Universidad de Alcalá
Escuela Politécnica Superior

Grado en ingeniería de computadores

Trabajo Fin de Grado

Implementación y validación de nuevas funcionalidades para el simulador LeonViP

Autor: Borja Losa Cruz

Tutor/es: Pablo Parra Espada

ESCUELA POLITECNICA SUPERIOR

TRIBUNAL:

Presidente: Antonio Da Silva Fariña

Vocal 1º: Elena Campo Montalvo

Vocal 2º: Pablo Parra Espada

2020 - 2021

Índice

Resumen

Uno de los puntos clave al realizar una misión espacial es asegurar de que todos los elementos que lo componen tengan un riesgo mínimo de fallo, y en el caso de haberlo, poder recuperarse de el. Por ello es esencial poder simular posibles fallos y validar la respuesta a ellos.

En este proyecto se plantea ampliar la funcionalidad del simulador LeonViP que está siendo desarrollado actualmente por el grupo de investigación Space Research Group de la Universidad de Alcalá, y se caracteriza por poder simular fallos de memoria como los que se producirían en caso de radiación cósmica.

Abstract

One of the key points when doing a space mission is to ensure that all elements that compose it have a very low risk of failing, and, in case of a fault, it has to be able to recover from it. Because of this, is essential to be able to simulate these faults and validate the response to them.

This project seeks to extend the LeonViP simulator, which is currently being developed by the Space Research Group of the University of Alcalá and is characterized for being able to simulate memory faults like the ones produced by cosmic radiation.

Resumen Extendido

Es común que cuando se diseña un nuevo hardware se busque que sea lo más rápido posible, pero también hay casos en los que se requiere dar una mayor prioridad a la fiabilidad del sistema. El caso más usual es cuando el hardware va a ser desplegado en un entorno de difícil acceso, lo cual dificulta o imposibilita poder hacer reparaciones de éste, y hace que un fallo pueda resultar en la pérdida total del sistema.

Los puntos de fallo más comunes a tener en cuenta a la hora de realizar despliegues en entornos difíciles suelen ser los producidos por temperaturas extremas, longevidad o contacto con líquidos, y, si el objetivo es desplegarlo en el espacio exterior, además es necesario considerar la radiación cósmica. Este tipo de radiación no suele tenerse en cuenta ya que la mayoría lo bloquea la atmosfera terrestre, pero es algo que no puede ignorarse en el caso de desplegar el sistema fuera de la atmosfera, ya que la radiación cósmica es capaz de alterar bits de memoria.

El los últimos años se ha visto un incremento considerable en la cantidad de misiones espaciales, resultando en una mayor necesidad de sistemas que funcionen con una mínima probabilidad de fallo en estos entornos. Esto también conlleva que se busquen herramientas que ayuden a crear este tipo de sistemas, como es el caso del simulador LeonViP.

LeonViP es un simulador actualmente en desarrollo por el grupo de investigación Space Research Group de la Universidad de Alcalá que busca imitar el comportamiento del procesador LEON3, que a su vez se basa en la arquitectura del conjunto de instrucciones (*Instruction Set Architecture* o ISA) SPARC versión 8 (SPARCv8). El simulador se caracteriza por permitir simular fallos como los que pueden ocurrir en caso de interferencias por radiación cósmica. Esto hace que, sin tener que usar hardware real, se pueda comprobar si el sistema a desplegar responde correctamente a este tipo de fallos. LeonViP ya es capaz de simular correctamente gran parte de las funcionalidades especificadas por LEON3, y ha sido empleado con éxito en el desarrollo del software de arranque y de aplicación de la unidad de control del instrumento Energetic Particle Dectector de la misión Solar Orbiter.

El objetivo de este proyecto es añadir ciertas funcionalidades presentes en LEON3 y la versión extendida de SPARC v8 que aún no han sido implementadas, en concreto, la unidad de manejo de memoria (*Memory Management Unit* o MMU), y el puntero de ventana alternativo (*Alternative Window Pointer* o AWP) que, aunque no son necesarios para el correcto funcionamiento del sistema, dan un mayor rango de opciones y flexibilidad que pueden ayudar a optimizarlo.

La MMU se encarga de virtualizar la memoria, lo que permite protegerla de accesos no permitidos, y facilita crear secciones de memoria lo cual puede resultar muy útil para casos en los que se necesario mover ciertos datos de un área de memoria a otra.

El AWP es una funcionalidad incluida en la versión extendida de SPARCv8 que ofrece una mayor flexibilidad a la hora de moverse por las ventanas de registros, y además habilita el poder crear particiones de estas ventanas.

Lista de acrónimos

AHB Advanced High-performance Bus

ASI Address Space Identifier

ASR Ancillary State Register

AWP Alternative Window Pointer

CP Coprocessor

CWP Current Window Pointer

FAR Fault Address Register

FAV Fault Address Valid

FPU Floating Point Unit

FSR Fault Status Register

ISA Instruction Set architecture

ISS Instruction Set Simulators

IU Integer Unit

MMU Memory management Unit

NPC Next Program Counter

PA Physical Address

PC Program Counter

PPN Physical Page Number

PSO Partial Store Ordering

PSR Processor State Register

PSZ Page Size

PTD Page Table Descryptor

PTE Page Table Entry

PTP Page Table Pointer

RISC Reduced Instruction Set Computer

TBR Trap Base Register

TLB Translation Lookaside Buffer

TSO Total Store Ordering

VA Virtual Address

WRPSR Write Processor State Register

1. Introducción

1.1 Objetivos

Este proyecto busca introducir nuevas mejoras en el simulador LeonViP, con el principal fin de acercar aún más este simulador a la implementación del procesador LEON3 que se quiere imitar. Estas mejoras no son algo nuevo, son elementos ya definidos en SPARCv8 e implementados en LEON3 que aún no se habían desarrollado sobre el simulador, y que pueden resultar útiles para la optimización de la ejecución de ciertas tareas.

Al final de este trabajo se pretende haber obtenido un conocimiento general de LEON3 y SPARCv8, un entendimiento detallado de los elementos a implementar, y lograr su correcta implementación sobre el simulador LeonViP.

1.2 Planteamiento

El problema se ha dividido en varias partes, comenzando por un estudio de la arquitectura y el simulador sobre el que se va a trabajar, seguido de un análisis más a fondo de los elementos a implementar, y terminando con el desarrollo y validación de éstos.

1.2.1 Estudio del procesador LEON3, SPARCv8 y LeonViP

El proyecto ha consistido en la implementación software sobre el simulador LeonViP de varias funcionalidades aún no presentes en éste, las cuales vienen descritas en la especificación del procesador LEON3 y definidas en la arquitectura SPARCv8. Por ello, ha sido necesario tener una idea previa de cómo funciona esta arquitectura y como se ha implementado sobre el simulador.

Primero se ha realizado un estudio de SPARCv8, que describe una ISA de tipo RISC sobre la que se basa el procesador LEON3 y, tras obtener un conocimiento general de éste, se ha analizado la forma en que LEON3 implementa dicha ISA, enfocándolo en las características que más interesan a este proyecto. Finalmente se ha comprobado como el simulador LeonViP implementa mediante software los elementos definidos por LEON3, de nuevo enfocándolo en los aspectos más relevantes para el trabajo a desarrollar.

Los primeros apartados de este proyecto ofrecen un resumen de los aspectos más importantes de los elementos estudiados, evitando entrar en muchos detalles excepto cuando se describen las funcionalidades a implementar.

1.2.2 Desarrollo y validación de la MMU

Tras el estudio general de las bases sobre las que se trabaja, se ha realizado un análisis más a fondo de cómo se implementa la MMU en LEON3 y SPARCv8. En este caso, LEON3 sigue casi al pie de la letra la descripción de la MMU ofrecida por SPARCv8, con pequeñas diferencias enfocadas en dar nuevas opciones que aumentan la flexibilidad de configuración de la MMU.

Tras comprender el funcionamiento de la MMU, se realizó una primera implementación software sobre LeonViP, que fue modificándose a la vez que se realizaban pruebas de la nueva implementación, comparando sus resultados con los obtenidos al ejecutar esas mismas pruebas sobre la implementación real de LEON3.

En este trabajo se proporciona una descripción de este elemento y se muestra el resultado final de la implementación y pruebas tras lograr conseguir los objetivos buscados, detallando el código realizado tanto para la MMU como las pruebas que validan su correcto funcionamiento.

1.2.3 Desarrollo y validación del AWP

Al igual que con el caso anterior, se comenzó realizando un análisis de cómo funciona el mecanismo de puntero de ventana alternativo (*Alternative Window Pointer* o AWP), estudiando la definición dada por la versión extendida de SPARCv8 y su implementación en LEON3.

Tras comprender éste, se comenzó con su desarrollo sobre LeonViP el cual evolucionó según los resultados obtenidos al compararlos con los producidos por LEON3, hasta lograr que se comportara como lo hace la implementación real.

Finalmente, se ha realizado una descripción del elemento y detallado el resultado de la implementación del AWP sobre LeonViP, además de mostrar cómo se han desarrollado las pruebas y resultados finales de estas.

1.3 Herramientas

Para lograr los objetivos buscados ha sido necesario tener acceso a los siguientes recursos:

- Ordenador con Ubuntu y el IDE Eclipse: se ha usado como el entorno sobre el que se desarrollará tanto la implementación de los nuevos elementos de LeonViP como la creación de los programas de prueba para su validación. Para facilitar el desarrollo sobre LeonViP se ha usado una nueva rama de su proyecto de GitHub, permitiendo hacer modificaciones sin alterar la versión maestra, y finalmente uniéndolo con el original tras realizar la oportuna validación. Para la creación de los programas de prueba se ha empleado Eclipse con el compilador cruzado BCC, que permite generar código para la ISA de SPARC.

- Placa GR-XC3S-1500: FPGA programada con la implementación de LEON3 configurado con la MMU y AWP habilitados. Se ha utilizado para ejecutar los programas de prueba creados, cuyos resultados han servido para comprobar y validar el buen funcionamiento de las nuevas funcionalidades programas en LeonViP.

1.4 Estructura del proyecto

La memoria actual se compone de cinco apartados. El apartado actual buscar dar una idea general de los objetivos planteados para este trabajo.

El resto de los apartados siguen una estructura parecida a la descrita en el planteamiento. El apartado 2 ofrece un resumen general de las bases estudiadas para la realización de este trabajo, comenzando con la ISA de SPARCv8, la implementación dada por LEON3, y finalmente el simulador LeonViP. Como el objetivo ha sido implementar ciertas funcionalidades en el simulador, únicamente se darán detalles de los aspectos que más interesen a dicho fin.

En el tercer apartado se agrupa todo lo relacionado con el desarrollo de la MMU. Se comienza dando una descripción detallada de cómo se define su implementación en LEON3, seguido de cómo se ha realizado la implementación en LeonViP. La última parte se dedica a explicar el desarrollo de las pruebas de validación y resultados de éstas.

En el cuarto apartado sigue la misma estructura que el anterior, pero sobre el AWP. Comienza detallando como se implementa AWP en LEON3, además de explicar el funcionamiento de las ventanas de registros sobre los que trabaja. Luego se describe cómo se ha implementado sobre el simulador de LeonViP y finalmente cómo se realizan las pruebas para su validación y sus resultados.

El apartado cinco contiene las conclusiones obtenidas como resultado del trabajo realizado, dificultades encontradas, y trabajos futuros en relación a éste.

2. Bases del proyecto

Este proyecto se realiza sobre el simulador LeonViP, el cual busca imitar el comportamiento de la implementación de LEON3 de la arquitectura SPARCv8, haciendo necesario un conocimiento previo de todos estos elementos.

Los siguientes subapartados se limitan a dar una breve descripción de las características que definen estas bases, siendo en los apartados centrados en el desarrollo de las nuevas funcionalidades donde se especifique más detalladamente los elementos más relevantes para este trabajo.

2.1 Arquitectura SPARCv8

SPARCv8 es un set de instrucciones (*instruction set architecture* o ISA) de CPU de tipo reducido (*reduced instruction set computer* o RISC), diseñado para un amplio rango de aplicaciones, siendo su fin principal la optimización de compiladores y facilidad de segmentación (*pipeline*) al implementarse en hardware.

2.1.1 Elementos del procesador

Los elementos principales del procesador son los siguientes:

- **Unidad de enteros (*Integer Unit* o IU)**: contiene los registros de propósito general y controla la operación del procesador. Se encarga de las operaciones de enteros, el cálculo de direcciones para instrucciones de lectura y escritura, mantenimiento de contadores y del control de la ejecución de instrucciones de la unidad de coma flotante y coprocesador.

  Su implementación puede contener de 40 a 520 registros de propósito general, divididos en 8 registros globales más una pila circular de 2 a 32 grupos de 16 registros, denominados ventanas de registros. Una instrucción puede acceder a cualquiera de los registros globales más los registros de la ventana de registros indicada por el campo de puntero de ventana actual (*current window pointer* o CWP) del registro de estado del procesador (*processor state register* o PSR). Uno de los fines de este trabajo consiste en implementar en el simulador una extensión de este sistema de ventanas, el cual se explica más detalladamente en el apartado 4.2.2 Sistema de ventanas de registros.

  Cuando se accede a una instrucción desde memoria, se le adjunta a la dirección un identificador de espacio de dirección (*address space identifier* o ASI) que codifica si el procesador está en modo usuario o supervisor y si el acceso es al área de instrucciones o datos. En el apartado 2.1.5 Identificador de espacio de dirección (ASI) se detalla el funcionamiento de estos espacios de direcciones.

- **Unidad de punto flotante (*Floating-point Unit* o FPU)**: Se compone de 32 registros de punto flotante de 32 bits. Permite el uso de valores en precisión simple, doble y cuádruple, ocupando 1, 2 y 4 registros respectivamente.

- Las instrucciones de lectura y escritura de punto flotante se usan para el movimiento de datos entre FPU y memoria, donde la dirección de memoria es calculada por la IU. Las instrucciones *operate* de punto flotante son las encargadas de realizar la aritmética en punto flotante. En caso de no implementarse o deshabilitarse la FPU, un intento de ejecutar una instrucción de punto flotante generará una excepción.

- **Coprocesador (*Coprocessor* o CP)**: La ISA de SPARC da soporte para un coprocesador opcional, que depende de la implementación. Este tiene su propio grupo de registros definidos según la implementación. Sus instrucciones de lectura y escritura se usan para mover datos entre sus registros y memoria. En caso de no estar implementado, o deshabilitado, una instrucción del coprocesador generará un*a excepción.*

2.1.2 Instrucciones

Las instrucciones codifican en formato de 32 bits y se leen desde memoria según la dirección dada por el registro contador de programa (*program counter* o PC). Además, la arquitectura SPARCv8 hace uso de huecos de retardo (*delay slots*), que es una técnica que permite aprovechar mejor el sistema de segmentación o *pipelining* del procesador. En el caso de la arquitectura SPARCv8, cuando el procesador ejecuta una instrucción que cambia el flujo de ejecución del programa como, por ejemplo, un salto condicional, el procesador ejecuta a continuación la instrucción siguiente a la del cambio de flujo antes de saltar a la nueva localización.

Una instrucción también puede generar lo que la arquitectura SPARCv8 denomina *trap*. Este se produce debido a una condición excepcional, interrupción o petición de reset, y provoca un cambio de control hacia el sistema operativo. Los *traps* se detallan en el apartado 2.1.4 *Traps*.

En caso de no haber *trap*, se copia el contador de programa siguiente (*next program counter* o NPC) en el contador de programa o PC, y se incrementa NPC en 4. En caso de instrucción de cambio de flujo de programa, el procesador escribe la dirección de destino en el NPC.

Las instrucciones pueden dividirse en 6 categorías:

- **Lectura y escritura (*Load/Store*)**: Son las únicas instrucciones que pueden acceder a memoria. Usan 2 registros, o un registro más un desplazamiento inmediato de 13 bits con signo para calcular la dirección a memoria. La IU adjunta un ASI a esta dirección que codifica si se está en modo usuario o supervisor, y si se accede al área de instrucciones o datos. Su campo destino determina si es un registro de enteros, de punto flotante o de coprocesador, el que da o recibe el dato según si es escritura o lectura.

- **Aritméticas, lógicas y de desplazamiento**: Usan 2 operadores para calcular el resultado de dicha operación, a excepción de la instrucción SETHI que escribe los bits más altos de un registro. El resultado se guarda o bien en un registro de destino o se descarta.

- **Cambio de control**: engloban las instrucciones de llamada, saltos indirectos y traps condicionales. La mayoría de las instrucciones de cambio de control son de tipo retardado, donde la instrucción que sigue a la instrucción de cambio de control (instrucción retardada) se ejecuta antes de la instrucción de cambio de control.

- **Acceso a registro de estado**: instrucciones para la lectura y escritura de registros de estado y registros auxiliares (A*ncillary State Register* o ASR). El privilegio de estas instrucciones depende de la implementación.

- ***Operate* de punto Flotante y Coprocesador**: las instrucciones de operación de punto flotante o *FPop* realizan los calculos de FPU, usando los registros de este para las operaciones. Lo mismo con las del coprocesador, denominadas *CPop*, pero usando los registros de dicho coprocesador.

2.1.3 Modelo de memoria

Define la semántica de las operaciones de memoria como su lectura y escritura, y especifica la relación entre el orden en el que el procesador lanza estas operaciones y el orden en el que se ejecutan. El modelo estándar se denomina orden de guardado total (*total store ordering* o TSO) y todas las implementaciones de SPARC deben incluirlo. También se incluye un modelo llamado orden de guardado parcial (*partial store ordering* o PSO) que permite implementar sistemas de memoria de alto rendimiento. La arquitectura LEON soporta únicamente TSO.
En cuanto a la entrada y salida, se usan una serie de registros accesibles mediante instrucciones de tipo lectura y escritura, de coprocesador o de lectura y escritura a registros auxiliares.

2.1.4 *Traps*

Un *trap* es un cambio de control vectorizado al sistema operativo mediante una tabla de *traps* que contiene las 4 primeras instrucciones de cada manejador de *trap*. La dirección base de la tabla se establece por software en el registro base de trap (*Trap Base Register* o TBR) de la IU. El desplazamiento en la tabla se codifica con el número de cada *trap*. La mitad de la tabla se reserva para *traps* de hardware, y la otra mitad para *traps* generadas por instrucciones software.

Un *trap* puede ser causado por una excepción, una petición de interrupción externa, o mediante el uso de la instrucción TA. Antes de ejecutar una instrucción, la IU comprueba si hay alguna excepción o petición de interrupción pendiente, y, en caso de haberla, la IU selecciona la de mayor prioridad y causa su ejecución. Al terminarse la ejecución de un *trap* se produce lo que se denomina un retorno de *trap*, lo cual se realiza con la instrucción RETT. Esta instrucción permite retornar al código que se estuviera ejecutando cuando saltó el *trap*.

Cuando ocurre un *trap*, se produce un cambio de ventana y el hardware escribe un valor de 8 bits el campo de tipo de *trap* (*trap type* o **tt**) del registro TBR. El sistema define 256 fuentes de *trap*. De ellas, la mitad inferior, esto es, del vector o tipo de *trap* 0 al 0x7F, están asignadas a excepciones e interrupciones externas, mientras que la mitad superior, del vector o tipo de *trap* 0x80 al 0xFF,

están reservados para *traps* software. Estos *traps* software se disparan empleando la instrucción TA antes mencionada. Esta instrucción presenta un operando que se corresponde con el número de *trap* software que se va a disparar.

La siguiente tabla muestra la asignación de excepciones e interrupciones hardware según su tipo y prioridad, siendo la de prioridad 1 la más prioritaria y 31 la menos. Ciertas prioridades son dependientes de la implementación.

Exccepción o interrupción	**Prioridad**	**tt**
reset	1	-
error de guardado de dato	2	0x2B
fallo de acceso a instrucción por MMU	2	0x3C
fallo de acceso a instrucción	3	0x21
fallo de acceso a registro	4	0x20
excepción por acceso a instrucción	5	0x01
instrucción privilegiada	6	0x03
instrucción ilegal	7	0x02
FPU deshabilitada	8	0x04
CP deshabilitado	8	0x24
flush no implementado	8	0x25
watchpoint detectado	8	0x0B
desbordamiento de ventana (*overflow*)	9	0x05
desbordamiento de ventana (*underflow*)	9	0x06
dirección de memoria no alineada	10	0x07
excepción de FPU	11	0x08
excepción de CP	11	0x28
error de acceso a datos	12	0x29
error de acceso a datos por MMU	12	0x2C
excepción por acceso a datos	13	0x09
desbordamiento de tag	14	0x0A
división por cero	15	0x2A
instrucción de *trap*	16	0x80 - 0xFF
interrupción de nivel 15	17	0x1F
interrupción de nivel 14	18	0x1E
interrupción de nivel 13	19	0x1D
interrupción de nivel 12	20	0x1C
interrupción de nivel 11	21	0x1B
interrupción de nivel 10	22	0x1A
interrupción de nivel 9	23	0x19
interrupción de nivel 8	24	0x18
interrupción de nivel 7	25	0x17
interrupción de nivel 6	26	0x16
interrupción de nivel 5	27	0x15
interrupción de nivel 4	28	0x14
interrupción de nivel 3	29	0x13

interrupción de nivel 2	30	0x12
interrupción de nivel 1	31	0x11
excepción dependiente de implementación	según implementación	0x60 - 0x7F

Tabla 1 - Tabla de tipos y prioridades de trap definida por la arquitectura SPARCv8

2.1.5 Identificador de espacio de dirección (ASI)

SPARCv8 define 4 de los 256 posibles ASI, que corresponden con las 4 combinaciones posibles a la hora de determinar si se está operando en modo usuario o supervisor, y si se accede al área de instrucciones o de datos. El resto no están asignados explícitamente por esta arquitectura, pero sí que se dan recomendaciones para sistemas basados en SPARC, buscando disuadir el particionado del espacio alternativo en regiones sin cohesión que puedan inducir incompatibilidades hardware. En la siguiente tabla se muestran las asignaciones recomendadas.

ASI	Función
0	reservado
1	sin asignar
2	sin asignar (registros del sistema)
3	*flush*/*probe* de MMU
4	registros de MMU
5	diagnóstico de TLB de instrucciones de MMU
6	diagnóstico de TLB de datos o compartida de MMU
7	diagnóstico de TLB de I/O de MMU
8	área de instrucciones de usuario
9	área de instrucciones de supervisor
A	área de datos de usuario
B	área de instrucciones de supervisor
C	tag de caché de instrucciones
D	datos de caché de instrucciones
E	tag de caché de datos o compartida
F	datos de caché de datos o compartida
10	*flush* de caché compartida (página)
11	*flush* de caché compartida (segmento)
12	*flush* de caché compartida (región)
13	*flush* de caché compartida (contexto)
14	*flush* de caché compartida (usuario)
15	reservado
16	reservado
17	copia de bloque
18	*flush* de caché de instrucciones (página)
19	*flush* de caché de instrucciones (segmento)
1A	*flush* de caché de instrucciones (región)
1B	*flush* de caché de instrucciones (contexto)

1C	*flush* de caché de instrucciones (usuario)
1D	reservado
1E	reservado
1F	llenar bloque
20 - 2F	acceso a memoria ignorando MMU
30 - 7F	sin asignar
80 - FF	reservado

Tabla 2 - Asignaciones ASI recomendadas por SPARCv8

2.1.6 Ventanas de registros

Una característica que define a la arquitectura SPARC es que, en lugar de usar una pila en memoria para el guardado de datos locales, paso de argumentos y retornos de funciones, utiliza grupos de registros, buscando acelerar la escritura y lectura de dichos datos.

Al implementar esta arquitectura, se pueden definir entre 40 y 520 registros de propósito general, de los cuales 8 son registros globales accesibles por cualquier instrucción, y el resto conforman una pila circular de 2 a 32 grupos de 16 registros, denominados ventanas.

Los registros de cada ventana se dividen en registros locales, registros de salida y registros de entrada, donde los registros de entrada son los mismos que los registros de salida de la ventana adyacente.

Los registros locales guardan datos o variables locales a la función o procedimiento actual. Los registros de entrada contienen los argumentos de entrada de la función, y también sirven para guardar los valores retornados por ésta. Los registros de salida se usan para guardar los argumentos de llamada a la siguiente función.

La ventana en uso viene indicada por el campo de puntero actual (*Current Window Pointer* o CWP) del registro de estado del procesador. Cada vez que quiera realizar un cambio de ventana, como por ejemplo a la entrada de una función, se debe ejecutar la instrucción SAVE la cual mueve el CWP a la ventana siguiente. Al retornar de dicha función se debe ejecutar la instrucción RESTORE, que vuelve a la ventana anterior. En caso de un *trap*, y de un retorno de *trap* (RETT), también se modifica el CWP, disminuyéndolo y aumentándolo respectivamente.

Uno de los temas a desarrollar en este proyecto se centra en la implementación en el simulador LeonViP de una nueva funcionalidad de estas ventanas de registros denominada puntero de ventana alternativo (A*lternative Window Pointer* o AWP). Por ello se da una explicación más detallada de cómo funciona este sistema en el apartado *4.2.2 Sistema de ventanas de registros*.

2.1.7 Unidad de manejo de memoria

La especificación SPARC también define una arquitectura de referencia para la implementación de la unidad de manejo de memoria (SPARC Reference *Memory Management Unit* o SRMMU). La razón de especificar una MMU de referencia es promover la estandarización de ésta, reduciendo el tiempo para portar un sistema operativo a nuevo hardware y reducir la probabilidad de nuevos errores hardware.

La unidad de manejo de memoria permite el uso y traducción de direcciones virtuales de memoria, lo que habilita el poder dividir y proteger la memoria física entre los diferentes contextos que se ejecuten. La SRMMU se caracteriza por el uso de direcciones virtuales de 32 bits y direcciones físicas de 36 bits, con un tamaño mínimo de página de 4KiB y soporte de 3 niveles de indirección, diferentes tamaños de página según nivel y varios contextos, además de protección a nivel de página y procesamiento hardware de fallos de página.

Para realizar la traducción se usan una serie de tablas de páginas las cuales se almacenan en memoria, y se organizan en 4 niveles partiendo de la tabla denominada tabla de contexto, cuyas entradas se indexan según el el identificador de la tarea en ejecución. Estas tablas pueden contener 2 tipos de entradas: un descriptor de tabla de página (*page table descryptor* o PTD), que apuntaría a un nuevo nivel de la tabla de páginas, o una entrada de tabla de página (*page table entry* o PTE) que da la dirección física buscada además de varios bits de permisos de acceso.

Además, cuenta con una caché TLB (*translation lookaside buffer*) que guarda las traducciones más recientes usando como referencia la dirección virtual y el contexto, lo cual permite acelerar las traducciones de direcciones virtuales a físicas.

Esta unidad de memoria es uno de los elementos que se plantea implementar en el simulador LeonViP durante el desarrollo de este trabajo, por ello se da una definición más detallada en el apartado *3.2 Especificación*.

2.2 Plataforma LEON

LEON es una plataforma que implementa la arquitectura SPARCv8 y de la cual hay varias versiones. Para este proyecto se estudia la especificación de la implementación de LEON3 además de la implementación de LEON4 en el *System-on-Chip* GR740. Ambas son muy parecidas y se diferencian principalmente en que la especificación de LEON3 da un amplio rango de configuraciones que fija en gran parte la implementación GR740. Esta última, en cambio, añade nuevas funcionalidades, pero que no afectan a este proyecto. Es decir, se usa la especificación de LEON3 para el estudio de esta plataforma, y la implementación GR740 solamente para fijar algunas de las configuraciones que ofrece la anterior.

2.2.1 Unidad de enteros

LEON3 implementa la parte de enteros de la ISA SPARCv8 usando un pipeline de 7 fases y arquitectura Harvard, es decir, con caché de instrucciones y datos. Dichas cachés se comunican con la MMU, la cual cuenta con dos TLB (T*ranslation Lookaside Buffer*), para datos e instrucciones, que finalmente se comunica con memoria a través del bus AHB (*Advanced High-performance Bus*). También incluye soporte para ventanas de registros, señales *trap*, interrupciones y predicción de salto estática.

El pipeline implementado consta de las siguientes fases:

- ***Fetch***: Obtiene siguiente instrucción de caché de instrucciones, o de memoria si la caché está deshabilitada.
- ***Decode***: Decodifica la instrucción y se genera la dirección de llamada o salto.
- ***Register** Access*: Lee operandos de registros.
- ***Execute***: Se ejecutan operaciones de ALU, lógicas o desplazamientos. En caso de operaciones de memoria como carga, saltos o retornos, se genera la dirección.
- ***Memory***: Se almacena resultados en caché de datos.
- ***Exception***: Se tratan posibles *traps* e interupciones. En caso de lectura de caché se alinean los datos.
- ***Write***: Se escriben en los registros los resultados obtenidos.

2.2.2 Caché

LEON3 implementa dos cachés, una para datos y otra para instrucciones, lo que permite realizar *fetch* y una instrucción de *load* o *store* en un mismo ciclo. Ambas comparten la misma conexión AHB y ciertas partes de la MMU. Además, incluyen un buffer de escritura, lo que habilita escrituras en paralelo a la ejecución de instrucciones.

Cada caché cuenta principalmente con datos de memoria y etiquetas (*tags*). Los *tags* dan información con respecto a la dirección de memoria que se busca acceder, como la dirección de caché, si es válida, si está libre o si la MMU está habilitada. En cada dirección de *tag* se almacena cierto número de entradas o vías (*ways*) para un cierto conjunto de posibles direcciones de memoria.

Cada caché puede configurarse para implementar una o varias vías (*single-way* o *multi-way*), con un tamaño configurable de 1 a 256KiB por camino, a 16 o 32 bytes por línea. Esta configuración se fija a 4KiB por camino de 32 bytes por línea en la implementación GR740. En caso de varios caminos se da la opción de cambiar la política de reemplazamiento entre LRU, LRR o aleatoria.

Si la MMU está habilitada, la caché usa *tags* basados en la dirección virtual evitando realizar traducciones cada vez que se acceda a esta, pero siempre que se necesite acceder al bus AHB se debe traducir dicha dirección.

2.2.3 Unidad de manejo de memoria

LEON3 incluye una unidad de manejo de memoria o MMU, compatible con la dada como referencia por la arquitectura SPARCv8, también denominada SRMMU (*SPARCv8 reference MMU*).

La implementación de LEON3 sigue en gran parte la definición dada por la SRMMU que se describe en el apartado 2.1.7 Unidad de manejo de memoria, pero extendiendo ciertas partes que se listan a continuación:

- Permite elegir distintos tamaños mínimos de página (4, 8, 16 o 32 KiB)
- Permite elegir entre tres tipos de TLB: una sola TLB, dos TLB separadas según sea acceso a datos o a instrucciones, o no TLB.
- Permite elegir de 2 a 64 entradas de TLB.
- Permite deshabilitar la MMU, lo cual implica que se traten las direcciones como físicas.
- Añade el campo *mmuen*, el cual habilita un bit en los tags de caché que indican el nivel de privilegio.

En LEON3, cuando se quiere realizar un acceso a memoria y la MMU está habilitada, primero se compara dicha dirección virtual y contexto con los tags guardados en caché. En caso de no encontrarlo, se envía dicha dirección a la MMU la cual busca su traducción comparándolo con los tags guardados en sus TLBs. Si no lo encuentra, realiza un recorrido de las tablas de páginas, también conocido como *table walk*. En caso de no encontrar la traducción en la tabla de páginas, se genera una excepción cuyo vector de *trap* depende de si fue por acceder a datos o instrucciones. Esta diferenciación es interesante debido al pipeline, ya que el área de memoria que produjo el error determina la fase de éste y por lo tanto el orden en el que debe tratarse la excepción. Dicha prioridad de tratamiento de excepción también depende del tipo de excepción, lo cual se verá más en detalle en el apartado del desarrollo de la MMU.

2.2.4 Otros elementos

La especificación de LEON incluye una gran cantidad de elementos, ya que está implementando la arquitectura completa de SPARCv8 además de añadir otras funcionalidades. Aun así, la mayoría de estos son de baja o ninguna relevancia para el proyecto entre manos. De estos, los más interesantes son los siguientes:

- **Unidad de punto flotante**: SPARCv8 define 2 co-procesadores opcionales, uno para una unidad en punto flotante y otro definido por el usuario. LEON3 implementa solamente la FPU de la cual da 2 opciones. Una FPU de alto rendimiento (*high-performance FPU* o GRFPU) y una versión reducida de esta (*GRFPU-Lite*) con recursos lógicos limitados.

- **Interfaz AMBA**: se compone de una interfaz AHB maestra para todos los datos, instrucciones y el recorrido de tablas realizado por traducciones de la MMU. Se usan señales denominadas HPROT para indicar si el acceso es de instrucción o datos y si es en

modo usuario o supervisor. Para un mejor manejo de errores, existen diferentes señales de *trap* en función de si la excepción ocurrió al acceder a instrucción o dato o si se estaba realizando un recorrido de tablas de página.

- **Tolerancia a fallos**: una característica interesante del procesador LEON3 es que permite añadir un sistema de tolerancia de fallos de evento único (*single-event upset* o SEU) usando distintas técnicas como comprobación de paridad, triplicado de memoria y suma de comprobación (*checksum*). Esto permite encontrar y corregir fallos en la unidad de enteros, FPU, memoria y cachés.

2.2.5 Ventanas de registros

LEON implementa el sistema de ventanas de registros de acuerdo con la arquitectura SPARCv8 para el guardado y obtención de datos locales y el paso de argumentos entre tareas. Por defecto se usan 8 ventanas, pero LEON3 permite cambiar este tamaño.

2.2.6 *Traps*

LEON sigue de forma general el modelo de *traps* dado por la arquitectura SPARCv8, aunque cambia bastantes de las prioridades. En la siguiente tabla se muestra la asignación de tipos de *trap* y prioridades dadas por LEON:

Exccepción o interrupción	Prioridad	tt
reset	1	0x00
error de guardado de dato	2	0x2B
excepción de acceso a instrucción	3	0x01
instrucción ilegal	5	0x02
FPU deshabilitada	6	0x04
CP deshabilitado	6	0x24
watchpoint detectado	7	0x0B
desbordamiento de ventana (*overflow*)	8	0x05
desbordamiento de ventana (*underflow*)	8	0x06
fallo de acceso a registro	9	0x20
dirección de memoria no alineada	10	0x07
excepción de FPU	11	0x08
excepción de CP	11	0x28
excepción por acceso a datos	13	0x09
desbordamiento de tag	14	0x0A
división por cero	15	0x2A
instrucción de *trap*	16	0x80 - 0xFF
interrupción de nivel 15	17	0x1F
interrupción de nivel 14	18	0x1E
interrupción de nivel 13	19	0x1D

interrupción de nivel 12	20	0x1C
interrupción de nivel 11	21	0x1B
interrupción de nivel 10	22	0x1A
interrupción de nivel 9	23	0x19
interrupción de nivel 8	24	0x18
interrupción de nivel 7	25	0x17
interrupción de nivel 6	26	0x16
interrupción de nivel 5	27	0x15
interrupción de nivel 4	28	0x14
interrupción de nivel 3	29	0x13
interrupción de nivel 2	30	0x12
interrupción de nivel 1	31	0x11

Tabla 3 - Tabla de tipos y prioridades de trap definida por la implementación de LEON3

2.2.7 Asignaciones ASI

LEON implementa solo un número reducido de los ASI recomendadas por la arquitectura SPARCv8. En la siguiente tabla se muestra que ASI implementa LEON3:

ASI	Función
0x01	Fallo de caché forzado
0x02	Registros de control del sistema
0x08-0x09	No soportado
0x0A	Si mmuen es 1, el nivel de acceso lo determina el bit S del registro PSR. Si mmuen es 2, se accede como usuario.
0x0B	Si mmuen es 1, el nivel de acceso lo determina el bit S del registro PSR. Si mmuen es 2, se accede como supervisor.
0x0C	tag de caché de instrucciones
0x0D	datos de caché de instrucciones
0x0E	tag de caché de datos
0x0F	datos de caché de datos
0x10	*flush* de caché de instrucciones (y de datos si el sistema implementa MMU)
0x11	*flush* de caché de datos
0x13	Solo si hay MMU: *flush* de ambas cachés
0x18 - 0x03	*flush* de TLB y ambas cachés
0x19 - 0x04	registros de MMU
0x1C	bypass de MMU y cachés

Tabla 4 - Asignaciones ASI implementadas por LEON3

2.3 Simulador LeonViP

LeonViP es un simulador que está siendo actualmente desarrollado por el grupo de investigación Space Research Group de la Universidad de Alcalá con el objetivo de imitar el comportamiento de la implementación LEON de la arquitectura SPARCv8.

