Instituto Politécnico Nacional

Centro de Investigación en Computación

Laboratorio de Microtecnología y Sistemas Embebidos

Diseño de la Unidad de Gestión de Memoria (MMU) para procesadores superescalares

T E S I S

QUE PARA OBTENER EL GRADO DE

MAESTRO EN CIENCIAS EN INGENIERÍA DE CÓMPUTO

CON OPCIÓN EN SISTEMAS DIGITALES

P R E S E N T A

Ing. Fernando Preciado Llanes

DIRECTORES DE TESIS:

Dr. Marco A. Ramírez Salinas

Dr. Luis A. Villa Vargas

MÉXICO, D.F. **Junio 2013**

SIP-14 bis

INSTITUTO POLITÉCNICO NACIONAL
SECRETARÍA DE INVESTIGACIÓN Y POSGRADO

ACTA DE REVISIÓN DE TESIS

En la Ciudad de México, D.F. siendo las 11:00 horas del día 21 del mes de junio de 2013 se reunieron los miembros de la Comisión Revisora de la Tesis, designada por el Colegio de Profesores de Estudios de Posgrado e Investigación del:

Centro de Investigación en Computación

para examinar la tesis titulada:

"Diseño de la unidad de gestión de memoria (MMU) para procesadores superescalares"

Presentada por el alumno:

PRECIADO	**LLANES**	**FERNANDO**
Apellido paterno	Apellido materno	Nombre(s)

Con registro: | A | 1 | 1 | 0 | 8 | 7 | 2 |

aspirante de: **MAESTRÍA EN CIENCIAS EN INGENIERÍA DE CÓMPUTO CON OPCIÓN EN SISTEMAS DIGITALES**

Después de intercambiar opiniones los miembros de la Comisión manifestaron ***APROBAR LA TESIS***, en virtud de que satisface los requisitos señalados por las disposiciones reglamentarias vigentes.

LA COMISIÓN REVISORA

Directores de Tesis

Dr. Luis Alfonso Villa Vargas

Dr. Marco Antonio Ramírez Salinas

Dr. Herón Molina Lozano

Dr. Víctor Hugo Ponce Ponce

M. en C. Osvaldo Espinosa Sosa

PRESIDENTE DEL COLEGIO DE PROFESORES

Dr. Luis Alfonso Villa Vargas

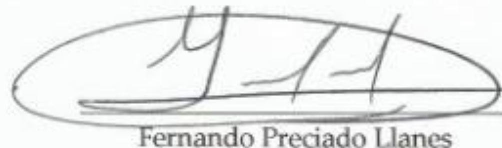
INSTITUTO POLITÉCNICO NACIONAL

SECRETARÍA DE INVESTIGACIÓN Y POSGRADO

CARTA CESIÓN DE DERECHOS

En la Ciudad de México, D. F. el día 24 del mes de Junio del año 2013, el (la) que suscribe Fernando Preciado Llanes alumno del Programa de Maestría en Ciencias en Ingeniería de Cómputo con opción en Sistemas Digitales con número de registro A110872 adscrita al Centro de Investigación en Computación, manifiesta que es autor (a) intelectual del presente trabajo de Tesis bajo la dirección de Dr. Luis Alfonso Villa Vargas y Dr. Marco A. Ramírez Salinas y cede los derechos del trabajo intitulado "Diseño de la Unidad de gestión de memoria (MMU) para procesadores superescalares", al Instituto Politécnico Nacional para su difusión, con fines académicos y de investigación.

Los usuarios de la información no deben reproducir el contenido textual, gráficas o datos del trabajo sin el permiso expreso del autor y/o director del trabajo. Este puede ser obtenido escribiendo a la siguiente dirección fdo.preciado@gmail.com Si el permiso se otorga, el usuario deberá dar el agradecimiento correspondiente y citar la fuente del mismo.



Fernando Preciado Llanes

Resumen

Los métodos para mejorar el desempeño de los procesadores han evolucionado durante estos años gracias a los avances en el diseño de arquitecturas y las tecnologías de fabricación de circuitos integrados, con los cuales se ha logrado el desarrollo de una microarquitectura conocida como superescalar.

La principal característica de esta arquitectura superescalar es la segmentación o *pipelining*, la cual mejora el rendimiento del procesador logrando ejecutar más de una instrucción por ciclo de reloj, explotando el paralelismo a nivel de instrucciones (ILP), así como también el paralelismo de flujo de datos.

El sistema de memoria de esta arquitectura es tan importante como la Unidad de Procesamiento Central (CPU) para determinar su rendimiento y utilidad. En la actualidad existen componentes de memoria tan rápidos como la ALU y otros elementos de ruta de datos. No obstante, aunque tales memorias se pueden construir, la factibilidad de uso está muy limitada por restricciones tecnológicas y económicas. La paradoja en el diseño de sistemas de memoria es que se requiere una gran cantidad de memoria para alojar conjuntos de datos extensos, así como programas complejos y que ésta sea tan rápida como para no frenar el flujo de datos de un proceso de ejecución cada día más veloz del CPU. Por lo anterior, el sistema de memoria es un factor principal en el rendimiento por lo que se trabaja en proyectos de diseño e investigación de arquitectura de computadoras.

La Unidad de Gestión de Memoria (MMU) es un dispositivo Hardware formado por un grupo de bloques funcionales que en conjunto con el Sistema Operativo forman un sistema de gestión de memoria que lleva a cabo la administración de la memoria virtual

que hace que el sistema aparente tener más memoria de la que realmente tiene compartiéndola entre múltiples procesos conforme la necesitan. El sistema de gestión de memoria provee más ventajas que sólo dar grandes espacios de direcciones, provee protección de la memoria entre procesos brindando a cada uno su propio espacio de direcciones virtual, además permite compartir el espacio de direcciones entre procesos y determinar el tipo de acceso a áreas de memoria ya sea para lectura, escritura y ejecución.

El presente trabajo de tesis tiene por objetivo obtener el diseño de una Unidad de Gestión de Memoria (MMU) para procesadores superescalares, para ello se realizó un análisis de las características de los diseños en procesadores actuales y de propuestas recientes encontradas en la literatura y con ello lograr un modelo eficiente mediante la investigación y combinación de técnicas de arquitectura de computadoras.

El diseño está basado en la SRMMU SPARCv8 del *softcore* LEON que es de código abierto y distribuido bajo una licencia GPL, el cual permite estudiar el funcionamiento, modificar o manipular el diseño y realizar evaluaciones para medir el rendimiento. Posteriormente se realizó la integración en un SoPC (System on Programmable Chip) que se sintetiza en un dispositivo programable FPGA (Field Programmable Gate Array) con el propósito de obtener una plataforma hardware que permita el diseño y simulación. La evaluación de caracterización y rendimiento de la MMU se llevó a cabo mediante la operación del SO Linux y ejecutando programas de verificación especializados *Benchmarks*. Además los resultados obtenidos fueron analizados para determinar el desempeño con respecto a modificaciones en el diseño y así poder obtener las conclusiones pertinentes sobre la arquitectura.

Abstract

Methods for improving the performance of processors have evolved over the years due to advances in the design of architectures and manufacturing technologies of integrated circuits, which have succeeded in developing the microarchitecture known as superscalar.

The main feature of this superscalar architecture is the segmentation or pipelining (in charge of improving processor performance), achieving to execute more than one instruction per clock cycle by exploiting the instruction level parallelism (ILP), along with the data parallel flow.

The memory system of this architecture is as important as the Central Processing Unit (CPU) to determine performance and utility. Nowadays there are memory modules as fast as the ALU and other datapath elements. However, while such memories can be built, the feasibility of its use is limited by economic and technological constraints. The paradox in the design of memory systems is that it requires a lot of memory to accommodate extended data sets, as well as complex programs, and that it needs to be fast enough to prevent the slowdown of data flow while running a process (which becomes faster everyday in the CPU). Therefore, memory systems are a major factor in CPU performance, leading to a major interest in design and computer architecture research projects.

The Memory Management Unit (MMU) is a hardware device consisting of a group of functional blocks which together with the OS form a memory management system that performs the management of the virtual memory: with it the system has apparently more

memory than it actually has by sharing it between multiple processes as needed. The memory management system provides more advantages than just giving large address spaces, it also provides memory protection between processes giving each its own virtual address space, as well as sharing the address space between processes and determining the type of memory access to read, write and execute.

This thesis aims to design a MMU for superscalar processors, thus an analysis of the current processor designs and recent research papers related to the topic was done, so that an efficient model was built according to fresh investigations and the combination of computer architecture techniques.

The design is based on the SRMMU SPARCv8 embedded in the LEON softcore which is open source and distributed under a GPL license; it allows the study of its performance, modifying the design and carrying out evaluations. Afterwards, integration was then accomplished on a SoPC (System on Programmable Chip) which is synthesized on a programmable device FPGA (Field Programmable Gate Array) for the purpose of obtaining a hardware platform that allowed the designing and simulation. The evaluation of characterization and MMU efficiency was achieved by the SO Linux operation and by running verification programs Benchmarks. In addition, the obtained results were analyzed to determine the performance according to modifications in the design, done so to determine the relevant conclusions on the computer architecture.

