
      if fallo = '1' then
        resp  <= HRESP_ERROR;
        ready <= '1';
      end if; -- END IF ya que podría continuar la transmisión de datos aún con error
      if ahbsi.hready = '1' then         -- bus disponible
        if ahbsi.hsel(hindex) = '1' then -- el LEON se comunica con nuestro módulo
          if ahbsi.hwrite = '1' then     -- transferencia del maestro al esclavo
            case (ahbsi.htrans) is       -- modo de transferencia
              when HTRANS_NONSEQ|HTRANS_SEQ =>
                if (ahbsi.haddr(19 downto addrsize+4) = zeros)then
                  -- los 12 bits más significativos forman el HSEL, los bits desde
                  -- 2 a addrsize+3 conforman la dirección del banco de registros
                  -- las direcciones en el bus amba aumentan de 4 en 4
                  -- dirección válida
                  resp       <= HRESP_OKAY;
                  ready      <= '1';
                  s_address  <= unsigned(ahbsi.haddr(addrsize+3 downto 2));
                  s_waddress <='1';
                else
                  -- direccion no válida
                  fallo <= '1';
                  resp  <= HRESP_ERROR;
                end if;
              when others =>
                ready <= '1';
            end case;
          else   -- transferencia del esclavo al maestro, se trata como un error
            fallo <= '1';
            resp  <= HRESP_ERROR;
          end if;
        else     -- transferencia con otro módulo
          ready <= '1';
        end if;
      end if;
    end if;
  end process;

-- asignación de interrupciones
  process (err_spy,err_w)
    variable irq : std_logic_vector (NAHBIRQ-1 downto 0) := (others => '0');
  begin
    if err_spy = '1' then
      irq(hirq_spy) := '1';
    else
```

```
      irq(hirq_spy) := '0';
    end if;
    if err_w = '1' then
      irq(hirq_w) := '1';
    else
      irq(hirq_w) := '0';
    end if;
    ahbso.hirq <= irq;
 end process;

-- asignación de salidas
  ahbso.hresp   <= resp;
  ahbso.hready  <= ready;
  ahbso.hconfig <= hconfig;
  ahbso.hindex  <= hindex;
  ahb_addr      <= ahbsi.haddr;
  data          <= ahbsi.hwdata;
  newdata       <= newtrans;
  address       <= s_address;
  waddress      <= s_waddress;
  write         <= newtrans_c;