Éste está siendo programado usando principalmente el lenguaje C++. Además, se está empleando la plataforma GitHub como sistema de control de versiones para poder compartir los cambios realizados y dividir fácilmente su desarrollo entre las diferentes funcionalidades del simulador y las personas que están trabajando sobre él al mismo tiempo.

Los siguientes subapartados dan una breve descripción de los componentes principales del simulador y un resumen de su funcionamiento.

2.3.1 Componentes principales

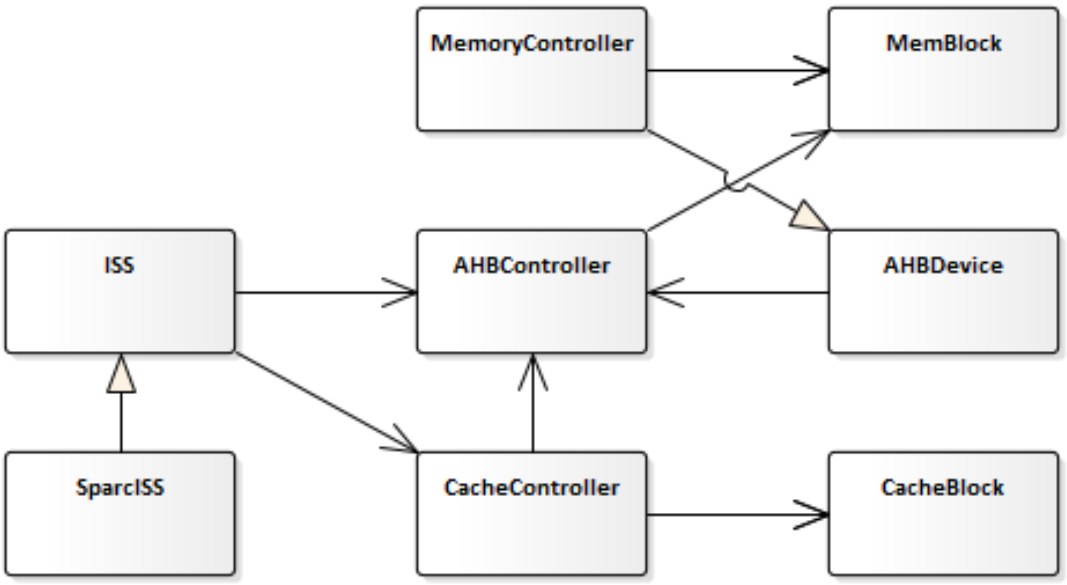


Ilustración 1 - Diagrama simplificado de componentes de LeonViP

· **Simulador del conjunto de instrucciones (*ISS*)**: representa la ISA a simular, intentando englobar cualquier conjunto de instrucciones que quiera implementarse sobre este simulador. En esta instancia, se implementa en la clase SparcISS que simula la ISA de SPARCv8.

· **Simulador del set de instrucciones de SPARCv8 (*SparcISS*)**: hereda de la ISS e imita el funcionamiento del procesador definido por la ISA de SPARCv8, simulando la decodificación y ejecución de instrucciones, registros y *traps*.

· **Controlador de memoria (*MemoryController*)**: simula los distintos bloques de la memoria del sistema (PROM, EEPROM, SDRAM...), representados mediante diferentes instancias de la clase *MemBlock*. Este controlador hereda de *AHBDevice* ya que se representa como uno de los varios elementos accesibles mediante el Bus.

· **Controlador de Caché (*CacheController*)**: simula la caché del sistema tal y como la define la especificación de LEON3. Usa instancias de la clase *CacheBlock* para los diferentes bloques de las cachés de datos e instrucciones. Necesita la conexión al *AHBController* para cuando requiera acceder a memoria.

· **Controlador de bus (*AHBController*)**: representa el bus por el que se intercambian los datos entre los distintos componentes del sistema. Cada elemento se representa con una instancia de la clase *AHBDevice*, entre los que se incluyen la memoria y dispositivos de E/S.

Además de las anteriores existen otros componentes para simular otros elementos del sistema como interrupciones, *timers*, dispositivos de entrada y salida, etc., que por simplicidad no se muestran en el diagrama, ya que no son necesarios para el fin de este proyecto.

2.3.2 Funcionamiento

Una vez creado un programa compilado para la ISA de SPARCv8, se le debe pasar el archivo binario generado al simulador y, opcionalmente, un archivo de configuración que fija las opciones dadas por la especificación LEON3 como por ejemplo el habilitado de cachés. En caso de no proveer de dicho archivo, el simulador usaría una configuración por defecto.

Al iniciar el simulador, este comienza instanciando todos los elementos que lo componen según la configuración provista (si la hay), y escribiendo las secciones definidas por el archivo binario en sus correspondientes bloques de memoria. A continuación, realiza un *reset* de la ISS y comienza la ejecución del programa llamando a la función que contiene el bucle principal para el decodificado y ejecución de instrucciones de la ISS implementada, en este caso SparcISS.

El bucle de la ISS va ejecutando las instrucciones, llamando a los distintos elementos como el *AHBController* o *CacheController* en caso de necesitarlo. Esta también se encarga de comprobar y tratar *traps*, y el movimiento de ventanas de registros.

El acceso al *CacheController* se hace a través de los métodos correspondientes. De forma general, primero comprueba los bloques para ver si el elemento buscado está en caché. En caso negativo, usa el *AHBController* para acceder a memoria.

Cuando el bucle de ejecución acaba, ya sea por fin del programa o por un error fatal del sistema, el simulador muestra información sobre la ejecución, como el tiempo empleado, e información sobre fallos si los hubo.

3. Desarrollo de la MMU

3.1 Concepto

La unidad de manejo de memoria tiene principalmente 2 fines: facilitar la asignación y manejo de segmentos de memoria para diferentes tareas y la protección de dichos segmentos para que solo sean accesibles bajo ciertas condiciones.

Para conseguir esto se introduce el concepto de virtualización de memoria, lo cual oculta la memoria real mediante un redireccionamiento usando lo que se denominan direcciones virtuales. En caso de usar MMU, cuando un proceso (o *contexto*, siguiendo la terminología empleada para la arquitectura SPARCv8) quiere acceder a memoria, la dirección se trata como virtual, la cual la MMU traduce a su correspondiente dirección real o física. Para realizar esta traducción se emplean una serie de tablas que almacenan la correspondencia entre la dirección virtual y contexto con su dirección física.

Esto de por sí ya permite hacer que un proceso solo se limite a ciertos rangos de memoria física, pero además cada uno de estos segmentos vienen marcados con unos bits que determinan el privilegio necesario para acceder a dichos datos, y el tipo de acceso que se permite.

3.2 Especificación

La MMU a simular se basa en la implementada por LEON, que a su vez se basa en la dada por referencia en la arquitectura de SPARCv8. Los siguientes subapartados definen en detalle la implementación dada por LEON junto a la configuración que decide fijarse por la implementación GR740 y que finalmente va a implementarse para el simulador LeonViP.

3.2.1 Resumen

La MMU a simular se sitúa entre la caché y la memoria y, en caso de estar habilitada, se solicita su uso cada vez que es necesario acceder a una dirección de memoria que no esté previamente en caché para que la traduzca a la dirección física. Para realizar las traducciones cuenta con un búfer de traducción anticipada (*translation lookaside buffe*r o TLB) y las tablas de páginas.

La TLB es una caché especial en la cual guarda las últimas traducciones que ha realizado indexadas por la dirección virtual y contexto. LEON permite dividirla en 2, una para datos y otra para instrucciones, además de poder elegir si habilitarlas o no y definir el número de entradas máximas que pueden contener.

Las tablas de páginas las crea el sistema operativo y están almacenadas en memoria física. Tras crear dichas tablas, el sistema operativo inicia la MMU. A la hora de traducir una dirección, la MMU primero comprueba la TLB y, si no encuentra la traducción, pasa a comprobar las tablas de

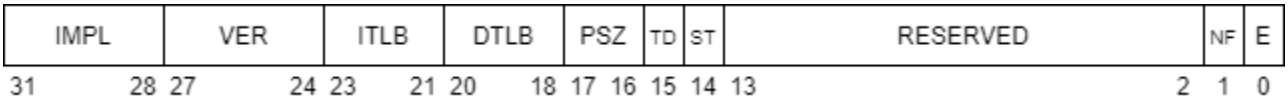
páginas. Si tras el recorrido de tablas no encuentra la traducción se lanza un *trap* para indicarlo. En caso de si encontrar la traducción, comprueba los permisos de dicha página y si algo no coincide envía el *trap* correspondiente.

3.2.2 Registros

La MMU cuenta con 5 registros que son accesibles mediante instrucciones *load* y *store* al ASI 0x19. En este apartado se definen en detalle dichos registros:

3.2.2.1 Registro de control de MMU

Define la configuración y ciertos aspectos de funcionamiento de la MMU. Se accede a través de la dirección 0 del ASI 0x19.



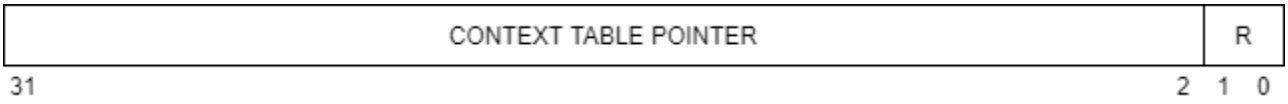
Campo	Bits	Descripción	R/W	Reset LEON3	Reset GR740
IMPL	31..28	ID de implementación de la MMU.	R	0	0
VER	27..24	ID de versión de la MMU.	R	1	0
ITLB	23..21	Número de entradas de la TLB de instrucciones. Se calcula como 2^{ITLB}. Si la TLB es compartida, este campo indica el número total de entradas.	R	*[a]	4
DTLB	20..18	Número de entradas de la TLB de datos. Se calcula como 2^{DTLB}. Si la TLB es compartida, este campo es 0.	R	*[a]	4
PSZ	17..16	Tamaño de la página más pequeña. 0 = 4 Kib; 1 = 8 Kib; 2 = 16 Kib; 3 = 32 Kib. En la implementación GR740 este campo no existe (reserved)	RW	0	0
TD	15	TLB *disable*. Si es 1, la TLB no se usa, haciendo que todos los accesos a memoria provoquen un recorrido de la tabla de páginas. En GR740 siempre debería estar habilitada.	RW	NR	NR
ST	14	*Separate* TLB. Si es 1, separa la TLB en 2, una para datos y otra para instrucciones.	R	*[a]	1
RES	13..2	Reservado, devuelve 0 en lectura.	R	0	0
NF	1	*No Fault*. Si es 0, cualquier fallo activará la señal de excepción correspondiente. Si es 1, solo fallos de área de instrucciones cuando se accede en modo supervisor producirán señal de excepción. Los registros de fallos se actualizan siempre, aunque no haya señal de excepción.	RW	0	0
E	0	MMU *enable*. Si es 0 no traduce direcciones por lo que se tratan como físicas.	RW	0	0

Tabla 5 - Registro de control de la MMU

*[a] Valores solo configurables antes del inicio del sistema.

3.2.2.2 Registro de puntero de contexto

Guarda la dirección de memoria que apunta a la primera entrada de la tabla de contexto. Se accede mediante la dirección 0x100 del ASI 0x19.

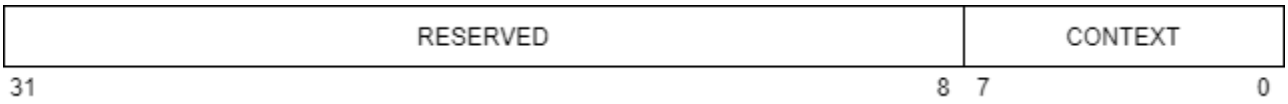


Campo	Bits	Descripción	R/W	Reset LEON3	Reset PLACA
CTP	31..2	Puntero a tabla de contexto, se compone de los bits 36...6 de la dirección física que apunta a la tabla de nivel 0. Nótese que la dirección está desplazada 4 bits.	RW	NR	NR
RES	1..0	Reservado, devuelve 0 en lectura.	R	0	0

Tabla 6 - Registro de puntero de contexto

3.2.2.3 Registro de contexto

Guarda el numero de contexto actual. Se accede mediante la dirección 0x200 del ASI 0x19.

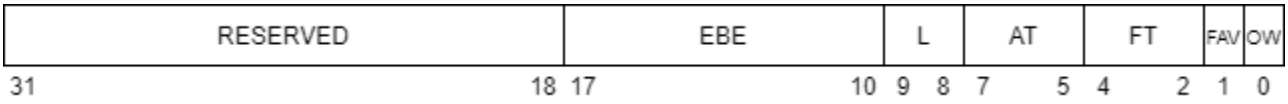


Campo	Bits	Descripción	R/W	Reset LEON3	Reset PLACA
RES	31..8	Reservado	R	0	0
CTX	7..0	Contexto actual, indica el número de entrada de la tabla de contexto al realizar una traducción.	RW	0	0

Tabla 7 - registro de contexto

3.2.2.4 Registro de estado de fallo

Contiene el tipo de fallo en caso de haberlo. Se accede mediante la dirección 0x300 del ASI 0x19.



Campo	Bits	Descripción	R/W	Reset LEON3	Reset PLACA
RES	31..18	Reservado.	R	0	0
EBE	17..10	*External bus error*. Nuca se usa en LEON.	R	0	0
L	9..8	Nivel de la entrada de página que causó el error.	R	0	0
AT	7..5	*Access Type*. Tipo de acceso cuando ocurrió el error. Mirar *Tabla 9 - Access Type*.	R	0	0
FT	4..2	*Fault Type*. Tipo de fallo. Mirar *Tabla 10 - Fault Type*.	R	0	0
FAV	1	*Fault address valid*. Se pone a 1 siempre que hay fallo, se limpia al leerse este registro.	R	0	0
OW	0	*Overwrite*. Indica a 1 si se sobrescribió un fallo de igual prioridad. Mirar apartado *3.2.3 Excepciones*.	R	0	0

Tabla 8 - MMU fault status register

AT	**Tipo de Acceso**		
	R/W	**Datos/Inst**	**User/Supervisor**
0	Lectura	Datos	modo Usuario
1	Lectura	Datos	modo Supervisor
2	Lectura	Instrucciones	modo Usuario
3	Lectura	Instrucciones	modo Supervisor
4	Escritura	Datos	modo Usuario
5	Escritura	Datos	modo Supervisor
6	Escritura	Instrucciones	modo Usuario
7	Escritura	Instrucciones	modo Supervisor

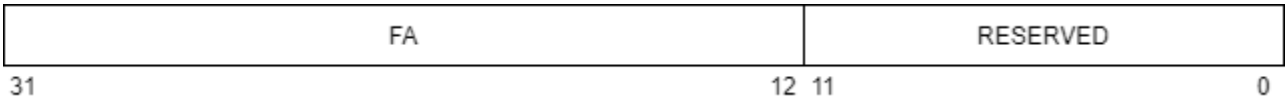
Tabla 9 - Access Type

FT	Orden de comprobación	Fallo	Descripción
0	-	*None*	No hay error.
1	3	*Invalid Address*	La entrada de página encontrada tenía marcado el bit de inválido.
2	5	*Protection Error*	Usuario intentó acceder a una página marcada como supervisor.
3	4	*Privilege violation*	La página encontrada no tiene los permisos de acceso pedidos.
4	2	*Translation Error*	Error al realizar el recorrido de tablas, ya sea por error al acceder a memoria, encontrar una entrada marcada como reservada, o encontrar una entrada descriptora de página en una tabla de nivel 3.
5	6	*Access Bus Error*	Error de acceso a bus, no sucede en LEON.
6	1	*Internal Error*	No sucede en LEON.
7	-	*Reserved*	Sin uso.

Tabla 10 - Fault Type

3.2.2.5 Registro de dirección de fallo

Contiene la dirección virtual que provocó fallo en caso de haberlo. Se accede mediante la dirección 0x400 del ASI 0x19.



Campo	Bits	Descripción	R/W	Reset LEON3	Reset PLACA
FA	31..12	*Fault Address*. Indica los bits más altos de la dirección virtual que provocó el fallo descrito en el *fault status register*.	R	NR	NR
RES	11..0	Reservado	R	0	0

Tabla 11 - MMU fault address register

3.2.3 Excepciones

Cada vez que se produce un fallo, se actualizan los registros de estado de fallo (*fault status register* o FSR) y de dirección de fallo (*fault address register* o FAR). Debido a que el procesador es segmentado (*pipelined*), existe la posibilidad de que ocurra más de un fallo antes de atenderse. Para solucionar esto, se dividen los fallos en 3 clases que determinan la prioridad del fallo y, por tanto, qué fallo mostrar al procesador cuando llegue el momento de atenderlo. Estas 3 clases son: fallos de traducción, fallos por acceso a área de datos y fallos por acceso a área de instrucciones.

Cualquier error pone el campo de dirección válida de fallo (*Fault Address Valid* o FAV) del registro FSR a 1. Una lectura del FSR limpia el bit de FAV. Si ocurre un fallo mientras FAV es 1, significa que el último fallo ocurrido en la MMU aún no se ha tratado, lo que hace que se tenga que comprobar las prioridades de clase para determinar si mantener el último fallo o escribir el nuevo en los registros.

Clase de Fallo		Prioridad de clase	Descripción
Tipo	Subtipo		
Fallo de Traducción	Por error de acceso a memoria	4	Fallo de traducción producido por un error de memoria mientras se realizaba un recorrido de tablas de página. Sobrescribe cualquier otro tipo de fallo, en cuyo caso pone OW a 0.
	Resto de causas.	3	Fallo de traducción por encontrar una entrada marcada como reservada o entrada descriptora en tabla de nivel 3 durante un recorrido de tablas de página. Sobrescribe cualquier fallo de igual o menor prioridad de clase, en cuyo caso pone OW a 0.
Resto de Fallos	Durante acceso a área de datos.	2	Fallo que no es de traducción, causado durante un acceso a un área de datos. Sobrescribe cualquier fallo de igual o menor prioridad de clase, en cuyo caso pone OW a 0.
	Durante acceso a área de Instrucciones.	1	Fallo que no es de traducción, causado durante a un acceso a un área de instrucciones. Sobrescribe cualquier fallo de igual prioridad de clase, en cuyo caso **pone OW a 1.**

Tabla 12 - Clases de Fallos

La MMU usa 2 *traps* para indicar que hubo una excepción, y se diferencian según el área en el que ocurrió el fallo. En caso de fallo por acceso a área de instrucciones, se envía la señal de *trap* 0x1, y en caso de ser por acceso a área de datos se envía la señal de *trap* 0x9.

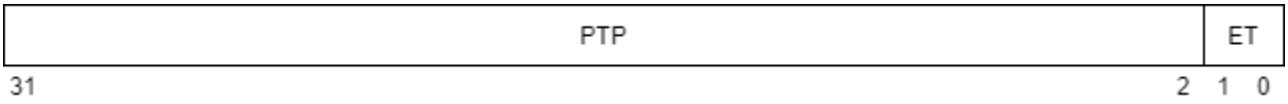
3.2.4 Tablas de páginas

El contenido y formato de las tablas de páginas se define en la especificación de la arquitectura de referencia de la MMU dada por SPARCv8. La implementación de LEON no entra en detalles sobre como implementa dichas tablas, pero las continuas pruebas realizadas a dicha implementación confirman que usan las mismas tablas que las que se especifican por SPARCv8.

Se definen 4 niveles de tablas cuyas entradas pueden contener o bien un descriptor de tabla de página (*page table descryptor* o PTD) o una entrada de tabla de página (*page table entry* o PTE). El tipo de entrada se identifica usando el campo ET (*entry type*) que corresponde con los 2 últimos bits de cada entrada.

3.2.4.1 Descriptor de tabla de página (PTD)

Apunta a la base de una tabla de página de siguiente nivel, la cual se indexa usando parte de la dirección virtual que depende del nivel actual al que se encuentra este.

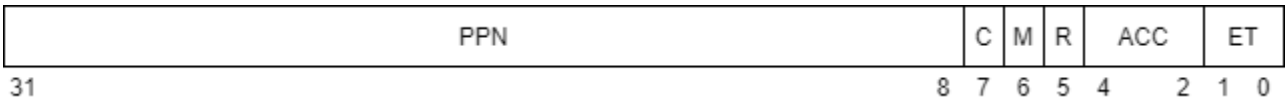


Campo	Bits	Descripción
PTP	31..2	*Page table pointer*. Constituye los bits 35 a 6 de la dirección física que apunta a la base de la tabla de siguiente nivel. (Notar que esto supone un desplazamiento de 4 bits)
ET	1..0	*Entry type*. Identifica el tipo de entrada, para el caso de un PTD debe ser 1.

Tabla 13 - Descriptor de página

3.2.4.2 Entrada de tabla de página (PTE)

Contiene la dirección física de la página buscada además de varios bits que definen ciertas propiedades y permisos de acceso de la página.



Campo	Bits	Descripción
PPN	31..8	*Physical page Number*. Constituye los 24 bits más altos de la dirección física de 36 bits de la página, es decir, forman los bits 35 a 12 de la dirección física.
C	7	*Cacheable*. Si es 1, indica que esta pagina es cacheable, es decir, si puede almacenarse en la caché.
M	6	*Modified*. Se pone a 1 cuando se accede una página para escribirla.
R	5	*Referenced*. Se pone a 1 cuando se accede a una página.
ACC	4..2	*Access Permissions*. Indican los permisos y privilegios que tiene la página. Mirar *Tabla 15 - Campo ACC* - Permisos de acceso.
ET	1..0	*Entry type*. Identifica el tipo de entrada, para el caso de un PTE debe ser 2. Mirar *Tabla 16 - Campo ET* - Tipo de entrada

Tabla 14 - Entrada de página

ACC	Acceso Permitido	
	en modo Usuario	en modo Supervisor
0	Lectura	Lectura
1	Lectura/Escritura	Lectura/Escritura
2	Lectura/Ejecución	Lectura/Ejecución
3	Lectura/Escritura/Ejecución	Lectura/Escritura/Ejecución
4	Ejecución	Ejecución
5	Lectura	Lectura/Escritura
6	Sin acceso (Página de supervisor)	Lectura/Ejecución
7	Sin acceso (Página de supervisor)	Lectura/Escritura/Ejecución

Tabla 15 - Campo ACC - Permisos de acceso

ET	Tipo de Entrada
0	Entrada inválida
1	Descriptor de tabla de página (PTD)
2	Entrada de página (PTE)
3	Reservado

Tabla 16 - Campo ET - Tipo de entrada

SPARCv8 define un sistema de 4 niveles de tablas para la asignación de páginas de memoria que luego usa la MMU para realizar la traducción de direcciones virtuales a físicas. El nivel 0 corresponde con la tabla de contexto y es donde se inicia la traducción usando el puntero de contexto y el contexto actual. Cada vez que encuentra un PTD, avanza a una tabla de siguiente nivel, hasta encontrar un PTE que indica la dirección física buscada.

El tamaño de página depende del nivel de tabla en el que se ponga el PTE, además de como se haya configurado el campo de tamaño de página (*Page Size* o PSZ) del registro de control de la MMU. Como se usa la dirección virtual para indexar los diferentes niveles de tabla y el offset final de la dirección física, dicha dirección virtual también cambia según el campo PSZ.

En la siguiente tabla se muestra como cambia el tamaño de página, número de entradas por tabla y formato de la dirección virtual en función del campo PSZ y el nivel en el que se encuentre el PTE. Se puede encontrar más información sobre el proceso de traducción y dirección virtual en el apartado *3.2.6.3 Recorrido de tablas de página (table walk)*.

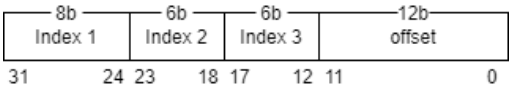
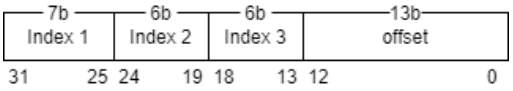
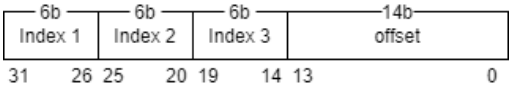
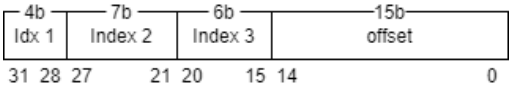
PSZ	Nivel	Tamaño de página	Tamaño por entrada (HEX)	Offset bits	Entradas por tabla	Dirección Virtual
0	3	4 KiB	0x0 0000 1000	12	64	
	2	256 KiB	0x0 0004 0000	18	64	
	1	16 MiB	0x0 0100 0000	24	256	
	0	4 GiB	0x1 0000 0000	32	256	
1	3	8 KiB	0x0 0000 2000	13	64	
	2	512 KiB	0x0 0008 0000	19	64	
	1	32 MiB	0x0 0200 0000	25	128	
	0	4 GiB	0x1 0000 0000	32	256	
2	3	16 KiB	0x0 0000 4000	14	64	
	2	1 MiB	0x0 0010 0000	20	64	
	1	64 MiB	0x0 0400 0000	26	64	
	0	4 GiB	0x1 0000 0000	32	256	
3	3	32 KiB	0x0 0000 8000	15	64	
	2	2 MiB	0x0 0020 0000	21	128	
	1	256 MiB	0x0 1000 0000	28	16	
	0	4 GiB	0x1 0000 0000	32	256	

Tabla 17 - Tamaños de páginas y Offsets según nivel de tabla y PSZ

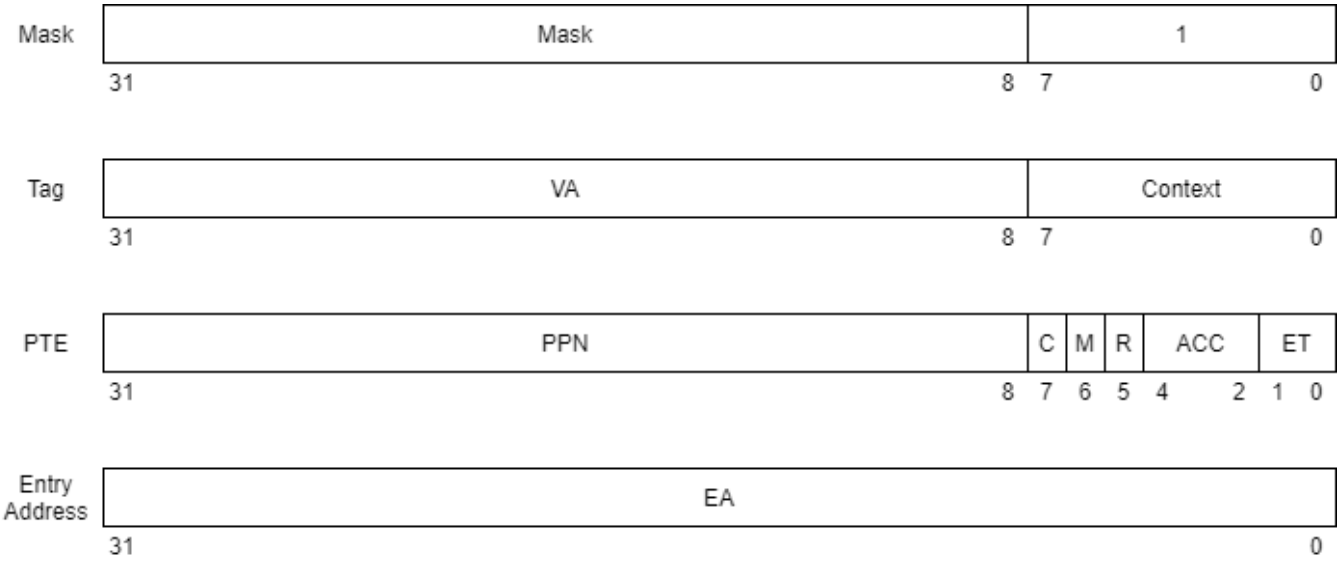
El campo PSZ es una opción de configuración dada solo por la implementación de LEON que no tiene equivalente en la especificación de SPARCv8, siendo la de referencia como la indicada por PSZ igual a 0. La implementación GR740 elimina PSZ, fijándolo a 0.

3.2.5 TLB

La TLB es una caché que usa la MMU para guardar las traducciones más recientes, acelerando el proceso de traducción de direcciones.

Las entradas y funcionamiento de TLB están muy vagamente descritas en la documentación de SPARCv8, y apenas se mencionan en la implementación de LEON, por lo que **la descripción dada en este apartado se refiere a cómo se ha decidido implementar dicho sistema de TLB para el simulador LeonViP**, el cual puede diferir de como se hace realmente en LEON, pero cuyo resultado termina siendo el mismo.

Las entradas de TLB definidas para LeonViP cuentan con 4 elementos: una máscara que determina los bits de offset, el tag que identifica las entradas, compuesto por la dirección virtual y contexto, el PTE que corresponde con dicho tag y finalmente la dirección física donde se encuentra dicho PTE y se usa en caso de necesitar modificar los bits de referencia y modificado.



3.2.5.1 Máscara

Indica el tamaño de página guardado en esta entrada, lo cual sirve tanto para conocer que bits de la dirección virtual comparar como cuantos bits usar de offset. Indica con '1' que bits se comparan con el tag y mantiene los últimos 8 bits a '1' para la comparación con el contexto.

PSZ	Nivel	Tamaño de página	Máscara
0	3	4 KiB	0xFFFF F0FF
	2	256 KiB	0xFFFC 00FF
	1	16 MiB	0xFF00 00FF
	0	4 GiB	0x0000 00FF
1	3	8 KiB	0xFFFF E0FF
	2	512 KiB	0xFFF8 00FF
	1	32 MiB	0xFE0000FF
	0	4 GiB	0x000000FF
2	3	16 KiB	0xFFFFC0FF
	2	1 MiB	0xFFF000FF
	1	64 MiB	0xFC0000FF
	0	4 GiB	0x000000FF
3	3	32 KiB	0xFFFF80FF
	2	2 MiB	0xFFE000FF
	1	256 MiB	0xF00000FF
	0	4 GiB	0x000000FF

Tabla 18 - Máscara de TLB según PSZ y nivel de Tabla

3.2.5.2 Tag

Es la parte que se compara al buscar la entrada en TLB.

Campo	Bits	Descripción
VA	31..8	Corresponde con los bits 31 a 8 de la dirección virtual que, junto al contexto, se traduce a la entrada de página guardada. Antes de asignarla, se limpian los bits de offset (por ejemplo, si es de una página de 4KiB, se pone a 0 los últimos 12 bits, y finalmente se guardan los bits 31 a 8 en el campo VA)
Context	7	Guarda el contexto que, junto a la VA, se traduce a la entrada de página guardada.

Tabla 19 - TLB Tag

3.2.5.3 PTE

Contiene la entrada de página que traduce la dirección virtual y coincide con la que se encuentra en la correspondiente tabla de páginas. Su descripción detallada puede verse en el apartado *3.2.4.2 Entrada de tabla de página (PTE)*.

3.2.5.4 Dirección de entrada (entry address – EA)

Guarda la dirección física donde se encuentra el PTE, se usa para cuando es necesario cambiar los bits de referencia y modificado en la entrada de tabla de página guardada en memoria.

3.2.6 Proceso de traducción de direcciones

Si la MMU está habilitada, todas las direcciones que se usen para accesos a memoria se tratan como direcciones virtuales, lo que hace que cada vez que se produce un acceso que no este en caché se pida a la MMU que realice la traducción de la dirección virtual a su correspondiente dirección física.

3.2.6.1 Dirección virtual (VA)

Son direcciones que asigna el software a los diferentes contextos que estén ejecutándose en el sistema y deben tener una traducción física en las tablas creadas por ese mismo software. Tal y como se vio en el apartado *3.2.4 Tablas de páginas*, la dirección virtual difiere ligeramente según el campo PSZ del registro de control de la MMU. Aquí se muestra la dirección virtual para PSZ igual a 0, es decir, de tamaño mínimo de página de 4KiB, cuya única diferencia con el resto es el tamaño de los campos.

Campo	Bits	Descripción
Index 1	31..24	En caso de PTE de nivel 0, forma parte del offset. En caso de PTD de nivel 0, indica el número de entrada de la tabla de nivel 1 apuntada por el PTD.
Index 2	23..18	En caso de PTE de nivel 1 o menor, forma parte del offset. En caso de PTD de nivel 1, indica el número de entrada de la tabla de nivel 2 apuntada por el PTD.
Index 3	17..12	En caso de PTE de nivel 2 o menor, forma parte del offset. En caso de PTD de nivel 2, indica el número de entrada de la tabla de nivel 3 apuntada por el PTD.
Offset	11..0	Offset del PPN del PTE encontrado.

Tabla 20 - Dirección virtual (PSZ = 0)

Es decir, cada índice indica el número de entrada de la tabla de dicho nivel, y en caso de ser PTE, ese índice y siguientes forman parte del offset de página.

3.2.6.2 Búsqueda en TLB

Al iniciar el proceso de traducción primero se comprueba si la TLB está habilitada y, en caso de estarlo, chequea si se usa una TLB compartida o separada. Si hay TLB, crea un tag usando la dirección virtual y contexto actual que compara con las entradas guardadas en el orden en el que estén almacenadas, usando para dicha comparación la máscara y tag de cada entrada, hasta encontrarla o llegar al fin de la TLB. El siguiente pseudocódigo muestra un ejemplo de como sería el proceso descrito.

```
1    // va = dirección virtual, ctx = contexto, inst = si es Instrucción
2    // TLB = lista doblemente enlazada, TLB.head = primera entrada de TLB
3    // entry.mask, entry.tag, entry.pte, entry.next -> elementos de la entrada de TLB
4
5    acierto = 0;
6    if(!MMUControl.TD){     // Si la TLB está habilitada
7       if(MMUControl.ST){   // y es separada
8          if (inst){        // Si es acceso a instrucción
9             TLB = instructionTLB;       // Usa TLB de instrucciones
10         else{
11            TLB = dataTLB;              // Si no, usa TLB de datos
12         }
13      else{
14         TLB = sharedTLB;  //Si no es separada, usa la única TLB (shared)
15      }
16
17      tag = (va & offsetMask) | ctx; // offsetMask es 0xFFFFF000 << MMUControl.PSZ
18      entry = TLB.head;    // Comienza con la primera entrada de la TLB
19      while(entry){        // Mientras haya entradas
20         if(entry.tag == (tag & entry.mask)){  // Compara tag con el traido AND mascara.
21            acierto = 1;      // Si coinciden, lo indica con acierto a 1.
22            pte = entry.pte;  // obtiene PTE
23            mask = not(entry.mask) | 0xFF; // Y la mascara para saber que offset darle
24            TLB.moveToHead(entry); // Función de TLB que mueve esta entrada al comienzo
25            break; // y finaliza la búsqueda
26         }
27         entry = entry.next; // Si no coincide, obtiene la entrada siguiente, será 0
28                             si no hay más entradas.
29      }
30   }
```

Código 1 - Pseudocódigo de búsqueda de entradas en TLB

3.2.6.3 Recorrido de tablas de página (table walk)

Cuando o bien la TLB está deshabilitada o no se encuentra la traducción en ella, la MMU realiza un recorrido de las tablas de páginas partiendo de la denominada tabla de contexto o tabla de nivel 0.