Agradecimientos

La culminación satisfactoria de este trabajo de investigación fue posible gracias al apoyo recibido por parte del Instituto Politécnico Nacional, el CONACYT y la Secretaría de Investigación y Posgrado por la beca otorgada.

Agradezco al CIC por darme la oportunidad de prepararme y colaborar con investigadores y personal increíble, particularmente a mis maestros y mentores, por transmitirme con paciencia y humildad la pasión por la investigación y profesionalismo. Especialmente agradezco al Dr. Marco A. Ramírez Salinas y al Dr. José Luis Oropeza Rodríguez por su colaboración, por haber compartido sus conocimientos y su ayuda durante el desarrollo de esta tesis.

A mis padres: Rosalva y Fernando, a mi hermana Lorena, a mi novia Rosa y a mis amigos: Diego, Troy, 3pio, Pere, Andrés, Edy y Toto, a la generación A11 del CIC especialmente a Nayeli, Javier y Antonio, a todos ellos por estar siempre presentes, incluso a la distancia, me apoyaron y alentaron para realizar este trabajo. Muchas gracias.

Índice general

Resumen I

Abstract III

Agradecimientos V

Índice general VI

Índice de figuras IX

Índice de tablas XI

Glosario de términos y lista de acrónimos XII

CAPÍTULO 1 Introducción. 2

1.1 Antecedentes 2

1.2 Planteamiento del problema 7

1.2.1 Descripción de cómo se afronta esta problemática en la actualidad 9

1.3 Justificación de la tesis 10

1.4 Objetivo general 11

1.5 Alcances del trabajo 12

1.6 Contribuciones 13

1.7 Método de investigación y desarrollo utilizado 14

1.8 Organización del trabajo 14

CAPÍTULO 2 Estado del arte 17

2.1 Microprocesador MIPS R10000 17

2.2 Microprocesador Pentium Intel x86 22

2.3 Microprocesador ARMv9 26

2.4 Softcore OpenRISC 1200 32

2.5 Microprocesador SPARC V8 40

2.6 Artículos 47

2.6.1 Mejora del desempeño de Tablas de Página 47

2.6.2 Mejora en TLB 50

2.6.3 Mejora en jerarquía de memoria. 51

2.7 Resumen del capítulo 53

CAPÍTULO 3 Marco teórico 55

3.1 Procesadores Superescalares 55

3.2 Unidad de gestión de memoria MMU 60

3.3 Elementos del proyecto 64

3.3.1 FPGA y Tarjeta de desarrollo 64

3.3.2 System on Chip (SoC) 67

3.3.3 Entorno Software para SoC 71

3.3.5 Software para la simulación y evaluación 76

3.4 Resumen del Capítulo 76

CAPÍTULO 4 Diseño de la Arquitectura propuesta 78

4.1 Metodología de diseño 78

4.2 Diseño del LEON-SoC 80

4.2.1 Softcore Leon3 81

4.2.2 Bus AMBA 2.0 83

4.2.3 Debug Support Unit DSU 87

4.2.4 JTAG Debug Link 87

4.2.5 UART (RS232) 87

4.2.6 Controlador SDRAM 87

4.3 Sistema de gestión de memoria 89

4.3.1 Sistema de caches 89

4.3.2 Operación SRMMU/Cache 94

4.3.3 Registro de control LEON/SRMMU 96

4.3.4 Mapeo SRMMU/ASI 97

4.3.5 Modelo de Interrupciones de memoria 97

4.4 Arquitectura del diseño propuesto de MMU superescalar 98

4.5 Resumen del Capítulo 100

CAPÍTULO 5 Pruebas y resultados **102**

5.1 Descripción de los factores a evaluar 102

5.2 Descripción del Procesador 102

5.3 Configuración de Caches 104

5.4 Configuración de la MMU 105

5.5 Benchmarks 106

5.5.1 Dhrystone 2.1 106

5.5.2 Stanford 107

5.5.3 Whetstone 107

5.5.4 LMbench 108

5.6 Resultados 109

5.7 Resumen del Capítulo 117

CAPÍTULO 6 Conclusiones y Trabajo futuro **118**

6.1 Conclusiones 118

6.2 Trabajo Futuro 122

Referencias **124**

Anexos **126**

Anexo A Código del diseño MMU

Anexo B The SPARC Architecture Manual Version 8, Revision SAV080SI9308

Anexo C LEON/GRLIB, Configuration and Development Guide, December 2012

Anexo D AMBA™ Specification, Rev. 2.0 ARM ARM IHI 0011A Limited 1999

Índice de figuras

Figura 2.1 Procesador R10000 fabricado por NEC Corporation ... 17
Figura 2.2 Pipeline de la arquitectura superescalar del MIPS R10000. ... 18
Figura 2.3 Microarquitectura del procesador MIPS R10K 19
Figura 2.4 Modelo de traducción de dirección del MIPS R10000. ... 20
Figura 2.5 Espacio de Direcciones virtuales Modo Usuario y Modo Supervisor. ... 21
Figura 2.6 Traducción de página de 4KB Intelx86. ... 22
Figura 2.7 Traducción de página de 4KB y 4MB Intelx86. ... 23
Figura 2.8 Diagrama de bloques del procesador ARM926EJ-S. ... 27
Figura 2.9 Tablas de traducción de página ARMv5. ... 28
Figura 2.10 Formato de registro c3 del ARM. ... 29
Figura 2.11 Diagrama de secuencia de traducción completa para una 4KB small page ... 30
Figura 2.12 Operación de caché tipo VIVT asociativa de 4-way 31
Figura 2.13 Arquitectura del softcore OpenRisc 1200 ... 32
Figura 2.14 Sistema de memoria OpenRISC 1200 ... 34
Figura 2.15 Diagrama de flujo de traducción de DV a VF OpenRISC 1000. ... 35
Figura 2.16 Sistema de MMU/TLB OpenRISC 1200 ... 36
Figura 2.17 Sistema 32 bits de Tablas de página de dos niveles OpenRISC 1200 37
Figura 2.18 Ventanas de registros. ... 41
Figura 2.19 Modelo de operación de la MMU del procesador SPARC. ... 42
Figura 2.20 Registro de Control. ... 43
Figura 2.21 Registro Apuntador de Contexto 44
Figura 2.22 Registro de Numero de Contexto ... 44
Figura 2.23 Registro de Estado de Fallo. ... 44
Figura 2.24 Registro de Dirección de Fallo 45
Figura 2.25 Tabla de contexto (Context Table) ... 45
Figura 2.26 Proceso de traducción de la SRMMU ... 46
Figura 2.27 Segmentación de dirección virtual x86_64 47
Figura 2.28 Ejemplo de contexto de UPTC de AMD ... 48
Figura 2.29 Ejemplo de contexto de STC de Intel. ... 49
Figura 2.30 (a)Acceso secuencial TLB/cache. (b) Acceso en paralelo a TLB/cache. ... 51
Figura 2.31 Diagrama de operación de cache con la Caché Víctima. 52
Figura 2.32 Fallos por conflicto removidos por la caché víctima. 52
Figura 2.33 Desempeño de caché víctima con variaciones de caché L1. ... 53
Figura 3.1 Pipeline de una arquitectura escalar. ... 55
Figura 3.2 Pipeline de un procesador superescalar de 4-way ... 56
Figura 3.3 Benchmark enfocado al renombrado de registros 58
Figura 3.4 Formatos de instrucciones tipo RISC del procesador MIPS. ... 59
Figura 3.5 Diagrama de bloques de un sistema con MMU 60
Figura 3.6 Diagrama de bloques de la operación de MMU 61
Figura 3.7 Traducción de dirección virtual a dirección física mediante el TLB. ... 62
Figura 3.8 Traducción de DV a DF mediante Tablas de marco de página 63
Figura 3.9 Diagrama de flujo de la operación de MMU 63

Figura 3.10 Estructura interna de una FPGA 64
Figura 3.11 Elemento lógico del FPGA Cyclone IV 65
Figura 3.12 Diagrama de recursos del FPGA Cyclone IV 65
Figura 3.13 Diagrama de componentes de la DE2-115 66
Figura 3.14 SoC basado en el procesador LEON3 70
Figura 3.15 Manejo de la memoria virtual Linux. 73
Figura 3.16 Estructura de mm_struct 75
Figura 4.1 Diagrama de flujo para el diseño propuesto.. 79
Figura 4.2 Diagrama de la arquitectura del SoC 80
Figura 4.3 Diagrama del pipeline de la Unidad de ejecución de enteros (IU) del Leon3 82
Figura 4.4 Vista conceptual del bus AMBA AHB/APB 84
Figura 4.5 Resumen de interconexión del bus AHB... 84
Figura 4.6 Resumen de interconexión del bus APB 86
Figura 4.7 Controlador SDRAM conectado al bus AHB y a la memoria SDRAM 88
Figura 4.8 Conexión de las SDRAM y el FPGA de la DE2-115 88
Figura 4.9 Registro de configuración (SDCFG) 88
Figura 4.10 Estructura de una Tag de la IC y DC 90
Figura 4.11 Registro de control de cache (CCR) 91
Figura 4.12 Registros de configuración de cache 92
Figura 4.13 Ciclos de operación en la etapa de EX (Execute) 93
Figura 4.14 Ciclos de operación en la etapa de FE (Fetch) 94
Figura 4.14 Estructura del Sistema de memoria, Caches, MMU y controlador SDRAM.................................. 95
Figura 4.15 Protocolo de la operación del Sistema de memoria 95
Figura 4.16 Write Buffer de la DC.. 96
Figura 4.17 Registro de control LEON/SRMMU 96
Figura 4.19 Diseño de componentes para obtener la MMU superescalar. 98
Figura 4.20 Implementación de caches dual_port 99
Figura 5.1 Gráfica de resultados Standford de comparación de configuraciones de caches......................... 111
Figura 5.2 Gráfica de resultados benchmark lam_mem_rd latencias (ns).. 115
Figura 5.3 Gráfica de resultados benchmark bw_mem ancho de banda (MB/s) 117