end interfaz;
```

B.4. Regbank.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
library grlib;
use grlib.amba.all;
use grlib.stdlib.all;
use grlib.devices.all;
library gaisler;
use gaisler.misc.all;
library sp;
use sp.genericos.all;

entity regbank is
  generic (
    hindex    : integer := 4;
    nummaster : integer := 0;  -- posición del maestro que se quiere espiar
    numslave  : integer := 0;  -- posición del esclavo que se quiere espiar
    addrsize  : integer := 2); -- número de bits para direccionar el banco de
                registros-resultados, el número de rutinas soportadas será 2**adrrsize
  port (
    data        : in std_logic_vector(31 downto 0);
                                       -- bus de datos, proveniente de la interfaz
    address     : in unsigned(addrsize+1 downto 0);
                      -- dirección del banco de registros, proveniente de la interfaz
    waddress    : in std_ulogic;           -- habilitar escritura de direcciones
    rst         : in std_ulogic;
    clk         : in std_ulogic;
    resultado   : in std_logic_vector(31 downto 0);  -- resultado
    raddress    : in integer range 0 to (2**addrsize)-1;
            -- dirección del banco de registros de resultado, proveniente del control
    wresult     : in std_ulogic;           -- habilitar escritura de resultado
    block_write : in std_ulogic;           -- protección contra escritura mientras se
                                            comprueba una rutina, indicado por control
    block_en    : in std_ulogic;
        -- habilitar modificación de protección contra escritura, indicado por control
    err_w       : out std_ulogic;          -- error de escritura
    iniaddress  : out banco_reg (0 to (2**addrsize)-1);
                                      -- dirección de inicio de todos los intervalos
    outaddress  : out std_logic_vector(31 downto 0);
```

```
                                          -- dirección de fin del intervalo actual
    timer       : out std_logic_vector(31 downto 0);
                                       -- salida del watchdog-timer hacia el control
    firstresult : out std_logic_vector(31 downto 0);
                               -- resultado de la primera iteración del intervalo actual
    block_readi : out std_ulogic_vector (0 to (2**addrsize)-1);
                                  -- protección contra lectura, dirección inicio => '1'
    block_readf : out std_ulogic_vector (0 to (2**addrsize)-1)
                                  -- protección contra lectura, dirección fin    => '1'
    );
end;

architecture regbank of regbank is

-- banco de registros
-- por cada rutina soportada hay 4 registros: direcciones de inicio y de fin, valor
   al que desborda el watchdog y resultado de la primera ejecución
signal reginicio : banco_reg (0 to (2**addrsize)-1);
signal regfin    : banco_reg (0 to (2**addrsize)-1);
signal regtime   : banco_reg (0 to (2**addrsize)-1);
signal regresult : banco_reg (0 to (2**addrsize)-1);
-- bits de protección, uno para cada una de las rutinas soportadas
signal block_ri  : std_ulogic_vector (0 to (2**addrsize)-1);
                                  -- protección contra lectura, dirección inicio => '1'
signal block_rf  : std_ulogic_vector (0 to (2**addrsize)-1);
                                  -- protección contra lectura, dirección fin    => '1'
signal block_w   : std_ulogic_vector (0 to (2**addrsize)-1);
                                                -- protección contra escritura => '1'

signal error_w : std_ulogic;  -- error de escritura

begin
  -- entrada de datos
  process(clk,rst)
    variable indice : integer range 0 to (2**addrsize)-1;
  begin
    if rst='0' then
      for i in 0 to (2**addrsize)-1 loop
        reginicio(i) <= (others => '0');
        regfin(i)    <= (others => '0');
        regtime(i)   <= (others => '0');
        regresult(i) <= (others => '0');
        block_ri(i)  <= '1';
        block_rf(i)  <= '1';
        block_w(i)   <= '0';
        error_w      <= '0';
      end loop;
    elsif clk'event and clk='1' then
      error_w <= '0';
      if block_en = '1' then            -- modificar protección contra escritura
        block_w(raddress) <= block_write;
      elsif waddress = '1' then         -- escritura en el bloque de direcciones, si
                                                lo permite la protección contra
escritura
        indice := to_integer(address(addrsize-1 downto 0));
        if block_w(indice) = '0' then   -- protección contra escritura desactivada
          if address(addrsize+1) = '1' then
                                          -- escritura del tiempo del watchdog-timer
            regtime(indice)<= data;
          else
            if address(addrsize) = '1' then  -- escritura de la dirección de inicio

              reginicio(indice)<= data;
              block_ri(indice) <= '0';       -- protección contra lectura desactivada
            elsif address(addrsize) = '0' then  -- escritura de la dirección de fin
              regfin(indice)   <= data;
```

```
              block_rf(indice) <= '0';       -- protección contra lectura desactivada
            end if;
          end if;
        elsif block_w(indice) = '1' then
                        -- error de escritura : intento de escritura en zona protegida
          error_w <= '1';
        end if;
      end if;
      if wresult = '1' then                  -- escritura en el bloque de resultados
        regresult(raddress)<= resultado;
      end if;
    end if;
  end process;

  -- salida de datos
  process(reginicio,regfin,regresult,raddress,error_w)
  begin
    iniaddress  <= reginicio;
    outaddress  <= regfin(raddress);
    timer       <= regtime(raddress);
    firstresult <= regresult(raddress);
    block_readi <= block_ri  ;
    block_readf <= block_rf  ;
    err_w       <= error_w;
  end process;

  process(clk)
    begin
      assert (error_w = '0')
        report "---Error de escritura---"
          severity note;
  end process;

end regbank;
```