Usando el puntero de tabla de contexto y el contexto actual, ambos obtenidos de los registros de MMU correspondientes, obtiene una entrada de la tabla de contexto a partir de la cual comienza a recorrer las tablas. En caso de encontrar una PTD usa la dirección guardada en ella más el índice correspondiente de la dirección virtual, que depende del nivel actual de la tabla, para obtener la entrada de tabla de siguiente nivel. En caso de encontrar una PTE, crea una nueva entrada en TLB

con esta nueva traducción, comprueba los permisos y finaliza obteniendo la dirección física usando la dirección dada por el PTE y el offset de la dirección virtual que, como antes, depende del nivel de la tabla. El pseudocódigo siguiente muestra el proceso descrito, y en *Ilustración 2 - Recorrido de tablas de página (PSZ = 0)* se muestra un diagrama de como sería el recorrido de tablas.

```
1    // va = dirección virtual, ctx = contexto, inst = es Instrucción, write = es escritura
2
3    if(!acierto){
4       eAdd = (PunteroTablaContexto << 4) | (ctx << 2);// Dirección de la entrada de tabla
5       entry = M(eAdd);  // lee la dirección eAdd de memoria, obtiene entrada de contexto
6       nivel = 0;        // nivel actual de tabla
7       while( ((entry & not ETMask) == 1 ) && (nivel < 3)){ // Si PTD y nivel menor que 3
8          nivel++; // incrementa nivel
9          // indexMask indica los bits del índice según el nivel y PSZ.
10         // indexDesp indica cuanto desplazarlo.
11         eAdd = ((entry & PTPMask) << 4) | ((va & indexMask) >> (indexDesp)).
12         entry = M(eAdd);
13      }
14
15      if((entry & not ETMask) != 2){ // Si no se encontró PTE
16         if((entry & not ETMask) == 0) { excepción de entrada Inválida } //invalida
17         else { excepción de Traducción } // Si no, es fallo de traducción
18      } // En caso de excepción debe terminar el proceso de traducción.
19      else{
20         auxEntry = entry; // Guarda temporalmente la entrada encontrada
21         entry.R = 1; // Modifica bit de referencia
22         if (write) { entry.M = 1 }            // Y bit de Modificado si es escritura
23         if(auxEntry != entry){ // Y si cambió la entrada, guarda la nueva en memoria
24            M(eAdd) = entry;
25         }
26
27         // Guarda la nueva entrada en TLB según si está habilitada y separada
28         if(!MMUControl.TD){     // Si la TLB está habilitada
29            tag = (va & not offsetMask) | ctx; // offsetMask depende de nivel de PTE y PSZ
30            if(MMUControl.ST){   // y es separada
31               if (inst){        // Si es acceso a instrucción, la guarda en TLB de inst
32                  instructionTLB.addEntry(entry, eAdd, (not offsetMask | 0xFF), tag);
33               else{             // Si no, la guarda en TLB de datos
34                  dataTLB.addEntry(entry, eAdd, (not offsetMask | 0xFF), tag);
35               }
36            else{    // Si no es separada, la guarda en la TLB compartida.
37               sharedTLB.addEntry(entry, eAdd, (not offsetMask | 0xFF), tag);
38         }
39      }
40   }
```

Código 2 - Pseudocódigo de recorrido de tablas de página

ET	Type of Entry
00	Invalid
01	Page Table Descriptor
10	Page Table Entry
11	Reserved

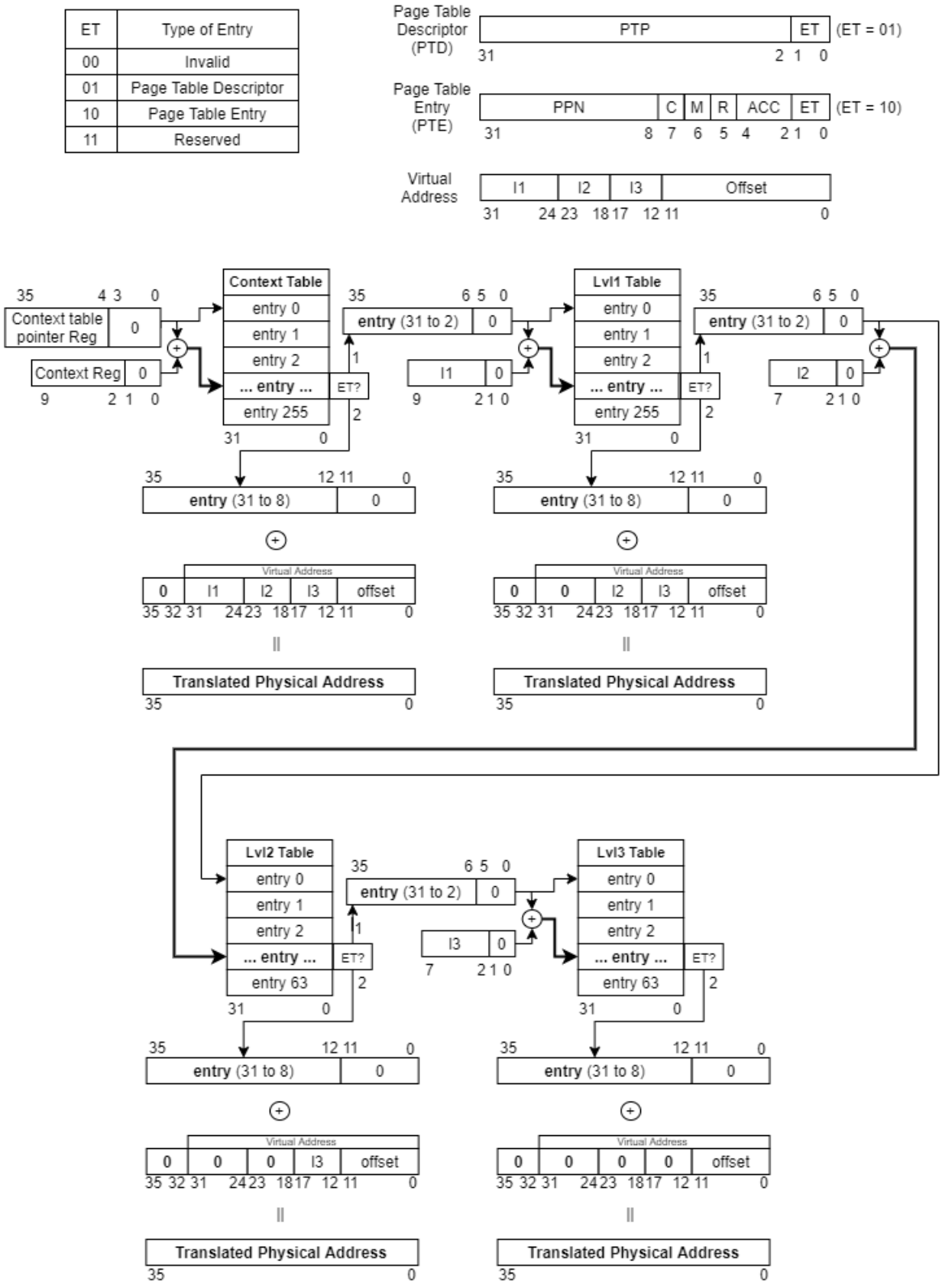


Ilustración 2 - Recorrido de tablas de página (PSZ = 0)

3.2.6.4 Comprobación de permisos

Una vez se obtiene un PTE válido se comprueban los bits de permisos de acceso a la página que se almacenan en el campo ACC del PTE y cuya descripción se puede ver en *Tabla 15 - Campo ACC* - Permisos de acceso. El orden de comprobación puede verse en el apartado *3.2.3 Excepciones* de la MMU. A continuación se adjunta un pseudocódigo que muestra un ejemplo de como se realizaría esta comprobación, y, para facilitar la comprensión de todo el proceso de traducción, se crean los diagramas *Ilustración 3 - Diagrama de flujo del proceso de traducción parte 1*, *Ilustración 4 - Diagrama de flujo del proceso de traducción parte 2* e *Ilustración 5 - Diagrama de flujo de excepciones*.

```
1    const bool accM[2][8][3]{ // accMatriz [privilegio(us/su)][acc(0-7)][tipoAcceso(r/w/x)]
2      // r, w, x,    acc
3        {1, 0, 0, //  0  Tabla de usuario
4         1, 1, 0, //  1
5         1, 0, 1, //  2
6         1, 1, 1, //  3
7         0, 0, 1, //  4
8         1, 0, 0, //  5
9         0, 0, 0, //  6
10        0, 0, 0}, // 7
11
12       {1, 0, 0, //  0  Tabla de supervisor
13        1, 1, 0, //  1
14        1, 0, 1, //  2
15        1, 1, 1, //  3
16        0, 0, 1, //  4
17        1, 1, 0, //  5
18        1, 0, 1, //  6
19        1, 1, 1} //  7
20   };
21
22   // pte = pte válido encontrado tras buscarlo en TLB o table walk
23   // su = modo supervisor, tipoAc = 0(read), 1(write), 2(execute)
24   // read, write, execute = tipo de acceso
25
26   if(read){         // Si el tipo de acceso es read
27      tipoAc = 0;    // tipoAc es 0
28   }
29   else if(write){   // Si no, si es escritura
30      tipoAc = 1;    // tipoAc será 1
31   else{             // Si no
32      tipoAc = 2;    // es ejecución
33   }
34
35   // Mira la matriz usando el modo de privilegio, acc del pte y tipo de acceso tipoAc
36   if(!accM[su][pte.acc][tipoAc]){  // y si es 0 no hay permisos/privilegios
37      if(!su && (acc >= 6){         // si es página de supervisor
38         excepción de Privilegios   // lanza excepción de privilegios
39      }
40      else {                        // Si no
41         excepción de Proteccion    // Excepción de protección
42      }
43   }
```

Código 3 - Pseudocódigo de comprobación de permisos

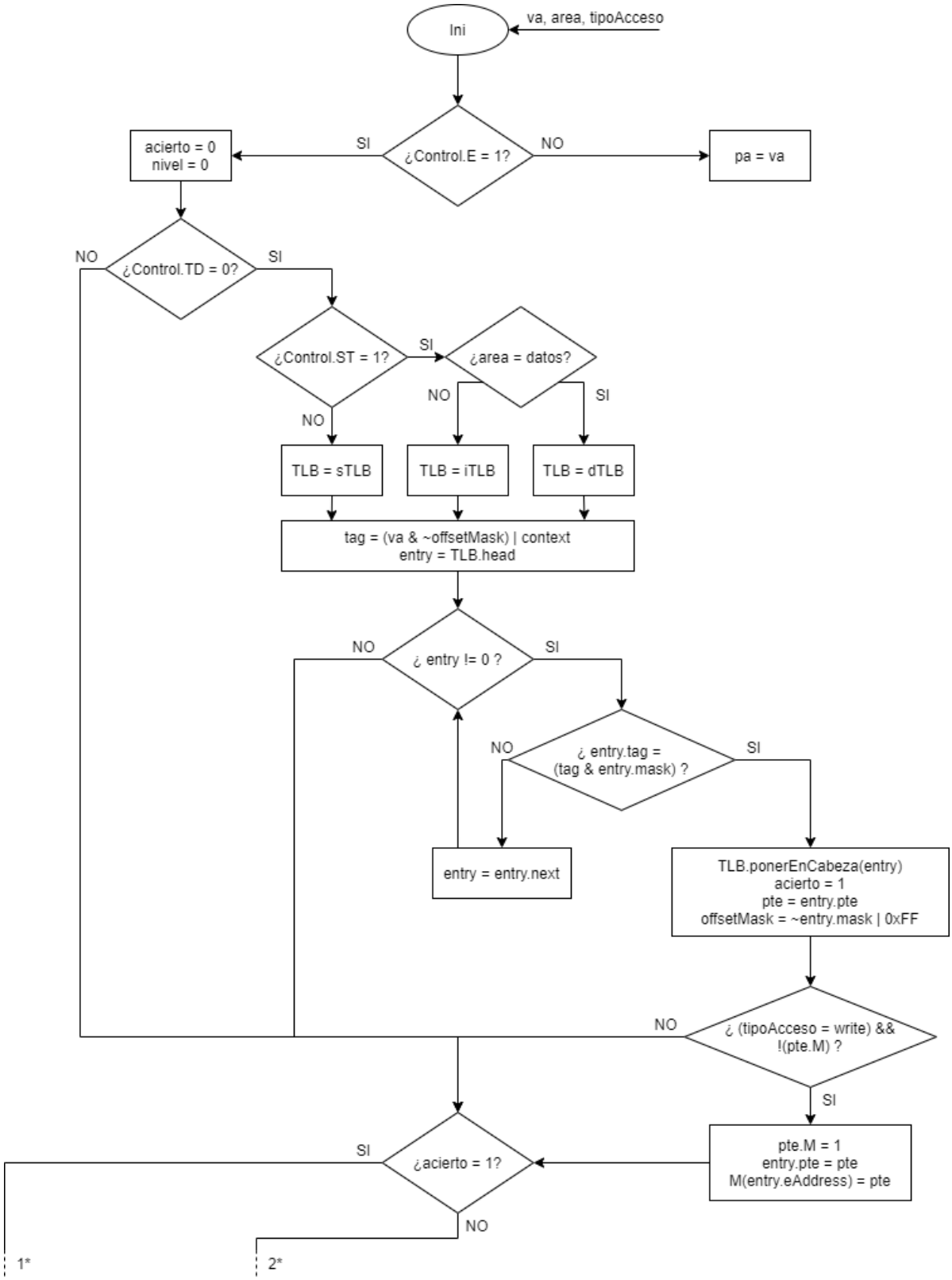


Ilustración 3 - Diagrama de flujo del proceso de traducción parte 1

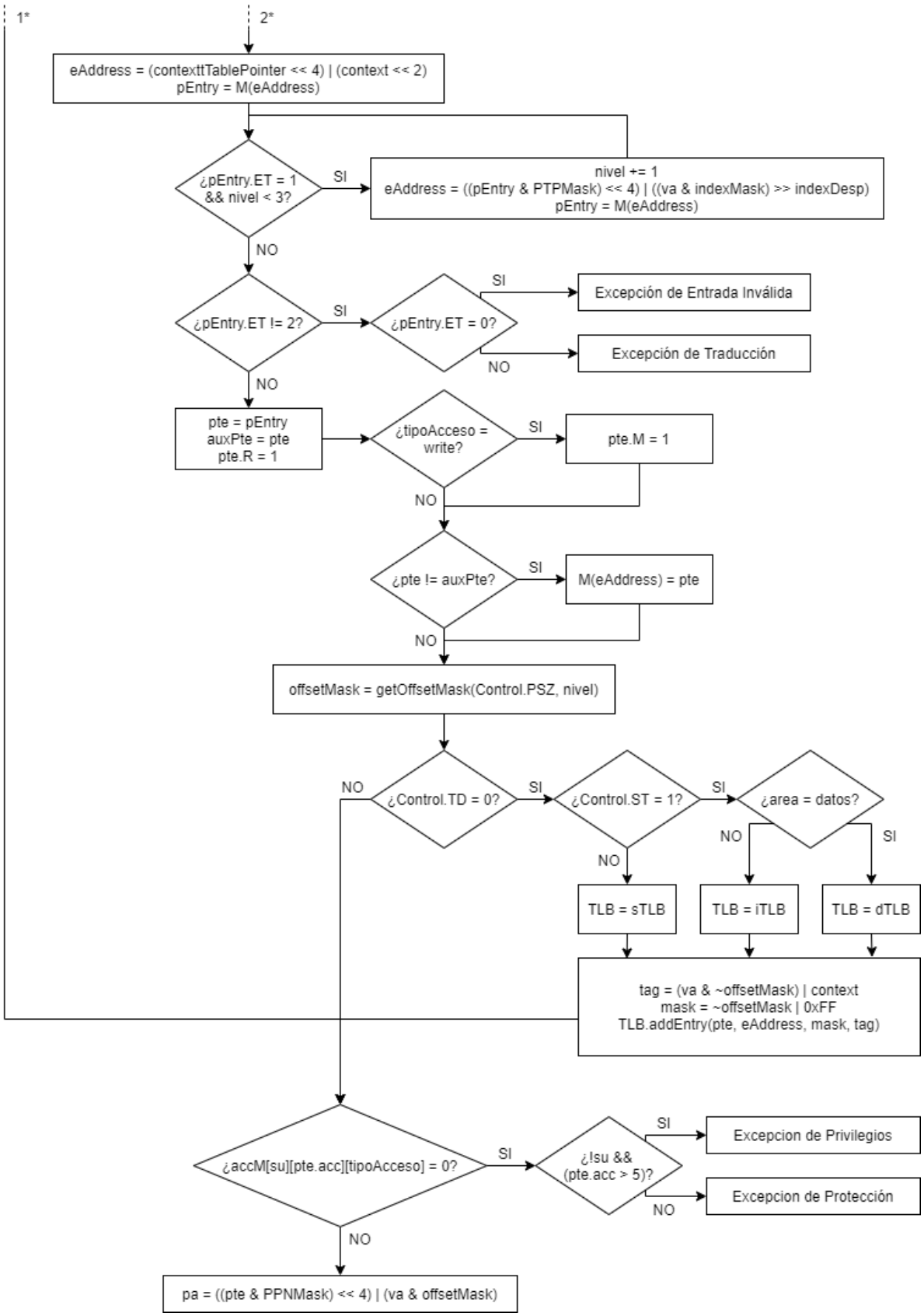


Ilustración 4 - Diagrama de flujo del proceso de traducción parte 2

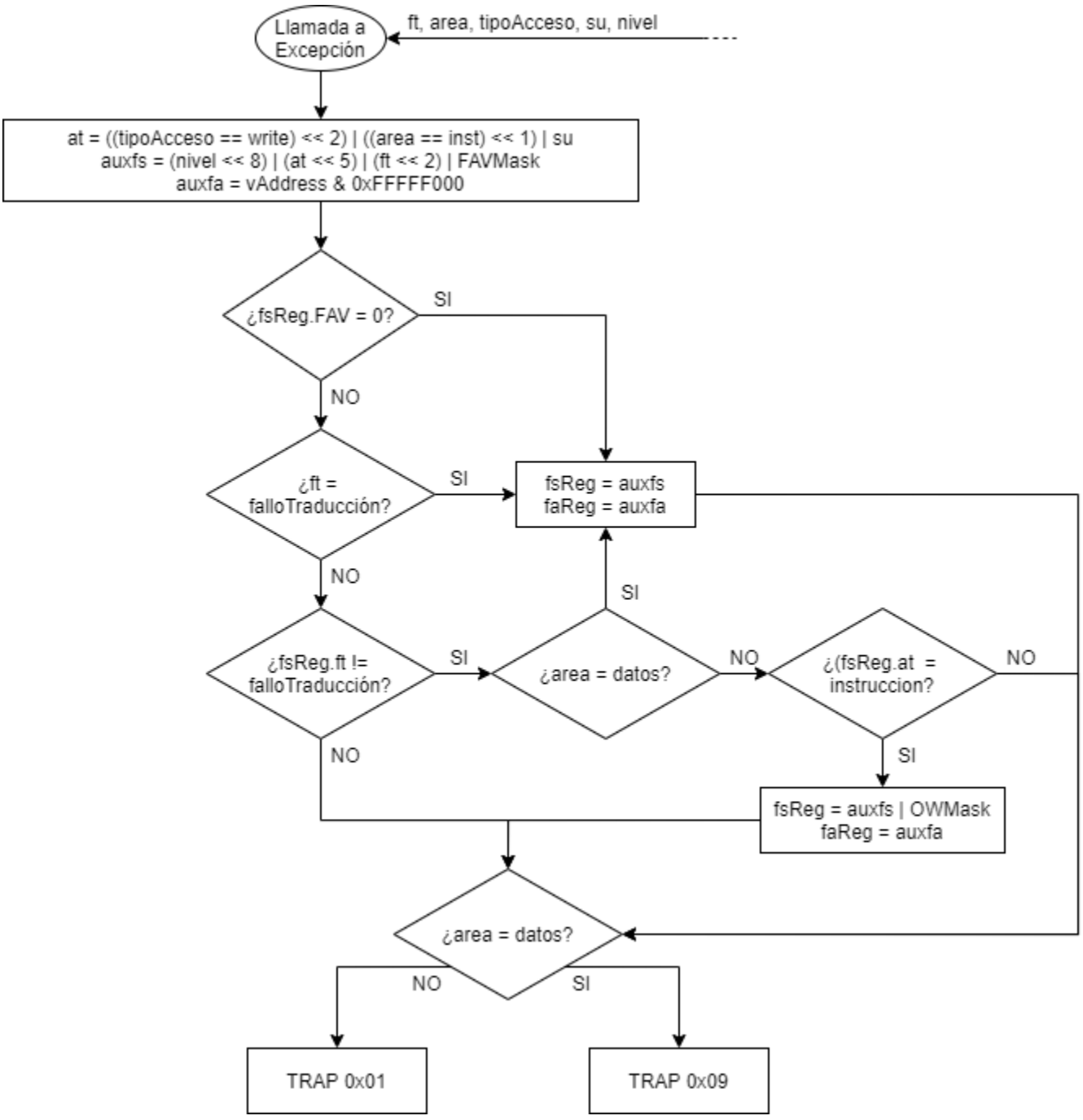


Ilustración 5 - Diagrama de flujo de excepciones

3.3 Implementación en LeonViP

Para poder implementar esta nueva funcionalidad en el simulador, ha sido necesario crear nuevas clases que faciliten la división ya existente en el programa para cada una de las funcionalidades implementadas, y en estas programar el comportamiento descrito en el apartado anterior.

En GitHub se ha creado una nueva rama específica para la implementación de esa nueva unidad, y sobre la cual se añaden las modificaciones descritas en los siguientes subapartados.

3.3.1 Componentes

Se añade 2 clases al simulador LeonViP, una para el controlador de la MMU que denomino *MMUController*, y otra para su TLB que denomino *MMUCache*, como se muestra a continuación:

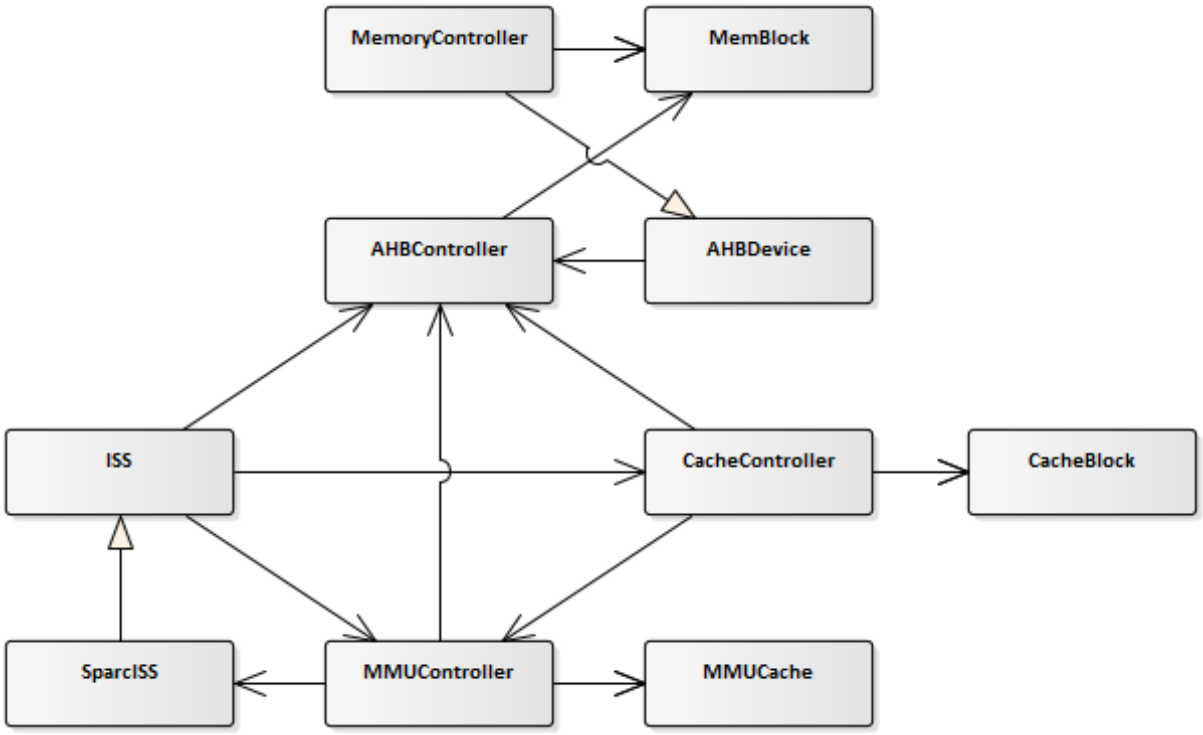


Ilustración 6 - Diagrama simplificado de componentes de LeonViP con MMU

Los elementos mostrados en este nuevo diagrama que coinciden con los definidos en el apartado *2.3 Simulador LeonViP* siguen teniendo las mismas funciones, excepto que ahora se añade la unidad de manejo de memoria.

El controlador de la MMU se conecta con el controlador de Caché para cuando esta necesita una traducción. La conexión con el controlador de AHB es para cuando necesita acceder a memoria por ejemplo para un recorrido de tablas. Además, cuenta con acceso directo a la ISS de SPARC ya

que necesita comprobar el registro PSR para conocer el modo de acceso. Finalmente, se conecta con la clase *MMUCache* para cuando necesite acceder a la TLB.

En el siguiente diagrama se muestra mas detalladamente el contenido de las nuevas clases.

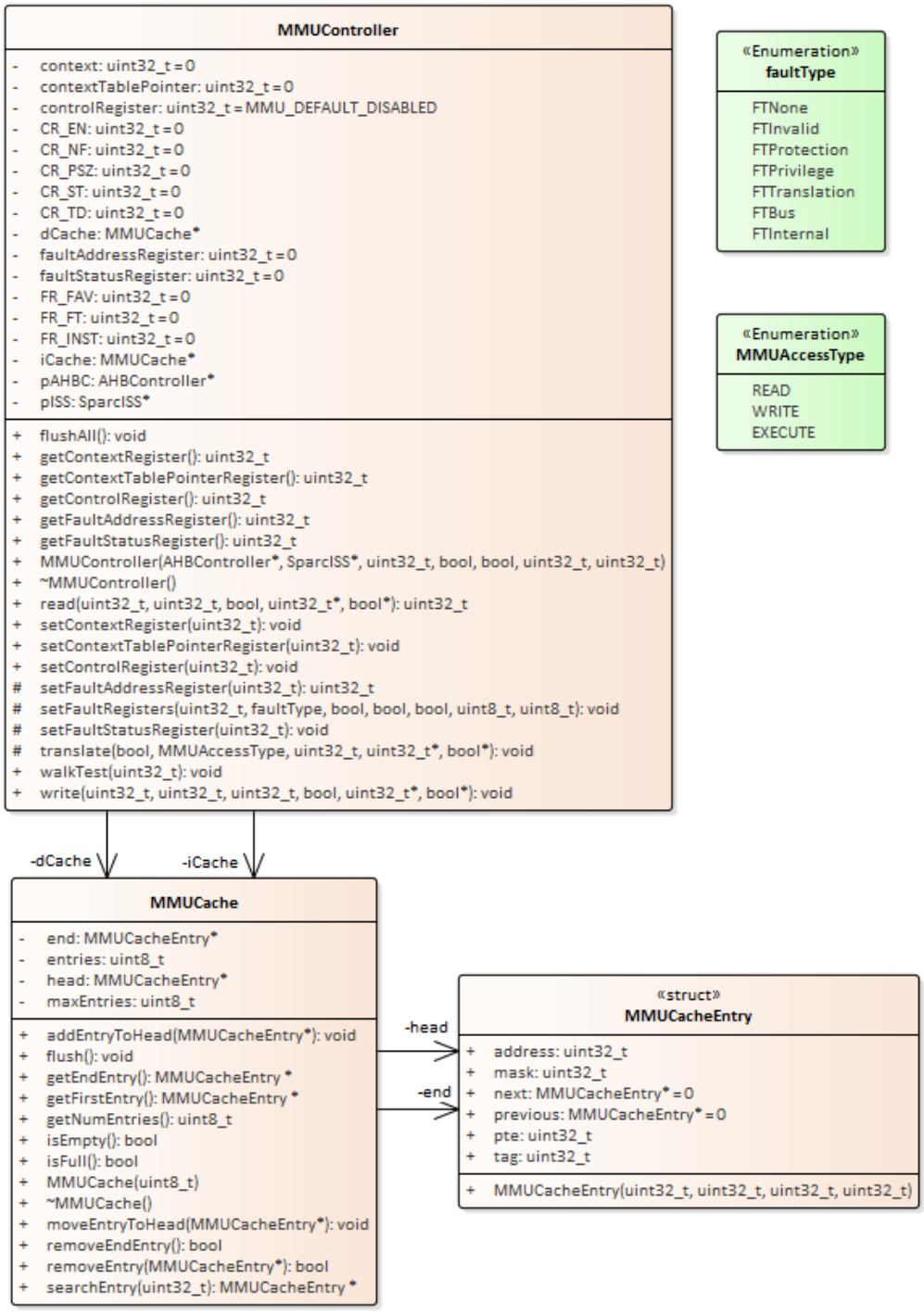


Ilustración 7 - Diagrama de clases de la MMU de LeonViP

3.3.2 Controlador de MMU (*MMUController*)

Simula el funcionamiento de la MMU, incluyendo registros y funciones para solicitar traducción o acceso a sus registros. Sus atributos pueden dividirse en 3 tipos: registros, campos de registro para acceso rápido y punteros a otros elementos del sistema. También incluye 2 tipos enumerados, uno que identifica los diferentes tipos de fallo y otro que representa los posibles tipos de acceso a la MMU.

3.3.2.1 Cabecera

En la primera parte de la cabecera, cuyo código se adjunta tras este párrafo, se comienza dando una definición a valores como la configuración por defecto o máscaras para ciertos campos de los registros de MMU, facilitando la legibilidad y creación del programa. También se crean los enumerados para el tipo de acceso y tipo de fallo.

-- Definiciones de la cabecera de *MMUController* --

```
1    // Valores por defecto de los campos configurables de la MMU
2    #define DEFAULT_MMU_ITLB 4
3    #define DEFAULT_MMU_DTLB 4
4    #define DEFAULT_MMU_PAGE_SIZE 0
5    #define DEFAULT_MMU_TLB_DISABLE 0
6    #define DEFAULT_MMU_TLB_SEPARATED 1
7    #define DEFAULT_MMUEN 1
8    #define CONTROL_REG_WR 0x00038003
9
10   // Máscaras del registro de control
11   #define MMU_CT_ENABLE 0x1
12   #define MMU_CT_NOFAULT 0x2
13   #define MMU_CT_SO 0x2000
14   #define MMU_CT_TLBSEP 0x4000
15   #define MMU_CT_TLBDIS 0x8000
16   #define MMU_CT_PAGESIZE 0x30000
17   #define MMU_CT_DTLB 0x1C0000
18   #define MMU_CT_ITLB 0xE00000
19
20   // Máscaras del registro de estado de fallo
21   #define MMU_FT_OW 0x1
22   #define MMU_FT_VALID 0x2
23   #define MMU_FT_FTYPE 0x1C
24   #define MMU_FT_ACC 0xE0
25   #define MMU_FT_LVL 0x300
26   #define MMU_FT_EBE 0x3FC00
27
28   // Máscaras de entradas de tablas de página
29   #define PT_ET 0x00000003
30   #define PTD_PTP 0xFFFFFFFC
31   #define PTE_PPN 0xFFFFFF00
32   #define PTE_ACC 0x0000001C
33   #define PTE_R 0x00000020
34   #define PTE_M 0x00000040
35   #define PTE_C 0x00000080
36
37   enum MMUAccessType {READ, WRITE, EXECUTE};
38
```

```
39  #define MMU_DEFAULT_DISABLED 0x01000000
40
41  enum faultType : uint8_t {
42      FTNone,
43      FTInvalid,
44      FTProtection,
45      FTPrivilege,
46      FTTranslation,
47      FTBus,
48      FTInternal
49  };
```

Código 4 - definiciones en la cabecera del controlador de MMU - MMUController.h

Al final de esta cabecera se declara los atributos y funciones que va a usar esta clase, como se muestra en el código siguiente:

-- Definición de la clase del controlador de MMU --

```
1   class MMUController {
2
3     private:
4       MMUCache *iCache, *dCache;
5       AHBController *pAHBC;
6       SparcISS *pISS;
7       uint32_t controlRegister = MMU_DEFAULT_DISABLED; // Registro de control
8
9       // Campos del registro de control para comparaciones rápidas
10      uint32_t CR_EN = 0;
11      uint32_t CR_NF = 0;
12      uint32_t CR_ST = 0;
13      uint32_t CR_TD = 0;
14      uint32_t CR_PSZ = 0;
15
16      uint32_t faultStatusRegister = 0; // Registro de estado de fallo
17
18      // Campos del registro de estado de fallo para comparaciones rápidas
19      uint32_t FR_FT = 0;
20      uint32_t FR_FAV = 0;
21      uint32_t FR_INST = 0;
22
23      uint32_t faultAddressRegister = 0; // Registro de dirección de fallo
24      uint32_t context = 0;              // Registro de contexto
25      uint32_t contextTablePointer = 0;  // Registro de puntero a tabla de contexto
26
27    protected:
28
29      void translate(bool isInstruction, MMUAccessType accType,
30                     uint32_t vAddress, uint32_t *phAddress,
31                     bool *isCacheable);
32
33      void setFaultRegisters(uint32_t vAddress, faultType statusType,
34                             bool isWrite, bool isInstruction,
35                             bool isSupervisor, uint8_t level,
36                             uint8_t externBus);
37
38      void setFaultStatusRegister(uint32_t data);
39      uint32_t setFaultAddressRegister(uint32_t data);
40
```

```
41      public:
42
43        MMUController(AHBController *pAHBC, SparcISS *pISS, uint32_t pageSize,
44                      bool enableCache, bool separatedCache, uint32_t iEntries,
45                      uint32_t dEntries);
46        ~MMUController();
47
48        void setControlRegister(uint32_t data);
49        uint32_t getControlRegister();
50        void setContextTablePointerRegister(uint32_t data);
51        uint32_t getContextTablePointerRegister();
52        void setContextRegister(uint32_t data);
53        uint32_t getContextRegister();
54        uint32_t getFaultStatusRegister();
55        uint32_t getFaultAddressRegister();
56
57        uint32_t read(uint32_t vAddress, uint32_t size, bool isInstruction,
58                      uint32_t *phAddress, bool *isCacheable);
59
60        void write(uint32_t vAddress, uint32_t size, uint32_t value,
61                   bool isInstruction, uint32_t *phAddress, bool *isCacheable);
62
63        void flushAll();
64        void walkTest(uint32_t vAddress);
65    };
66
```

Código 5 - definición de la clase del controlador de MMU - MMUController.h

3.3.2.2 Implementación

El fichero de implementación *MMUController.cpp* comienza creando una serie de constantes para facilitar la obtención de máscaras de offset de la dirección virtual, además de las matrices de comprobación de permisos, como se muestra a continuación:

-- Declaración de constantes de offset de dirección virtual y comprobación de permisos --

```
1   #include <mmucontroller.h>
2
3   // Máscaras de offsets de la dirección virtual en función de PSZ y nivel
4   static const uint32_t vOff[4][4] = {
5       {0xFFFFFFFF, 0x00FFFFFF, 0x0003FFFF, 0x00000FFF},
6       {0xFFFFFFFF, 0x01FFFFFF, 0x0007FFFF, 0x00001FFF},
7       {0xFFFFFFFF, 0x03FFFFFF, 0x000FFFFF, 0x00003FFF},
8       {0xFFFFFFFF, 0x0FFFFFFF, 0x001FFFFF, 0x00007FFF}};
9
10  // Máscaras de índices de la dirección virtual según PSZ y nivel
11  static const uint32_t vM[4][4] = {{0, 0xFF000000, 0x00FC0000, 0x0003F000},
12                                    {0, 0xFE000000, 0x01F80000, 0x0007E000},
13                                    {0, 0xFC000000, 0x03F00000, 0x000FC000},
14                                    {0, 0xF0000000, 0x0FE00000, 0x001F8000}};
15
16  // Desplazamientos necesarios de la dirección virtual según PSZ y nivel
17  static const uint32_t vSh[4][4] = {
18      {0, 22, 16, 10}, {0, 23, 17, 11}, {0, 24, 18, 12}, {0, 26, 19, 13}};
19
20  // Número de entradas máxima de TLB según campos ITLB y DTLB del registro de control
21  static const uint8_t entries[7] = {0, 0, 4, 8, 16, 32, 64};
```

```
22
23  // Permisos en función de privilegio, ACC y tipo de acceso
24  static const bool accTable[2][8][3]{ // [user/super][ACC][r/w/x]
25      // UserTable      acc
26      {1, 0, 0,  //     0
27       1, 1, 0,  //     1
28       1, 0, 1,  //     2
29       1, 1, 1,  //     3
30       0, 0, 1,  //     4
31       1, 0, 0,  //     5
32       0, 0, 0,  //     6
33       0, 0, 0}, //     7
34
35      // SuperTable     acc
36      {1, 0, 0, //      0
37       1, 1, 0, //      1
38       1, 0, 1, //      2
39       1, 1, 1, //      3
40       0, 0, 1, //      4
41       1, 1, 0, //      5
42       1, 0, 1, //      6
43       1, 1, 1} //      7
44  };
```

Código 6 -Declaración Constantes para offset de dirección virtual y comprobación de permisos - MMUController.cpp

Tras ello, se definen las funciones declaradas en la cabecera. Comienza con el método de instanciación de clase *MMUController*, adjuntado tras este párrafo, el cual primero comprueba si los argumentos son válidos y luego asigna la configuración, creando 1 o 2 TLB según si activa la opción de TLB compartida o no. Finalmente escribe el registro de control según la configuración dada.