Índice de tablas

Tabla 2.1 Descripción de los registros especiales x86 . 24
Tabla 2.2 Descripcion de los registros de control Openrisc MMU. 38
Tabla 2.3 Espacio de direcciones Openrisc. 39
Tabla 2.4 Vectores de interrupción MMU Openrisc. 39
Tabla 2.5 Modos de acceso a memoria OpenRISC 1200. 39
Tabla 2.6 Direcciones de registros internos de la SRMMU. 43
Tabla 3.1 Tabla comparativa entre Leon3 SPARC-MMU y OpenRisc 1200 MMU. 69
Tabla 4.1 Mapeo ASI = 2 a registros de Control y Configuración de Caches. 92
Tabla 4.2 Bits de dirección ASI y descripción de uso. 97
Tabla 4.3 Tabla de interrupciones del Sistema de memoria. 97
Tabla 5.1 Configuración base del softcore Leon3 y controlador de memoria. 103
Tabla 5.2 Configuraciones de caches a evaluar 104
Tabla 5.3 Configuraciones de la MMU a evaluar. 105
Tabla 5.4 Tabla de resultados Dhrystone sobre caches. 110
Tabla 5.5 Tabla de resultados Stanford sobre caches. 111
Tabla 5.6. Tabla de recursos implementados enfocado a caches 112
Tabla 5.7 Tabla de resultados de benchmarks enfocados a MMU/SO. 114
Tabla 5. 8 Tabla de recursos implementados por la arquitectura SoC/MMU. 115
Tabla 5. 9 Tabla comparativa de latencias (ns) del sistema de memoria.. 116
Tabla 5.10 Tabla comparativa de ancho de banda (MB/s) del sistema de memoria 117

Glosario de términos y lista de acrónimos

A continuación se muestra una lista de términos y acrónimos que se utilizan en la presente tesis. Las palabras que se encuentran entre paréntesis dentro de la definición contribuyen con el contexto de la definición dada. Además de que se presentan dentro de corchetes el significado de los acrónimos heredados de la literatura anglosajona.

AHB	Bus AMBA de alta velocidad. [Advanced High-performance Bus]
AMBA	(Bus) Se utiliza como la interfaz de comunicación estándar entre los núcleos IP en GRLIB. [Advance Microcontroller Bus Architecture]
API	(Interfaz de programación de aplicaciones) proporcionar un entorno de programación con un conjunto de funciones de uso general a los programadores. [Application Programming Interface]
ASI	El procesador SPARC genera un identificador de espacio de dirección de 8 bits (ASI), que proporciona hasta 256 espacios separados de direcciones de 32 bits. Durante el funcionamiento normal, el procesador LEON3 accede a instrucciones y datos utilizando ASI 0x8 - 0xB tal como se definen en la norma SPARCv8. Con las instrucciones LDA / STA, espacios de direcciones alternativas pueden ser accesadas. [Alternate Space Identifiers]
Benchmark	Es una técnica utilizada para medir el rendimiento de un sistema o componente del mismo, mediante la comparación.
CMOS	(Tecnología) Metal–Óxido–Semiconductor Complementario. [Complementary Metal–Oxide Semiconductor]
Cross_compilation	Es cuando el compilador produce un ejecutable para ejecutarse en una plataforma diferente, ARQ-A a ARQ-B (Sistema Embebido)
DC	Cache de datos. [Data Cache]
DF	Dirección Física.
DSU	Unidad de depuración. [Debug Support Unit]

DV Dirección Virtual.

Escalar Arquitectura capaz de ejecutar una instrucción por ciclo de reloj.

FPGA Dispositivo semiconductor constituido de bloques de lógica cuya interconexión y funcionalidad se puede configurar in situ a través de un lenguaje de descripción especializado.[Field Programmable Gate Array]

FSM (modelo computacional) Máquina de estado finito, realiza operaciones en forma automática sobre una entrada para producir una salida. [Finite-State Machine]

GPIO Puerto de entradas y salidas de propósito general. [General Purpose Input/Output]

GRLIB (HDL) Librería VHDL de IPcores. [Gaisler Research Library]

GRMON Software para la comunicación y descarga de aplicaciones al LEONSoC. [Gaisler Research Debug Monitor]

HDL Lenguajes de descripción de Hardware para el modelado de sistemas digitales. [Hardware Description Language]

IC Cache de instrucciones. [Instruction Cache]

IPcores Módulos HDL reutilizables, que junto con el software realizan una función o controlan la operación de un periférico.
[Intellectual Property]

JTAG Mecanismo de comunicación que permite al programador acceder al módulo de depuración que se encuentra integrado dentro de la CPU.
[Joint Test Action Group]

LRU (Algoritmo de remplazo) Menos usado recientemente.
[Least Recently Used]

MMU (Unidad de Gestión de Memoria) dispositivo hardware formado por un grupo de bloques funcionales que en conjunto con el SO llevan a cavo la administración de la MV.
[Memory Management Unit]

MV	(Memoria Virtual) Técnica de gestión de la memoria que permite mayor cantidad de memoria que la disponible físicamente, protección y compartición de la memoria entre procesos y mapeo de archivos a memoria.
Pipeline	Estructura segmentada.
PIPT	(Cache física) utilizan la DF para acceder a la localdiad. [Physically Indexed, Physically Tagged]
PTD	Descriptor de Tabla de Pagina. [Page Table Descriptor]
PTE	Entrada de Tabla de Página. [Page Table Entry]
RISC	*Set* o conjunto de instrucciones reducidas, características de ser instrucciones de tamaño fijo y presentadas en un reducido número de formatos. [Reduced Instruction Set Compute]
SDRAM	Memoria síncrona dinámica de acceso aleatorio. [Synchronous Dynamic Random-Access Memory]
SO	Sistema Operativo.
SoC	(Sistema en Chip) integran componentes electrónicos o para crear un sistema informático en un único circuito integrado o chip. [System on Chip]
Softcore	Es aquel procesador descrito en HDL, configurable, de aplicación especifica y que sintetizable en FPGA.
SPARC	Procesador escalar RISC de Sun Microsystems. [Scalable Processor Architecture]
SRAM	Memoria Estática de Acceso Aleatorio, es capaz de mantener los datos, mientras esté alimentada, sin necesidad de circuito de refresco. [Static Random-Access Memory]
SRMMU	Unidad de gestión de memoria MMU del procesador SPARC. [SPARC Reference MMU]

Superescalar Arquitectura en *pipeline* que logra ejecutar más de una instrucción por ciclo de reloj.

TLB (Cache de PTEs) es una memoria caché administrada por la MMU, que contiene partes de la Tabla de páginas, es decir, es una memoria que mantiene traducciones entre direcciones virtuales y reales. [Table Lookaside Buffer]

Toolchain Es un proyecto que contiene herramientas de programación y desarrollo de Sowftware y SO, principalmente para sistemas embebidos.

UART Es un chip o sub-componente de un microcontrolador que proporciona el hardware para generar un flujo serie asíncrono tales como RS-232 o RS-485.
[Universal Asynchronous Receiver/Transmitter]

VHDL Lenguaje de descripción de hardware. [VHSIC(Very-High-Speed Integrated Circuit) Hardware Description Language]

VIPT (Cache Hibrida) utilizan la DV como índice y la DF como etiqueta para acceder a la localidad de la memoria. [Virtually Indexed, Physically Tagged]

VIVT (Cache Virtual) utilizan la DV acceder a la localidad de la memoria. [Virtually Indexed, Virtually Tagged]

CAPÍTULO 1.

INTRODUCCIÓN

En este capítulo se introduce al Procesador superescalar, el cual logra ejecutar más de una instrucción por ciclo de reloj, gracias a su arquitectura segmentada *pipeline* y otras técnicas hardware. Principalmente se presentan los antecedentes y funciones del dispositivo hardware llamado Unidad de Gestión de Memoria (MMU) cuya función es la administración de la Memoria Virtual que permite ambientes de multiprocesos y grandes espacios de direcciones de memoria. Se muestran tanto los objetivos así como los alcances de esta tesis, además de las contribuciones y justificación de la misma. Al final de este capítulo, se presenta una pequeña descripción de la organización y contenido del resto de los capítulos del presente trabajo de tesis.