B.5. Control.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
library grlib;
use grlib.amba.all;
use grlib.stdlib.all;
use grlib.devices.all;
library gaisler;
use gaisler.misc.all;
library sp;
use sp.genericos.all;


entity control is
  generic (
    hindex    : integer := 4;
    nummaster : integer := 0;  -- posición del maestro que se quiere espiar
    numslave  : integer := 0;  -- posición del esclavo que se quiere espiar
    addrsize  : integer := 2); -- número máximo de rutinas soportadas
  port (
    result_spy    : in std_logic_vector(31 downto 0);
                                                -- resultado proveniente del espía
    rst           : in std_ulogic;
    clk           : in std_ulogic;
    ahb_addr      : in std_logic_vector(31 downto 0);
                          -- dirección del bus amba ahb proveniente de la interfaz
    write         : in std_ulogic;         -- escritura del maestro al esclavo
    iniaddress    : in banco_reg (0 to (2**addrsize)-1);
```

```
                                        -- dirección de inicio de todos los intervalos
    outaddress    : in  std_logic_vector(31 downto 0);
                                              -- dirección de fin del intervalo actual
    timer         : in std_logic_vector(31 downto 0);
                                         -- salida del watchdog-timer hacia el control
    firstresult   : in std_logic_vector(31 downto 0);
                              -- resultado de la primera iteración del intervalo actual
    block_readi   :   in std_ulogic_vector (0 to (2**addrsize)-1);
                                  -- protección contra lectura, dirección inicio => '1'
    block_readf   :   in std_ulogic_vector (0 to (2**addrsize)-1);
                                  -- protección contra lectura, dirección fin    => '1'
    err_spy       : out std_ulogic;        -- error de ejecución
    enable        : out std_ulogic;
       -- habilitación, en el rango de direcciones de algun bloque que se debe espiar
    clear         : out std_ulogic;        -- borra el resultado, indicado por
control
    result_regbank: out std_logic_vector(31 downto 0);
                                              -- resultado hacia el banco de registros
    raddress      : out integer range 0 to (2**addrsize)-1;
                                      -- dirección del banco de registros de resultado
    wresult       : out std_ulogic;         -- habilitar escritura de resultado
    block_write   : out std_ulogic;        -- protección contra escritura mientras se
                                           comprueba una rutina, indicado por control
    block_en      : out std_ulogic         -- habilitar modificación de protección
                                                  contra escritura, indicado por
control
    );

end;

architecture control of control is

  type estado is (espera, rutina, fin_rutina, iteracion_1, iteracion_2);
  signal actual, siguiente: estado;
  signal check            : std_ulogic_vector (0 to (2**addrsize)-1);
                                  -- iteración 1 '0' o iteración 2 '1' para cada rutina
  signal checki           : std_ulogic;
    -- toma el valor de uno de los bits de check para operar en la parte combinacional
  signal pos_intervalo    : integer range 0 to (2**addrsize)-1;
                                        -- posición de la rutina actual (combinacional)
  signal pos_en           : std_ulogic;
                                        -- habilitar escritura del registro de posición
  signal pos              : integer range 0 to (2**addrsize)-1;
                                           -- posición de la rutina actual (secuencial)
  signal error_spy        : std_ulogic;
  signal watchdog         : unsigned (31 downto 0); -- temporizador

begin

-- determinación del siguiente estado
  process(actual,iniaddress,outaddress,firstresult,result_spy,ahb_addr,pos,watchdog,
  write)
    variable indice       : integer range 0 to (2**addrsize)-1;
    variable coincidencia : std_ulogic;
  begin
    enable         <= '0';
    clear          <= '0';
    error_spy      <= '0';
    wresult        <= '0';
    result_regbank <= (others => '0');
    block_write    <= '0';
    block_en       <= '0';
    pos_en         <= '0';
    pos_intervalo  <= pos;
    checki         <= check(pos);
    case (actual) is
```