-- Método de instanciación de la clase del controlador de MMU --

```
1   MMUController::MMUController(AHBController *pAHBC, SparcISS *pISS,
2                                uint32_t pageSize, bool enableCache,
3                                bool separatedCache, uint32_t iEntries,
4                                uint32_t dEntries) {
5
6       if ((iEntries < 2 || iEntries > 6) ||
7           (separatedCache && (dEntries < 2 || dEntries > 6))) {
8           exit(EXIT_FAILURE);
9       }
10      if (pageSize > 3) { exit(EXIT_FAILURE);}
11
12      this->pAHBC = pAHBC;
13      this->pISS = pISS;
14
15      CR_PSZ = pageSize;
16      CR_TD = !enableCache;
17      CR_ST = separatedCache;
18
19      if (separatedCache) {
20          dCache = new MMUCache(entries[dEntries]);
21      } else {
22          dEntries = 0;
23          dCache = 0;
24      }
```

```
25
26      iCache = new MMUCache(entries[iEntries]);
27
28      controlRegister |= (iEntries << 21) | (dEntries << 18) | (CR_PSZ << 16) |
29                         (CR_TD << 15) | (CR_ST << 14);
30  }
31  MMUController::~MMUController() {}
```

Código 7 - Método de instanciación del controlador de MMU - MMUController.cpp

Seguidamente se definen las funciones para la obtención y escritura de los registros. En las funciones de los registros cuyos campos se tratan por separado mediante atributos para un acceso más rápido, se reasignan estos atributos cuando se modifica el registro.

-- Métodos de lectura y escritura de registros de la MMU --

```
1   void MMUController::setControlRegister(uint32_t data) {
2       controlRegister &= (~CONTROL_REG_WR);
3       controlRegister |= (data & CONTROL_REG_WR);
4       CR_EN = controlRegister & MMU_CT_ENABLE;
5       CR_NF = (controlRegister & MMU_CT_NOFAULT) ? 1 : 0;
6       CR_TD = (controlRegister & MMU_CT_TLBDIS) ? 1 : 0;
7       CR_PSZ = (controlRegister & MMU_CT_PAGESIZE) >> 16;
8   }
9
10  uint32_t MMUController::getControlRegister() { return controlRegister; }
11
12  void MMUController::setContextRegister(uint32_t data) {
13      this->context = data & 0x0F;
14  }
15
16  uint32_t MMUController::getContextRegister() { return context; }
17
18  void MMUController::setContextTablePointerRegister(uint32_t data) {
19      contextTablePointer = data & 0xFFFFFFF0;
20  }
21
22  uint32_t MMUController::getContextTablePointerRegister() {
23      return contextTablePointer;
24  }
25
26  void MMUController::setFaultStatusRegister(uint32_t data) {
27      faultStatusRegister = data;
28  }
29
30  uint32_t MMUController::getFaultStatusRegister() {
31      uint32_t value = faultStatusRegister;
32      faultStatusRegister &= ~(MMU_FT_VALID);
33      FR_FAV = 0;
34      return value;
35  }
36
37  uint32_t MMUController::getFaultAddressRegister() {
38      return faultAddressRegister;
39  }
```

Código 8 - Método de escritura y lectura de registros de la MMU - MMUController.cpp

Los métodos *write* y *read*, mostrados en *Código 9 - Método de escritura y lectura a través de la MMU - MMUController.cpp*, se usan para solicitar un acceso a memoria, pero en lugar de solo traducir, directamente escriben o devuelven el dato respectivamente, facilitando la comunicación entre las distintas clases del simulador.

Ambos métodos reciben de argumento la dirección virtual a traducir, información sobre el tamaño del dato y el área de acceso. En caso de estar la MMU habilitada, obtiene la dirección física llamando al método *translate*, y si no trata la dirección virtual como física. Una vez tiene la dirección física realiza directamente la escritura o lectura accediendo al bus AHB, pero solamente si no se generó una excepción durante la traducción.

-- Métodos de lectura y escritura a través de la MMU --

```
1   uint32_t MMUController::read(uint32_t vAddress, uint32_t size,
2                                bool isInstruction, uint32_t *phAddress,
3                                bool *isCacheable) {
4
5       if (CR_EN) {
6           MMUAccessType acc = (isInstruction) ? EXECUTE : READ;
7
8           translate(isInstruction, acc, vAddress, phAddress, isCacheable);
9
10          return (pISS->trap) ? 0 : pAHBC->read(*phAddress, size);
11
12      } else {
13          *phAddress = vAddress;
14          *isCacheable = 1;
15          return pAHBC->read(vAddress, size);
16      }
17  }
18
19  void MMUController::write(uint32_t vAddress, uint32_t size, uint32_t value,
20                            bool isInstruction, uint32_t *phAddress,
21                            bool *isCacheable) {
22
23      if (CR_EN) {
24
25          translate(isInstruction, WRITE, vAddress, phAddress, isCacheable);
26
27          if (!pISS->trap) {
28              pAHBC->write(*phAddress, value, size);
29          }
30
31      } else {
32          pAHBC->write(vAddress, value, size);
33          *phAddress = vAddress;
34          *isCacheable = 1;
35      }
36  }
```

Código 9 - Método de escritura y lectura a través de la MMU - MMUController.cpp

El método *translate* hace algo muy parecido a lo descrito en el apartado *3.2.6 Proceso de traducción de direcciones*, es decir, traduce una dirección virtual a la física. Este se muestra en *Código 10 - Método de traducción de la MMU - MMUController.cpp.*

El método comienza obteniendo el nivel de privilegio usando el registro PSR de la ISS y, tras ello, primero comprueba si la TLB está habilitada, en cuyo caso elije la TLB correspondiente según si es compartida o no y del área de acceso.

En caso de haber TLB, hace una búsqueda comparando los tags de la TLB con el tag creado, que se compone de la dirección virtual y contexto actual. Si hay acierto, mueve dicha entrada a la cabeza de la TLB y obtiene el PTE y máscara de la dirección virtual. También, en caso de escritura, actualiza el bit de modificado en la entrada de TLB y en memoria.

Si no hubo acierto de TLB, realiza un recorrido de tablas (*table walk*), partiendo de la dirección que da el registro de puntero de contexto más el contexto actual, y en un bucle va recorriendo las tablas hasta encontrar una entrada de tabla de página.

En caso de encontrar el PTE, comprueba y asigna los bits de referencia y modificado según requiera, obtiene la máscara de la dirección virtual según el PSZ y nivel al que encontró el PTE y crea la nueva entrada de TLB.

En caso de no haber encontrado el PTE, llama a la función *setFaultRegisters*, que se encarga de asignar los registros de fallo y enviar la señal de *trap* correspondiente.

Si se obtuvo una entrada de página válida, comprueba sus permisos mirando el campo ACC y los compara usando la tabla de permisos, mirando el bit que corresponde con los privilegios, ACC y tipo de acceso actual. Si no tiene permisos, llama a *setFaultRegisters.*

Finalmente, si todo fue bien, obtiene la dirección física usando la dirección dada por la entrada de página y el offset de la dirección virtual, devolviendo por referencia dicha dirección y el atributo que indica si es *cacheable*.

-- Método de traducción de la MMU --

```
1   void MMUController::translate(bool isInstruction, MMUAccessType accType,
2                                 uint32_t vAddress, uint32_t *phAddress,
3                                 bool *isCacheable) {
4
5       uint32_t entry, eAddress, vMask;
6       uint8_t level = 0;
7       *isCacheable = 0;
8
9       bool isSupervisor = (pISS->getRegisterValue(PSR_id) & 0x80);
10
11      bool hit = false;
12
13      if (!CR_TD) {
14          MMUCache *cache = (isInstruction || !CR_ST) ? iCache : dCache;
15          MMUCacheEntry *CacheEntry =
16              cache->searchEntry((vAddress & ~vOff[CR_PSZ][3]) | context);
17          if (CacheEntry) {
18              cache->moveEntryToHead(CacheEntry);
19              entry = CacheEntry->pte;
20              vMask = ~CacheEntry->mask | 0xFF;
21
22              if ((accType == WRITE) && !(entry & PTE_M)) {
```

```
23                  entry |= PTE_M;
24                  pAHBC->write(CacheEntry->address, entry, FOUR_BYTES);
25                  CacheEntry->pte = entry;
26              }
27              hit = true;
28          }
29      }
30
31      if (!hit) {
32          eAddress = (contextTablePointer << 4) + (context << 2);
33          entry = pAHBC->read(eAddress, FOUR_BYTES);
34
35          while (((entry & PT_ET) == 1) && (level < 3)) {
36              level += 1;
37              eAddress = ((entry & PTD_PTP) << 4) +
38                         ((vAddress & vM[CR_PSZ][level]) >> vSh[CR_PSZ][level]);
39              entry = pAHBC->read(eAddress, FOUR_BYTES);
40          }
41
42          if ((entry & PT_ET) != 2) {
43              faultType fault = (entry & PT_ET) ? FTTranslation : FTInvalid;
44              setFaultRegisters(vAddress, fault, (accType == WRITE),
45                                      isInstruction, isSupervisor, level, 0);
46              return;
47          }
48
49          uint32_t nEntry = entry;
50          entry |= PTE_R | (PTE_M * (accType == WRITE));
51
52          if (nEntry != entry) {
53              pAHBC->write(eAddress, entry, FOUR_BYTES);
54          }
55
56          vMask = vOff[CR_PSZ][level];
57
58          if (!CR_TD) {
59              if (isInstruction || !CR_ST) {
60                  iCache->addEntryToHead(
61                      new MMUCacheEntry(entry, eAddress, ~vMask | 0xFF,
62                                        (vAddress & ~vMask) | context));
63              } else {
64                  dCache->addEntryToHead(
65                      new MMUCacheEntry(entry, eAddress, ~vMask | 0xFF,
66                                        (vAddress & ~vMask) | context));
67              }
68          }
69      }
70
71      uint8_t acc = (entry & PTE_ACC) >> 2;
72
73      if (!accTable[isSupervisor][acc][accType]) {
74          faultType fault =
75              (!isSupervisor && (acc > 5)) ? FTPrivilege : FTProtection;
76          setFaultRegisters(vAddress, fault, (accType == WRITE),
77                                    isInstruction, isSupervisor, level, 0);
78          return;
79      }
80      *phAddress = ((entry & PTE_PPN) << 4) | (vAddress & vMask);
81      *isCacheable = (entry & PTE_C) != 0;
82  }
```

Código 10 - Método de traducción de la MMU - MMUController.cpp

La función *setFaultRegisters*, mostrada en *Código 11 - Método tratamiento de fallos de la MMU - MMUController.cpp*, se encarga de tratar los fallos, asignando los registros de estado y dirección de fallo y enviando la señal de *trap* correspondiente.

Primero comprueba si hay un fallo sin atender mirando el campo de dirección de fallo válida (*FAV*). En caso de haberlo, debe comprobar las prioridades de clase de fallo, primero mirando si es de traducción y, si no, comprobando el área donde ocurrió el fallo. En caso de nuevo fallo (*FAV* es 0) o sobreescritura de fallo, sustituye los registros según los datos dados de argumento. Finalmente, comprueba el campo de "no fallo" (*No fault* o NF), que usa para ignorar ciertos tipos de fallo y, según su valor y el área donde se produjo el fallo, envía la señal de *trap* correspondiente.

-- Método de tratamiento de fallos de la MMU --

```
1   void MMUController::setFaultRegisters(uint32_t vAddress,
2                                         faultType statusType, bool isWrite,
3                                         bool isInstruction, bool isSupervisor,
4                                         uint8_t level, uint8_t externBus) {
5
6       uint8_t accessType = ((isWrite << 2) + (isInstruction << 1) + isSupervisor);
7       uint32_t fs = (externBus << 10) + (level << 8) + (accessType << 5) +
8                     (statusType << 2) + MMU_FT_VALID;
9       uint32_t fa = vAddress & 0xFFFFF000;
10
11      if (FR_FAV) {
12          if (statusType == FTTranslation) {
13              faultStatusRegister = fs;
14              faultAddressRegister = fa;
15              FR_FT = statusType;
16              FR_INST = isInstruction;
17          } else if (FR_FT != FTTranslation) {
18              if (!isInstruction) {
19                  faultStatusRegister = fs;
20                  faultAddressRegister = fa;
21                  FR_FT = statusType;
22                  FR_INST = isInstruction;
23              } else if (FR_INST) {
24                  faultStatusRegister = fs | MMU_FT_OW;
25                  faultAddressRegister = fa;
26                  FR_FT = statusType;
27                  FR_INST = isInstruction;
28              }
29          }
30      } else {
31          faultStatusRegister = fs;
32          faultAddressRegister = fa;
33          FR_FT = statusType;
34          FR_INST = isInstruction;
35          FR_FAV = 1;
36      }
37
38      if (!isInstruction && !CR_NF) {
39          pISS->trap = TRAP_DEXC;
40      } else if (isInstruction && (isSupervisor || !CR_NF)) {
41          pISS->trap = TRAP_IEXC;
42      }
43  }
```

Código 11 - Método tratamiento de fallos de la MMU - MMUController.cpp

El método de *flush*, mostrado a continuación, elimina las entradas de TLB llamando a las correspondientes funciones de *MMUCache*.

-- Método de flush de la MMU --

```
1  void MMUController::flushAll() {
2      iCache->flush();
3      if (CR_ST) {
4          dCache->flush();
5      }
6  }
```

Código 12 - Método de flush MMU - MMUController.cpp

Para finalizar, se define el método *walkTest*, el cual se muestra en *Código 13 - Método de test de recorrido de tablas de página - MMUController.cpp*. Este no forma parte del comportamiento a simular de la MMU, sino que facilita la realización de pruebas de recorrido de tablas de páginas, para lo cual realiza algo parecido a la función *translate*. En este caso, el método no comprueba la TLB ni trata los fallos, sino únicamente los accesos que hace y sus resultados, y finalmente los muestra por pantalla.

-- Método de test de recorrido de tablas de página de la MMU --

```
1   void MMUController::walkTest(uint32_t vAddress) {
2
3       if (CR_EN) {
4           std::cout << std::hex << "Walk: 0x" << vAddress << std::dec
5                     << " Context: " << context << endl;
6
7           int level = 0;
8           uint32_t offset = 0;
9           uint32_t address = (contextTablePointer << 4) + (context << 2);
10          uint32_t entry = pAHBC->read(address, FOUR_BYTES);
11
12          std::cout << std::dec << "lv 0: 0x" << std::hex << address << " -> 0x"
13                    << entry << endl;
14
15          while (((entry & PT_ET) == 1) && (level < 3)) {
16              level += 1;
17              offset = (vAddress & vM[CR_PSZ][level]) >> vSh[CR_PSZ][level];
18              vAddress &= ~vM[CR_PSZ][level];
19              address = ((entry & PTD_PTP) << 4) + offset;
20              entry = pAHBC->read(address, FOUR_BYTES);
21              std::cout << std::dec << "lv " << level << ": 0x" << std::hex
22                        << address << " -> 0x" << entry << endl;
23          }
24          uint32_t PhAddress = ((entry & PTE_PPN) << 4) + vAddress;
25
26          int acc = (entry & PTE_ACC) >> 2;
27
28          switch (entry & PT_ET) {
29          case 0:
30              std::cout << "Invalid Entry.\n" << endl;
31              break;
32          case 1:
33              std::cout << "Page Descriptor.\n" << endl;
34              break;
35          case 3:
```

```
36              std::cout << "Reserved.\n" << endl;
37              break;
38          default:
39
40              string txtAccAllow[8] = {
41                  "RD for User and Supervisor",
42                  "RD/WR for User and Supervisor",
43                  "RD/X for User and Supervisor",
44                  "RD/WR/X for User and Supervisor",
45                  "X for User and Supervisor",
46                  "RD for User || RD/WR for Supervisor",
47                  "NO access for User || RD/X for Supervisor",
48                  "NO access for User || RD/WR/X for Supervisor"};
49
50              std::cout << std::dec << "Page Entry ("
51                        << "Ref: " << ((entry & PTE_R) >> 5)
52                        << " Mod: " << ((entry & PTE_M) >> 6)
53                        << " Cachable: " << ((entry & PTE_C) >> 7)
54                        << " Acc: " << acc
55                        << ")\nPermissions: " << txtAccAllow[acc] << std::hex
56                        << "\nPteLv3: 0x" << entry << "\nResult Address: 0x"
57                        << PhAddress << "\n"
58                        << endl;
59
60              break;
61          }
62      } else {
63          std::cout << "MMU is Disabled.\n " << endl;
64      }
65  }
```

Código 13 - Método de test de recorrido de tablas de página - MMUController.cpp

3.3.3 Caché de MMU (*MMUCache*)

Simula la TLB de la MMU. Esta se define como una lista doblemente enlazada, en la que se usan los punteros *head* y *end* para indicar el primer y ultimo elemento de la lista, el valor *entries* que indica el numero de entradas, y *maxEntries* que indica el máximo de entradas. Sus funciones incluyen utilidades para la inserción, eliminado, movimiento y búsqueda de entradas.

También define el formato de cada entrada, que como se describió en el apartado *3.2.5 TLB*, incluye los campos de máscara, tag, PTE y *address*, además de los punteros a la entrada anterior y siguiente de la lista.

3.3.3.1 Cabecera

La cabecera define primero la estructura de una entrada de TLB (*MMUCacheEntry*), y luego declara las funciones a usar por esta clase. Los atributos son punteros al comienzo y final de la lista, y variables que indican el número de entradas y máximo de entradas. Esto se muestra en el código siguiente:

-- Cabecera de la caché de MMU --

```
struct MMUCacheEntry {
    MMUCacheEntry(uint32_t pte, uint32_t address, uint32_t mask, uint32_t tag) {
        this->pte = pte;
        this->address = address;
        this->mask = mask;
        this->tag = tag;
    }
    uint32_t pte, address, mask, tag;
    MMUCacheEntry *next = 0, *previous = 0;
};

class MMUCache {
  private:
    MMUCacheEntry *head, *end;
    uint8_t entries, maxEntries;

  public:

    MMUCache(uint8_t maxEntries);
    ~MMUCache();

    bool isFull();
    bool isEmpty();
    uint8_t getNumEntries();
    MMUCacheEntry *searchEntry(uint32_t tag);
    MMUCacheEntry *getEndEntry();
    MMUCacheEntry *getFirstEntry();
    void moveEntryToHead(MMUCacheEntry *entry);
    void addEntryToHead(MMUCacheEntry *entry);
    bool removeEntry(MMUCacheEntry *entry);
    bool removeEndEntry();
    void flush();
};
```

Código 14 - Cabecera de la caché de MMU - MMUCache.h

3.3.3.2 Implementación

El fichero de implementación *MMUCache.cpp* comienza definiendo el método de instanciación de clase en el que simplemente se inicializan los punteros y el número de entradas a 0, indicando que está vacía, y se asigna el número máximo de entradas que puede contener:

-- Método de instanciación de la caché de MMU --

```
MMUCache::MMUCache(uint8_t maxEntries) {
    this->head = 0;
    this->end = 0;
    this->entries = 0;
    this->maxEntries = maxEntries;
}
MMUCache::~MMUCache() {}
```

Código 15 - Cabecera de la caché de MMU - MMUCache.cpp

Tras lo anterior se definen los métodos sencillos típicos de una estructura de datos de lista doblemente enlazada, como el *isFull* que devuelve si la TLB está llena o el *isEmpty* que devuelve si está vacía. También se incluye un método que devuelve el número de entradas, y otros 2 para obtener la primera y la última entrada. Todos éstos se muestran a continuación:

-- Métodos cortos de la caché de MMU --

```
1    bool MMUCache::isFull() { return (entries == maxEntries); }
2    bool MMUCache::isEmpty() { return (entries == 0); }
3    uint8_t MMUCache::getNumEntries() { return entries; }
4    MMUCacheEntry *MMUCache::getEndEntry() { return this->end; }
5    MMUCacheEntry *MMUCache::getFirstEntry() { return this->head; }
```

Código 16 - Métodos cortos de la caché de MMU - MMUCache.cpp

Además, se incluyen varios métodos para la inserción, la eliminación y el movimiento de entradas, como *moveEntryToHead* que mueve la entrada indicada en el argumento a la cabeza de la lista, o *addEntryToHead* que añade la nueva entrada de argumento a la cabeza de la lista, mostrados en el siguiente código:

-- Métodos de tratamiento de entradas de la caché de MMU --

```
1    void MMUCache::moveEntryToHead(MMUCacheEntry *entry) {
2        if (head != entry) {
3            if (end == entry) { end = entry->previous;
4            } else {
5                entry->next->previous = entry->previous;
6            }
7            entry->previous->next = entry->next;
8            head->previous = entry;
9            entry->next = head;
10           entry->previous = 0;
11           head = entry;
12       }
13   }
14   void MMUCache::addEntryToHead(MMUCacheEntry *entry) {
15
16       if (isFull()) { removeEndEntry(); }
17
18       entry->previous = 0;
19
20       if (entries == 0) {
21           head = entry;
22           end = entry;
23           entry->next = 0;
24       } else {
25           entry->next = head;
26           head->previous = entry;
27           head = entry;
28       }
29       entries++;
30   }
31
32   bool MMUCache::removeEntry(MMUCacheEntry *entry) {
33
34       if (entries > 0) {
```

```
35          if (entries > 1) {
36              if (head == entry) {
37                  entry->next->previous = 0;
38                  head = entry->next;
39              } else if (end == entry) {
40                  entry->previous->next = 0;
41                  end = entry->previous;
42              } else {
43                  entry->next->previous = entry->previous;
44                  entry->previous->next = entry->next;
45              }
46          } else {
47              head = 0; end = 0;
48          }
49          entries--;
50          delete entry;
51          return true;
52      }
53      return false;
54  }
55
56  bool MMUCache::removeEndEntry() {
57      MMUCacheEntry *entry;
58      entry = end;
59      if (entries > 0) {
60          if (entries > 1) {
61              entry->previous->next = 0;
62              end = entry->previous;
63          } else {
64              head = 0; end = 0;
65          }
66          entries--;
67          delete entry;
68          return true;
69      }
70
71      return false;
72  }
```

Código 17 - Métodos de tratamiento de entradas de la caché de MMU - MMUCache.cpp

El siguiente método que se define es *searchEntry*, mostrado en *Código 18 - Método de búsqueda de entrada de la caché de MMU - MMUCache.cpp*. Este método se usa a la hora de comprobar si se tiene guardada en la TLB la dirección virtual que se quiere traducir. Para ello la MMU llama a esta función con un tag que se compone de la dirección virtual con los bits de offset a 0 más el contexto.

Como el offset depende del nivel de la página, pero la MMU no conoce ese nivel hasta que realiza el recorrido de páginas, esta función hace un *AND* entre la máscara guardada y el tag traído por la MMU, lo cual elimina los bits de offset según el nivel al que se encontró la entrada guardada en TLB.

-- Método de búsqueda de entrada de la caché de MMU --

```
1   MMUCacheEntry *MMUCache::searchEntry(uint32_t tag) {
2
3       MMUCacheEntry *entry = head;
4
5       while (entry && (entry->tag != (tag & entry->mask))) {
6           entry = entry->next;
7       }
8
9       return entry;
10  }
```

Código 18 - Método de búsqueda de entrada de la caché de MMU - MMUCache.cpp

Por último, se define la función *flush*, que simplemente elimina todas las entradas de la TLB, cuyo código sería el siguiente:

-- Método de flush de la caché de MMU --

```
1   void MMUCache::flush() {
2       MMUCacheEntry *entry, *next;
3       entry = head;
4       while (entry) {
5           next = entry->next;
6           delete entry;
7           entry = next;
8           entries--;
9       }
10      head = 0;
11      end = 0;
12  }
```

Código 19 - Método de flush de la caché de MMU - MMUCache.cpp

3.3.4 Unión de la MMU con el resto de elementos de LeonViP

Una vez definida y creada la MMU, es necesario conectarla con el resto del sistema del simulador LeonViP.

3.3.4.1 Conexión con la ISS

La ISS, en concreto la clase SparcISS, necesita de la conexión a la MMU para llamar a las funciones de lectura y escritura de registros de MMU, además de la función de *flush*, que dependen del campo ASI de las instrucciones de carga y guardado. Se muestra un ejemplo de éste a continuación:

-- Uso de función de flush de MMU mediante ASI en SparcISS --

```
1    uint32_t checkASI(SparcISS *pISS, uint32_t address, uint32_t size,
2                      uint32_t data, uint32_t asi, bool isLoad) {
3        uint32_t retData = 0;
4        switch (asi) {
5
6            (...)
7
8            case ASI_MMU_FLUSH_TLB_AND_CACHE:
9                pISS->pCacheController->setFlushICache();
10               pISS->pCacheController->setFlushDCache();
11               pISS->pMMU->flushAll();
12               break;
13
14           (...)
15
16       }
17   }
```

Código 20 - Extracto de la función de tratamiento de ASI en SparcISS.cpp

3.3.4.2 Conexión con la caché

En la implementación original del simulador LeonViP, la caché accedía directamente a memoria usando el bus AHB, pero ahora debe hacerlo a través de la MMU. Para lograr ésto, se han creado unas funciones de lectura y escritura hacia la MMU, que sustituyen a las que se usaban para acceder al bus, las cuales se muestran en el siguiente código:

-- Uso de MMU en CacheController --

```
1    uint32_t CacheController::readMMU(uint32_t address, uint32_t size,
2                                      CacheBlock *cb, uint32_t *phAddress,
3                                      bool *isCacheable) {
4
5        return pMMU->read(address, size, !cb->getIsDataCache(), phAddress,
6                          isCacheable);
7    }
8    void CacheController::writeMMU(uint32_t address, uint32_t data, uint32_t size,
9                                   CacheBlock *cb, uint32_t *phAddress,
```

```
                                bool *isCacheable) {

    pMMU->write(address, size, data, !cb->getIsDataCache(), phAddress,
                isCacheable);
}
```

Código 21 - funciones de llamada a MMU en CacheController.cpp

-- Uso de MMU en CacheController --

```
void CacheController::writeCache(uint32_t address, uint32_t data, uint32_t size,
                                 CacheBlock *cb) {
    if (cb->isCached(address)) {
        cb->write(address, data, size);
    }

    uint32_t phAddress;
    bool isCacheable;
    writeMMU(address, data, size, cb, &phAddress, &isCacheable);
}
```

Código 22 - función de escritura de caché en CacheControler.cpp

3.3.4.3 Conexión con el bus AHB

Por último, la MMU se conecta con el bus AHB para poder acceder a memoria, ya que ahora está de intermediaria cuando se necesitan leer o escribir datos, además de los accesos a memoria necesarios para hacer el recorrido de tablas de páginas. Se puede comprobar dicha conexión en el código mostrado de la clase de controlador de MMU, para las funciones *read*, *write* y *translate*, que corresponden con *Código 9 - Método de escritura y lectura a través de la MMU - MMUController.cpp* y *Código 10 - Método de traducción de la MMU - MMUController.cpp*.

3.4 Validación

Para comprobar que el funcionamiento de la MMU implementada en el simulador LeonViP se corresponde con la implementada por LEON3 se ha creado un programa en C usando el compilador cruzado BCC que lo compila para la ISA de SPARCv8.

Este programa luego se ejecuta usando LeonViP y en la placa de desarrollo GR-XC3S-1500 programada con la implementación LEON3 y se comparan los resultados, los cuales deben coincidir.

3.4.1 Objetivos de la validación

La MMU, aunque no es excesivamente compleja, puede producir una gran cantidad de resultados según las diferentes opciones de configuración, sobre todo en cuanto al tratamiento de fallos.

Principalmente se busca comprobar que las traducciones se realizan correctamente y se tratan lo fallos tal y como lo hace la placa con la que se va a comparar. El mayor problema a la hora de realizar la validación, es que hay muchos factores que afectan al tratamiento de fallos, ya que estos dependen de el nivel en el que se encuentra la entrada, los 7 posibles permisos (ACC), el privilegio, el tipo de acceso y si la entrada está en caché o no.

Como la MMU comienza deshabilitada, y se mantiene así hasta que se escriben las tablas de páginas y se habilita manualmente, el hecho de que se llegue al punto en el que se habilita sirve para demostrar que la MMU deshabilitada funciona correctamente.

Una vez con la MMU habilitada, se realizan lecturas y escrituras, tanto de datos como de instrucción, en lugares de memoria definidos con entradas de páginas con diferentes bits de permisos, y se comprueba que el resultado en los registros de fallo y señales de *trap* coinciden.

3.4.2 Diseño del programa de validación

Para facilitar la validación de la MMU, el programa de validación va a realizar impresiones de los resultados obtenidos por pantalla o por la puerta serie, dependiendo de si se trata del simulador o de la placa de desarrollo. De esta manera, si lo que ha impreso el simulador LeonViP coincide con lo producido por la placa con la que se compara, se puede decir que la MMU implementada en el simulador funciona como la original.

A la hora de crear un programa que use la MMU, una de las cosas más importantes es la realización de un mapeado de memoria virtual a física. Para conseguir esto se opta por hacer una asignación 1 a 1 de direcciones virtuales a físicas, es decir, que, tras realizar la traducción, la dirección física obtenida debe coincidir con la dirección virtual, lo cual facilita la automatización de la creación de las tablas de páginas y la verificación de que la traducción se ha realizado correctamente.

Una vez creadas las tablas de páginas, se tiene que habilitar la MMU para lo cual se va a usar la configuración dada por defecto en la implementación del system-on-chip GR740.

Finalmente, se asignan ciertas entradas de direcciones conocidas en partes de memoria que no afecten a la ejecución del programa, y se realizan accesos a dichas direcciones modificando los diferentes atributos de la entrada de la tabla de página que las traducen con diferentes tipos de acceso.

En cada paso se imprime qué tipo de acceso a qué tipo de entrada se hace, el dato leído o escrito, y los registros de fallo de la MMU.

3.4.3 Estructura del programa de validación

El programa que se ha creado para validar la MMU se divide en varias partes:

- **defines.h**: da nombre a diferentes valores usados en la arquitectura SPARCv8, como vectores de *trap* y máscaras para campos de registros, mejorando la legibilidad del programa.

- **assembly.S**: ofrece varias funciones escritas en ensamblador usando la ISA SPARCv8 para realizar ciertas operaciones que solo pueden realizarse en ensamblador.

- **leon3_traps.c**: define una función que permite la creación y sobreescritura de *traps*. Se obtiene de los ejemplos dados en el manual de uso del compilador BCC.

- **linker.ld**: versión ligeramente modificada del *linker* dado originalmente por el compilador BCC, en el que se añade una nueva sección de instrucciones para realizar pruebas de *fetch*.

- **main.c**: contiene todas las funciones para la creación de tablas de páginas y pruebas.

3.4.4 Descripción del programa de Validación

3.4.4.1 Definiciones

Para mejorar la legibilidad del programa se crea la cabecera *defines.h*. Este archivo contiene un conjunto de macros que dan nombre a los diferentes vectores de *trap* y máscaras de los campos de los registros que se usan en la creación de este programa. Su código se muestra a continuación:

-- defines.h --

```
1   #ifndef DEFINES_H_
2   #define DEFINES_H_
3
4   /** Traps */
5   #define LEON3_MMUINSTERROR_TRAP 0x01
```

```
6   #define LEON3_MMUDATAERROR_TRAP 0x09
7   #define LEON3_LOAD_MMU_ASI_TRAP 0x90
8   #define LEON3_STORE_MMU_ASI_TRAP 0x91
9   #define LEON3_FLUSH_TRAP 0x92
10  #define LEON3_GETPSR_TRAP 0x93
11
12  /** Máscaras de registro psr */
13  #define S_MASK 0x00000080    // bit supervisor de PSR
14
15  /* Máscaras de entradas de tablas de páginas */
16  #define PET_MASK 0x3    // Máscara de campo ET de PTE
17  #define ACC_MASK 0x1C   // Máscara de campo ACC de PTE
18  #define REF_MASK 0x20   // Máscara de campo R de PTE
19  #define MOD_MASK 0x40   // Máscara de campo M de PTE
20  #define CACHE_MASK 0x80 // Máscara de campo C de PTE
21  #define PPN_MASK 0xFFFFFF00    // Máscara de campo PPN de PTE
22  #define PTP_MASK 0xFFFFFFFC    // Máscara de campo PTP de PTD
23
24  /* Valores de campos de entradas de tabla de páginas */
25  #define CACHEABLE 0x00000080   // Valor si es cacheable
26  #define MODIFIED 0x00000040    // Valor si es modificada
27  #define REFERENCED 0x00000020  // Valor si es referenciada
28  #define PG_ENTRY 0x00000002    // Valor si es entrada
29  #define PG_DESC 0x00000001     // Valor si es descriptor
30  #define PG_INVALID 0x00000000  // Valor si es inválida
31  #define ACC(num) (num << 2)    // Valor de acc
32
33  /* Máscaras del registro de estado de fallo */
34  #define FSR_EBE 0x0003FC00   // Máscara de campo EBE
35  #define FSR_LVL 0x00000300   // Máscara de campo L
36  #define FSR_ACC 0x000000E0   // Máscara de campo AT
37  #define FSR_TYPE 0x0000001C  // Máscara de campo FT
38  #define FSR_VALID 0x00000002 // Máscara de campo FAV
39  #define FSR_OW 0x00000001    // Máscara de campo OW
40
41  #endif /* DEFINES_H_ */
```

Código 23 - Definiciones del programa de validación - defines.h

3.4.4.2 Creación de tablas de páginas

En *main.c* se define primero la estructura de las tablas, asegurando que están alineadas, y se incorporan en un array para que sean fácilmente accesibles simplemente indicando el nivel de la tabla.

Luego se crea la función *createTables()*, que se encarga llenar dichas tablas con entradas descriptoras o entradas de página, creando un mapeado 1 a 1. El objetivo es mapear toda la memoria, pero que haya al menos 1 entrada de cada nivel para hacer pruebas.

El código que define estas estructuras y método de creación de tablas se muestra en *Código 24 - Estructura y creación de tablas de páginas del programa de validación - main.c*.

Para lograr esto, es necesario tener claro el proceso de traducción de la MMU mediante el recorrido de tablas de páginas, para para lo cual se puede usar de referencia lo descrito en el apartado *3.2.6.3 Recorrido de tablas de página (table walk)*.

De forma resumida, el recorrido de tablas de página parte de la tabla de contexto, de la que obtiene la entrada especificada por el contexto actual. A partir de ésta, si se encuentra una entrada descriptora (PTD), busca la entrada de la tabla de siguiente nivel usando la dirección dada por el PTD y el offset indicado por los bits de índice correspondientes de la dirección virtual. En caso de encontrarse una entrada de página (PTE), obtiene la base de la dirección física.

La especificación define direcciones de 36 bits, aún usando espacios de 32 bits para su almacenamiento. Para lograr esto, cada vez que se obtiene una dirección durante el recorrido de tablas de páginas, ésta se desplaza 4 bits a la izquierda antes de usarse. Es decir, las direcciones guardadas tanto en el registro de puntero de contexto, como las del campo PTP de las entradas descriptoras (PTD) y el campo PPN de las entradas de página (PTE), se desplazan 4 bits a la izquierda antes de usarlas.

Si, por ejemplo, se obtiene una entrada con valor 0x04000001, se realiza lo siguiente:

1. Comprueba el tipo de entrada (ET), que son los 2 últimos bits. En este caso ET vale 1 por lo que determina que es una entrada descriptora (PTD).
2. Como es descriptora, obtiene el campo PTP, rellenando con ceros a la derecha hasta tener 32 bits, lo que en este caso resulta en 0x04000000
3. **Desplaza 4 bits a la izquierda lo anterior**, resultando en 0x40000000, que, como estamos tratando un PTD, referencia la dirección de la primera entrada de la tabla de siguiente nivel.
4. Finalmente, como la entrada es descriptora, añade a la dirección anterior el índice correspondiente de la dirección virtual, lo cual dará la entrada de tabla de siguiente nivel.

En caso de ser entrada de página el proceso es el mismo, solo que se usa el campo PPN el cual da la base de la dirección física, y al final se le añade el offset de la dirección virtual. Estas entradas se describen detalladamente en el apartado *3.2.4 Tablas de páginas*.

Es decir, a la hora de crear las tablas de páginas, se deben almacenar las direcciones desplazadas 4 bits a la derecha. Como para esta prueba se busca realizar un mapeado 1 a 1, las tablas y sus entradas se pueden rellenar de la siguiente manera:

- Definir cada entrada como una estructura de N valores de 32 bits, donde N depende del nivel de entrada. Por ejemplo, la tabla de nivel 0 sería un array de 256 valores de 32 bits.
- Para crear una entrada descriptora, usar la dirección base de la tabla de siguiente nivel desplazada 4 bits a la derecha, y añadirle 1 para indicar que el tipo de entrada (ET) es descriptora.
- Para crear una entrada de página, usar la dirección del marco físico de memoria, asegurándose que su offset es 0, y desplazarla 4 bits a la derecha. Finalmente añadirle los atributos y poner su valor de tipo a 2, indicando que es entrada de página.