1.1 Antecedentes
1.2 Planteamiento del problema
1.3 Justificación de la tesis
1.4 Objetivo general
1.4.1 Objetivos particulares
1.5 Alcances del trabajo
1.6 Contribuciones
1.7 Método de investigación y desarrollo utilizado
1.8 Organización del trabajo

CAPÍTULO 1 Introducción.

1.1 Antecedentes

Las nuevas ideas de investigación en el área de arquitectura de computadoras, generaron la estructura de los procesadores superescalares, su microarquitectura es capaz de ejecutar más de una instrucción por ciclo de reloj, mediante el paralelismo a nivel de instrucciones (ILP), además del paralelismo de flujo de datos, ésto gracias a la estructura en *pipeline*. Este *pipeline* consta de las siguientes etapas: Lectura (*fetch*), Decodificación (*decode*), Lanzamiento (*dispatch*), Ejecución (*execute*), Escritura (*writeback*), Finalización (*commit*), a través de las cuales se lleva a cabo el procesamiento de instrucciones de su ISA (*Instruction Set Architecture*).

El término superescalar (*superscalar*) se empleó, por primera vez en un reporte técnico escrito por John Cocke y Tilak Agerwala dentro de IBM, el cual fue titulado High Performance Reduced Instruction Set Processors, en 1987, a partir de entonces, dicho término se popularizó, aunque los primeros diseños de procesadores superescalares fueron fabricados en la década de 1960 por la misma compañía.

El procesador IBM 7030 desarrollado a principios de 1960, se considera el primer procesador escalar, el cual contaba con cuatro etapas en su pipeline para la ejecución de las instrucciones que brindó avances tecnológicos escalables.

Consecuentemente surgieron más diseños, en 1964 el procesador CDC 6600, el cual contaba con diez unidades funcionales, era capaz de decodificar y emitir una instrucción por ciclo, pero hasta tres instrucciones podían iniciar su ejecución a la vez. El procesador IBM S/360 Modelo 91, el cual decodificaba una instrucción por ciclo y además permitía la ejecución de instrucciones fuera de orden del programa en su unidad funcional de punto flotante, usando la técnica conocida como algoritmo de Tomasulo, llamado así en honor de su inventor Robert Tomasulo.

A partir de entonces, la investigación y desarrollo de éste tipo de arquitecturas, continuó de un modo lento hasta principios de la década de 1980 cuando surgieron los primeros procesadores RISC, los cuales fueron diseñados usando el principio de una estructura segmentada, con el único fin de incrementar drásticamente el rendimiento de la CPU, tal es el caso del procesador PowerPC basado en RISC lanzado en 1990 por IBM.

Esto motivó a otras compañías a generar procesadores RISC superescalares, como es el caso de la línea i960 y posteriormente la arquitectura IA-64 de Intel con Hewlett-Packard, Sun Sparc de Sun Microsystems, Am29000 de AMD, la línea Alpha de DEC y el PowerPC desarrollado por Apple, IBM y Motorola.

Los diseños desarrollados de procesadores superescalares en el área académica, se puede destacar la arquitectura prototipo de la Universidad Norteamericana de Stanford, creada por el grupo de trabajo dirigido por John Hennessy llamado MIPS en 1981, quién junto a David Patterson fundó en 1984 la compañía MIPS Computer Systems. Dicha compañía sacó al mercado en 1985 el procesador R2000 que fue el primero de una línea de procesadores RISC superescalares que está vigente hasta la actualidad.

Lo anterior impulsó al diseño de procesadores CISC superescalares como el Pentium de Intel y el MC 68060 de Motorola lanzados en 1993. Ambos procesadores fueron el resultado de convertir una línea de procesadores CISC escalares en arquitecturas superescalares. Cabe mencionar que este tipo de procesadores, fueron implementados usando un núcleo RISC; es decir, las instrucciones complejas primero se convierten en micro-operaciones las cuales son ejecutadas posteriormente por este núcleo RISC.

Los microprocesadores diseñados por la Academia China de Ciencias llamados Godson, fueron basados en la arquitectura y conjunto de instrucciones MIPS. A partir del 2006, una serie de empresas empezaron a comercializar equipos de hardware con el procesador, incluyendo netbooks con un bajo consumo. Un ejemplo importante de

crecimiento tecnológico fué en Julio de 2008, que anunció la entrada al mercado de un ordenador portátil de bajo costo denominado Yeeloong del fabricante Lemote, con el procesador Loongson 2F, lector de DVD y sistema operativo GNU/Linux. Con este proyecto el gobierno chino realizó la propuesta de que todo el mundo pudiera tener acceso a un ordenador personal con un bajo poder adquisitivo.

Actualmente los procesadores superescalares, son la arquitectura y tecnología dominante en el mercado, existiendo diferentes diseños que varían en su microarquitectura y desempeño, dependiendo de la aplicación, objetivos y necesidades.

La estructura típica de un procesador superescalar consta de un *pipeline* que maneja más de una instrucción en cada etapa, para lograr esto, necesita de una eficaz Unidad de gestión de memoria (MMU) responsable del manejo de los accesos a la memoria.

Históricamente el sistema de gestión de memoria a evolucionado, principalmente por la necesidad de administrar ambientes *multitasking* en el cual múltiples procesos comparten un mismo sistema de memoria, prohibiendo el acceso accidental que podría bloquear todo el sistema, o el acceso no autorizado para proteger los datos privados.

En sistemas de cómputo antiguos que no disponían de hardware de traducción (MMU), se implementaba una reubicación de las direcciones que contiene el programa en el momento de su carga en memoria. Con esta estrategia el código del programa cargado en memoria ya contiene las direcciones traducidas propiamente. Durante su carga en memoria, el Sistema Operativo detectaría qué instrucciones hacen referencia a direcciones de memoria y las modificaría añadiendo un desplazamiento según la partición. Por ejemplo, la dirección del programa 100 + 100k para la partición 1, 100 + 200k para la partición 2. Esto permite ejecutar simultáneamente múltiples procesos y aplicaciones. Pero esta solución no permitía al Sistema Operativo y al usuario brindarle un espacio de direcciones mayor que la memoria real o física.

Las aplicaciones requieren el acceso a más información (código y datos) que la que se puede mantener en memoria física. Una solución al problema de necesitar mayor cantidad de memoria de la que se posee, consiste en que las aplicaciones mantengan parte de su información en disco, moviéndola a la memoria principal cuando sea necesario. Hay varias formas de hacer esto.

Una opción es que la aplicación misma sea responsable de decidir qué información será guardada en cada sitio (segmentación), y de traerla y llevarla. La desventaja de ésto, además de la dificultad en el diseño e implementación del programa, es que es muy probable que los intereses sobre la memoria de dos o varios programas generen conflictos entre sí, cada programador podría realizar su diseño teniendo en cuenta que es el único programa ejecutándose en el sistema improductivo en los SO multitarea actuales.

El método más eficiente se conoce actualmente como Memoria Virtual (Fotheringham, 1961). La idea es que cada programa tiene su propio espacio de direcciones, el cual se divide en trozos llamados páginas. Cada página es un rango continuo de direcciones. Estas páginas se asocian a la memoria física, pero no todas tienen que estar en la memoria física para poder ejecutar el programa. Cuando el programa hace referencia a una parte de su espacio de direcciones que está en la memoria física, el hardware realiza la asociación necesaria al instante. Cuando el programa hace referencia a una parte de su espacio de dirección que no está en la memoria física, el SO recibe una alerta (interrupción) para buscar la parte faltante y volver a ejecutar la instrucción de fallo.

Uno de los primeros SO que fue pionero en los conceptos fundamentales de sistema multitarea/multiusuario fue MULTICS, que data de finales de los años '60, el cual implementa el concepto de memoria virtual en el hardware por 2 técnicas principales: paginación y la segmentación. Estos principios se mantienen hasta la fecha y son usados en los SO actuales (más complejos) y multiprocesamiento.

Implementando la MV la cantidad de memoria máxima que se puede hacer ver, depende de las características del procesador. Por ejemplo, en un sistema de 32 bits, lo que da 4096 Megabytes (4GB). Lo cual, hace que el trabajo del programador de aplicaciones se realice con facilidad, al poder ignorar completamente la necesidad de mover datos entre los distintos espacios de memoria.

Debido a las necesidades del SO y de los usuarios, ha obligado a los diseñadores del hardware a realizar investigación e incluir nuevos mecanismos que, a su vez, han posibilitado realizar la gestión de memoria de modo eficaz y eficiente.

Así el SO con el apoyo del hardware de gestión de memoria del procesador, debe administrar la memoria existente, proporcionando un espacio independiente para cada proceso en ejecución y protección entre los procesos.