```
    when espera =>
      coincidencia := '0';
      indice := 0;
      for i in 0 to (2**addrsize)-1 loop
        if (block_readi(i) = '0') and (block_readf(i) = '0') then
                                        -- se comprueba la protección contra lectura
          if iniaddress(i)= ahb_addr then
            indice :=i;
            coincidencia := coincidencia or '1';
          else
            coincidencia := coincidencia or '0';
          end if;
        else
          coincidencia := coincidencia or '0';
        end if;
      end loop;
      if  coincidencia = '1' and write = '1' then
        siguiente     <= rutina;
        pos_intervalo <= indice;
        pos_en        <= '1';
        block_write   <= '1';
        block_en      <= '1';
      else
        siguiente <= espera;
      end if;
    when rutina =>
      enable <= '1';
      if watchdog < unsigned(timer) then -- comprobación del watchdog
        if ahb_addr = outaddress and write = '1' then
          siguiente <= fin_rutina;
        else
          siguiente <= rutina;
        end if;
      else                               -- error de ejecución - límite de tiempo
        siguiente <= espera;
        error_spy <= '1';
        clear     <= '1';
        checki    <= '0';
        block_en  <= '1';
      end if;
    when fin_rutina =>
      enable <= '1';
      if checki= '0' then
        siguiente <= iteracion_1;
      else
        siguiente <= iteracion_2;
      end if;
    when iteracion_1 =>
              -- toma el resultado del espía y lo guarda en el banco de registros
      clear          <= '1';
      wresult        <= '1';
      result_regbank <= result_spy;
      checki         <= '1';
      siguiente      <= espera;
    when iteracion_2 =>
         -- toma el resultado del banco de registros y lo compara con el del espía
      checki   <= '0';
      block_en <= '1';
      if (result_spy = firstresult) then -- correcto, ambas iteraciones coinciden
        error_spy <= '0';
      else              -- error de ejecución - las ejecuciones han sido distintas
        error_spy <= '1';
      end if;
      clear     <= '1';
      siguiente <= espera;
  end case;
```

```
  end process;

-- actualización del estado, watchdog, check y registro de posición
  process(clk,rst)
  begin
    if rst='0' then
      actual   <= espera;
      pos      <= 0;
      watchdog <= to_unsigned(0,32);
      for i in 0 to (2**addrsize)-1 loop
        check(i) <= '0';
      end loop;
    elsif clk'event and clk = '1' then
      actual <= siguiente;
      if actual = espera then
        watchdog <= to_unsigned(0,32);
      elsif actual = rutina then
        watchdog <= watchdog + to_unsigned(1,32);
      end if;
      if pos_en = '1' then
        pos <= pos_intervalo;
      end if;
      check(pos)<=checki;
    end if;
  end process;

  process(clk)
  begin
    assert (error_spy = '0')
      report "---Error de ejecución---"
        severity note;
  end process;

-- asignación de salidas
  raddress <= pos_intervalo;
  err_spy  <= error_spy;

end control;
```

B.6. Spy.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
library grlib;
use grlib.amba.all;
use grlib.stdlib.all;
use grlib.devices.all;
library gaisler;
use gaisler.misc.all;
library sp;
use sp.genericos.all;

entity spy is
  generic (
    hindex  : integer := 4;
    nummaster: integer:= 0;  -- posición del maestro que se quiere espiar
    numslave: integer := 0); -- posición del esclavo que se quiere espiar
 port (
    enable    : in std_ulogic;
       -- habilitación, en el rango de direcciones de algun bloque que se debe espiar
    clear     : in std_ulogic;             -- borra el resultado, indicado por
control
    rst       : in std_ulogic;
    clk       : in std_ulogic;
```

```
    newdata   : in std_ulogic;             -- nuevo dato en el bus
    data      : in std_logic_vector(31 downto 0);
                                          -- bus de datos, proveniente de la interfaz
    resultado : out std_logic_vector(31 downto 0) -- se pasa el resultado al control
    );
end;

architecture spy of spy is

-- registros del modulo
signal result: std_logic_vector (31 downto 0);  -- total acumulado
constant poli_prim : unsigned (32 downto 0) := "100011000000000000000000000000011";

begin

  process (clk,rst)
  begin
    -- reset
    if rst = '0' then
      result <= (others => '0');
    elsif clk'event and clk = '1' then
      if clear = '1' then
                -- inicialización del resultado, tiene preferencia clear sobre enable
        result <= (others => '0');
      elsif enable = '1' then
                            -- lectura del dato del bus y actualización del resultado
        if newdata = '1' then
          -- generación de la firma
          result(0) <= data(0) xor (result(31) and poli_prim(32));
          for i in 1 to 31 loop
            result(i) <= data(i) xor result(i-1) xor (result(31) and
            poli_prim(32-i));
          end loop;
        end if;
      end if;
    end if;
  end process;
  -- salida del resultado
  resultado <= result;
end spy;
```