Una manera muy sencilla de mapear toda la memoria sería con una sola entrada de página en la tabla de contexto, pero en esta validación se busca probar varios niveles, por lo que se va a usar una tabla de cada nivel, que se asignan de la siguiente manera:

- Tabla de contexto: Cada entrada de esta tabla referencia 4 GiB de memoria, así que una entrada es suficiente para referenciar toda la memoria, por lo que se crea una sola entrada descriptora que apunta a la tabla de nivel 1. Esta se guarda en la entrada 0 de la tabla (es decir, para el contexto 0).

- Tabla de nivel 1: Cada entrada de esta tabla referencia 16 MiB de memoria, por lo que para mapear los 4 GiB se deben asignar todas las entradas, por lo tanto, se crean 256 entradas de página, excepto la entrada 64 (contando desde 0) que es una descriptora que apunta a una tabla de nivel 2. Se usa la entrada 64 por que ésta se traduce a la dirección 0x40000000 que es a partir de donde se guardan las instrucciones y datos del programa. (64 se usa en el *index1* que en hexadecimal es 0x40).

- Tabla de nivel 2: Cada entrada de esta tabla referencia 256 KiB, y como se quiere mapear el boque completo, se llena entera con 64 entradas de página, excepto la entrada 0 que es una entrada descriptora que apunta a la tabla de nivel 3. Como antes, esta entrada al traducirse da la dirección 0x40000000.

- Tabla de nivel 3: Cada entrada referencia 4 KiB de memoria, que al igual que antes, se llena entera con 64 entradas de página. Como esta tabla viene apuntada por la de nivel 2 anterior, que a su vez la apunta la de nivel 1, y cada entrada abarca 4 KiB o 0x1000 bytes, entonces esta tabla engloba las direcciones desde la 0x40000000 a la 0x4003FFFF.

-- Creación de tabla de páginas del programa de validación --

```
1   typedef struct table {
2       uint32_t *entry;
3   } table;
4
5   #define TABLE_STRUCT(x) { &(x) }
6   struct __attribute__ ((packed, aligned(1024))) ctxTable {
7           uint32_t entry[256];
8   } ctxTable;
9
10  struct __attribute__ ((packed, aligned(1024))) lv1Table {
11          uint32_t entry[256];
12  } lv1Table;
13
14  struct __attribute__ ((packed, aligned(1024))) lv2Table {
15          uint32_t entry[64];
16  } lv2Table;
17
18  struct __attribute__ ((packed, aligned(1024))) lv3Table {
19          uint32_t entry[64];
20  } lv3Table;
21
22  table tables[] = {TABLE_STRUCT(ctxTable), TABLE_STRUCT(lv1Table),
23  TABLE_STRUCT(lv2Table), TABLE_STRUCT(lv3Table)};
24
```

```
/* Función de creación de tablas
 * Tabla contexto, entrada 0 = PTD a tabla de nivel 1
 * Tabla nivel 1, entrada 64 = PTD a tabla de nivel 2 (0x40000000 a 0x40FFFFFF)
 * Tabla nivel 1, resto de entradas = PTE's de 0x0 a 0xFFFFFFFF
 * Tabla nivel 2, entrada 0 = PTD a tabla de nivel 3
 * Tabla nivel 2, entrada 1 a 63 = PTE's de 0x40040000 a 0x40FFFFFF
 * Tabla nivel 3, entrada 0 a 63 = PTE's de 0x40000000 a 0x4003FFFF
 */
void createTables(){

	uint32_t Table_lvl1_address = (uint32_t)&lv1Table;
	uint32_t Table_lvl2_address = (uint32_t)&lv2Table;
	uint32_t Table_lvl3_address = (uint32_t)&lv3Table;

	// Entrada 0 de tabla de contexto -> descriptor de pagina a tabla de nivel 1.
	uint32_t Table0_E0 = ((Table_lvl1_address >> 4) & ~PET_MASK)| PG_DESC;
	ctxTable.entry[0] = Table0_E0;


	// Entradas de tabla de nivel 1. La número 64 apunta a la tabla de nivel 2,
	// el resto son entradas de página.
	// Primero se define el tamaño de las entradas de tabla de nivel 1 y creo
	// una entrada de página genérica con todos los permisos (acc 3)
	uint32_t Table_lvl1_PageSize = 0x01000000; // tamaño de cada entrada de lvl1
	uint32_t Table_PageEntry = CACHEABLE | ACC(3) | PG_ENTRY; // PTE genérica

	int i = 0;
	for(i = 0; i < 256; i++){    // llena la tabla de nivel 1 con la PTE generica
		lv1Table.entry[i] = ((Table_lvl1_PageSize * i) >> 4) | Table_PageEntry;
	}

	// Y la entrada 64 la sustituyo por el PTD que apunta a la tabla de nivel 2
	uint32_t Table1_E64 = ((Table_lvl2_address >> 4) & ~PET_MASK) | PG_DESC;
	lv1Table.entry[64] = Table1_E64;


	// Entradas de tabla de nivel 2. La número 0 apunta a la tabla de nivel 3,
	// el resto son entradas de página.
	// Comienza asignando la entrada 0 como PTD a la tabla de nivel 3
	uint32_t Table2_E0 = ((Table_lvl3_address >> 4) & ~PET_MASK) | PG_DESC;
	lv2Table.entry[0] = Table2_E0;

	// Y, tras definir el tamaño de cada entrada de nivel 2 y donde comienza
	uint32_t Table_lvl2_PageSize = 0x40000;
	uint32_t Table_lvl2_Ini = 0x40000000;
	for(i = 1; i < 64; i++){     // rellena el resto de entradas de página
		lv2Table.entry[i] = ( (Table_lvl2_Ini+(Table_lvl2_PageSize*i)) >> 4)
				| Table_PageEntry;
	}

	// Entradas de tabla de nivel 3. Todas serán entradas de página
	// Defino el tamaño de entrada y donde comienza
	uint32_t Table_lvl3_PageSize = 0x1000;
	uint32_t Table_lvl3_Ini = 0x40000000;
	for(i = 0; i < 64; i++){     // y rellena la tabla
		lv3Table.entry[i] = ( (Table_lvl3_Ini+(Table_lvl3_PageSize*i)) >> 4)
				| Table_PageEntry;
	}
}
```

Código 24 - Estructura y creación de tablas de páginas del programa de validación - main.c

3.4.4.3 Funciones en ensamblador

Hay ciertas operaciones que solo pueden realizarse empleando el lenguaje ensamblador, o que en general resultan mucho más sencillas de esta manera. Un ejemplo es el acceso a los registros de la MMU, lo cual se debe hacer mediante una instrucción de load o store al ASI 0x19.

Además, muchas de estas operaciones solo se pueden en modo supervisor. Puesto que varias pruebas se realizan en modo usuario, es necesario realizar el acceso a estas funciones a través de un *trap*. Por ejemplo, como se muestra en *Código 25 - Función de lectura de registro de MMU - assembly.S*, para el caso de lectura de registros de MMU se crea la función *trap_call_loadMMUControlAsi* cuya primera instrucción es un *TA Vector*, lo cual provoca un *trap a dicho vector*. En la entrada correspondiente de la tabla de vectores se coloca un código que salta al método *trap_loadMMUControlASI_handler*, el cual realiza la lectura del registro de la MMU asociado.

Todas estas funciones se crean dentro del archivo *assembly.S* y se definen en el *main* usando el comando *extern*, lo cual hace que sea posible llamarlas como a cualquier otra función.

-- Función en ensamblador de lectura de registros de MMU --

```
1    // Lectura de registros de MMU
2        .globl trap_loadMMUControlASI_handler
3    trap_loadMMUControlASI_handler:
4    mov %l2, %l1
5    add %l2, 4, %l2     // Crea dirección de retorno
6
7    lda [%i0]0x19, %i0  // Instrucción de carga con Asi 0x19, dirección i0
8
9    jmp %l1
10   rett %l2 // Retorna usando la instrucción rett ya que esta función se llama como trap
11
12
13   // Trap para lectura de registros de MMU
14       .globl trap_call_loadMMUControlASI
15   trap_call_loadMMUControlASI:
16   ta LEON3_LOAD_MMU_ASI_TRAP // Genera trap del vector asignado a la lectura de
17                              // registros de MMU
18   retl // Vuelve
19   nop
```

Código 25 - Función de lectura de registro de MMU - assembly.S

3.4.4.4 Tratamiento de excepciones

Durante esta validación se van provocar fallos de MMU, lo que a su vez va a generar señales de *trap*. Estas se deben definir y tratar adecuadamente de tal forma que no se detenga la ejecución del programa.

Las excepciones provocadas por la MMU activan los vectores *trap* 0x01 y 0x09 para errores en área de instrucciones y datos respectivamente. Por lo que, para obligar a que continúe el programa tras dichas excepciones, es necesario configurar esos *traps* con manejadores que retornen al

programa principal. En *Código 26 - Función para el tratamiento del trap 0x09 - assembly.S* se muestra el código del manejador creado para la excepción correspondiente a un error de área de datos.

Usando la función de *leon3_install_handler* del archivo *leon3_traps.c*, implementada como parte de la biblioteca de funciones del compilador BCC, es posible configurar las entradas de la tabla de *traps*. De esta forma, hemos creado en ensamblador unos métodos que retornan al programa principal al final de su ejecución. Estos métodos tienen asignados los *traps* 0x01 y 0x09. Se adjunta el código de instalación de manejadores de traps en *Código 27 - Función para instalar manejadores de trap - leon3_traps.c*, y como se usa éste desde el método principal en *Código 28 - Instalación de los manejadores de trap - main.c*.

-- Función de trap 0x09 (Error en área de datos) --

```
1    // Función para el tratamiento de trap 0x09, imprime "Trap 0x09" por pantalla
2        .globl trap_SRMMUData_Error_handler
3    trap_SRMMUData_Error_handler:
4        mov 0x54, %i0
5        call write_char
6        nop
7        mov 0x72, %i0
8        call write_char
9
10       (...)
11
12       jmp %l2
13       rett %l2+4
14       nop
```

Código 26 - Función para el tratamiento del trap 0x09 - assembly.S

-- Función de instalación de manejadores de trap --

```
1    uint32_t _rdtbr();
2
3    asm(".text\n"
4    "_rdtbr:\n"
5    "    retl \n"
6    "    mov %tbr, %o0");
7
8    int32_t leon3_install_handler(uint32_t trap_num, void (* handl_routine) (void))
9    {
10          uint32_t handler_routine=(uint32_t)handl_routine;
11          uint32_t * trap_address;
12          uint32_t tbr = _rdtbr();
13
14          trap_address = (uint32_t *) ((tbr & ~0x0fff) | ((uint32_t) trap_num << 4));
15          *trap_address = 0xA1480000UL;
16          trap_address++;
17          *trap_address = (0x29000000UL | (handler_routine >> 10));
18          trap_address++;
19          *trap_address = (0x81C52000UL | ((handler_routine & ((1 << 10) - 1))));
20          trap_address++;
21          *trap_address = (0xA6102000UL | (trap_num & 0x1FFF));
22
23       return LEON3_ERR_SUCCESS;
24   }
```

Código 27 - Función para instalar manejadores de trap - leon3_traps.c

-- Instalación de manejadores de trap --

```
int main() {
        leon3_install_handler(LEON3_MMUINSTERROR_TRAP, trap_SRMMUInst_Error_handler);
        leon3_install_handler(LEON3_MMUDATAERROR_TRAP, trap_SRMMUData_Error_handler);
        leon3_install_handler(LEON3_LOAD_MMU_ASI_TRAP, trap_loadMMUControlASI_handler);
        leon3_install_handler(LEON3_STORE_MMU_ASI_TRAP,
                              trap_writeMMUControlASI_handler);
        leon3_install_handler(LEON3_FLUSH_TRAP, trap_flushMMU_handler);
        leon3_install_handler(LEON3_GETPSR_TRAP, trap_getPSR_handler);

         (...)

}
```

Código 28 - Instalación de los manejadores de trap - main.c

3.4.4.5 Funciones para las pruebas de acceso a memoria

A la hora de realizar pruebas de lectura y escritura en área de datos solo es necesario crear un puntero a una dirección de memoria que no contenga instrucciones ni datos del programa, y escribir o leer un dato de dicha dirección. También sería posible usar los *ASI* 0x0A y 0x0B, obteniendo el mismo resultado. Para estas pruebas se usan la entrada 63 de la tabla de nivel 3 (que se traduce a la dirección 0x4003F000) y la entrada 1 de la tabla de nivel 2 (que se traduce a la dirección 0x40040000).

Para poder realizar pruebas en el área de instrucciones (*fetch*) es necesario hacerlo en la sección de memoria definida para ello, la cual se denomina *text*. Esto genera el problema de que, en caso de realizar una prueba que genere una excepción en esta área, le será imposible volver a la ejecución normal del programa, ya que la siguiente instrucción a ejecutar está en la misma área, lo cual provocaría una segunda excepción y, por lo tanto, un error fatal del procesador.

Por defecto, el enlazador (*linker*) ajusta el tamaño del área de *text* a la cantidad de instrucciones del programa, por lo que será necesario crear una segunda área de instrucciones para poder realizar pruebas de *fetch*.

Entonces, se crea una nueva área de instrucciones entre el área original y el área de datos llamada *itst*, tal y como se puede ver en el extracto de código mostrado en *Código 29 - definición de sección itst - linker.ld*. Se define esta área de forma que comience en la dirección de *offset* 0x20000 hasta 0x21000. Como el área de instrucciones comienza en 0x40000000, entonces la sección *itst* comprenderá desde 0x40020000 hasta 0x40021000, que corresponde con la entrada 32 de la tabla de página de nivel 3 (*index3* = 100000b = 32).

-- Definición de la sección itst --

```
SECTIONS
{
  .text   : {

    (...)

    . = 0x20000;
    *(.itst)
    . = ALIGN (16);
    . = 0x21000;
    etext  =  .;
  }  > ram
(...)
```

Código 29 - definición de sección itst - linker.ld

Dentro de la nueva área de instrucciones se introduce el método *tlbi_test* el cual simplemente retorna al llamante, como se muestra debajo:

-- Función de prueba de fetch --

```
.section ".itst"

    .globl tlbi_test
tlbi_test:
    nop
    retl
    nop
```

Código 30 - función de prueba de fetch definida para el área itst - assembly.S

Aún con una nueva área para instrucciones, sigue existiendo el problema de que, si se produce una excepción, el manejador retorna a la misma área provocando múltiples excepciones. Para evitar esto se hace que el manejador de excepciones por acceso a área de instrucciones retorne a la dirección indicada en su primer argumento. Finalmente, se crea el método *fetch()* para la realización de pruebas en el área de instrucciones, el cual llama al método *tlbi_test* poniendo de argumento la dirección siguiente a ésta llamada. Es decir, cuando se quiere realizar una prueba de *fetch*, se siguen los siguientes pasos:

1. Se modifica la entrada de página que se traduciría a la sección *itst*.
2. Se llama a el método *fetch()*.
3. El método *fetch()* llama a tlbi_test, poniendo como argumento la dirección de la siguiente instrucción a esta llamada.
4. En caso de producirse una excepción, se llama a el manejador de excepciones por acceso a área de instrucciones, el cual retorna a la dirección que puso de argumento el método *fetch()*, haciendo que se vuelva a la instrucción siguiente a la llamada a *tlbi_test*.
5. En caso de no haber excepción, se ejecuta el método *tlbi_test* de manera normal, el cual también retorna a la instrucción siguiente a su llamada.

Ambos, el método de *fetch()* y manejador de excepciones por acceso a área de instrucciones se muestran a continuación:

-- Función para prueba de fetch --

```
void fetch(){
        uint32_t rAdd;
        rAdd = &&ra;     // obtiene la dirección de retorno usando etiqueta 'ra'
        tlbi_test(rAdd); // llamada a función de test de fetch
        ra:              // etiqueta para obtener la dirección de retorno
        nada();          // función vacía, debe haber algo tras una etiqueta
}
```

Código 31 - función para test de fetch - main.c

-- Función de trap 0x01 (Error en área de instrucciones) --

```
// Función para el tratamiento de trap 0x01, imprime "Trap Inst" por pantalla
        .globl trap_SRMMUInst_Error_handler
trap_SRMMUInst_Error_handler:
    mov %i0, %l2 //Solo para tests, obtiene el argumento pasado que contiene la
                        dirección de retorno a la función de fetch de main.c
    mov 0x54, %i0
    call write_char
    nop

    (...)

    jmp %l2
    rett %l2+4
    nop
```

Código 32 - Función para el tratamiento del trap 0x01 - assembly.S

Para facilitar la realización de múltiples pruebas de acceso a entradas con diferentes atributos, se ha creado la función *TestEntry*, que recibe como argumentos el nivel al que se quiere hacer la prueba, los atributos de la entrada de página y el dato a leer o escribir. También se da la opción de si hacer prueba de *fetch* o no. Ésto es importante porque también se van a probar resultados con el campo *No Fault* de la MMU a 1, el cual hace que se ignoren ciertos fallos, pero ignorar fallos de *fetch* puede provocar el *trap* 0x02 por intentar ejecutar una instrucción no implementada.

La función de *TestEntry* crea una nueva entrada de página con los atributos dados y la sustituye en la entrada correspondiente de la tabla de páginas. A continuación, realiza varias pruebas de lectura y escritura, probando un acceso y dos accesos seguidos, haciendo un *flush* de las cachés en cada acceso para que las pruebas anteriores no afecten a la actual. Al inicio de la prueba, imprime un resumen de cuál es su objeto, y por cada acceso se imprimen los registros de fallo de MMU para comprobar que dan el resultado esperado. A continuación, se adjunta el código que define los textos a imprimir y el método de pruebas de entradas:

-- Funciónes y constantes para imprimir resultados --

```
const char * txtPageType[4] = {"invalid entry", "page descriptor",
                               "page entry", "Reserved"};

const char * txtAccAllow[8] = {"RD for User and Supervisor",
                               "RD/WR for User and Supervisor",
                               "RD/X for User and Supervisor",
                               "RD/WR/X for User and Supervisor",
```

```
                                "X for User and Supervisor",
                                "RD for User || RD/WR for Supervisor",
                                "NO access for User || RD/X for Supervisor",
                                "NO access for User || RD/WR/X for Supervisor"};

const char * txtPriv[2] = {"as User", "as Supervisor"};

const char * txtAttAcc[8] = {"Load from user Data Space",
                             "Load from Supervisor Data Space",
                             "Load/Execute from User Instruction Space",
                             "Load/Execute from Supervisor Instruction Space",
                             "Store to User Data Space",
                             "Store to Supervisor Data Space",
                             "Store to User Instruction Space",
                             "Store to Supervisor Instruction Space"};

const char * errType[8] = {"None",
                           "Invalid Address Error",
                           "Protection Error",
                           "Privilege violation error",
                           "Translation Error",
                           "Access bus Error",
                           "Internal Error",
                           "Reserved"};

void printFault(){
       uint32_t FaultReg = trap_call_loadMMUControlASI(0x300);
       printf("MMU Fault Status:     0x%x\n", FaultReg);
       printf("MMU Fault Address:    0x%x\n\n", trap_call_loadMMUControlASI(0x400));
       printf("Error Description:\n");
       printf("External Bus: 0x%x\n", (FaultReg & FSR_EBE) >> 10);
       printf("Entry Level: %u\n", (FaultReg & FSR_LVL) >> 8);
       printf("Attempted Access: %s\n", txtAttAcc[(FaultReg & FSR_ACC) >> 5]);
       printf("Error Type: %s\n", errType[(FaultReg & FSR_TYPE) >> 2]);
       printf("Valid Address: %u\n", (FaultReg & FSR_VALID) >> 1);
       printf("Overwritten: %u\n\n", (FaultReg & FSR_OW) >> 0);
}

void printEntry(uint32_t entry){
       printf("Entry Type: %s\n", txtPageType[(entry & PET_MASK)]);
       printf("Permissions: %s\n", txtAccAllow[(entry & ACC_MASK)>>2]);
       printf("Ref: %u, Mod: %u, Cacheable: %u\n", (entry & REF_MASK)>>5,
              (entry & MOD_MASK)>>6,(entry & CACHE_MASK)>>7);
       printf("PPN shifted: 0x%x\n", (entry & PPN_MASK)<<4);
       printf("PTP shifted: 0x%x (If Descriptor)\n\n", (entry & PTP_MASK)<<4);
}
```

Código 33 - constantes y funciones para la impresión de resultados - main.c

-- Función de test de entradas --

```
const uint32_t test_address[4] = {0,0x41000000,0x40040000,0x4003F000};
const uint32_t entry_Num[4] = {0,65,1,63};

void TestEntry(uint32_t level, uint32_t isCacheable, uint32_t access,
                        uint32_t pageType, uint32_t f, uint32_t dato){

       uint32_t *dir = (uint32_t *) test_address[level];
       tstStore();

       uint32_t original = tables[level].entry[entry_Num[level]]; // Entrada de datos
       uint32_t pageEntry = (test_address[level] >> 4) | (isCacheable<<7) |
```

```
                                        ACC(access) | pageType;

        tables[level].entry[entry_Num[level]] = pageEntry;

        uint32_t originalI = tables[3].entry[32]; // Entrada de fetch
        pageEntry = (originalI & PPN_MASK) | (isCacheable<<7) | ACC(access) | pageType;
        tables[3].entry[32] = pageEntry;

        trap_call_flushMMU(); // Flush Caché

        printf("################ ENTRY ################\n");
        printEntry(tables[level].entry[entry_Num[level]]);
        printf("\nLvl%u Entry. address 0x%x\n", level, test_address[level]);
        printf("Reading and Writting %s using Pointers\n", txtPriv[priv]);
        printf("#########################################\n\n");

        printf("############## WRITE ################\n");
        printf("--Writing 0x%x %s in entry at 0x%x\n\n", dato, txtPriv[priv]
               ,test_address[level]);

        *dir = dato;
        printf("MMU Control Register: 0x%x\n", trap_call_loadMMUControlASI(0));
        trap_call_flushMMU();
        printEntry(tables[level].entry[entry_Num[level]]);
        printFault();

        printf("**Second read of Fault register**: \n");
        printFault();

        trap_call_flushMMU();
        printf("--Writing %s 2 times without flushing\n\n", txtPriv[priv]);
        *dir = dato;
        *dir = dato;
        printFault();

        printf("**Second read of Fault register without flushing**: \n");
        printFault();

        trap_call_flushMMU();
        printf("################ READ ################\n");
        printf("--Reading %s entry at 0x%x\n", txtPriv[priv],test_address[level]);
        printf("Value Read: 0x%x\n\n", *dir);
        printf("MMU Control Register: 0x%x\n", trap_call_loadMMUControlASI(0));
        printFault();

        if (f){
               trap_call_flushMMU();
               printf("--Fetch %s entry at 0x40020000\n", txtPriv[priv]);
               fetch();
               printf("MMU Control Register: 0x%x\n", trap_call_loadMMUControlASI(0));
               printFault();

               trap_call_flushMMU();
               printf("--Double fetch %s without flushing\n", txtPriv[priv]);
               fetch();
               fetch();
               printFault();

               printf("**Second read of Fault register without flushing**: \n");
               printFault();

```

```
                trap_call_flushMMU();
                printf("--fetch after read %s without flushing\n", txtPriv[priv]);
                printf("--Reading %s entry at 0x%x\n", txtPriv[priv],
                        test_address[level]);
                fetch();
                printFault();

                printf("**Second read of Fault register without flushing**: \n");
                printFault();
        }

        trap_call_flushMMU();
        tables[level].entry[entry_Num[level]] = original;
        tables[3].entry[32] = originalI;
}
```

Código 34 - Función de pruebas de acceso - main.c

3.4.4.6 Inicio y realización de pruebas

Con todo ya definido para la creación de tablas de páginas y realización de pruebas, lo único que falta es llamar a las diferentes funciones y comparar resultados. En la función *main*, tras la instalación de los manejadores de *trap*, se llama a la función de creación de tablas seguido de las funciones para la escritura de registros de la MMU, para establecer el puntero de tabla de contexto, el contexto y finalmente habilitar la MMU. Tras ello, se llama a la función de pruebas de acceso con diferentes argumentos, cambiando entre supervisor y usuario, y el bit de *No Fault* para comprobar distintas opciones. A continuación, se muestra un extracto del código del método principal que realiza lo comentado.

-- Función principal del programa de validación --

```
int main() {
        // Instalación de manejadores de trap
        leon3_install_handler(LEON3_MMUINSTERROR_TRAP, trap_SRMMUInst_Error_handler);
        leon3_install_handler(LEON3_MMUDATAERROR_TRAP, trap_SRMMUData_Error_handler);
        leon3_install_handler(LEON3_LOAD_MMU_ASI_TRAP, trap_loadMMUControlASI_handler);
        leon3_install_handler(LEON3_STORE_MMU_ASI_TRAP,
                              trap_writeMMUControlASI_handler);
        leon3_install_handler(LEON3_FLUSH_TRAP, trap_flushMMU_handler);
        leon3_install_handler(LEON3_GETPSR_TRAP, trap_getPSR_handler);

        createTables(); // Creates Table entries

        // Establece el registro de puntero de contexto a la tabla de nivel 0
        trap_call_writeMMUControlASI(0x100, ((uint32_t)&ctxTable) >> 4);
        trap_call_writeMMUControlASI(0x200, 0); // Pone el registro de contexto a 0
        trap_call_flushMMU(); // realiza un flush de la MMU
        trap_call_writeMMUControlASI(0, 0x01904001); // Habilita MMU

        printf("MMU Control Register: 0x%x\n", trap_call_loadMMUControlASI(0));
        printf("MMU Context Pointer:  0x%x\n", trap_call_loadMMUControlASI(0x100));
        printf("MMU Context Number:   0x%x\n\n", trap_call_loadMMUControlASI(0x200));

        // Pruebas en modo supervisor con bit No Fault a 0
        printf("###############################################\n");
```

```
25          printf("#####  Tests as Superv using pointers  #####\n");
26          printf("#####         and No Fault = 0         #####\n");
27          printf("#############################################\n\n");
28          trap_call_writeMMUControlASI(0, 0x01904001); // No fault to 0
29          TestEntry(3, 0, 0, 2, 1, 0x1111); //lvl 3, ACC 0, PTE
30          TestEntry(3, 0, 1, 2, 1, 0x2222); //lvl 3, ACC 1, PTE
31          TestEntry(3, 0, 4, 2, 1, 0x3333); //lvl 3, ACC 4, PTE
32          TestEntry(3, 0, 7, 2, 1, 0x4444); //lvl 3, ACC 7, PTE
33          TestEntry(3, 0, 3, 0, 1, 0x5555); //lvl 3, ACC 3, invalid
34          TestEntry(3, 0, 3, 1, 1, 0x6666); //lvl 3, ACC 3, PTD
35          TestEntry(3, 0, 3, 3, 1, 0x7777); //lvl 3, ACC 3, PTE
36          TestEntry(2, 0, 0, 3, 1, 0x8888); //lvl 2, ACC 0, Reserved
37
38          (...)
39
40          // Pruebas en modo supervisor con bit No Fault a 1
41          printf("#############################################\n");
42          printf("#####  Tests as Superv using pointers  #####\n");
43          printf("#####         and No Fault = 1         #####\n");
44          printf("#############################################\n\n");
45          trap_call_writeMMUControlASI(0, 0x01904003); // No fault to 1
46          TestEntry(3, 0, 0, 2, 1, 0x1111);
47          TestEntry(3, 0, 1, 2, 1, 0x2222);
48          TestEntry(3, 0, 4, 2, 1, 0x3333);
49          TestEntry(3, 0, 7, 2, 1, 0x4444);
50          TestEntry(3, 0, 3, 0, 1, 0x5555);
51          TestEntry(3, 0, 3, 1, 1, 0x6666);
52          TestEntry(3, 0, 3, 3, 1, 0x7777);
53          TestEntry(2, 0, 0, 3, 1, 0x8888);
54
55          (...)
56
57          setUsuario(); // Cambia a modo usuario
58
59          (...) //Mas pruebas, ahora en modo usuario
60
61          printf("fin\n");
62
63          return 0;
64      }
```

Código 35 - Función principal del programa de validación - main.c

3.4.5 Resultados de validación

La ejecución del programa anterior genera un texto con varios miles de líneas, por lo que para facilitar la comprobación se ha utilizado un plugin del programa *notepad++* que compara ficheros de texto marcando las diferencias entre ambos:

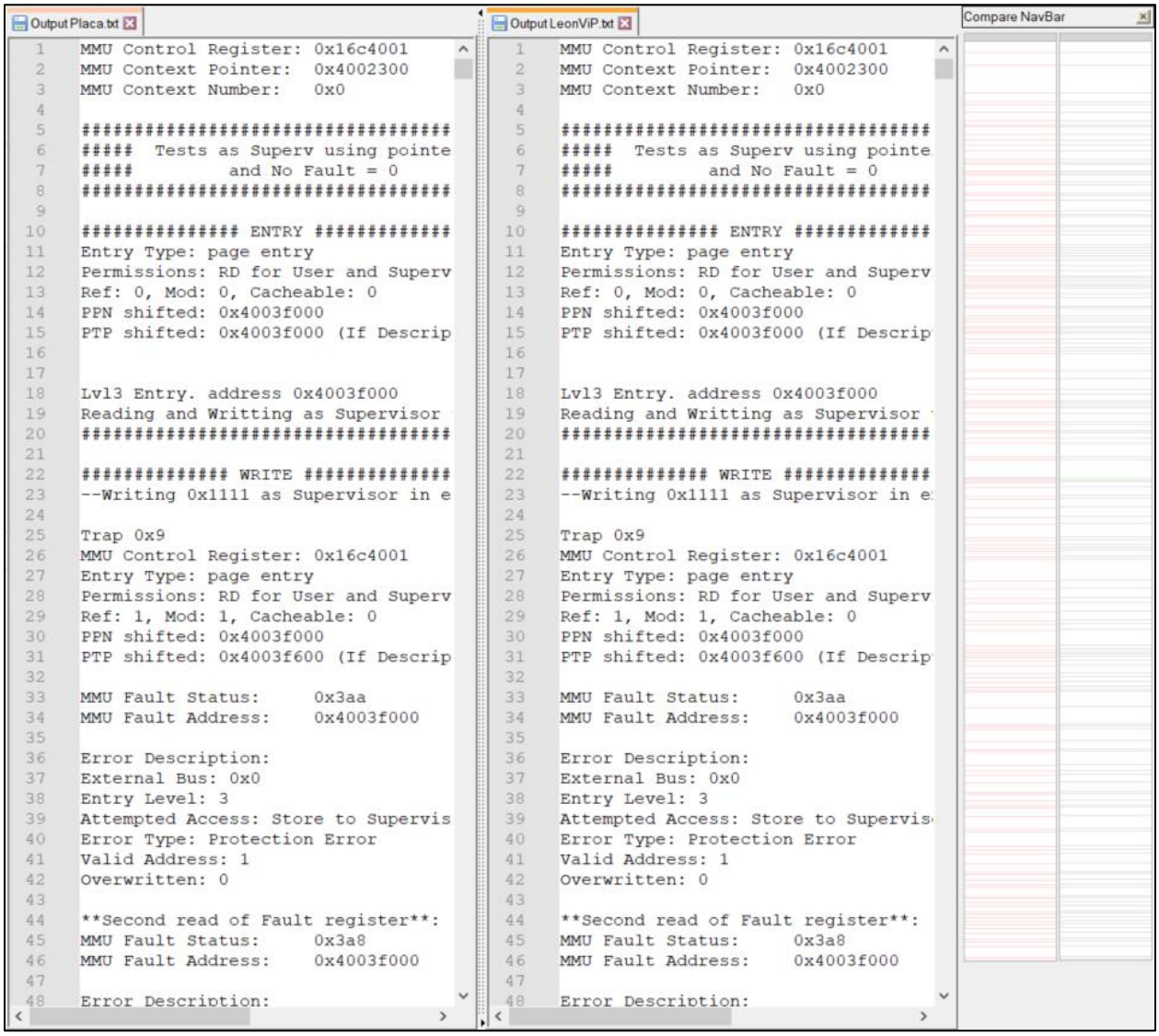


Ilustración 8 - Comparación de resultados de inicio y escritura

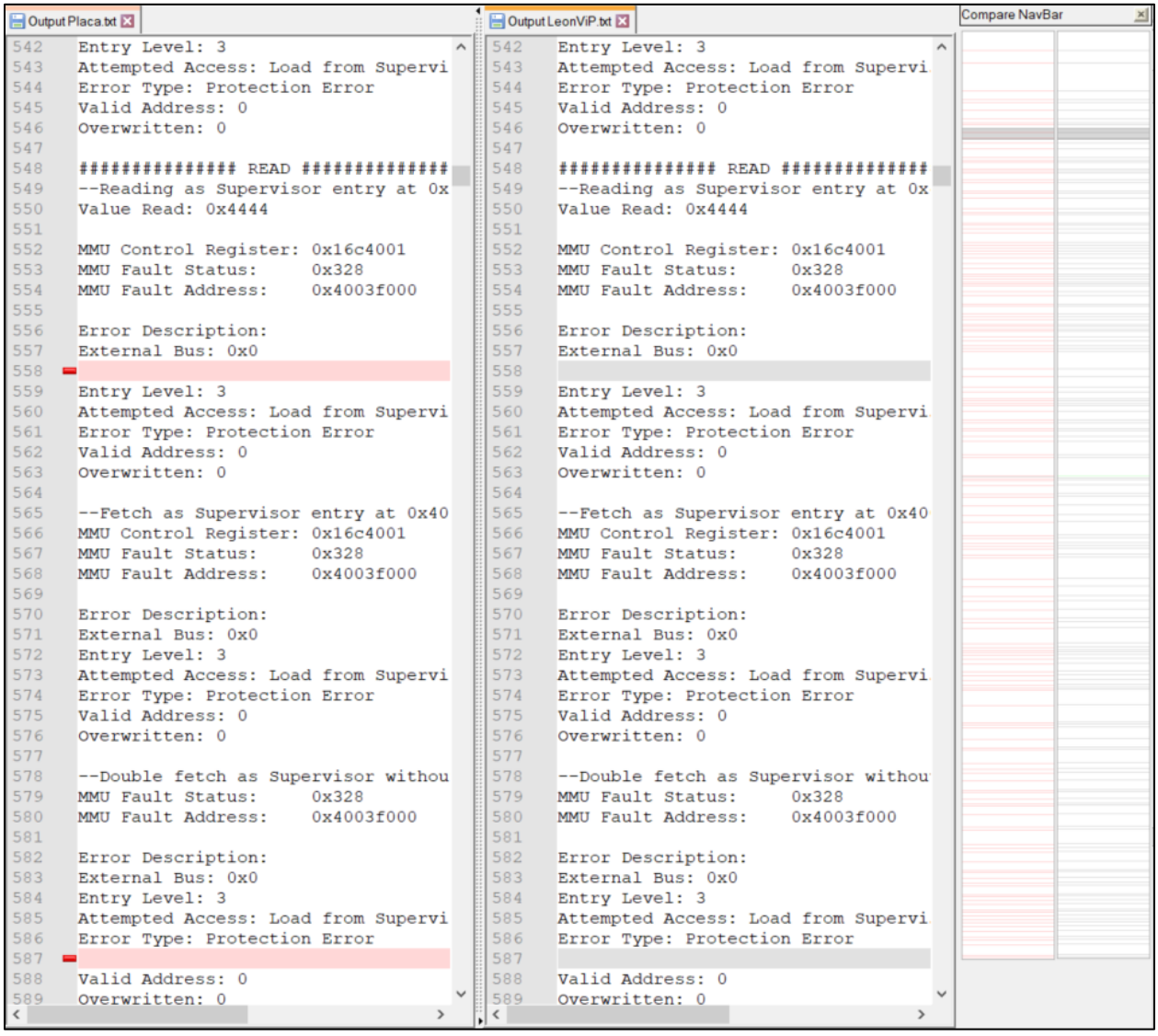


Ilustración 9 - Comparación de resultados en lectura

Las ilustraciones anteriores muestran la comparación entre el resultado dado por el simulador LeonViP y la implementación de LEON3 en la placa. A la derecha se muestra, con los colores rojo y gris, las líneas que no coinciden al comparar ambos textos. En todos los casos las diferencias siempre son debidas a saltos de línea que se imprimen en la salida de la placa y no en el LeonViP.

Como, a excepción de saltos de línea extras, las salidas de ambos coinciden, se concluye que el simulador LeonViP se está comportando igual que la implementación de LEON3.