Los procesadores superescalares como el MIPS también hacen uso de este hardware para gestionar y mapear direcciones virtuales de un programa a direcciones físicas en memoria. En algunos diseños de la serie R implementan el Coprocesador 0 y utilizan instrucciones para su programación y en otros casos se basan en un sistema más simple con asignación estática de DV a DF, además de que se cuenta con un sistema de protección de acceso para hacer referencia a áreas de memoria. Estos y otros principios de diseño existen por parte de procesadores Intel 80386, i486, i860, Motorola 88200 y arquitecturas como ARM, SPARC, entre otros.

Por las ventajas de la MV implementarla en sistemas de múltiples procesos genera gran cantidad de direcciones virtuales que deben ser traducidas eficaz y eficientemente a direcciones físicas de la memoria. Por lo anterior es necesario que sea el procesador el encargado de realizar esta traducción, la función de traducción la lleva a cabo concretamente el módulo específico del procesador: Unidad de gestión de memoria (MMU, *Memory Management Unit*).

1.2 Planteamiento del problema

Una de las características clave en los procesadores modernos para extraer el máximo paralelismo de las aplicaciones, es la capacidad de ejecutar múltiples instrucciones en un mismo ciclo de reloj; para lograr esto, la arquitectura del procesador se ha hecho muy compleja, haciendo cada vez más difícil su evaluación y su diseño.

De igual forma, el diseño de las etapas del *pipeline* que constituyen las estructuras funcionales de la arquitectura de un procesador superescalar con ejecución de instrucciones fuera de orden, busca lograr la eficiencia y un mayor desempeño, con esto se incrementa la complejidad de las estructuras que la componen.

La evaluación de nuevas ideas de investigación en el área de arquitectura de computadoras, se lleva a cabo por simulación por software que juega un papel importante y todo indica que es la mejor opción. Sin embargo, para un diseño físico o un análisis de comportamiento y área más real es necesario su diseño digital en hardware de base, donde las nuevas ideas puedan ser incorporadas y más tarde puedan ser sintetizadas a un dispositivo lógico programable.

Para el caso de la estructura funcional conocida como Unidad de gestión de Memoria (MMU), el reto de diseño consiste en llegar a la propuesta de una arquitectura que administre la MV, que integre un buffer de traducción de direcciones virtuales a físicas TLB, la protección de la memoria, una gestión de jerarquía de memoria y que sea responsable del manejo de los accesos a memoria por parte de un procesador superescalar.

Además, se busca que se implementen o mejoren las características que presentan las arquitecturas actuales mediante la investigación y combinación de diseños, logrando la eficiencia y el menor consumo de energía, de modo que al final se obtenga un diseño novedoso que se caracterice por su baja complejidad y alto performance.

El diseño de la MMU de procesadores modernos, es área poco difundida en la literatura científica, pues el impacto que ésta tiene en el desempeño de los procesadores es muy importante y las compañías de procesadores guardan sus diseños como secretos industriales. Las principales compañías de procesadores publican los manuales de las familias de sus procesadores, en los cuales omiten nuevas características de tecnologías así como de programación avanzada, esa información se registra en un documento formalmente conocido como Suplemento al Manual de Familias de Procesadores, el cual, al ser solicitado por el programador o el usuario debido a que el acceso a la información beneficia a los diseños de sus programas, los receptores deben firmar un contrato de no divulgación (nondisclosure agreement NDA) de 15 años con esto, los fabricantes protegen su propiedad intelectual deteniendo así el intercambio de conocimiento y beneficios de uso de la tecnología. Los fabricantes afirman que comparten sus nuevos avances tecnológicos y toda clase de información a sus usuarios, nunca los niegan, siempre y cuando se firme el acuerdo de confidencialidad. No obstante, varias historias han circulado con respecto a la negación de la información hacia los solicitantes (programadores, usuarios) debido a que los fabricantes declaran que no necesitan la información. Esto ha dado lugar a una comunidad de programadores en conjunto con universidades, centros de investigación y diseñadores de tecnología dedicados a la ingeniería inversa de estas tecnologías y a la publicación de sus investigaciones en diferentes medios de comunicación con el propósito de avanzar y mejorar en el conocimiento de arquitectura de computadoras.

El objetivo del proyecto entonces, es obtener el diseño de una MMU con un impacto en el desempeño ideal de un procesador superescalar moderno, integrando técnicas de arquitectura de computadoras. El diseño se realizará en primera instancia a nivel de microarquitectura, una vez evaluados los diseños se modelarán en código HDL, para posteriormente su síntesis en FPGA. Los resultados serán reportes técnicos, manuales de diseño de hardware y publicaciones.

1.2.1 Descripción de cómo se afronta esta problemática en la actualidad

En la actualidad la Arquitectura de Computadoras, se estudia y desarrolla a menudo haciendo análisis y diseños por medio de simuladores software.

El gran costo en complejidad, tiempo y dinero que supone diseñar físicamente componentes hardware, obliga a depender de la simulación software para desarrollar nuevas ideas en arquitectura de computadores.

Por otro lado existen algunos trabajos del área académica y centros de investigación, que han desarrollado métodos más allá de software de simulación mediante la aplicación de hardware real.

En la Universidad de Washington en el departamento de Ingeniería Eléctrica e investigadores de la IEEE, han desarrollado proyectos los cuales permiten diseñar, implementar y ejecutar un MIPS de un solo ciclo procesador en una FPGA.

Otro gran ejemplo son sitios como OpenCores, que es la mayor comunidad de desarrollo de núcleos IP de hardware de código abierto. Es anfitrión de código fuente para diferentes proyectos digitales HW (IPcores, SoC, sistemas embebidos, etc.) y apoyar a los usuarios con diferentes herramientas, plataformas, foros y otras informaciones útiles.

La compañía AeroFlex Gaisler, que en los inicios fue un outsourcing de la Agencia Espacial Europea (ESA) denominada Gaisler Research, es proveedora de IPcores y desarrolla herramientas para procesadores para sistemas embebidos basados en la arquitectura SPARC. Uno de sus productos es la familia de procesadores sintetizables LEON que surgió con la búsqueda de desarrollar un procesador resistente a radiaciones, modular, fácilmente portable, con interfaces estándar y que ejecuta hasta 100 MIPS. Además, el nuevo diseño, escrito en VHDL, se licenció bajo GPL, lo que permite integrarlo en un System-on-Chip (SoC). Esta compañía cuenta con actividades de investigación,

diplomas de tesis y proyectos basados en el procesador LEON en cooperación con otras universidades.

1.3 Justificación de la tesis

El laboratorio de Microtecnología y Sistemas Embebidos (MICROSE), contempla en uno de sus programas prioritarios, el desarrollo de las arquitecturas de procesadores de alto desempeño con bajo consumo de energía, en este programa se contempla el proyecto Alligator/Lagarto, el cual consiste en el diseño e implementación de un procesador superescalar con ejecución fuera de orden, que contempla en la medida de lo posible, integrar el estado del arte en técnicas de microarquitectura de bajo consumo de energía y alto desempeño. Se busca promover y desarrollar herramientas para el diseño y la evaluación de los procesadores superescalares por medio de: código, reportes técnicos, manuales y material de apoyo para el investigador que se distribuyen en línea, para las universidades y su distribución a código abierto hará posible que los usuarios puedan extender y mejorar el diseño de manera continua.

Para el desarrollo de circuitos integrados de mayor desempeño, los diseñadores han basado sus estrategias en las técnicas de integración de transistores en un chip, iniciando con bajas escalas de integración MSI, pasando por escalas de alta integración LSI hasta llegar a una alta capacidad de integración VLSI (millones de compuertas), cumpliendo la Ley del cofundador de la compañía Intel, Gordon Moore que establece: "El número de elementos transistores por microprocesador se duplicará cada dos años"

Hoy se habla incluso de niveles superiores de integración, como la escala ULSI con capacidad de almacenar miles de millones de transistores en una placa de silicio. Pero, se ha demostrado que la capacidad de integrar transistores en una placa de silicio fuera del factor económico, tienen límites físicos como: disipación de potencia, capacitancias

parásitas y densidad de integración y al parecer dichos límites no se encuentran muy distante de los niveles que se alcanzan en la actualidad.

Una vez alcanzado el límite físico impuesto por el material, será mucho más complicado o casi imposible seguir aumentando la capacidad de procesamiento de sistemas por medio de la integración. Estas razones han obligado a los diseñadores de procesadores a pensar en nuevas alternativas para mejorar los diseños. Surge de esta manera, la idea de desarrollar nuevas ideas y arquitecturas de procesadores como propuesta para mejorar el rendimiento del sistema.

En esta tesis se pretende generar el diseño y la caracterización, desempeño ideal de una Unidad de Gestión de Memoria (MMU) de un procesador moderno, implementar o mejorar diseños que presentan las arquitecturas actuales, mediante la investigación y combinación de diseños.

Enfocado a la eficiencia, con un diseño novedoso de baja complejidad y alto performance. Se proyecta implementar en tarjetas de desarrollo con FPGA, escalable y que trabaje con los demás módulos del proyecto Lagarto para la evaluación de los procesadores superescalares.