B.7. Spy.in

```
# Configuración para el diseño leon3-gr-xc3s-1500
#
# hindex para memory controller en xilinx es 3
#
#

mainmenu_option next_comment
  comment 'SPY'
  bool 'SPY Enable' CONFIG_SPY_ENABLE
  if["$CONFIG_SPY_ENABLE" = "y"]; then
    int 'rutinas soportadas' CONFIG_SPY_ADDRSIZE 2
    choice 'Target Master' \
      "Leon3 CONFIG_SPY_LEON3 \
      Dma CONFIG_SPY_DMA" leon3
    choice 'Target Slave' \
      "Memory_controller CONFIG_SPY_MEMORY \
      APB_bridge CONFIG_SPY_APB" Memory_controller
    hex 'Spy Address' CONFIG_SPY_ADDRESS C00
    int 'interrupción para error de ejecución' CONFIG_SPY_HIRQ_SPY 10
    int 'interrupción para error de escritura' CONFIG_SPY_HIRQ_W 9
  fi
endmenu
```

B.8. Spy.in.h

```
#if defined CONFIG_SPY_LEON3
#define CONFIG_SPY_NUMMASTER leon3
#elif defined CONFIG_SPY_DMA
#define CONFIG_SPY_NUMMASTER dma
#endif

#if defined CONFIG_SPY_MEMORY
#define CONFIG_SPY_NUMSLAVE memory
#elif defined CONFIG_SPY_APB
#define CONFIG_SPY_NUMSLAVE apb
#endif

#ifndef CONFIG_SPY_ENABLE
#define CONFIG_SPY_ENABLE 0
#endif
#ifndef CONFIG_SPY_NUMMASTER
#define CONFIG_SPY_NUMMASTER 0
#endif
#ifndef CONFIG_SPY_ADDRSIZE
#define CONFIG_SPY_ADDRSIZE 0
#endif
#ifndef CONFIG_SPY_NUMSLAVE
#define CONFIG_SPY_NUMSLAVE 0
#endif
#ifndef CONFIG_SPY_ADDRESS
#define CONFIG_SPY_ADDRESS C00
#endif

#ifndef CONFIG_SPY_HIRQ_SPY
#define CONFIG_SPY_HIRQ_SPY 10
#endif
#ifndef CONFIG_SPY_HIRQ_W
#define CONFIG_SPY_HIRQ_W 9
#endif
```

B.9. Spy.in.vhd

```
-- Configuración del espía
  constant CFG_SPY_EN       : integer := CONFIG_SPY_ENABLE;
  constant CFG_SPY_NUMMASTER: integer := CONFIG_SPY_NUMMASTER;
  constant CFG_SPY_NUMSLAVE : integer := CONFIG_SPY_NUMSLAVE;
  constant CFG_SPY_ADDRSIZE : integer := CONFIG_SPY_ADDRSIZE;
  constant CFG_SPY_ADDR     : integer := 16#CONFIG_SPY_ADDRESS#;
  constant CFG_SPY_HIRQ_SPY : integer := CONFIG_SPY_HIRQ_SPY;
  constant CFG_SPY_HIRQ_W   : integer := CONFIG_SPY_HIRQ_W;
```

B.10. Spy.in.help

```
SPY Enable
CONFIG_SPY_ENABLE
  Say Y here to enable SPY

rutinas soportadas
CONFIG_SPY_ADDRSIZE
  Indicar el numero de bits para direccionar el banco de registros (el numero de
rutinas soportadas será 2**addrsize)

Prompt for Target master
CONFIG_SPY_LEON3
  Indicar el Maestro a espiar
```

```
Prompt for Target slave
CONFIG_SPY_MEMORY
  Indicar el esclavo a espiar

Prompt for Spy Address
CONFIG_SPY_ADDRESS
  Indicar la dirección del modulo espía

Interrupciones
CONFIG_SPY_HIRQ_SPY
  Indicar el numero para la interrupción de error de ejecución

Interrupciones
CONFIG_SPY_HIRQ_W
  Indicar el numero para la interrupción de error de escritura
```