4. Desarrollo del AWP

4.1 Concepto

La ISA SPARCv8 define un método específico para el almacenamiento de los datos globales y el paso de parámetros entre funciones. Este mecanismo, que persigue mejorar el rendimiento de las aplicaciones evitando usar, en la medida de lo posible, una pila en memoria, consiste en la agrupación de los registros de propósito general del procesador en ventanas. Cada vez que se produce una llamada a la una función, el programa cambia la ventana de registros en uso, la cual se indica usando un puntero llamado puntero de ventana actual (*current window pointer* o CWP) contenido en el registro PSR.

Como hay un número limitado de ventanas, se puede dar el caso en el que el programa alcance una profundidad de llamadas suficiente y que se quede sin ventanas libres. Para solucionar esto, se usa el registro de máscara de ventana inválida (*Window invalid Mask* o WIM). Este registro permite marcar qué ventana es la última libre.

El puntero de ventana alternativo (*alternative window pointer* o AWP) es una nueva funcionalidad especificada por la versión extendida de SPARCv8 que aumenta la flexibilidad del sistema de ventanas de registros, añadiendo un segundo puntero y permitiendo particionar el conjunto de ventanas de registros.

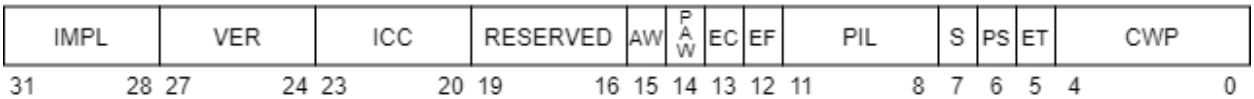
4.2 Especificación

El aspecto más importante para poder implementar esta nueva funcionalidad es entender cómo funciona el mecanismo de las ventanas de registros, ya que el AWP es una extensión de éste. Por ello, hemos hecho primero un análisis a fondo de como funcionan las ventanas de registros, que seguidamente se ha extendido con la especificación del AWP.

4.2.1 Registros

A continuación, se describen los registros del sistema que se usan para poder manejar las ventanas de registros. En la descripción de los campos se marcan en negrita los que afectan al sistema de ventanas.

4.2.1.1 Registro de estado del procesador (processor state register o PSR)

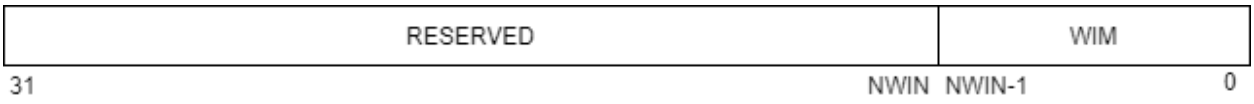


Campo	Bits	Descripción	R/W	Reset
IMPL	28..31	ID de implementación, fijado en hardware a '1111' (15)	R	0xF
VER	24..27	Versión de implementación, fijado en hardware a '0011' (3)	R	0x3

ICC	20..23	Códigos de condición de enteros.	R	0
RES	16..19	Reservado.	R	0
AW	15	Selección de puntero alternativo. *Es de solo lectura si AWP no está implementado.	RW*	0
PAW	14	Selección de puntero alternativo previo. *Es de solo lectura si AWP no está implementado.	RW*	0
EC	13	Habilitar coprocesador. Es de solo lectura si el coprocesador no está implementado.	R	0
EF	12	Habilitar punto flotante. *Es de solo lectura si la FPU no está implementada.	RW*	0
PIL	8..11	Nivel de interrupción de procesador, indica el menor número de interrupción que puede generar un *trap*.	RW	0
S	7	Indica si se está en modo supervisor.	RW	1
PS	6	Indica si el modo previo era supervisor.	RW	1
ET	5	Habilitar *traps*.	RW	0
CWP	0..4	Puntero de ventana actual.	RW	0

Tabla 21 - Registro de estado del procesador - PSR

4.2.1.2 Registro de máscara de ventana inválida (Window invalid mask o WIM)



Campo	Bits	Descripción	R/W	Reset
RES	NWIN..31	Reservado.	R	0
WIM	0..NWIN-1	Máscara de ventana inválida. Marca con '1' el bit que corresponde con el número de la última ventana libre. Su longitud depende del número de ventanas habilitadas, por defecto 8 ventanas, haciendo este campo de 8 bits (0..7).	RW	NR

Tabla 22 - Registro de máscara de ventana inválida - WIM

4.2.2 Sistema de ventanas de registros

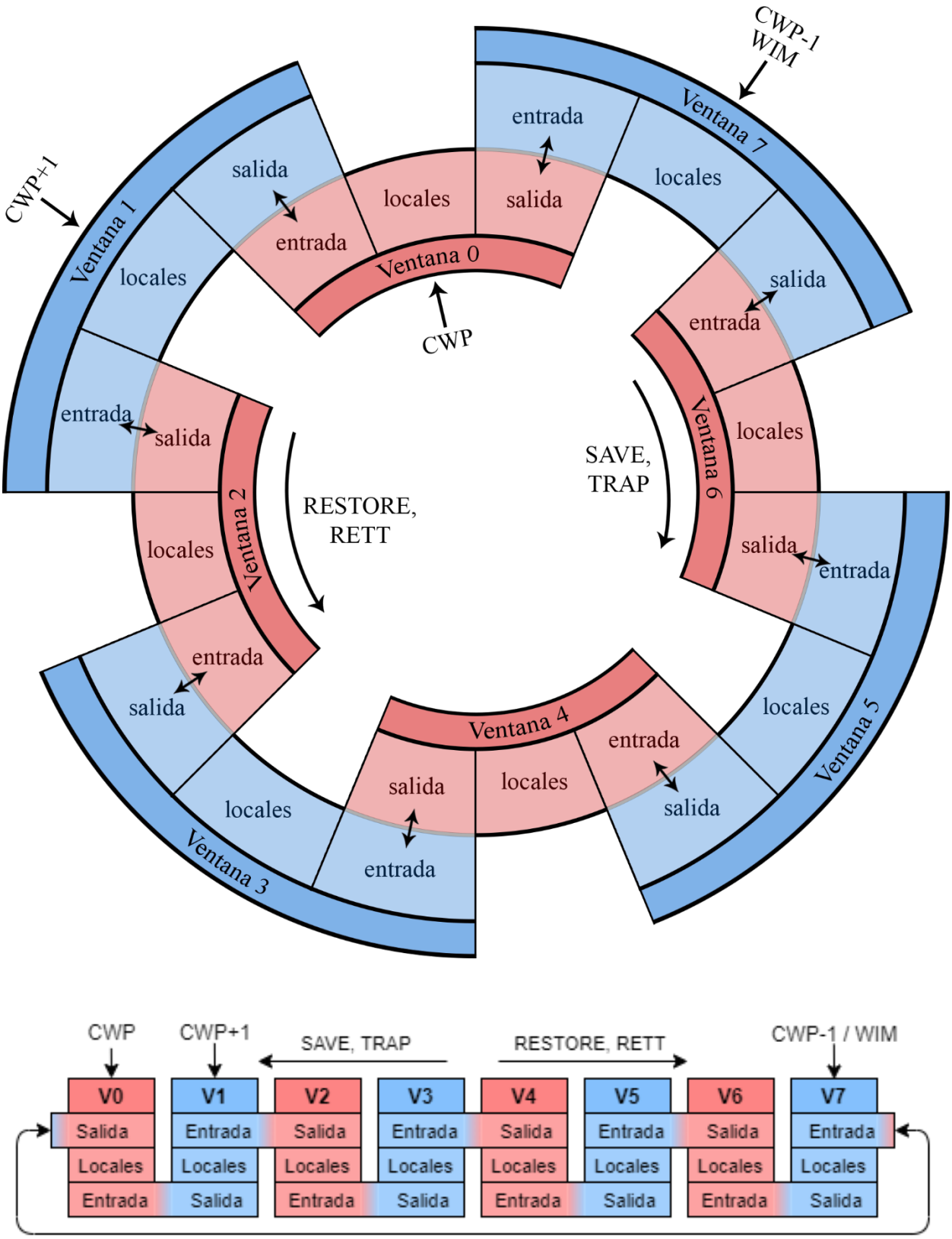


Ilustración 10 - Sistema de ventanas de registros

El sistema de ventanas de registros consta de un número finito de ventanas. En este ejemplo se muestran 8, ya que es la configuración por defecto, numeradas de la 0 a la 7, como se muestra en *Ilustración 10 - Sistema de ventanas de registros*. El puntero de ventana actual (CWP) indica la ventana que está usando la función que se esté ejecutando. Cada una de estas ventanas se compone de 3 tipos de registros:

· Registros locales: almacenan datos locales a la función actual.
· Registros de entrada: almacena los argumentos con los que la función anterior llamó a la actual. También se utiliza para almacenar el valor de retorno de esta función.
· Registros de salida: almacena los argumentos para la siguiente función a llamar y el valor de retorno de dicha función.

Cada uno de estos tipos (locales, salida y entrada) cuenta con 8 registros de 32 bits, donde cada ventana tiene acceso a 24 registros. Tal y como se muestra en la *Ilustración 10*, los registros de salida de una ventana están solapados con los registros de entrada de la siguiente ventana, formando una lista circular, de tal forma que los registros de salida de la ventana con el identificador más bajo están solapados con los registros de entrada de la ventana con el identificador más alto. En el caso de ejemplo, con 8 ventanas, el número total de registros es de 8 ventanas por 16 registros/ventana, o lo que es lo mismo, 128 registros. Además, existe otro grupo de 8 registros denominados globales y que son accesibles desde cualquier ventana.

La ISA SPARCv8 proporciona dos instrucciones que permiten modificar el CWP. La instrucción de SAVE permite avanzar una ventana, restando una unidad al valor del CWP. Por el contrario, la instrucción RESTORE permite retroceder una ventana, sumando una unidad al valor del CWP.

Cuando se está ejecutando una función, los datos que se usen localmente se almacenan y modifican usando los registros locales de la ventana que está siendo apuntada actualmente por el CWP. Si en algún momento se llama a otra función, la función actual guarda los argumentos de entrada en los registros de salida de la ventana actual y el programa ejecuta la instrucción SAVE, que automáticamente resta 1 al CWP, avanzando de ventana. Al retornar, la función llamante guarda sus resultados en sus registros de entrada y se ejecuta la instrucción RESTORE, que suma 1 al CWP.

Suponer el siguiente programa de ejemplo:

```
1    int op1 = 1;
2
3    void main(){
4           int op2 = 2;
5           int op3 = 3;
6           int res = suma(op2, op3);
7           printf("Resultado = ", res);
8    }
9
10   int suma(int op2, int op3){
11          int op4 = 4;
12          return(op1 + op2 + op3 + op4);
13   }
```

Código 36 - Programa de ejemplo para el sistema de ventanas

1. El programa comienza en main() con CWP = 0 (ventana 0). Antes de llegar aquí ya se ha guardado la global op1 en un registro global.

```
int op1 = 1;
void main()
```

	Glob
R0	**1**
R1	...
R2	...
R3	...
...	...
R7	...

Ventana 0			Ventana 7		
Ent	Loc	Sal	Ent	Loc	Sal
...	...	...		...	...
...	...	...		...	...
...	...	...		...	...
...	...	...		...	...
...	...	...		...	...
...	...	...		...	...

2. Se ejecutan las líneas 4 y 5, lo que provoca que se guarden las variables op2 y op3 en los registros locales de la ventana actual.

```
int op2 = 2;
int op3 = 3;
```

	Glob
R0	1
R1	...
R2	...
R3	...
...	...
R7	...

Ventana 0			Ventana 7		
Ent	Loc	Sal	Ent	Loc	Sal
...	**2**	...		...	...
...	**3**	...		...	...
...	...	...		...	...
...	...	...		...	...
...	...	...		...	...
...	...	...		...	...

3. Se va a llamar al método suma, pero antes se guardan sus argumentos en los registros de salida.

```
suma(op2, op3);
```

	Glob
R0	1
R1	...
R2	...
R3	...
...	...
R7	...

Ventana 0			Ventana 7		
Ent	Loc	Sal	Ent	Loc	Sal
...	2	**2**		...	...
...	3	**3**		...	...
...	...	...		...	...
...	...	...		...	...
...	...	...		...	...
...	...	...		...	...

4. Con los argumentos de llamada listos, se ejecuta la instrucción SAVE, lo que provoca que se reste 1 al CWP, y hace que se cambie a la ventana 7.

	Glob
R0	1
R1	...
R2	...
R3	...
...	...
R7	...

Ventana 0			**Ventana 7**		
Ent	Loc	Sal	Ent	Loc	Sal
...	2	2		...	...
...	3	3		...	...
...	...	...		...	...
...	...	...		...	...
...	...	...		...	...
...	...	...		...	...

5. Comienza la ejecución del método suma, la cual parte declarando una nueva variable que se guarda en sus registros locales.

```
int op4 = 4;
```

	Glob
R0	1
R1	...
R2	...
R3	...
...	...
R7	...

Ventana 0			**Ventana 7**		
Ent	Loc	Sal	Ent	Loc	Sal
...	2	2		**4**	...
...	3	3		...	...
...	...	...		...	...
...	...	...		...	...
...	...	...		...	...
...	...	...		...	...

6. Continúa haciendo la operación op1+op2+op3+op4, donde op1 lo obtiene de los globales, op2 y op3 de los argumentos (entrada) y op4 de sus locales. Como es retorno lo guarda en los de entrada.

```
return(op1 + op2 + op3 + op4);
```

	Glob
R0	1
R1	...
R2	...
R3	...
...	...
R7	...

Ventana 0			**Ventana 7**		
Ent	Loc	Sal	Ent	Loc	Sal
...	2	**10**		4	...
...	3	3		...	...
...	...	...		...	...
...	...	...		...	...
...	...	...		...	...
...	...	...		...	...

7. Finaliza el método suma, lo cual provoca que se ejecute la instrucción RESTORE que suma 1 al CWP, volviendo a la ventana 0.

	Glob
R0	1
R1	...
R2	...
R3	...
...	...
R7	...

Ventana 0			Ventana 7		
Ent	Loc	Sal	Ent	Loc	Sal
...	2	10		4	...
...	3	3		...	...
...	...	...		...	...
...	...	...		...	...
...	...	...		...	...
...	...	...		...	...

8. De nuevo en main, guarda el retorno de suma en una nueva variable que llama res, y termina imprimiendo dicho valor.

```
int res = suma(op2, op3);
```

	Glob
R0	1
R1	...
R2	...
R3	...
...	...
R7	...

Ventana 0			Ventana 7		
Ent	Loc	Sal	Ent	Loc	Sal
...	2	10		4	...
...	3	3		...	...
...	**10**	...		...	...
...	...	...		...	...
...	...	...		...	...
...	...	...		...	...

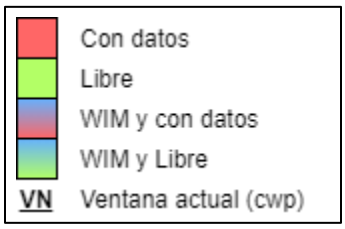
4.2.3 Desbordamiento de ventanas

Como el número de ventanas es limitado, es posible que se de el caso en el que, al cambiar de ventana, no quede ninguna libre. Para tratar esto se usan 2 *traps*, el de desbordamiento por no quedar ventanas libres (*window overflow*) y el de desbordamiento por no quedar ventanas de retorno (*window underflow*).

Para saber el estado de las ventanas se usa el registro WIM, que guarda un campo de tantos bits como ventanas existen, es decir, para 8 ventanas WIM usa un campo de 8 bits, donde cada bit representa dicho número de ventana. En la configuración de uso estándar, este campo solo tendrá un bit a '1', el cual referencia la última ventana libre, y el resto de bits serán 0. Esta ventana se denomina generalmente la ventana “inválida”. Por ejemplo, si WIM es "0000 0100b", está indicando que la ventana 2 es la ventana inválida, es decir, es la última libre, y que las ventanas de la 1 a la apuntada por el CWP están ocupadas con datos válidos.

4.2.3.1 Trap de window overflow

Cada vez que se avanza a una ventana nueva, mediante la instrucción de SAVE o al saltar un *trap*, antes de mover la ventana el procesador comprueba si la ventana destino es la última libre, es decir, si la ventana CWP-1 es la marcada como inválida en el WIM, en cuyo caso se dispara el *trap* de *window overflow*. Este manejador se encarga de salvar en la pila el valor de los registros de la ventana más antigua, liberándola para su uso. Ejemplo del proceso:



Las ventanas en rojo contienen datos de funciones que las han usado previamente y actual, y las verdes están libres.
Las azules son las marcadas por WIM, que se degradan en rojo o verde según si tienen datos o no de funciones previas.
Si el texto interno está subrayado y en negrita, es que el CWP está apuntando a dicha ventana.

1. Por ejemplo, se parte de cierta función que está ejecutándose en la ventana 5 (CWP = 5), con el resto de las ventanas ya usadas excepto la marcada como inválida (0001 0000b = V4), y quiere realizar un SAVE.

2. Como la ventana CWP-1 (V4) está marcada como inválida, salta un *trap*, lo que le hace moverse a CWP-1 y llamar al manejador de *window overflow*.

3. En el manejador de *window overflow*, primero se hace un SAVE (CWP-1), lo que le hace ir a V3.

4. Tras ello modifica el WIM haciendo un desplazamiento a derecha, por lo que ahora WIM = 0000 1000b = V3

5. Seguidamente, se guarda el contenido de esta ventana en pila, así que V3 pasa a estar libre.

6. El manejador de *window overflow* termina primero realizando un RESTORE (CWP+1), volviendo a V4.

7. Y finalmente un RETT (retorno de *trap*), que provoca un CWP+1, volviendo a V5 y a la **misma instrucción** de SAVE que provocó el *trap*.

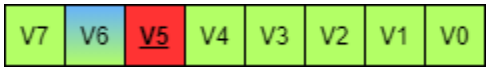
8. De nuevo en la primera función, se ejecuta el SAVE, lo que le hace ir a V4 que ya está libre.

Tabla 23 - Manejador de desbordamiento de ventana por falta de ventanas libres (window overflow)

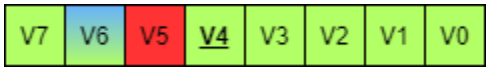
4.2.3.2 Trap de window underflow

Éste ocurre cuando se hace un retorno a una ventana, mediante una instrucción RESTORE, cuyos registros se habían guardado previamente en pila debido a un *overflow*, es decir, si la ventana a la que se va a retornar (CWP+1) está marcada como inválida en el WIM, se produce un trap. La rutina de atención deberá recuperar el valor de registros para que la ventana pueda ser utilizada normalmente por el programa. Un ejemplo de este proceso sería el siguiente:

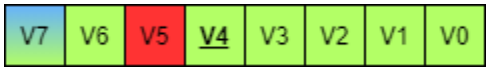
1. Se parte de cierta función que está ejecutándose en la ventana 5 (CWP = 5), con el resto de ventanas ya libres, por lo que WIM marca la anterior como inválida (0100 0000b = V6), y se quiere realizar un RESTORE.



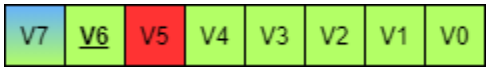
2. Como la ventana CWP+1 (V6) está marcada como inválida, salta un TRAP, lo que le hace moverse a CWP-1 (V4) y llamar a *window underflow*.



3. En el manejador de *window underflow*, primero se desplaza el WIM un bit a la izquierda, por lo que ahora WIM = 1000 0000b = V7

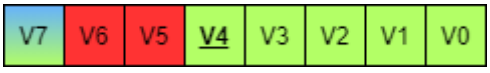


4. Tras ello, el manejador ejecuta dos instrucciones RESTORE, (CWP+2), lo que le hace ir a V6

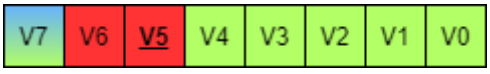


5. Seguidamente, carga el contenido de V6 desde pila que previamente se habría guardado por un *window overflow*.

6. El manejador de *window underflow* termina primero ejecutando dos instrucciones SAVE (CWP-2), volviendo a V4.



7. Y finalmente un RETT, que provoca un CWP+1, volviendo a V5 y a la **misma instrucción** de RESTORE que provocó el *trap*.



8. De nuevo en la primera función, se ejecuta el RESTORE, lo que le hace ir a V6, que ya tiene cargados los datos de la función que lo usó. Como es RESTORE, se considera V5 como libre.

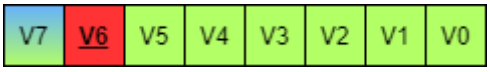
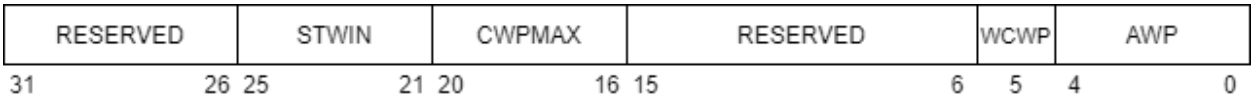


Tabla 24 - Manejador de desbordamiento de ventana por ser todas libres (window underflow)

4.2.4 Puntero de ventana alternativo – AWP

En caso de habilitarse el AWP en LEON3, se activa el uso del registro ASR20, el cual se describe en el siguiente apartado. Además, se usan los campos AW y PAW del registro PSR, descritos en el apartado 4.2.1.1 Registro de estado del procesador (*processor state register o PSR)*, los cuales sirven para indicar cuales de los punteros de ventana utilizar.

4.2.4.1 Registro de estado auxiliar 20 (*ancillary state register 20* o ASR20)



Campo	Bits	Descripción	R/W	Reset
RES	26..31	Reservado.	R	0
STWIN	21..25	Ventana de inicio de partición.	RW*	0
CWPMAX	16..20	Valor máximo del puntero de ventana actual, que equivale al tamaño de la partición menos 1. * El valor inicial es el número de ventanas menos 1, por lo que con STWIN = 0 asigna todas las ventanas en la partición. Si se escribe este campo a 0, este y STWIN no se modifican.	RW*	*
RES	6..15	Reservado.	R	0
WCWP	5	Escribe CWP. En caso de ponerse a 1 durante una escritura a este registro, se copia el valor que se vaya a escribir en AWP en CWP.	W	-
AWP	0..4	Puntero de ventana alternativo. *En caso de estar deshabilitado, se actualiza continuamente con el valor de CWP.	RW	*

Tabla 25 - Registro de estado auxiliar 20 - ASR20

4.2.4.2 Funcionamiento del AWP

Para activar el puntero alternativo o AWP se debe poner el campo AW del registro PSR a 1, momento a partir el cual la ventana que se usa deja de ser la marcada por el campo CWP de PSR, y pasa a ser la indicada por el campo AWP del registro ASR20.

Mientras AW sea 0, el funcionamiento del sistema de ventanas de registros es como el indicado en el apartado *4.2.2 Sistema de ventanas de registros*, pero si además PAW es 0, el valor del campo CWP se escribe simultáneamente en AWP.

Mientras AW sea 1, las instrucciones para cambiar de ventana solo modifican el campo AWP (SAVE hace AWP-1 y RESTORE hace AWP+1), pero con 3 peculiaridades:

1. No se comprueban desbordamientos de ventanas, obligando a considerar que ventanas se pueden usar o no mientras se utilice el AWP.
2. En caso de *trap*, se copia AW en PAW y se pone AW a 0. Esto hace que durante la ejecución del un *trap* siempre se comience usando CWP.
3. En caso de RETT (retorno de *trap*), se copia PAW en AW, por lo que, si antes del *trap* se estaba usando AWP, al retornar de un *trap* vuelve a activarse.

La siguiente tabla resume el comportamiento del sistema de ventanas en función del valor de los bits AW y PAW, y, tras ésta, se adjunta *Ilustración 11 - Diagramas de flujo de las instrucciones de cambio de ventana con AWP habilitado*, la cual muestra este comportamiento usando diagramas de flujo.

PAW	AW	Funcionamiento del sistema de ventanas
0	0	· La ventana en uso es la apuntada por el campo CWP del registro PSR. · En caso de cambiar de ventana (SAVE, *trap*, RESTORE o RETT) se modifica CWP. · Todos los cambios en el campo CWP se copian en el campo AWP. · Se generan *traps* de desbordamiento en caso de llegar a la ventana marcada por WIM.
0	1	· La ventana en uso es la apuntada por el campo AWP del registro ASR20. · En caso de cambio de ventana por SAVE o RESTORE, se modifica AWP. · En caso de *trap*, PAW pasa a 1 y AW pasa a 0, y se resta 1 a CWP. · En caso de RETT, se suma 1 a CWP y AW pasa a 0. · No se generan *traps* por desbordamiento de ventana.
1	0	· La ventana en uso es la apuntada por el campo CWP del registro PSR. · En caso de cambio de ventana por SAVE o RESTORE, se modifica CWP. · En caso de *trap*, PAW pasa a 0, y se resta 1 a CWP. · En caso de RETT, se suma 1 a CWP y AW pasa a 1. No se comprueba desbordamiento. · Se generan *traps* de desbordamiento en caso de llegar a la ventana marcada por WIM.
1	1	· La ventana en uso es la apuntada por el campo AWP del registro ASR20. · En caso de cambio de ventana por SAVE o RESTORE, se modifica AWP. · En caso de *trap*, PAW pasa a 1 y AW pasa a 0, y se resta 1 a CWP. · En caso de RETT, se suma 1 a CWP u AW pasa a 1. No se comprueba desbordamiento. · No se generan *traps* por desbordamiento de ventana.

Tabla 26 - Resumen de funcionamiento del sistema de puntero alternativo AWP según los campos AW y PAW

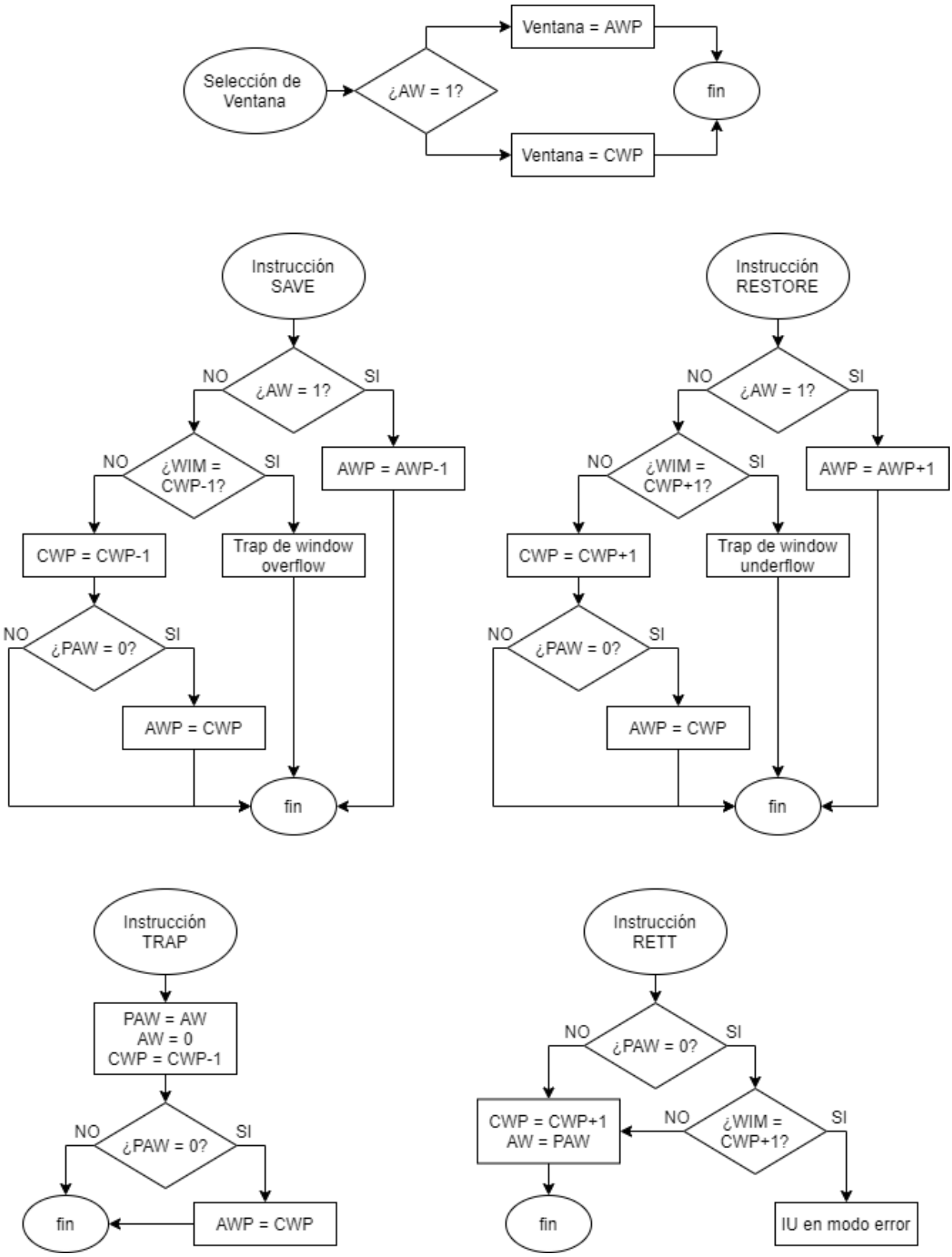


Ilustración 11 - Diagramas de flujo de las instrucciones de cambio de ventana con AWP habilitado

4.2.4.3 Funcionamiento de las particiones de registros

Al activar AWP en LEON3, además del puntero alternativo, se activa el uso de los campos STWIN y CWPMAX del registro ASR20, los cuales permiten definir particiones de ventanas.

Hasta ahora, al cambiar de ventana mediante una de las instrucciones que modifican CWP y AWP, siempre se hacía dentro del rango desde la ventana 0 a la ventana máxima (7 por defecto). Pero este rango depende del valor de los campos STWIN y CWPMAX, y el cambiarlo permite limitar que ventanas se pueden usar.

El campo de ventana de inicio (*Starting window* o STWIN) indica en qué ventana comienza la partición. Independientemente del valor de STWIN, el puntero de ventana siempre parte de 0, por lo que, si por ejemplo STWIN vale 2, entonces, cuando el puntero de ventana vale 0, la ventana en uso será la 2. Es decir, numero de ventana en uso = CWP (o AWP si AW es 1) + STWIN. Por defecto STWIN es 0, indicando que la primera ventana de la partición es la 0.

El campo de puntero de ventana actual máximo (*Current window pointer maximum* o CWPMAX) indica el valor máximo que puede tomar el puntero de ventana (ya sea CWP o AWP) al hacer cambios de ventana. También sirve para indicar el tamaño de la partición, ya que da igual en que ventana comience la partición (STWIN), el puntero de ventana siempre parte de 0, por lo que el tamaño de la partición coincide con CWPMAX+1 (Es decir, CWPMAX = tamaño partición - 1). Por defecto, CWPMAX es el número de ventanas disponibles menos 1, que en el caso de 8 ventanas haría que valiese 7.

En resumen, la partición de ventanas en uso se define como:

- Rango de ventanas = desde la ventana STWIN hasta la ventana STWIN+CWPMAX.
- Numero de ventana = puntero de ventana (CWP o AWP si AW es 1) + STWIN.

Es necesario hacer resaltar en que, como el rango es hasta STWIN+CWPMAX, esta suma no debería superar el número de ventanas disponibles. Es decir, STWIN+CWPMAX < Número de ventanas disponibles.

Es importante tener en cuenta que el uso de particiones diferentes a la de por defecto no modifica el funcionamiento del registro WIM, solo enmascara los bits mayores que CWPMAX. Esto obliga a tener que definir nuevos manejadores de *trap* de desbordamiento de ventana, ya que los manejadores predefinidos por el compilador BCC no consideran particiones diferentes a la de por defecto.

Por último, otra cosa a considerar es que no está definido el comportamiento al cambiar de ventanas si se está fuera de la partición creada, por lo que existe el peligro de modificar ventanas que estén siendo usadas por otras funciones si se sale de la partición. De todas formas, la única manera de salirse de la partición es si se modifica manualmente el puntero de ventana, en cuyo caso debe hacerse con un proceso específico y considerando en todo momento que ventanas pueden usarse.

En la siguiente tabla se muestran varios ejemplos de como sería el rango del puntero y ventanas en función de los campos STWIN y CWPMAX:

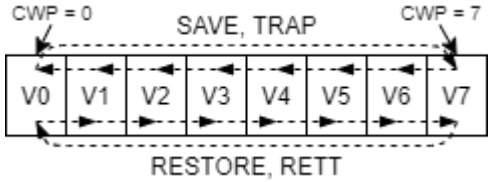
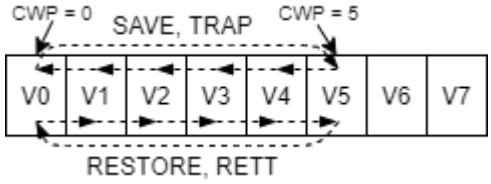
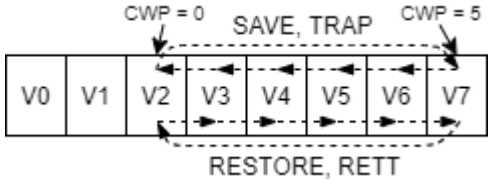
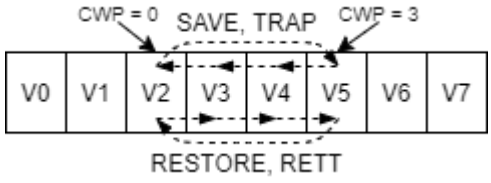
STWIN	CWPMAX	Rango Ventanas	Rango CWP	Ilustración
0	7	0 a 7	0 a 7	
0	5	0 a 5	0 a 5	
2	5	2 a 7	0 a 5	
2	3	2 a 5	0 a 3	

Tabla 27 - Ejemplos de particiones de ventanas usando STWIN y CWPMAX.

4.2.4.4 Uso del AWP y particiones de registros

La razón principal de implementar el AWP es el poder crear particiones de registros, ya que esto permite hacer que el resto de las ventanas esté siempre libre y dejarse para ciertas funciones. Esto es importante porque al cambiar de función se produce un cambio de ventana, lo cual puede provocar un *trap* de desbordamiento. Esto puede suponer un problema en el caso de utilizarse manejadores de interrupción que tengan un plazo máximo de ejecución, ya que es imposible saber cuál es el estado inicial de las ventanas y si se producirá o algún desbordamiento, lo que provoca que haya variaciones en los tiempos de latencia (*jitter*).

Originalmente, cuando se produce una interrupción por un evento, se llama al manejador de interrupciones el cual primero comprueba si hay desbordamiento y finalmente pasa la ejecución a la función definida para dicha interrupción. Como la interrupción puede ocurrir en cualquier punto de la ejecución del programa, es difícil predecir el estado de las ventanas en ese momento, por lo que la misma interrupción puede tardar más o menos tiempo en ejecutarse dependiendo de si hay un desbordamiento de ventana o no. Pero si en lugar de cambiar de ventana, se cambia de partición

a un rango de ventanas que previamente se ha dejado libre para esta el manejador de interrupción, no es necesario comprobar nada, y la interrupción siempre tardará el mismo tiempo.

4.3 Implementación en LeonViP

El simulador LeonViP ya implementa el sistema de ventanas de registros, por lo que, para implementar el AWP, primero se describe cómo simula dicho sistema y seguidamente se detallan los cambios realizados para añadir la funcionalidad del AWP.

4.3.1 Sistema de ventanas de registros en LeonViP

4.3.1.1 Ventanas de registros de enteros

LeonViP usa un array de enteros sin signo de 32 bits denominado *registers[]* para representar todas las ventanas de registros de enteros. Según el índice de dicho array, se referencia o bien a los registros locales, de entrada, o de salida de cierta ventana, tal y como se muestra en la siguiente figura:

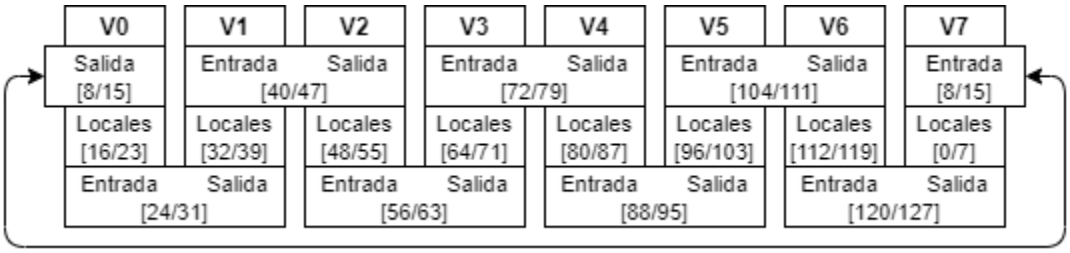


Ilustración 12 - Sistema de ventanas por índices de array en LeonViP

Este array se declara con un tamaño de *N_Windows*Window_Size*, donde *N_Windows* son el número de ventanas disponibles para el sistema, que por defecto son 8, y *Window_Size* que es el número de registros por ventana, y por defecto es 16. Además, se declara un array denominado *globals* de 8 enteros sin signo de 32 bits, el cual representa los registros globales.