1.4 Objetivo general

Diseñar y evaluar una Unidad de Gestión de Memoria (MMU) para un procesador superescalar, utilizando lenguajes de descripción de hardware (HDL), para ser implementado en dispositivos lógicos programables (FPGA) y circuitos integrados CMOS VLSI.

1.4.1 Objetivos particulares

1. Diseñar MMU a nivel de microarquitectura para un procesador de arquitectura superescalar e integrar técnicas de diseño para mejora de desempeño de caches.
2. Desarrollar plataforma de evaluación (LEON-SoC + SO-Linux).
3. Desarrollar diferentes configuraciones y arquitecturas sobre el diseño SRMMU SPARCv8 del softcore LEON y evaluar su desempeño y caracterización mediante la operación del SO Linux y ejecutando programas de verificación especializados Benchmarks.
4. Obtener el circuito electrónico MMU y diseñar para un procesador superescalar utilizando VHDL haciendo el uso de herramientas de diseño del EDA Quartus II de Altera.
5. Profundizar y realizar investigación del sistema de gestión de memoria del SO Linux en *softcores* a fin de determinar y poder obtener el software para controlar nuestro diseño MMU.
6. Elaborar un diseño escalable y que converja con la arquitectura del procesador Lagarto, para interconectarlo con los módulos del *pipeline* que lo conforman.
7. Desarrollar documentación e investigación.

1.5 Alcances del trabajo

Se diseñará a nivel de microarquitectura un diseño escalable y que converja con la arquitectura de un procesador superescalar, para interconectarlo con los módulos del *pipeline* que lo conforman.

Se diseñará el comportamiento del circuito electrónico, utilizando lenguaje de descripción de hardware HDL (VHDL/Verilog) usando herramientas del EDA Quartus II de Altera.

El diseño es basado en la SRMMU SPARCv8 del *soft-processor* Leon3 que es de código abierto y distribuido bajo una licencia GPL el cual nos permite diseñar y realizar evaluaciones buscando un mejor performance e implementar una plataforma hardware-software tomando en cuenta los recursos de la tarjeta de desarrollo con la que se cuenta en el laboratorio.

Obtener el circuito electrónico MMU y diseñar para un procesador superescalar que lleve a cabo la gestión de la Memoria Virtual con el SO-Linux.

Desarrollar documentación e investigación para las áreas relacionadas con Sistemas Embebebidos y Arquitectura de Computadoras del Laboratorio de Microtecnología y Sistemas Embebidos (MICROSE).

1.6 Contribuciones

Se espera obtener el diseño y la caracterización de la Unidad de Gestión de Memoria (MMU), para ser implementado en *softcores* y sintetizado en una tarjeta de desarrollo con FPGA que además, sea escalable, configurable y trabaje principalmente con los demás módulos del proyecto Lagarto del Laboratorio de MICROSE del CIC y beneficie para el desarrollo de diseños de los procesadores superescalares.

Obtener plataforma de evaluación (LEON-SoC + SO-Linux) que sirva para realizar investigación, implementar nuevas ideas generadas para arquitecturas de procesadores y diseño y creación de hardware (IPcores) y software (drivers, aplicaciones).

Generación de conocimiento acerca del diseño y funcionamiento de una MMU. El código, reportes técnicos, manuales y material de apoyo para el profesor se distribuirán en línea, a fin de promover el desarrollo y la investigación en arquitectura de computadoras.

Instalar herramientas de programación y desarrollo de software y SO embebidos.

1.7 Método de investigación y desarrollo utilizado

El trabajo se abordó realizando investigación de los diferentes *soft-processors* que hay en el mercado y sus recursos como: IPcores y entorno de desarrollo de software para la arquitectura. Con esto poder desarrollar una plataforma Hardware-Software que está compuesta por un SoC sintetizado en un FPGA y el SO-Linux con la finalidad de evaluar el rendimiento de diferentes configuraciones y diseños de MMU.

1.8 Organización del trabajo

El presente trabajo de tesis se divide en 6 capítulos, los cuales presentan el contenido siguiente:

Capítulo 1. Introducción. En este capítulo se presentan los antecedentes de procesadores superescalares y sistemas de gestión de memoria, el planteamiento del problema, justificación, los objetivos generales y particulares, además, de los alcances de la tesis.

Capítulo 2. Estado del arte. En este capítulo se presenta el estado del arte del diseño MMU de varias arquitecturas y algunas técnicas de mejora para sistemas de memoria.

Capítulo 3. Marco teórico. En este capítulo se presenta el conjunto de elementos y teoría que se utilizaron para desarrollar la tesis. Se explica las funciones, operación y estructura de la MMU.

Capítulo 4. Modelo propuesto. En este capítulo se muestra la metodología de diseño para obtener el diseño propuesto de la Unidad de Gestión de Memoria (MMU) para un procesador superescalar. Se presenta el diseño y operación de los elementos del Sistema de gestión de memoria implementado que comprende la MMU y el Sistema de memoria.

Capítulo 5. Pruebas y resultados. Este capítulo presenta el análisis de las estadísticas obtenidas tras realizar *benchmarks* a la arquitectura diseñada mediante la evaluación de

diferentes esquemas y configuraciones de los elementos que conforman el Sistema de memoria (MMU y Caches).

Capítulo 6. Aspectos finales. En este capítulo se comentan los resultados obtenidos a través de las simulaciones, justificando cada uno de los objetivos propuestos en la tesis. Además se mencionan los trabajos futuros en relación al presente trabajo de tesis.

CAPÍTULO 2.

ESTADO DEL ARTE

En este Capítulo se presenta un análisis que se enfoca en lo distintivo de los diferentes diseños de arquitecturas MMU que se encargan de la administración de la MV, protección de segmentos de memoria y manejo de los accesos a la memoria del procesador y además, mostrar los elementos y características del sistema de memoria cache, implementados en micro-arquitecturas de procesadores escalares y superescalares con ejecución fuera de orden. Las microarquitecturas analizadas son las correspondientes a los procesadores populares MIPS R10K, Intel Pentium x86, ARMv9, OpenRISC 1200 y SPARC v8, se detalla su diseño y caracterización en las secciones 2.1, 2.2, 2.3, 2.4 y 2.5 respectivamente. Por último se analizan algunos artículos relevantes al diseño del sistema de memoria y la MMU.

2.1 Microprocesador MIPS R10000
2.2 Microprocesador Pentium Intel x86.
2.3 Microprocesador ARMv9
2.4 Softcore OpenRISC 1200
2.5 Microprocesador SPARC V8
2.6 Artículos
2.7 Resumen del Capítulo 2

CAPÍTULO 2 Estado del arte

2.1 Microprocesador MIPS R10000

El MIPS R10K [1] es un procesador superescalar, efectúa el paralelismo de instrucciones además del paralelismo de flujo, ésto gracias a su microarquitectura en *pipeline* de 7 etapas. En su diseño implementa el renombrado de registros, ejecución especulativa y predictor de saltos, con las cuales logra ejecutar hasta 4 instrucciones en paralelo fuera de orden (*out-of-order*) en cada ciclo de reloj; implementa el conjunto de instrucciones (ISA) MIPS IV arquitectura RISC de 64bits y al ser escalable permite ejecutar las anteriores versiones MIPS. El R10K es fabricado usando tecnología CMOS VLSI de 0.35 µm en una superficie de 17x18 mm y contiene alrededor de 6.8 millones de transistores a una frecuencia de operación de 175MHz a 195 MHz.

Fig. 2.1 Procesador R10000 fabricado por NEC Corporation

El R10K obtiene cuatro instrucciones por ciclo de su cache de instrucciones (*fetch 4-way*). Estas instrucciones se decodifican (*decode*) y se colocan en las entradas de las colas de instrucciones (*instruction queues*) de enteros, punto flotante o de *load/store* según el tipo de la instrucción para posteriormente ser emitidas (issue) de forma dinámica dependiendo la disponibilidad de las unidades de ejecución (*execute*). Como una mejora la Unidad de decode es asistida por la cache de instrucciones que pre-decodifica las mismas, anexando cuatro bits a cada instrucción para permitir a la Unidad de decode identificar rápidamente

a qué unidad de ejecución corresponde la instrucción a ejecutar de esa forma se reordena el formato de la instrucción para optimizar la decodificación proceso.

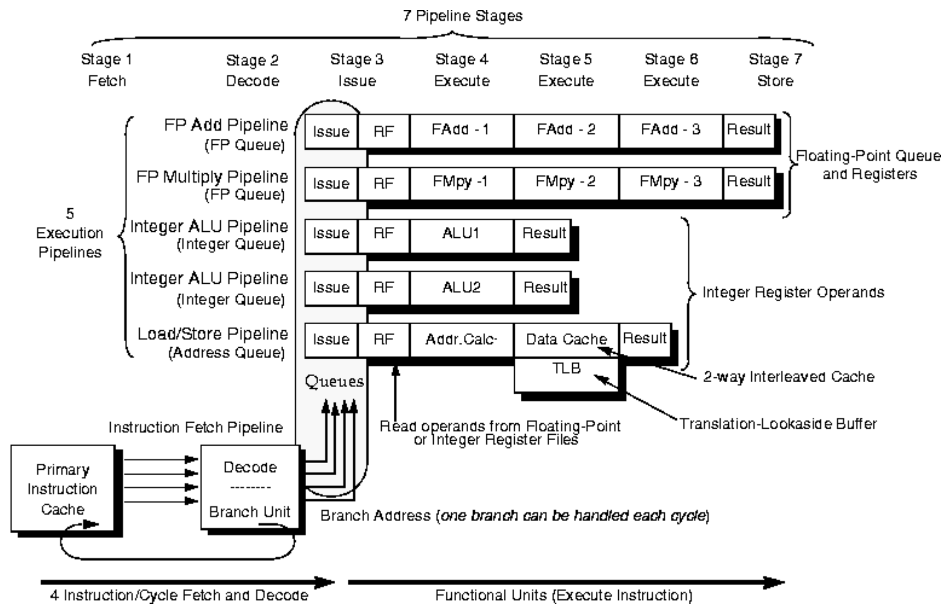


Fig. 2.2 Pipeline de la arquitectura superescalar del MIPS R10000.

El sistema de memoria del R10000 fué un gran avance, ya que proporcionó gran capacidad de almacenamiento y velocidad. La arquitectura de caches son de tipo Hardvard las principales L1 son *on-chip* 32KB de instrucciones y 32KB para datos, siendo la cache de instrucciones asociativa de dos vías (*2-way set-associative*) y con un tamaño de línea (*line-size*) de 128-bytes. La cache de datos es de dos puertos y dos vías, que consiste en dos bancos de 16KB cada uno con sus dos vías de asociatividad, cuenta con un tamaño de línea de 64-bytes, es virtualmente indexada, físicamente etiquetada (VIPT) y utiliza un protocolo *write-back*, es decir, escribir solo en la cache principal y escribir en el siguiente nivel de la jerarquía de memoria que es la cache secundaria L2, solo al reemplazar el bloque. Ambas implementan una política de reemplazo (LRU).

La cache secundaria L2 es externa puede tener capacidades de 512KB y 16MB según la versión del procesador. Es de tipo memoria de acceso aleatorio estática síncrona

(SSRAM) asociativa de dos vías (*2-ways*). El acceso a esta cache es mediante un bus de 128-bits el cual cuenta con protección de 9-bits para código de corrección de error (ECC), ambos la cache y el bus operan a una frecuencia de 200MHz aproximadamente 3.2GB/s.

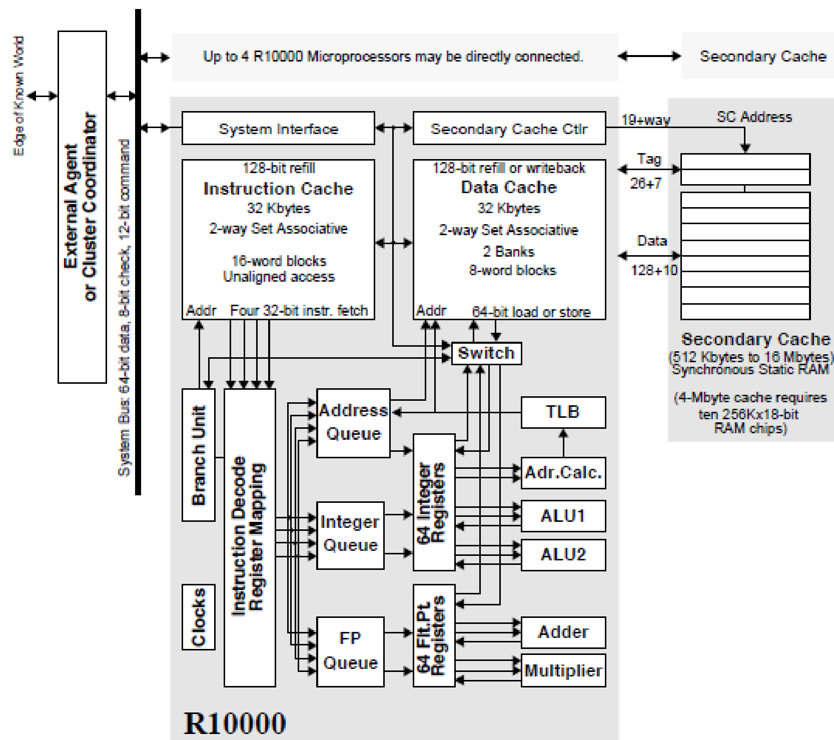


Fig. 2.3 Microarquitectura del procesador MIPS R10K

El ISA MIPS IV es de 64-bit, pero el R10K no implementa toda la dirección virtual y física para reducir los costos. En lugar de eso, tiene una dirección virtual de 44-bits que se traduce a una dirección física de 40-bits, por lo que es capaz de mapear a 16TB de memoria virtual y 1TB de memoria física.

El procesador puede operar con direcciones de memoria de 32 y 64 bits físicas o virtuales, las virtuales deben de ser traducidas a direcciones físicas para accesar a la cache. Las traducciones son mantenidas por el SO, mediante un modelo de tablas de páginas de 3 niveles en la memoria principal, que la MMU accede para obtener la dirección física. Las

referencias de memoria del R10K-MMU serían demasiado lentas, si cada una necesitara accesar a las tablas de páginas en la memoria principal para traducir de una dirección virtual a una dirección física. Por lo tanto, un subconjunto de estas traducciones se carga en una pequeña cache hardware llamada *Translation-Lookaside Buffer* ó TLB. El contenido de esta cache es mantenido por el SO y si una instrucción necesita una traducción que no se encontrara en las entradas del TLB y en las tablas de página, se generaría una interrupción para actualizar y cargar la traducción necesaria.

El TLB es de 64-entradas totalmente asociativo, que convierte las direcciones virtuales a direcciones físicas de 40-bits. Cada entrada traduce dos páginas físicas que son seleccionadas con el bit MSB de la dirección virtual, estas páginas pueden ser de tamaños que van desde 4KB hasta 16MB, en potencias de 4. El SO carga y mediante una política *random* reemplaza las entradas además, identifica entradas como *wired* que son irremplazables después de su carga en el TLB las cuales se usan para *root* PTE y el *kernel*.

A la dirección virtual se le adicionan 8bits ASID (A*ddress Space IDentifier*) que permiten mantener hasta 256 procesos en memoria y en TLB, identifica más procesos que anteriores diseños MIPS, lo que genera ahorros de cambio de contexto entre procesos.

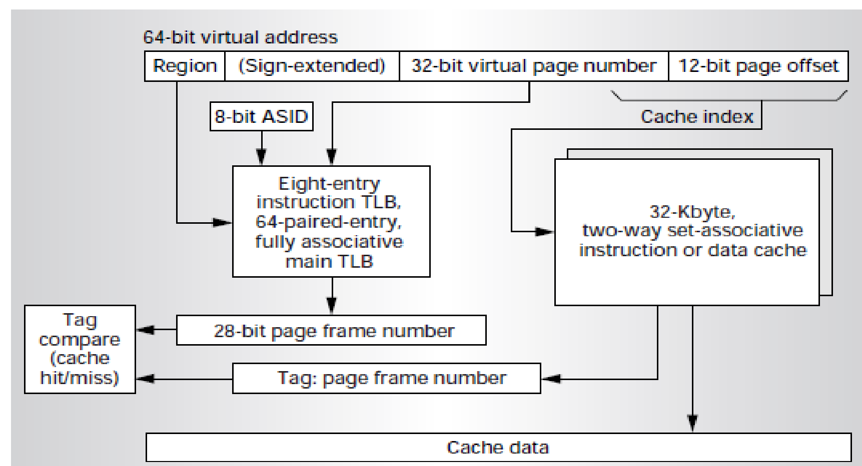


Fig. 2.4 Modelo de traducción de dirección del MIPS R10000.

El procesador tiene tres modos de operación con privilegios, segmentos e instrucciones ISA MIPS III y IV, estos son *kernel*, supervisor y usuario que se distinguen con los dos primeros bits de los *Region bits* de la dirección virtual. Los *Region bits* restantes determinan las áreas de direcciones mapeables y cacheables (*cached/uncached, mapped/unmapped*).

Usuario : 32-bits (*useg*), 2GB (231 bytes) y 64-bits (*xuseg*), 16 TB (244 bytes)

Supervisor : 32-bits (*suseg, sseg*), 2GB (231 bytes) y 64-bits (*xsuseg*), 16 TB (244 bytes)

Kernel : 32-bits (*kuseg, ksseg, kseg3*), 2GB (231 bytes)

64-bits (*xkuseg, xkseg, cksseg, ckseg3*), 16 TB (244 bytes)

Fig. 2.5 Espacio de Direcciones virtuales Modo Usuario y Modo Supervisor.