LeonViP usa los métodos *setRegisterValue* y *getRegisterValue* para escribir y leer respectivamente los datos de los registros. A estos se les da de argumento *regNum* que, además de indicar el número de registro, infiere que tipo de registros acceder usando 6 rangos posibles:

valor de *regNum*	**Registros a los que se accede**
de 0 a 7	Registros globales.
de 8 a 15	Registros de salida de la ventana actual de los registros de enteros.
de 16 a 23	Registros locales de la ventana actual de los registros de enteros.
de 24 a 31	Registros de entrada de la ventana actual de los registros de enteros.
de 32 a 63	Registros de punto flotante.
mayor de 63	Registros del sistema. (PSR, WIM...)

Tabla 28 - Rango de valores de la variable de registros regNum

Para facilitar la indexación de del array de registros de enteros se emplea el atributo *cwp_mult_WINDOW_SIZE* cuyo valor es la ventana actual por el tamaño de ventana. Este atributo se actualiza cada vez que se produce un cambio de ventana, por lo que, en el caso de acceder a los registros de enteros, solo es necesario sumar el *regNum* a *cwp_mult_WINDOW_SIZE*, lo que resulta en el índice del registro pedido. Como la lista de registros es circular, es necesario tener en cuenta los casos en los que se esté en uno de los extremos del array y el registro pedido esté en el otro extremo. Para tratarlo se realiza una operación AND usando el atributo totalWindowSize, el cual guarda el número de registros totales. Entonces, el cálculo de índice del array de registros de enteros se realiza como:

índice = (*cwp_mult_WINDOW_SIZE* + *regNum*) AND *totalWindowSize* =>
índice = ((CWP**Window_Size*) + *regNum*) AND ((*N_Windows* * *Window_Size*) - 1)

Por ejemplo, con valores por defecto (*Window_Size* = 16, *N_Windows* = 8), y estando en la ventana 7 (CWP = 7), si se quiere acceder a su primer registro de entrada (*i0*), *regNum* será 24:

índice = ((7*16)+24) AND ((8*16)-1) = 136 AND 127 = 8

Y, estando en la ventana 0 (CWP = 0), si se quiere acceder a su primer registro de salida (*o0*), *regNum* será 8, el cual debería coincidir con el índice anterior:

índice = ((0*16)+8) AND ((8*16)-1) = 8 AND 127 = 8

Como se puede comprobar, ambos dan el índice 8, que tal y como muestra la *Ilustración 12* - Sistema de ventanas por índices de array en LeonViP, *registers[8]* es el primer registro de entrada de la ventana 7, y el primero de salida de la ventana 0. A continuación se muestra el código utilizado para acceder originalmente a estos registros:

-- Sistema de registros en LeonViP --

```
1    uint32_t globals[8];   // Registros globales
2    uint32_t registers[WINDOW_SIZE * N_WINDOWS];   // Ventanas de registros
3
4    uint32_t SparcISS::getRegisterValue(uint32_t regNUM) {
5        if (regNUM <= 31) {
6            if (regNUM < 8){  // Globales
7                return globals[regNUM];
8            } else {  // Ventanas
9                return registers[(cwp_mult_WINDOW_SIZE + regNUM) & totalWindowsSize];
10           }
11
12           (...) Comprobación de registros de punto flotante y de Sistema
```

Código 37 - Sistema de registros en LeonViP - SparcISS.cpp

4.3.1.2 Movimiento de ventanas de registros

En LeonViP, al ejecutar una instrucción que cambia la ventana de registros, el procedimiento comienza, en la mayoría de los casos, por comprobar a qué CWP se va a cambiar, lanzar un *trap* de desbordamiento si la nueva ventana está marcada como inválida en el WIM, y

finalmente cambiar el CWP. Estos métodos también modifican la variable *cwp_mult_WINDOW_SIZE*, que, como se comentó anteriormente, facilitan la indexación del array de registros de enteros. A continuación se adjunta el código que se encarga de lo descrito:

-- Instrucciones de cambio de ventana en LeonViP --

```
1   void _SAVE(SparcISS *pISS) { // Instrucción Save (CWP - 1)
2
3       // Calcula CWP siguiente
4       uint32_t newCWP = ((pISS->psr & CWP_MASK) - 1) & (N_WINDOWS - 1);
5
6       if ((pISS->wim & (1 << newCWP))) { // Comprueba si hay overflow
7           pISS->trap = TRAP_WOFL;
8           return;
9       }
10      pISS->psr &= ~CWP_MASK;
11      pISS->psr |= newCWP; // Cambia campo CWP del registro PSR
12      pISS->cwp_mult_WINDOW_SIZE = newCWP * WINDOW_SIZE;
13      (...)
14  }
15
16  void _RESTORE(SparcISS *pISS) { // Instrucción Restore (CWP + 1)
17
18      // Calcula CWP siguiente
19      uint32_t newCWP = ((pISS->psr & CWP_MASK) + 1) & (N_WINDOWS - 1);
20
21      if (pISS->wim & (1 << newCWP)) { // Comprueba si hay underflow
22          pISS->trap = TRAP_WUFL;
23          return;
24      }
25      pISS->psr &= ~CWP_MASK;
26      pISS->psr |= newCWP; // Cambia campo CWP del registro PSR
27      pISS->cwp_mult_WINDOW_SIZE = newCWP * WINDOW_SIZE;
28      (...)
29  }
30
31  const uint8_t *SparcISS::executeTRAP() { // Instrucción Trap (CWP - 1)
32
33      (...) Comprobación de traps
34
35      // Calcula CWP siguiente
36      uint32_t newCWP = ((psr & CWP_MASK) - 1) & (N_WINDOWS - 1);
37      psr &= ~CWP_MASK;
38      psr |= newCWP; // Cambia campo CWP del registro PSR
39
40      cwp_mult_WINDOW_SIZE = newCWP * WINDOW_SIZE;
41
42      (...) Salto a manejador de Trap o Error
43  }
44
45  void _RETT(SparcISS *pISS) { // Instrucción Rett (CWP + 1)
46
47      (...)
48
49      // Calcula CWP siguiente
50      uint32_t newCWP = ((psr & CWP_MASK) + 1) & (N_WINDOWS - 1);
51
52      if (pISS->wim & (1 << newCWP)) { // Comprueba si hay underflow
53          pISS->trap = TRAP_WUFL;
```

```
54              return;
55          }
56
57          (...)
58
59          psr &= ~CWP_MASK;
60          psr |= newCWP; // Cambia campo CWP del registro PSR
61          pISS->cwp_mult_WINDOW_SIZE = newCWP * WINDOW_SIZE;
62
63          (...)
64
65      }
```

Código 38 - Instrucciones de cambio de ventana en LeonViP - SparcISS.cpp

Además de estas, hay que considerar la posibilidad de que se cambie el CWP manualmente mediante una escritura al registro de PSR, lo cual se hace con la instrucción WRPSR. En este caso no se comprueba WIM, ya que, aunque haya desbordamiento, es imposible saber si fue por *overflow* o *underflow*. El siguiente código muestra el método encargado de ésta escritura al registro PSR:

-- Escritura al registro PSR en LeonViP --

```
1   void _WRPSR(SparcISS *pISS) {
2
3       (...) Escritura de PSR
4
5       // Cambia el valor de CWP*Window_Size por si cambió CWP
6       pISS->cwp_mult_WINDOW_SIZE = (pISS->psr & CWP_MASK) * WINDOW_SIZE;
7
8       (...)
9
10  }
```

Código 39 - Escritura al registro PSR LeonViP - SparcISS.cpp

4.3.2 Cambios en LeonViP para la implementación del AWP

4.3.2.1 Atributos auxiliares

Se declaran varios atributos en *SparcISS.h* para facilitar comparaciones y añadir otras utilidades:

- ***aw*** y ***paw***: guardan los valores de los campos de mismo nombre del registro PSR.
- ***cmax:*** indica si el valor de CWPMAX es diferente al de por defecto.
- ***cwpmax:*** guarda el valor del campo de mismo nombre del registro ASR20.
- **stwin_mul_win_*size***: contiene el valor del campo STWIN por el tamaño de ventana *WindowSize*. Por defecto es STWIN*16, y funciona como offset al array de registros de enteros, ya que la ventana indicada por CWP = 0 cambia según STWIN.

- ***outsCwp0**:* guarda un offset a los registros de salida cuando CWP = 0, ya que, como ahora es posible cambiar qué ventanas son la primera y última, es necesario conectar los registros de entrada de la ventana CWP = CWPMAX con los de salida de la ventana inicial STWIN (CWP = 0). Este *offset* se usa solamente cuando se quieren obtener los registros de salida si CWP = 0. Se dan más detalles sobre esto en el apartado 4.3.2.4 Cambios en el sistema de acceso a registros de enteros.

- ***wimMask***: contiene la máscara a aplicar cuando se hace una lectura del registro de WIM. Esta depende de CWPMAX, enmascarando las ventanas superiores. Se dan más detalles en el apartado 4.3.2.3 Cambios en los registros del sistema.

-- Nuevos atributos en SparcISS --

```
1    bool aw, paw, cmax; // Campos aw y paw de PSR, y cmax para indicar cwpMax diferente
2    uint32_t cwpmax; // Campo cwpmax del registro ASR20
3    uint32_t stwin_mul_win_size; // Offset de registros, STWIN*Window_Size
4    uint32_t outsCwp0; // Offset del array de registros de salida cuando la ventana es 0
5    uint32_t wimMask; // Máscara de lectura de WIM según valor de CWPMAX
```

Código 40 - Nuevos atributos para la implementación de AWP - SparcISS.cpp

4.3.2.2 Nuevos métodos para la selección de la ventana de registros

Se crean 3 funciones para facilitar el cambio de ventana en uso según los valores de los campos AW y PAW del registro PSR. Estos métodos se llaman cuando hay un movimiento de ventana, ya sea por una instrucción de cambio de ventana o por una escritura en el registro PSR o en el ASR20.

Primero se define el método *getRegisterWindow()* para cuando se quiere saber a qué ventana se va a mover antes de asignarla. El código de la función se muestra en *Código 41 - Método de obtención de siguiente ventana - SparcISS.cpp*. A éste se le da de argumento el movimiento de ventana, '-1' para un SAVE y '1' para un RESTORE, y devuelve el número de ventana que corresponde a ese movimiento en función de si se está usando CWP o AWP y si CWPMAX es diferente del de por defecto, lo cual se comprueba con la variable auxiliar *cmax*.

Por ejemplo, suponer el caso en que AW es 0, CWP es 3 y CWPMAX es 3. Si se ejecuta un RESTORE, se llama a este método con argumento '1', lo que le hace devolver el número 0, ya que como CWPMAX es 3, a partir de esa ventana vuelve a CWP = 0. Si AW fuera 1 usaría AWP en vez de CWP.

-- Método de obtención de siguiente ventana --

```
1    uint32_t SparcISS::getRegisterWindow(int shift) {
2
3        // Obtiene siguiente ventana (CWP o AWP + shift) según si AW es 0 o 1
4        uint32_t cwp = (aw) ? ((asr[20] & AWP_MASK) + shift) & (N_WINDOWS - 1)
5                            : ((psr & CWP_MASK) + shift) & (N_WINDOWS - 1);
6
```

```
7         if (cmax) {    // Si CWPMAX es diferente al de por defecto
8             if (shift > 0 && cwp == (cwpmax + 1)) { // Comprueba si se supera
9                 cwp = 0;
10            } else if (shift < 0 && cwp == (N_WINDOWS - 1)) { // O si se vuelve
11                cwp = cwpmax;
12            }
13        }
14
15        return cwp;
16    }
```

Código 41 - Método de obtención de siguiente ventana - SparcISS.cpp

También se define un segundo método *getRegisterWindow()*, pero ahora sin argumentos, el cual simplemente devuelve la ventana actual indicada por CWP o AWP según el estado del campo AW:

-- Método de obtención de ventana actual --

```
1     uint32_t SparcISS::getRegisterWindow() {
2       // Devuelve CWP o AWP en función del bit AW de PSR
3       return (aw) ? asr[20] & AWP_MASK : psr & CWP_MASK;
4     }
```

Código 42 - Método de obtención de ventana actual - SparcISS.cpp

Por último, se define el método *setRegisterWindow()*, mostrado tras éste párrafo, que cambia el CWP o el AWP según el argumento, y además copia CWP en AWP si se cumple que AW y PAW son 0. Este se llamaría, entre otras, desde la función que ejecuta la instrucción SAVE, tras comprobar si la ventana a la que se cambia provoca un *trap* de *window overflow*.

-- Método de cambio de ventana --

```
1     void SparcISS::setRegisterWindow(uint32_t window) {
2         if (aw) { // Si AW es 1, cambia AWP
3             asr[20] = (asr[20] & ~AWP_MASK) | window;
4         } else { // Si no, cambia CWP
5             psr = (psr & ~CWP_MASK) | window;
6             if (!paw) { // Y si además PAW es 0, copia CWP en AWP
7                 asr[20] = (asr[20] & ~AWP_MASK) | window;
8             }
9         }
10        // Y finalmente restablece cwp_mult
11        cwp_mult_WINDOW_SIZE = window * WINDOW_SIZE;
12    }
```

Código 43 - Método de cambio de ventana - SparcISS.cpp

4.3.2.3 Cambios en los registros del sistema

El registro ASR20 no estaba definido en la versión original de LeonViP, por lo que se ha tenido que modificar el método *_WRY*, que es el encargado de la escritura de ciertos registros como los ASR.

El registro ASR20 tiene bastantes peculiaridades que hay que tener en cuenta, ya que, a parte de definir particiones de registros y el nuevo puntero de ventana AWP, también cambia la forma en la qué se escriben ciertas cosas según el valor pasado:

- STWIN y CWPMAX desbordan si superan el máximo de ventanas disponibles
- Si se escribe 0 en CWPMAX solo se modifica AWP.
- Si se escribe 1 en WCWP, se copia AWP en CWP.
- Si cambia STWIN, CWPMAX o AWP, hay que cambiar la ventana de registros de enteros, además de actualizar los atributos definidos al inicio que dependen de estos valores.

El siguiente código muestra los cambios realizados en el método *_WRY* para simular el registro ASR20:

-- Cambios del método de escritura de registros ASR --

```
1    void _WRY(SparcISS *pISS) {
2
3    uint32_t rd = pISS->currentInst->rd; // Obtiene registro a escribir
4
5    if (rd){
6
7       (...)
8
9       if (rd == 20) {  // Si es ASR20
10         uint32_t data = pISS->operand1 ^ pISS->operand2; // Obtiene dato a escribir
11         uint32_t cwpmax, stwin; // Temporales
12
13         // Cwpmax a escribir, teniendo en cuenta desbordamiento
14         cwpmax = ((data & CWPMAX_MASK) >> 16) & (N_WINDOWS - 1);
15         if (cwpmax) { // Si cwpMax NO es 0
16            // Obtiene stwin teniendo en cuenta desbordamiento
17            stwin = ((data & STWIN_MASK) >> 21) & (N_WINDOWS - 1);
18
19            // Reescribe el dato con el cwpmax y stwim tras comprobar desbordamiento
20            data = (data & ~CWPMAX_MASK) + (cwpmax << 16);
21            data = (data & ~STWIN_MASK) + (stwin << 21);
22
23            if (data & WCWP_MASK) { // Si WCWP es 1, copia AWP en CWP de PSR
24               pISS->psr = (pISS->psr & ~CWP_MASK) | (data & AWP_MASK);
25            }
26
27            // Reescribe asr20, aplicando una mascara que indica que campos se escriben
28            pISS->asr[20] = data & 0x03FF001F;
29
30            // Modifica atributos según los cambios en asr20
31            pISS->cwpmax = cwpmax;
32            pISS->cmax = cwpmax < (N_WINDOWS - 1);
33            pISS->stwin_mul_win_size = stwin * WINDOW_SIZE;
34            pISS->outsCwp0 = ((stwin + cwpmax + 1)*WINDOW_SIZE) & pISS->totalWindowsSize;
35            pISS->wimMask = (1 << (pISS->cwpmax + 1)) - 1;
36
37         } else { // Si cwpMax es 0, solo modifica el campo AWP
38            pISS->asr[20] =(pISS->asr[20] & ~AWP_MASK) | (data & AWP_MASK);
39         }
40
41         // Finalmente, como es posible que haya cambiado la ventana, se obtiene la
42         // ventana actual y se la pasa de argumento al método de cambio de ventana
43         pISS->setRegisterWindow(pISS->getRegisterWindow());
```

```
44
45      }
46
47  (...)
```

Código 44 - Cambios en el método _WRY - SparcISS.cpp

A continuación, se ha tenido que modificar el método *_WRPSR*, que se encarga de la escritura del registro PSR. Éste ya tenía en cuenta la posible modificación de CWP, pero ahora además hay que considerar cambios en AW y PAW, lo cual se hace fácilmente usando los métodos *setRegisterWindow* y *getRegisterWindow*. A continuación, se muestra esta modificación:

-- Cambios del método de escritura del registro PSR --

```
1   void _WRPSR(SparcISS *pISS) {
2
3       if (pISS->rd) { // Si rd distinto de 0, solo se escribe el campo ET
4           pISS->psr &= ~ET_MASK;
5           pISS->psr |= (pISS->operand1 ^ pISS->operand2) & ET_MASK;
6       } else { // Si no, se escribe todos los campos escribibles de PSR
7           pISS->psr &= 0xFF000000; // limpia campos escribibles
8
9           // Reescribe PSR
10          pISS->psr |= (pISS->operand1 ^ pISS->operand2) & 0x00FFFFFF;
11
12          // Restablece atributos aw y paw
13          pISS->aw = pISS->psr & AW_MASK;
14          pISS->paw = pISS->psr & PAW_MASK;
15
16          // Y llama a los métodos de cambio de ventana por si hubo algún cambio
17          // en AW o CWP que provoque un cambio de ventana.
18          pISS->setRegisterWindow(pISS->getRegisterWindow());
19      }
20
21      (...)
22
23  }
```

Código 45 - Cambios en el método _WRPSR - SparcISS.cpp

Tras ello se modifica el método *_RDWIM*, utilizado para leer el registro WIM, ya que ahora se le aplica una máscara según CWPMAX. Para ello solo es necesario hacer que devuelva 0 en los bits que representen las ventanas que sean superiores a CWPMAX, es decir, si por ejemplo CWPMAX es 5, los bits 6 y 7 de WIM siempre devuelven 0.

Para conseguir esto se define el atributo *wimMask*, que cambia según CWPMAX, de forma que da valor '1' a los bits de las ventanas que se deban mostrar, y '0' a el resto. Por ejemplo, si CWPMAX es 5, *wimMask* valdrá "0011 1111b".

De esta forma, si se hace la operación *Wim AND wimMask*, como se muestra en el código siguiente, siempre se devuelven a 0 los bits que representan las ventanas superiores a CWPMAX.

-- Cambios del método de lectura del registro WIM --

```
1    void _RDWIM(SparcISS *pISS) {
2
3        // Devuelve WIM enmascarado por wimMask
4        pISS->setRegisterValue(pISS->currentInst->rd, (pISS->wim & pISS->wimMask));
5
6        (...)
7
8    }
```

Código 46 - Cambios en el método _RDWIM - SparcISS.cpp

Por último, como se añadió el nuevo registro ASR20 además de varios atributos, es necesario modificar la función de inicialización para asignarles un valor inicial. Para ello se modifica el método *reset*, el cual se encarga del inicio de registros y otros valores del sistema. El siguiente código muestra los cambios realizados en el método de *reset* para lograr esta inicialización:

-- Cambios del método de inicio del sistema --

```
1    void SparcISS::reset(uint32_t pc) {
2
3         (...)
4
5        aw = psr & AW_MASK;
6        paw = psr & PAW_MASK;
7        asr[20] = ((N_WINDOWS - 1) << 16);
8        cwpmax = N_WINDOWS - 1;
9        cmax = false;
10       outsCwp0 = ((cwpmax + 1) * WINDOW_SIZE) & totalWindowsSize;
11       stwin_mul_win_size = 0;
12       wimMask = (1 << N_WINDOWS) - 1;
13
14       setRegisterWindow(getRegisterWindow());
15
16        (...)
17
18   }
```

Código 47 - Cambios en el método reset - SparcISS.cpp

4.3.2.4 Cambios en el sistema de acceso a registros de enteros

Debido a la inclusión del mecanismo de AWP, hay que reconsiderar como acceder al array de registros de enteros. Se debe comprobar si la ventana es la indicada por CWP o AWP y unir los registros de salida de la primera ventana de la partición con los de entrada de la última.

Para determinar si la ventana es AWP o CWP, se usa el atributo *cwp_mult_WINDOW_SIZE*, el cual ya estaba definido en la versión original de LeonViP, pero ahora, en lugar de depender de CWP, depende de CWP o AWP según si AW está activado o no:

Si AW = 0 entonces *cwp_mult_WINDOW_SIZE = CWP * Window_Size*
Si AW = 1 entonves *cwp_mult_WINDOW_SIZE = AWP * Window_Size*

Para el cálculo de ventana de inicio se usa el atributo *stwin_mul_win_size*, que determina el offset a sumar a *regNum* para obtener el índice:

$$stwin\_mul\_win\_size = \text{STWIN} * Window\_Size$$

Por último, se emplea el atributo *outsCwp0* para conocer el offset del índice de los registros de salida de la ventana indicada cuando CWP = 0 (o AWP si AW es 1), ya que ahora los registros de salida de la ventana STWIN deben coincidir con los registros de entrada de la ventana STWIN+CWPMAX, porque el rango de ventanas es de STWIN a STWIN+CWPMAX.

$$outsCwp0 = ((\text{STWIN} + \text{CWPMAX} + 1) * Window\_Size) \text{ AND } totalWindowSize$$

El offset *outsCwp0* se aplica únicamente en el caso de que se quieran obtener los registros de salida cuando el puntero de ventana es 0, es decir, cuando *cwp_mult_WINDOW_SIZE* es 0 y *regNum* está en el rango de 8 a 15, que indica que son de salida.

En resumen, en caso de querer los registros de salida cuando CWP = 0 (o AWP = 0 si AW = 1):

$$\text{índice} = outsCwp0 + regNum$$

En cualquier otro caso:

$$\text{índice} = (cwp\_mult\_WINDOW\_SIZE + stwin\_mul\_win\_size + regNum) \text{ AND } totalWindowSize$$

Ambos métodos, *setRegisterValue* y *getRegisterValue* se modifican con estas nuevas formulas de obtención del índice del array de registros, y se muestran a continuación:

-- Cambios en los métodos de obtención de registros --

```
1   uint32_t SparcISS::getRegisterValue(uint32_t regNUM) {
2       if (regNUM <= I7) {
3           if (regNUM < 8){ // Globales
4               return globals[regNUM];
5           } else {
6               uint32_t regN;
7
8               // Si se buscan los registros de salida con CWP 0
9               if (!cwp_mult_WINDOW_SIZE && regNUM < 16) {
10                  regN = outsCwp0 + regNUM;
11              } else {  // Si no, se usa stwin y cwp
12                  regN = (cwp_mult_WINDOW_SIZE + stwin_mul_win_size + regNUM) &
13                         totalWindowsSize;
14              }
15              return registers[regN];
16          (...)
17  }
18
19  void SparcISS::setRegisterValue(uint32_t regNUM, uint32_t value) {
20      if (regNUM <= I7) {
21          if (regNUM < 8) {
22              if (regNUM != 0) {
23                  globals[regNUM] = value;
```

```
            }
        } else {
            uint32_t regN;
            if (!cwp_mult_WINDOW_SIZE && regNUM < 16) {
                regN = outsCwp0 + regNUM;
            } else {
                regN = (cwp_mult_WINDOW_SIZE + stwin_mul_win_size + regNUM) &
                       totalWindowsSize;
            }

            registers[regN] = value;
       (...)
    }
```

Código 48 - Cambios en los métodos de obtención de registros - SparcISS.cpp

4.3.2.5 Cambios en las instrucciones de movimiento de ventana

Finalmente, se modifican los métodos que ejecutan las instrucciones de cambio de ventana, entre los que se incluyen *_SAVE, _RESTORE, _RETT* y *executeTRAP()*.

Los métodos *_SAVE* y *_RESTORE* simplemente llaman a *getRegisterWindow()* para obtener la ventana a la que se va a mover, y comprueban si hay desbordamiento solo si AW es 0, tras lo cual modifican la ventana usando el método *setRegisterWindow()*, dando de argumento la ventana obtenida al llamar a *getRegisterWindow()*. Las modificaciones se muestran a continuación:

-- Cambios en los métodos de SAVE y RESTORE --

```
void _SAVE(SparcISS *pISS) {
    // Obtiene ventana de CWP-1 o AWP-1
    uint32_t newCWP = pISS->getRegisterWindow(-1);

    // Si coincide con wim Y aw es 0, genera window overflow
    if ((pISS->wim & (1 << newCWP)) && !pISS->aw) {
        pISS->trap = TRAP_WOFL;
        return;
    }
    pISS->setRegisterWindow(newCWP); // Establece la nueva ventana

  (...)

}

void _RESTORE(SparcISS *pISS) {
    // Obtiene ventana de CWP+1 o AWP+1
    uint32_t newCWP = pISS->getRegisterWindow(1);

    // Si coincide con wim Y aw es 0, genera window underflow
    if ((pISS->wim & (1 << newCWP)) && !pISS->aw) {
        pISS->trap = TRAP_WUFL;
        return;
    }
    pISS->setRegisterWindow(newCWP); // Establece la nueva ventana

  (...)

}
```

Código 49 - Cambios en los métodos de SAVE y RESTORE - SparcISS.cpp

Los métodos para *trap* y RETT además añaden los cambios en AW y PAW, ya que como define la funcionalidad AWP, un *trap* copia AW en PAW y pone AW a 0, y un RETT copia PAW en AW y no comprueba *window underflow* si PAW es 0, como se muestra en el siguiente código:

-- Cambios en los métodos de executeTRAP y RETT --

```
1   const uint8_t *SparcISS::executeTRAP() {
2
3       if (aw) { // Si aw es 1, pone paw a 1 y aw a 0
4           psr = (psr & ~AW_MASK) | PAW_MASK;
5           aw = 0;
6           paw = 1;
7       } else { // Si aw es 0, ambos paw y aw se ponen a 0
8           psr &= ~PAW_MASK; // PAW and AW '0'
9           paw = 0;
10      }
11
12      (...) Comprobación de traps
13
14      // Establece la ventana CWP-1 o AWP-1
15      setRegisterWindow(getRegisterWindow(-1));
16
17      (...) Salto a manejador de Trap o Error
18
19  }
20
21  void _RETT(SparcISS *pISS) {
22
23      (...)
24
25      // Obtiene ventana de CWP+1 o AWP+1
26      uint32_t newCWP = pISS->getRegisterWindow(1);
27
28      // Comprueba si hay underflow solo si paw es 0
29      if ((pISS->wim & (1 << newCWP)) && !pISS->paw) {
30          pISS->trap = TRAP_WUFL;
31          return;
32      }
33
34      (...)
35
36      // Establece ventana
37      pISS->setRegisterWindow(newCWP);
38
39      // copia paw en aw
40      pISS->aw = pISS->paw;
41
42      if (pISS->aw) {
43          pISS->psr |= AW_MASK;
44      } else {
45          pISS->psr &= ~AW_MASK;
46      }
47
48      // Y vuelve a establecer ventana en caso de que aw haya cambiado
49      pISS->setRegisterWindow(pISS->getRegisterWindow());
50
51      (...)
52  }
```

Código 50 - Cambios en los métodos de executeTRAP y RETT - SparcISS.cpp

4.4 Validación

Al igual que con la validación de la MMU simulada, se va a crear una serie de programas usando el compilador cruzado BCC, comprobando que los resultados obtenidos por el simulador y la placa con LEON3 coinciden.

4.4.1 Objetivos de la validación

La funcionalidad implementada añade un puntero alternativo a las ventanas de registros, además de poder particionar dichas ventanas, por lo que se debe demostrar que el nuevo puntero maneja las ventanas de forma correcta, y que las particiones de ventanas limitan las ventanas al rango definido además de unir la primera ventana de la partición con la última.

Para el puntero alternativo se busca demostrar que, una vez en uso, se mueve a las ventanas seleccionadas de forma correcta, que no comprueba excepciones por desbordamiento, y que en caso de excepción se cambian los bits AW y PAW de PSR como se define.

Para las particiones de ventanas, comprobar que, tras crear una partición, el puntero de ventana que esté en uso se mantiene siempre dentro de dicha partición, además de que la salida de la primera ventana de la partición se une a la entrada de la última ventana.

4.4.2 Diseño del programa de validación

Se han implementado 2 programas para realizar la validación de la implementación: uno para validar el puntero alternativo y otro para las particiones.

El test del puntero alternativo consistirá en simplemente hacer varias instrucciones SAVE y RESTORE, tanto con el puntero normal CWP como con el alternativo AWP, y escribiendo en cada cambio de ventana el estado de los valores de registros relevantes, como AW, PAW, CWP y AWP. Se busca llegar al punto en el que se produciría un desbordamiento usando ambos punteros, demostrando que con AWP no se produce el *trap*. Finalmente, se fuerza un *trap* y ver como éste cambia los valores de AW y PAW.

En cuanto al test de particiones de registros, se crea una partición diferente a la de por defecto, y se realizan varios SAVE y RESTORE mostrando como los punteros se mueven solamente entre las ventanas de dicha partición, y que los registros de entrada de la última ventana se unen con los de salida de la primera.

Ambos test presentan el problema de que las funciones de impresión en pantalla ofrecidas por las librerías de C realizan múltiples cambios de ventana, lo cual puede entrar en conflicto con el resultado buscado. Para solucionar esto se va a crear un método simple de escritura directamente en ensamblador, usando la salida de la UART, con el fin de imprimir por pantalla sin realizar ningún cambio de ventana.

Por último, ha sido necesario modificar los manejadores de desbordamiento predefinidos por el compilador cruzado ya que los originales no tienen en cuenta la posibilidad de que se usen menos ventanas de las disponibles.

4.4.3 Estructura del programa de validación

- **defines.h**: da nombre a diferentes valores usados en la arquitectura SPARCv8, como vectores de *trap* y máscaras para campos de registros, mejorando la legibilidad del programa.

- **assembly.S**: ofrece varias funciones escritas en ensamblador usando la ISA SPARCv8 para realizar ciertas operaciones que solo pueden realizarse en ensamblador.

- **leon3_traps.c**: define una función que permite la creación y sobreescritura de *traps*. Se obtiene de los ejemplos dados en el manual de uso del compilador BCC.

- **main.c**: llama a las diferentes funciones creadas para la realización de los tests.

4.4.4 Descripción del programa de validación

4.4.4.1 Definiciones

Para mejorar la legibilidad del programa se crea la cabecera *defines.h* la cual da un nombre a los diferentes vectores de *trap* y máscaras de campos de registros que se usan en la creación de este programa. El código de esta cabecera se adjunta a continuación:

-- definiciones para pruebas de validación de AWP --

```
1   #ifndef DEFINES_H_
2   #define DEFINES_H_
3
4   /** Traps */
5   #define LEON3_WINDOW_OVERFLOW_TRAPNUM 5  //Numero de trap de overflow
6   #define LEON3_WINDOW_UNDERFLOW_TRAPNUM 6 //Numero de trap de underflow
7   #define LEON3_SPARC_TEST_TRAPNUM 0x85    //Numero de trap de test
8
9   /* others */
10  #define N_WINDOWS 8   // Número máximo de ventanas
11
12  /** Masks */
13  #define CWP_MASK 0x0000001F  //Máscara del campo CWP de PSR
14  #define AW_MASK 0x00008000   //Máscara del campo AW de PSR
15  #define AW_POS 0xF           //Bit de inicio del campo AW de PSR
16  #define PAW_MASK 0x00004000  //Máscara del campo PAW de PSR
17  #define PAW_POS 0xE          //Bit de inicio del campo PAW de PSR
18  #define AWP_MASK 0x0000001F  //Máscara del campo AWP de ASR20
19  #define STWIN_MASK 0x03E00000       //Máscara del campo STWIN de ASR20
20  #define STWIN_POS 0x15              //Bit de inicio del campo STWIN de ASR20
21  #define CWPMAX_MASK 0x001F0000      //Máscara del campo CWPMAX de ASR20
22  #define CWPMAX_POS 0x10             //Bit de inicio del campo CWPMAX de ASR20
23  #define WCWP_MASK 0x00000020 //Máscara del campo WCWP de ASR20
24  #define WCWP_POS 0x5         //Bit de inicio del campo WCWP de ASR20
```

```
25
26  #endif /* DEFINES_H_ */
```

Código 51 - Definiciones del programa de validación de AWP - defines.h

4.4.4.2 Método de impresión por pantalla

El objetivo del programa de validación es mostrar de forma clara qué esta ocurriendo. El problema es que los métodos ofrecidos para realizar impresiones por pantalla realizan múltiples cambios de ventana, lo cual entra en conflicto con las pruebas que se quieren realizar para esta validación. Por ello se ha creado un método en ensamblador, mostrado en *Código 52 - Métodos para impresión por pantalla - assembly.S*, encargado de realizar estas impresiones sin hacer cambios de ventana. Cuando se llama a este método no se producen instrucciones de SAVE ni RESTORE, pero es necesario considerar en todo momento qué registros se usan, ya que se usa la ventana de la función que llamó a este método.

Para evitar sobrescribir registros locales que puedan estar siendo usados, se emplean los registros de salida (*o0*-*o5*). Cada vez que se hace una llamada (*call*), se guarda automáticamente la dirección de retorno en el registro de salida 7 (*o7*), por lo que, si se hacen varias llamadas, hay que guardar previamente dicha dirección en otro registro para evitar perderla. Además, el registro de salida 6 (*o6*) guarda el puntero de marco de pila, por lo que no debe modificarse en ningún caso.

Cada vez que se quiere imprimir algo, la función principal hace una llamada a un método de inicio en ensamblador que comienza mostrando la situación actual (Si se está en ejecución normal, o SAVE, o RESTORE...). Desde éste se llama a un segundo método llamado *AWPStatusPrint*, que obtiene y manda imprimir valores de los campos de registros relevantes para las pruebas (AW, PAW, CWP, AWP...). Ambos métodos realizan las impresiones usando un tercero denominado *write_char*, que es el encargado de usar la UART para imprimir por pantalla.

Por ejemplo, se busca imprimir el estado tras un SAVE, lo cual primero llama al método en ensamblador *savePrint* que imprime "SAVE: " usando *write_char*. Tras imprimir eso, llama a *AWPStatusPrint* para que mande imprimir los valores de los registros comentados. Como al llamar a *write_char* y *AWPStatusPrint* se sustituye el registro de salida 7 por la nueva dirección de retorno, antes de realizar dichas llamadas guarda su dirección de retorno en un registro que previamente se ha comprobado no se va a modificar.

-- Métodos para impresión en pantalla --

```
1   // write_char: Imprime el caracter que haya en el registro de salida 0 (o0)
2   write_char:
3       %hi(0x80000000), %o1
4       or %o1, 0x100, %o1  // o1 = 0x80001000 = dirección de salida de UART
5       mov 0xFFF, %o2      // o2 = 0xFFF = contador de espera
6   wait:
7       subcc %o2, 0x1, %o2  // o2 = o2 - 1, comprobando si llega a 0
8       bnz wait   // si o2 no es 0, vuelve a la etiqueta 'wait'
9       nop        // Hueco de retardo (Delay Slot)
10
```

```
11      st  %o0, [%o1]  // Salida Uart = o0 = caracter a imprimir
12      retl           // retorno
13      nop
14
15
16  // AWPStatusPrint: Obtiene y manda imprimir AW, PAW, CWP, AWP, STWIN, CWPMAX y WIM
17      .globl AWPStatusPrint
18  AWPStatusPrint:
19      mov %o7, %o3   // guarda dirección de retorno en o3
20
21      // Escribe AW: Valor
22      mov 'A', %o0
23      call write_char
24      nop
25      mov 'W', %o0
26      call write_char
27      nop
28      mov ' ', %o0
29      call write_char
30      nop
31      mov %psr, %o0         // o0 = psr
32      srl %o0, AW_POS, %o0  // o0 = psr >> 15
33      and %o0, 0x1, %o0     // o0 = o0 & 1
34      add %o0, 0x30, %o0    // o0 = o0 + 0x30 (decimal de 1 dígito a carácter)
35
36      (...) Escribe el resto de valores
37
38      mov %o3, %o7      // restaura dirección de retorno
39      retl  // retorna
40      nop
41
42  // Método de escritura de estado para llamar cuando se realice save
43      .globl savePrint
44  savePrint:
45      mov %o7, %o5  // guarda dirección de retorno
46
47      // Manda escribir SAVE:
48      mov 'S', %o0
49      call write_char
50      nop
51      mov 'A', %o0
52      call write_char
53      nop
54      mov 'V', %o0
55      call write_char
56      nop
57      mov 'E', %o0
58      call write_char
59      nop
60      mov ':', %o0
61      call write_char
62      nop
63      mov ' ', %o0
64      call write_char
65      nop
66      call AWPStatusPrint    // llama al método de escritura de estado
67      nop
68      mov %o5, %o7           // restaura dirección de retorno
69      retl   // retorna
70      nop
```

Código 52 - Métodos para impresión por pantalla - assembly.S

4.4.4.3 Manejadores de desbordamiento

Los manejadores de desbordamiento ofrecidos por el compilador cruzado modifican el registro WIM con un desplazamiento a derechas o izquierdas dependiendo de si es por SAVE (*overflow*) o RESTORE (*underflow*) respectivamente, sin considerar que haya un número de ventanas diferente al de por defecto. Por lo que, si se utiliza una partición de menos ventanas, en el momento que haya un desbordamiento en uno de los límites de dicha partición, el manejador original modificará el registro WIM de manera que se invalide una ventana que no está en uso, lo cual hace que el programa se comporte de forma inesperada, normalmente acabando en un error. Esto hace necesario definir nuevos manejadores de desbordamiento que modifiquen WIM en función del tamaño de partición actual, lo cual se puede comprobar mirando el campo CWPMAX del registro ASR20.

Ambos, el de *window overflow* y *window underflow*, comienzan usando *write_char* y *AWPStatusPrint* para dar a conocer que ha ocurrido un desbordamiento, por ello, antes de llamarlos, se guardan los registros de entrada y salida que contienen los marcos de pila y retornos, para asegurar que no se modifiquen. Tras ello, dependiendo de si es *overflow* o *underflow*, se realizan los pasos descritos en el *4.2.3 Desbordamiento de ventanas*, pero al desplazar WIM se considera la partición indicada por CWPMAX. El código de ambos se adjunta a continuación:

-- Nuevos métodos para manejar desbordamientos --

```
1    .globl wind_overflow  // Método de window overflow
2    wind_overflow:
3
4        // Guarda registros de marco de pila y dirección de retorno
5        // Solo es necesario si se va a llamar a otro metodo
6        mov %o6, %l3
7        mov %o7, %l4
8        mov %i6, %l5
9        mov %i7, %l6
10
11       // Escribe *OVR:
12       mov '\n', %o0
13       call write_char
14       nop
15       mov '*', %o0
16       call write_char
17       nop
18       mov 'O', %o0
19       call write_char
20       nop
21       mov 'V', %o0
22       call write_char
23       nop
24       mov 'R', %o0
25       call write_char
26       nop
27       mov ':', %o0
28       call write_char
29       nop
30       mov ' ', %o0
31       call write_char
32       nop
```

```
33
34      call AWPStatusPrint  // Escribe estado de campos de ventanas
35      nop
36
37      // Restaura registros de marco de pila y dirección de retorno
38      mov %l3, %o6
39      mov %l4, %o7
40      mov %l5, %i6
41      mov %l6, %i7
42
43      // Realiza pasos de window overflow considerando CWPMAX
44      mov  %wim, %l3           // l3 = wim register
45      mov  %g1, %l7            // l7 = g1
46      mov %asr20, %l4          // l4 = asr20
47      sethi  %hi(CWPMAX_MASK), %l5    //l5 = máscara CWPMAX
48      andcc  %l5, %l4, %l4            // l4 (numero ventanas) = valor CWPMAX
49      bnz save_window                 // Si CWPMAX != 0, ve a save_window
50      srl %l4, CWPMAX_POS, %l4        // (Delay Slot) CWPMAX >> 16
51      mov N_WINDOWS - 1, %l4   // Si CWPMAX era 0, l4 = N_WINDOWS-1
52  save_window:
53      srl  %l3, 1, %g1   // g1 = wim >> 1
54      sll  %l3, %l4, %l4 // l4 = l3 << l4 (wim << MaxWindows-1)
55      or  %l4, %g1, %g1  // g1 = g1 | l4 ((wim >> 1) | (wim << MaxWindows-1))
56      save               // cwp+1 (window to save)
57      mov  %g1, %wim     // wim = g1
58      std  %l0, [ %sp ]  // Guarda registros en pila
59      std  %l2, [ %sp + 8 ]
60      std  %l4, [ %sp + 0x10 ]
61      std  %l6, [ %sp + 0x18 ]
62      std  %i0, [ %sp + 0x20 ]
63      std  %i2, [ %sp + 0x28 ]
64      std  %i4, [ %sp + 0x30 ]
65      std  %fp, [ %sp + 0x38 ]
66      restore        //cwp-1, vuelta a ventana de trap
67      mov  %l7, %g1  //restaura g1
68      jmp  %l1
69      rett  %l2      //rett
70      nop
71
72
73  .globl wind_underflow  // Método de window underflow
74  wind_underflow:
75
76      // Guarda registros de marco de pila y dirección de retorno
77      // Solo es necesario si se va a llamar a otro metodo
78      mov %o6, %l3
79      mov %o7, %l4
80      mov %i6, %l5
81      mov %i7, %l6
82
83      // Escribe *UND:
84      mov '\n', %o0
85      call write_char
86      nop
87      mov '*', %o0
88      call write_char
89      nop
90      mov 'U', %o0
91      call write_char
92      nop
93      mov 'N', %o0
```

```
94      call write_char
95      nop
96      mov 'D', %o0
97      call write_char
98      nop
99      mov ':', %o0
100     call write_char
101     nop
102     mov ' ', %o0
103     call write_char
104     nop
105
106     call AWPStatusPrint
107     nop
108
109     // Restaura registros de marco de pila y dirección de retorno
110     mov %l3, %o6
111     mov %l4, %o7
112     mov %l5, %i6
113     mov %l6, %i7
114
115     // Realiza pasos de window underflow considerando CWPMAX
116     rd  %wim, %l3        // l3 = wim
117     mov %asr20, %l4      // l4 = asr20
118     sethi  %hi(CWPMAX_MASK), %l5    //l5 = máscara CWPMAX
119     andcc  %l5, %l4, %l4 // l4 (número ventanas) = campo CWPMAX
120     bnz restore_window   // Si CWPMAX != 0, ir a restore_window
121     srl %l4, CWPMAX_POS, %l4  // (Delay Slot) CWPMAX >> 16
122     mov N_WINDOWS - 1, %l4    // Si CWPMAX era 0, l4 = N_WINDOWS-1
123 restore_window:
124     sll  %l3, 1, %l5     // l5 = wim << 1
125     srl  %l3, %l4, %l3   // l3 = wim >> N_Windows-1
126     or  %l3, %l5, %l3    // l3 = (wim << 1) | (wim >> N_Windows-1)
127     mov  %l3, %wim       // wim = l3
128     nop
129     nop
130     nop
131     restore
132     restore    // Va a la ventana a cargar (cwp+2)
133     ldd  [ %sp ], %l0      // Carga ventana desde pila
134     ldd  [ %sp + 8 ], %l2
135     ldd  [ %sp + 0x10 ], %l4
136     ldd  [ %sp + 0x18 ], %l6
137     ldd  [ %sp + 0x20 ], %i0
138     ldd  [ %sp + 0x28 ], %i2
139     ldd  [ %sp + 0x30 ], %i4
140     ldd  [ %sp + 0x38 ], %fp
141     save
142     save    // Vuelve a la ventana de trap (cwp-2)
143     jmp  %l1
144     rett  %l2    // Rett
145   nop
```

Código 53 - Nuevos métodos para manejar desbordamientos - assembly.S

4.4.4.4 Método de test de excepción en AWP y escritura de manejadores de trap.

Para comprobar como cambian los campos AW y PAW cuando ocurre un *trap*, se ha definido un manejador de *trap* usando uno de los vectores software libres, en este caso el vector 0x85. Este provoca un trap haciendo uso de la instrucción TA, y lo único que hace es imprimir que ha ocurrido un *trap* y muestra el estado de los valores de registros relevantes usando el método *AWPStatusPrint*, como se muestra en el siguiente código:

-- Métodos de test de excepción --

```
1       .globl trap_testTrap_handler  // Manejador de trap de test
2   trap_testTrap_handler:
3
4       // Escribe TRAP:
5       mov 'T', %o0
6       call write_char
7       nop
8       mov 'R', %o0
9       call write_char
10      nop
11      mov 'A', %o0
12      call write_char
13      nop
14      mov 'P', %o0
15      call write_char
16      nop
17      mov ':', %o0
18      call write_char
19      nop
20      mov ' ', %o0
21      call write_char
22      nop
23
24      call AWPStatusPrint  // Llamada a impresión de registros
25      nop
26
27      jmp %l2
28      rett %l2+4
29      nop
30
31      .globl trap_call_testTrap  // Llamada al manejador de trap de test
32  trap_call_testTrap:
33      ta LEON3_SPARC_TEST_TRAPNUM
34      retl
35      nop
```

Código 54 - Métodos de test de excepción - assembly.S

En cuanto a la reescritura de manejadores de *trap*, se usa el mismo sistema que se empleó en la validación de la MMU, descrito en el apartado *3.4.4.4 Tratamiento de excepciones*, es decir, empleando el método de instalación de manejadores de *trap leon3_install_handler* escrito en *leon3_traps.c*, se instalan los nuevos manejadores de *trap*s a utilizar como se muestra debajo:

-- Instalación de manejadores de *trap* --

```
1    int main() {
2
3        leon3_install_handler(LEON3_SPARC_TEST_TRAPNUM, trap_testTrap_handler);
4        leon3_install_handler(LEON3_WINDOW_OVERFLOW_TRAPNUM, wind_overflow);
5        leon3_install_handler(LEON3_WINDOW_UNDERFLOW_TRAPNUM, wind_underflow);
6
7        (...)
8
9    }
```

Código 55 - instalación de manejadores de trap - main.c

4.4.4.5 Otros métodos útiles

Al igual que pasaba con la validación de la MMU, ciertas operaciones solo pueden realizarse usando directamente ensamblador, como cuando se quiere leer o escribir ciertos registros. Por ello se definen varios métodos para la lectura y escritura de los registros PSR, ASR20 y WIM, mostrados en el código siguiente:

-- Métodos de lectura y escritura de registros del sistema --

```
1    .globl wrAsr20  // Escribe ASR20
2    wrAsr20:
3        retl
4        mov %o0, %asr20
5
6    .globl rdAsr20  // Lee ASR20
7    rdAsr20:
8        retl
9        mov %asr20, %o0
10
11   .globl wrPsr  // Escribe PSR
12   wrPsr:
13       retl
14       mov %o0, %psr
15
16   .globl rdPsr  // Lee PSR
17   rdPsr:
18       retl
19       mov %psr, %o0
20
21   .globl wrWim  // Escribe WIM
22   wrWim:
23       retl
24       mov %o0, %wim
25
26   .globl rdWim  // Lee WIM
27   rdWim:
28       retl
29       mov %wim, %o0
```

Código 56 - Métodos de lectura y escritura de registros del sistema - assembly.S

En el *main()* además se definen 2 métodos para facilitar las pruebas a realizar. El primero de ellos es la función *createAsr20Val*, que crea el valor a escribir en el registro ASR20 en función de los valores STWIN, CWPMAX, WCWP y AWP que se le indique en los argumentos. El segundo es el método *recursion* que se llama a si mismo tantas veces como se le diga en el argumento, y resulta útil para realizar varios SAVE y RESTORES seguidos. Ambos métodos se muestran en el siguiente código:

-- Utilidades para pruebas --

```
1    // Retorna valor a escribir en ASR20 según los valores de los campos dados
2    uint32_t createAsr20Val(uint32_t STWIN, uint32_t CWPMAX, uint32_t WCWP, uint32_t awp){
3        return awp + (WCWP << 5) + (CWPMAX << 16) + (STWIN << 21);
4    }
5
6    // Método recursivo que se llama a si mismo tantas veces como 'times'
7    void recursion(int times){
8        savePrint();   // indica que ocurrió un SAVE, imprime registros
9        if (times > 1){
10           recursion(times-1); // se llama a sí mismo
11       }
12       restorePrint();// Indica que ocurrió un restore, imprime registros
13   }
```

Código 57 - Métodos útiles para pruebas de ventanas - main.c

4.4.4.6 Test de puntero alternativo

El test del puntero alternativo comienza haciendo un *'printf'* indicando qué puntero se va a usar. No se utiliza el método definido en ensamblador para imprimir caracteres ya que, como se comienza usando el CWP, no hay problema en que haga varios cambios de ventana.

A continuación realiza una recursión de 7 llamadas usando CWP para demostrar el desbordamiento y se cambia a usar AWP poniendo el campo AW de PSR a 1. Tras ello, vuelve a hacer una recursión de 7 llamadas que demuestra que se usa AWP, además de que no se produce desbordamiento, aunque se llegue a la ventana marcada como inválida en el WIM. Se termina haciendo una llamada al *trap* de test, mostrando como cambian los bits de AW y PAW al producirse un *trap* y retornar de éste. El siguiente código muestra el método principal, donde se realiza el proceso descrito:

-- Test de puntero alternativo --

```
1    int main() {
2        leon3_install_handler(LEON3_SPARC_TEST_TRAPNUM, trap_testTrap_handler);
3        leon3_install_handler(LEON3_WINDOW_OVERFLOW_TRAPNUM, wind_overflow);
4        leon3_install_handler(LEON3_WINDOW_UNDERFLOW_TRAPNUM, wind_underflow);
5
6        CurAwpPrint(); // Situación actual
7        printf("CWP SAVE/RESTORE test \n");
8        recursion(7); // Recursión 7 veces usando CWP
9
10       printf("\nAWP SAVE/RESTORE test \n");
11
12       uint32_t awPsr = rdPsr() | AW_MASK; // Se obtiene valor de PSR + AW
13       wrPsr(awPsr); // Se escribe el valor anterior, lo que activa AW
```

```
14        __asm__("nop\n" "nop\n" "nop\n");  // Se debe esperar a que se escriba el registro
15
16        CurAwpPrint(); // Situación actual usando AWP
17        recursion(7);  // recursión 7 veces usando AWP
18        CurAwpPrint(); // Situación actual usando AWP
19
20        trap_call_testTrap(); // Llamada a trap de test usando AWP
21        CurAwpPrint(); // Situación actual tras retorno del trap
22
23        return 0;
24    }
```

Código 58 - Test de puntero alternativo - main.c

4.4.4.7 Resultados del test de puntero alternativo

Se ha ejecutado el programa creado en el simulador LeonViP, mostrando su resultado en el archivo AwpTestWim.txt, y luego en la FPGA programada con la implementación de LEON3 con el AWP activado, el cual imprime su resultado en el archivo AwpTestPlaca.txt. Finalmente, usando un plugin de comparación de Notepad++, se ha comprobado que los resultados coinciden.

AwpTestSim.txt

```
1   CURR: AW 0, PAW 0, CWP 6, AWP 6, STW 0, CMX 7, WIM 1
2
3   *OVR: AW 0, PAW 0, CWP 1, AWP 1, STW 0, CMX 7, WIM 1
4
5   *OVR: AW 0, PAW 0, CWP 0, AWP 0, STW 0, CMX 7, WIM 0
6   CWP SAVE/RESTORE test
7   SAVE: AW 0, PAW 0, CWP 5, AWP 5, STW 0, CMX 7, WIM 7
8   SAVE: AW 0, PAW 0, CWP 4, AWP 4, STW 0, CMX 7, WIM 7
9   SAVE: AW 0, PAW 0, CWP 3, AWP 3, STW 0, CMX 7, WIM 7
10  SAVE: AW 0, PAW 0, CWP 2, AWP 2, STW 0, CMX 7, WIM 7
11  SAVE: AW 0, PAW 0, CWP 1, AWP 1, STW 0, CMX 7, WIM 7
12  SAVE: AW 0, PAW 0, CWP 0, AWP 0, STW 0, CMX 7, WIM 7
13
14  *OVR: AW 0, PAW 0, CWP 7, AWP 7, STW 0, CMX 7, WIM 7
15  SAVE: AW 0, PAW 0, CWP 7, AWP 7, STW 0, CMX 7, WIM 6
16  REST: AW 0, PAW 0, CWP 7, AWP 7, STW 0, CMX 7, WIM 6
17  REST: AW 0, PAW 0, CWP 0, AWP 0, STW 0, CMX 7, WIM 6
18  REST: AW 0, PAW 0, CWP 1, AWP 1, STW 0, CMX 7, WIM 6
19  REST: AW 0, PAW 0, CWP 2, AWP 2, STW 0, CMX 7, WIM 6
20  REST: AW 0, PAW 0, CWP 3, AWP 3, STW 0, CMX 7, WIM 6
21  REST: AW 0, PAW 0, CWP 4, AWP 4, STW 0, CMX 7, WIM 6
22  REST: AW 0, PAW 0, CWP 5, AWP 5, STW 0, CMX 7, WIM 6
23
24  *UND: AW 0, PAW 0, CWP 4, AWP 4, STW 0, CMX 7, WIM 6
25
26  AWP SAVE/RESTORE test
27  CURR: AW 1, PAW 0, CWP 6, AWP 6, STW 0, CMX 7, WIM 7
28  SAVE: AW 1, PAW 0, CWP 6, AWP 5, STW 0, CMX 7, WIM 7
29  SAVE: AW 1, PAW 0, CWP 6, AWP 4, STW 0, CMX 7, WIM 7
30  SAVE: AW 1, PAW 0, CWP 6, AWP 3, STW 0, CMX 7, WIM 7
31  SAVE: AW 1, PAW 0, CWP 6, AWP 2, STW 0, CMX 7, WIM 7
32  SAVE: AW 1, PAW 0, CWP 6, AWP 1, STW 0, CMX 7, WIM 7
33  SAVE: AW 1, PAW 0, CWP 6, AWP 0, STW 0, CMX 7, WIM 7
34  SAVE: AW 1, PAW 0, CWP 6, AWP 7, STW 0, CMX 7, WIM 7
35  REST: AW 1, PAW 0, CWP 6, AWP 7, STW 0, CMX 7, WIM 7
36  REST: AW 1, PAW 0, CWP 6, AWP 0, STW 0, CMX 7, WIM 7
37  REST: AW 1, PAW 0, CWP 6, AWP 1, STW 0, CMX 7, WIM 7
38  REST: AW 1, PAW 0, CWP 6, AWP 2, STW 0, CMX 7, WIM 7
39  REST: AW 1, PAW 0, CWP 6, AWP 3, STW 0, CMX 7, WIM 7
40  REST: AW 1, PAW 0, CWP 6, AWP 4, STW 0, CMX 7, WIM 7
41  REST: AW 1, PAW 0, CWP 6, AWP 5, STW 0, CMX 7, WIM 7
42  CURR: AW 1, PAW 0, CWP 6, AWP 6, STW 0, CMX 7, WIM 7
43  TRAP: AW 0, PAW 1, CWP 5, AWP 6, STW 0, CMX 7, WIM 7
44  CURR: AW 1, PAW 1, CWP 6, AWP 6, STW 0, CMX 7, WIM 7
45
46  40001FFC : TRAP: Unimplemented...
47
```

AWPTestPlaca.txt

```
1   CURR: AW 0, PAW 0, CWP 6, AWP 6, STW 0, CMX 7, WIM 1
2
3   *OVR: AW 0, PAW 0, CWP 1, AWP 1, STW 0, CMX 7, WIM 1
4
5   *OVR: AW 0, PAW 0, CWP 0, AWP 0, STW 0, CMX 7, WIM 0
6   CWP SAVE/RESTORE test
7   SAVE: AW 0, PAW 0, CWP 5, AWP 5, STW 0, CMX 7, WIM 7
8   SAVE: AW 0, PAW 0, CWP 4, AWP 4, STW 0, CMX 7, WIM 7
9   SAVE: AW 0, PAW 0, CWP 3, AWP 3, STW 0, CMX 7, WIM 7
10  SAVE: AW 0, PAW 0, CWP 2, AWP 2, STW 0, CMX 7, WIM 7
11  SAVE: AW 0, PAW 0, CWP 1, AWP 1, STW 0, CMX 7, WIM 7
12  SAVE: AW 0, PAW 0, CWP 0, AWP 0, STW 0, CMX 7, WIM 7
13
14  *OVR: AW 0, PAW 0, CWP 7, AWP 7, STW 0, CMX 7, WIM 7
15  SAVE: AW 0, PAW 0, CWP 7, AWP 7, STW 0, CMX 7, WIM 6
16  REST: AW 0, PAW 0, CWP 7, AWP 7, STW 0, CMX 7, WIM 6
17  REST: AW 0, PAW 0, CWP 0, AWP 0, STW 0, CMX 7, WIM 6
18  REST: AW 0, PAW 0, CWP 1, AWP 1, STW 0, CMX 7, WIM 6
19  REST: AW 0, PAW 0, CWP 2, AWP 2, STW 0, CMX 7, WIM 6
20  REST: AW 0, PAW 0, CWP 3, AWP 3, STW 0, CMX 7, WIM 6
21  REST: AW 0, PAW 0, CWP 4, AWP 4, STW 0, CMX 7, WIM 6
22  REST: AW 0, PAW 0, CWP 5, AWP 5, STW 0, CMX 7, WIM 6
23
24  *UND: AW 0, PAW 0, CWP 4, AWP 4, STW 0, CMX 7, WIM 6
25
26  AWP SAVE/RESTORE test
27  CURR: AW 1, PAW 0, CWP 6, AWP 6, STW 0, CMX 7, WIM 7
28  SAVE: AW 1, PAW 0, CWP 6, AWP 5, STW 0, CMX 7, WIM 7
29  SAVE: AW 1, PAW 0, CWP 6, AWP 4, STW 0, CMX 7, WIM 7
30  SAVE: AW 1, PAW 0, CWP 6, AWP 3, STW 0, CMX 7, WIM 7
31  SAVE: AW 1, PAW 0, CWP 6, AWP 2, STW 0, CMX 7, WIM 7
32  SAVE: AW 1, PAW 0, CWP 6, AWP 1, STW 0, CMX 7, WIM 7
33  SAVE: AW 1, PAW 0, CWP 6, AWP 0, STW 0, CMX 7, WIM 7
34  SAVE: AW 1, PAW 0, CWP 6, AWP 7, STW 0, CMX 7, WIM 7
35  REST: AW 1, PAW 0, CWP 6, AWP 7, STW 0, CMX 7, WIM 7
36  REST: AW 1, PAW 0, CWP 6, AWP 0, STW 0, CMX 7, WIM 7
37  REST: AW 1, PAW 0, CWP 6, AWP 1, STW 0, CMX 7, WIM 7
38  REST: AW 1, PAW 0, CWP 6, AWP 2, STW 0, CMX 7, WIM 7
39  REST: AW 1, PAW 0, CWP 6, AWP 3, STW 0, CMX 7, WIM 7
40  REST: AW 1, PAW 0, CWP 6, AWP 4, STW 0, CMX 7, WIM 7
41  REST: AW 1, PAW 0, CWP 6, AWP 5, STW 0, CMX 7, WIM 7
42  CURR: AW 1, PAW 0, CWP 6, AWP 6, STW 0, CMX 7, WIM 7
43  TRAP: AW 0, PAW 1, CWP 5, AWP 6, STW 0, CMX 7, WIM 7
44  CURR: AW 1, PAW 1, CWP 6, AWP 6, STW 0, CMX 7, WIM 7
45
46  Error mode  (tt = 0x02, illegal instruction)
47  0x40000020: 91d02000  ta  0
```

Ilustración 13 - Resultados de test de puntero alternativo

Como se puede comprobar en la ilustración anterior, cada vez que se muestran los valores de los registros a analizar, se preceden de un texto el cual indica la situación en la que se está:

- CURR indica que se está en la función principal (*main*).
- SAVE indica que acaba de realizarse un SAVE (comienzo de método de recursión).
- REST indica que va a realizarse un RESTORE (final del método de recursión).
- TRAP indica que se está dentro del manejador del test de *trap*.
- *OVR indica que se está dentro del manejador de *window overflow*.
- *UND indica que se está dentro del manejador de *window underflow*.

El plugin de comparación marca las líneas que tienen algún carácter que no coincide, y como se puede comprobar en la ilustración anterior, todo coincide excepto las 2 últimas líneas, las cuales solo sirven para indicar que ha ocurrido una excepción fatal por intentar ejecutar una instrucción inválida y que muestran de manera diferente sin afectar al resultado.

Como puede verse, tras escribir el estado de inicio en la línea 1, se producen 2 *overflow* debido al *printf*. Tras ello se realizan 7 SAVE y RESTORES por la llamada a recursión usando CWP, y se provoca un *overflow* cuando CWP coincide con WIM, y un *underflow* al volver al método principal. Tras esto, se cambia AW a 1, lo cual se muestra en la línea 27, y seguidamente se realizan de nuevo 7 SAVE y RESTORES mediante el método de recursión. Como puede verse, ahora solo se modifica el puntero de AWP, y además al llegar a la ventana marcada por WIM en la línea 34, no se produce *overflow*. Finalmente, tras volver a mostrar la situación actual tras finalizar la recursión, se llama al *trap* de test, mostrando su estado en la línea 43 y retorno en la línea 44. Como se puede observar, los campos AW y PAW cambian tal y como se describió en el funcionamiento del AWP.

Al final, cuando se retorna del *main* y finaliza el programa, ocurre un error por intentar ejecutar una instrucción no válida. Esto se debe a que, al haberse movido entre ventanas usando AWP sin considerar los posibles desbordamientos, se han sobrescrito valores clave para el retorno final, y termina yendo a una dirección con una instrucción no válida.

4.4.4.8 Test de particiones de ventanas de registros

El test de particiones comienza mostrando la situación inicial, tras lo cual se crea una partición de 4 ventanas, de la 4 a la 7 incluidas. Para ello se pone STWIN a 4, indicando que la ventana inicial es la 4, y CWPMAX a 3, indicando que el puntero máximo es 3 (de 0 a 3 = 4 ventanas).

Como tras entrar al *main* el CWP es 6 y se busca mantenerse en esa misma ventana, es necesario modificar el CWP al realizar esta partición. Como se describió en el apartado de particiones de registros, la ventana actual es igual a STWIN+CWP. De esta forma. si se quiere estar en la ventana 6, el CWP será el Ventana-STWIN, es decir 6-4 que es 2. Por lo tanto, se debe cambiar el CWP a 2 para mantenerse en la ventana 6. Para ello simplemente se deben poner los campos WCWP a 1

y AWP a 2, lo cual reescribe CWP a lo indicado en AWP. Tras escribir en ASR20, se cambia WIM a 1, lo cual indica que la última ventana libre es la apuntada por CWP = 0, es decir, en este caso la 4.

Una vez creada la partición, se vuelve a imprimir el estado actual demostrando que se cambió STWIN, CWPMAX, CWP y WIM, y se realiza una recursión de 4 llamadas, lo cual debe demostrar que el puntero en uso (en este caso CWP), se mantiene dentro de la partición creada. Esto también demuestra que se están uniendo de forma correcta las ventanas a los límites de la partición, ya que, si no fuese así, se perdería el valor que está indicando cuantas veces realizar el método recursivo, terminando en un resultado inesperado.

El siguiente código muestra el proceso principal en el cual se realiza lo descrito antes:

-- Test de partición de registros --

```
1   int main() {
2       leon3_install_handler(LEON3_SPARC_TEST_TRAPNUM, trap_testTrap_handler);
3       leon3_install_handler(LEON3_WINDOW_OVERFLOW_TRAPNUM, wind_overflow);
4       leon3_install_handler(LEON3_WINDOW_UNDERFLOW_TRAPNUM, wind_underflow);
5
6       CurAwpPrint();  // Situación actual
7
8       // Valor a escribir en asr20 con STWIN = 4, CWPMAX = 3, WCWP = 1 y AWP = 2
9       uint32_t part = createAsr20Val(4, 3, 1, 2);
10
11      wrAsr20(part);  // Escribe el valor obtenido en asr20
12      __asm__("nop\n" "nop\n" "nop\n");   // Espera fin de escritura del registro
13      wrWim(0x1);  // Escribe Wim marcando la ventana 0
14      __asm__("nop\n" "nop\n" "nop\n");   // Espera fin de escritura del registro
15
16      CurAwpPrint();  // Muestra situación actual, mostrando nueva partición
17      recursion(4);   // Recursión 4 veces
18      CurAwpPrint();  // Termina volviendo a mostrar situación actual
19
        return 0;
    }
```

Código 59 - Test de partición de registros - main.c

4.4.4.9 Resultados del test de partición de registros

Al igual que en los casos anteriores, se ha ejecutado el programa en el simulador y en la placa y se han comparado los resultados usando un plugin de *Notepad++*.

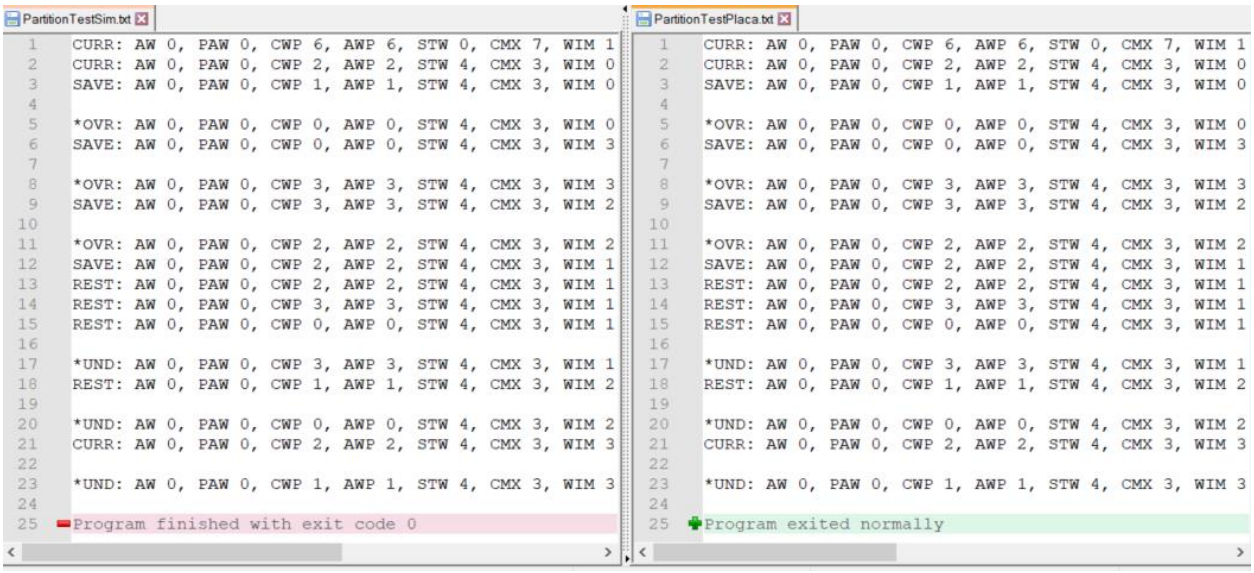


Ilustración 14 - Resultados de test de partición de registros

Como se puede ver en la ilustración anterior, en la línea 1 se muestra el estado de inicio, y en la línea 2 se muestra el estado tras crear la partición de registros. Como se puede observar, tras crear la partición CWP se cambia a 2, por lo tanto, se está en la ventana CWP+STWIN = 6, que es la misma en la que se estaba al inicio.

Tras ello se comienza haciendo SAVE por la recursión. Al realizar el segundo SAVE se produce un *overflow* por llegar a CWP 0, que es la ventana marcada como inválida en el WIM, lo que provoca que todos los SAVE siguientes resulten en *overflow*. Debido a la partición, la ejecución de la instrucción SAVE desde CWP 0 provoca que se vuelva a CWP 3, como se muestra en las líneas 6 y 8. Tras terminar los 4 SAVE, realiza los RESTORES correspondientes que, tras volver a la última ventana que marcada como inválida en el WIM, producen *underflow*. Finalmente, en la línea 21 se muestra el estado final y se termina el programa, que esta vez termina correctamente ya que la partición creada se ha mantenido dentro de las ventanas en las que comenzó el programa y los *trap* de desbordamiento han ido guardando y cargando las ventanas correspondientes de pila. Al final, la única línea que no coincide es la última por que el simulador y placa indican el fin del programa de maneras diferentes.

5. Conclusiones y trabajos futuros

Este trabajo se ha centrado en la implementación de la unidad de manejo de memoria y puntero de ventana alternativo definidos en los sistemas LEON3 y SPARCv8 sobre el simulador LeonVip, con el fin de acercarse más al funcionamiento definido en las arquitecturas en las que se basa el simulador y usar éstas para optimizar la ejecución de ciertos programas.

Este proyecto ha mostrado lo importante que es tener en cuenta las bases sobre la que se soporta lo que se va a desarrollar, la complejidad que tiene un sistema pensado para desplegarse en misiones espaciales y la necesidad de que este tipo de sistemas sean lo más fiables posibles.

Ahora que se tiene un simulador de LEON3 con unidad de manejo de memoria y puntero de ventana alternativo es posible comprobar si programas que usen estos elementos funcionan como se espera aún sin necesidad de tener que desplegarlo sobre hardware real, además de facilita la creación de software optimizado usando estas nuevas funcionalidades.

La unidad de manejo de memoria permite la protección de áreas de memoria según el tipo de acceso y privilegios, pero lo más interesante es que, como se pueden cambiar direcciones de memoria a otras, es posible reubicar programas sin tener que modificarlos. Es decir, si se diese el caso de un error de memoria en el área donde se despliega el programa, es posible cambiar donde se despliega el programa y configurar la MMU para que traduzca las direcciones originales a las nuevas. El problema es que la MMU implementada es bastante compleja y tiene muchos elementos ya implementados directamente en hardware lo cual dificulta dicho caso de uso. Por ello se ha planteado como trabajo futuro el diseño e implementación de una nueva unidad de manejo de memoria más sencilla, por ejemplo, solo basada en TLB, y cuyo objetivo sea dar una interfaz software fácil y rápida para reubicar áreas de memoria.

El puntero de ventana alternativo da un mayor control sobre las ventanas de registros, pero lo más interesante es que posibilita restringir las ventanas que usa el proceso principal, lo que a su vez permite reservar ventanas para otros procesos o para los manejadores de interrupción, evitando retardos inesperados por quedarse sin ventanas a las que ir o volver. Entonces, un claro caso de uso y trabajo futuro es la modificación de los métodos de interrupción que son sensibles al tiempo de ejecución. El plan consistiría en modificar el programa para que parta creando una partición de ventanas, haciendo que el programa principal solo use cierto número de ventanas, dejando el resto siempre libres. Finalmente, modificar la llamada a las interrupciones para que partan usando las ventanas reservadas, lo que asegura que la interrupción siempre tarde el mismo tiempo en ejecutarse. Esto genera nuevos problemas que se tendrán que tener en cuenta, como tratar el caso de que salte una interrupción mientras se está ejecutando otra.

Posiblemente la parte más difícil a la hora de desarrollar el proyecto ha sido validar su funcionamiento. Debido a la complejidad de las funcionalidades implementadas y el sistema donde se implementan, existen un gran número de casos posibles a validar. Esto ha hecho que se haya tenido que dedicar más tiempo al diseño y creación pruebas que demuestren el buen funcionamiento de los elementos implementados, que a la implementación de dichos elementos.

6. Bibliografía

[1] The SPARC Architecture Manual, SPARC International Inc., U.S.A, 1992, Version 8.

[2] SPARC-V8 Embedded Architecture Specification, SPARC International Inc., U.S.A, 1996, Version 1.0.

[3] GRLIB IP Core User's Manual, Cobham Gaisler, Sweden, 2020, Version 2020.2.

[4] GR740 Data Sheet and User's Manual, Cobham Gaisler, Sweden, 2018, Version 1.10.

[5] GRMON3 User's Manual, Cobham Gaisler, Sweden, 2020, Version 3.2.1.

[6] H.-J. Wunderlich, "Design of a Memory Management Unit for System-on-a-Chip Platform "LEON"", Diploma Thesis, Division of Computer Architecture, University of Stuttgart, Stuttgart, 2002.

[7] Da Silva, "Nuevas técnicas de inyección de fallos en sistemas embebidos mediante el uso de modelos virtuales descritos en el nivel de transacción", Tesis Doctoral, Departamento de Automática, Universidad de Alcalá, Alcalá de Henares, 2015.

Universidad de Alcalá
Escuela Politécnica Superior

ESCUELA POLITECNICA
SUPERIOR