2.2 Microprocesador Pentium Intel x86

El x86 [2], implementa la memoria virtual que permite ejecutar programas mucho más grandes que la cantidad disponible del ordenador de la memoria. Lo anterior se logra mediante la MMU que mapea segmentos de memoria llamadas marcos de página a través de una serie de tablas, dos tipos para ser exactos. Ellas son el directorio de paginación y la tabla de páginas. Ambas tablas contienen 1024 entradas de 4bytes. En el directorio de página, cada entrada apunta a una tabla de página. En la tabla de páginas, cada entrada apunta una dirección física que es asignada a la dirección virtual. La forma de obtener la dirección física es mediante la segmentación en grupos bits de la dirección virtual con los cuales se realiza el desplazamiento dentro del directorio y el desplazamiento dentro de la tabla. De esta forma el sistema de tablas para 32bits representa 4GB de direcciones de memoria virtual con páginas de 4KB ó 4MB.

Al habilitar la paginación 4KB, las direcciones virtuales son mapeadas a las direcciones físicas. El Registro base de directorio de página (CR3) apunta a la base directorio de página, cada entrada en el directorio (PDE) apunta a una tabla de páginas y cada entrada de la tabla de páginas (PTE) apunta a la página. La dirección virtual es segmentada en tres partes: los 10 bits más altos para indexar en el directorio de página, los 10 bits siguientes indexan en la tabla de página y los restantes 12 bits son para indexar en la página de 4Kb.

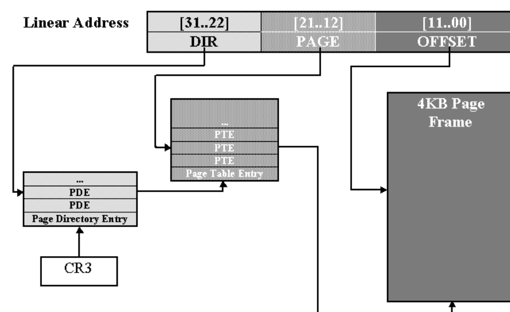


Fig. 2.6 Traducción de página de 4KB Intelx86.

Para activar la paginación x86 se necesita cargar el registro CR3 con la dirección del directorio de página y activar el bit CR0 de paginación.

```
mov eax, [page_directory]
mov cr3, eax
mov eax, cr0
or eax, 0x80000000
mov cr0, eax
```

Con marcos de página de 4MB [3] se permite al SO accesar a una gran cantidad de memoria sin constantes referencias y cambios de entradas en las tablas de páginas. Esta característica es muy útil para el SO que desea una página de memoria dedicada al *kernel* o una gran estructura de datos, como un buffer de video y para desarrolladores de software. Para su implementación se activa el bit del registro especial CR4.PSE y el modelo de traducción de tabla de páginas se reduce a un nivel, siendo el directorio el que apunta a los marcos de página de 4MB. La dirección virtual de 32bits es segmentada en un *offset* de 22bits y el directorio de 10bits y es convertida de la misma manera que una página de 4KB.

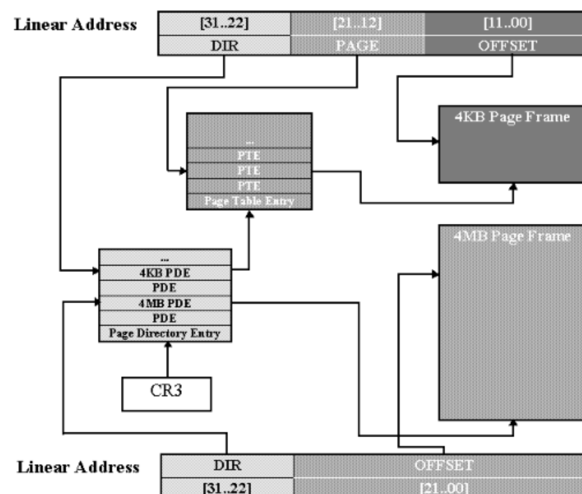


Fig. 2.7 Traducción de página de 4KB y 4MB Intelx86.

Para activar PSE (4MB) se requiere el siguiente código:

```
mov eax, cr4
or eax, 0x00000010
mov cr4, eax
```

Todos los procesadores Intel desde el Pentium Pro (con la excepción del Pentium M en 400 Mhz) y todo el AMD desde la serie Athlon implementan la Extensión de Dirección Física (PAE). Este rasgo le permite tener acceso hasta 64 GB (2^36) de RAM esto requiere del soporte de un SO que lo implemente y se activa poniendo el bit CR4.PSE =1 (CR4[5]=1), además las entradas en el directorio de páginas tienen un indicador en el bit 7, llamado PS para tamaño de página. Si el sistema ha establecido este bit a 1, la entrada de directorio de páginas apunta a una página de 2 MB (*Page Size Extension*).

La arquitectura x86 cuenta con los registros de control (CR0, CR1, CR2, CR3, y CR4) de 32bits [4] que determinan el modo de funcionamiento del procesador y las características de la tarea que se está ejecutando.

Tabla 2.1 Descripción de los registros especiales x86

Registro de control	Descripción
CR0	Contiene las banderas de control de operación que controlan el modo de funcionamiento y los estados del procesador.
CR1	Reservado.
CR2	Contiene la dirección virtual de la página que ocasionó el fallo.
CR3	Contiene la dirección física de la base del directorio de página (PDBR) y dos banderas (PCD y PWT).
CR4	Contiene el PSE bit para activar la paginación de 4MB y el PAE bit para activar direcciones físicas de 36 bits trabajando con paginación de 4KB o 2MB.

El SO realiza la gestión de la MV. Para obtener la dirección física de una dirección virtual en el rango de 0x00000000-0xFFFFF000 puede realizar lo siguiente:

```
void * get_physaddr(void * virtualaddr)

{

  unsigned int pdindex = (unsigned int)virtualaddr >> 22;

  unsigned int ptindex = (unsigned int)virtualaddr >> 12 & 0x03FF;

  unsigned int * pd = (unsigned int *)0xFFFFF000;

   // Here you need to check whether the PD entry is present.

   unsigned int * pt = ((unsigned int *)0xFFC00000) + (0x400 * pdindex);

    // Here you need to check whether the PT entry is present.

   return (void *)((pt[ptindex] & ~0xFFF) + ((unsigned int)virtualaddr &
0xFFF));

}
```

Para asignar una DV a una DF se puede realizar lo siguiente:

```
void map_page(void * physaddr, void * virtualaddr, unsigned int flags)

{

    // Make sure that both addresses are page-aligned.

    unsigned int pdindex = (unsigned int)virtualaddr >> 22;

    unsigned int ptindex = (unsigned int)virtualaddr >> 12 & 0x03FF;

    unsigned int * pd = (unsigned int *)0xFFFFF000;

    // Here you need to check whether the PD entry is present.

    // When it is not present, you need to create a new empty PT and
adjust the PDE accordingly.

    unsigned int * pt = ((unsigned int *)0xFFC00000) + (0x400 * pdindex);

    // Here you need to check whether the PT entry is present.

    // When it is, then there is already a mapping present. Then...

   pt[ptindex] = ((unsigned int)physaddr) | (flags & 0xFFF) | 0x01;

    // Now you need to flush the entry in the TLB or you might not notice
the change.}
```

2.3 Microprocesador ARMv9

El procesador ARM (*Advanced Risc Machine*) [5] es de arquitectura RISC de 32-64bits, desarrollado por la empresa Acorn Computers Ltd. bajo el mando de Roger Wilson y Steve Furber en 1987. Debido al éxito de la arquitectura, posteriormente se decidió crear una nueva compañía llamada Advanced RISC Machines, que sería la encargada del diseño y gestión de las nuevas generaciones de procesadores ARM en el año 1990. La arquitectura ARM es licenciable, empresas como Qualcomm, Samsung, TexasInstruments, Nvidia, Apple, Nintendo, Marvell Technology, Freescale, Broadcom, Alcatel-Lucent son algunas de las titulares con licencias de ARM.

La arquitectura ARM es ideal para aplicaciones de bajo consumo de potencia. A partir del 2005, alrededor del 98% de los más de mil millones de teléfonos móviles vendidos cada año, utilizan al menos un procesador ARM. Desde 2009, los procesadores ARM son aproximadamente el 90% de todos los procesadores RISC de 32 bits integrados y se utilizan ampliamente en la electrónica de consumo, incluyendo PDA, tabletas, teléfono inteligente, teléfonos móviles, videoconsolas portátiles, calculadoras, reproductores digitales de música y video y periféricos de ordenador como discos duros y routers.

Se han desarrollado mejoras en arquitecturas clasificadas en versiones. ARMv1 fué el diseño preliminar y el primer prototipo del procesador con direcciones de 26 bits. La primera versión utilizada comercialmente se bautizó como ARMv2 y se lanzó en el año 1986, que posee un bus de datos de 32 bits y ofrece un espacio de direcciones de 26 bits, junto con 16 registros de 32 bits, es probablemente el procesador de 32 bits útil más simple del mundo, ya que posee sólo 30.000 transistores, como era común en aquella época, no incluye cache. La siguiente versión fue el ARMv3 con direcciones de 32 bits y se incluye una pequeña cache unificada de 4KB. La mayor utilización de la tecnología ARM se alcanzó con ARMv4T de 32bits del procesador ARMv7TDMI (ARM7-Thumb+Debug+Multiplier+ICE), con millones de unidades en teléfonos móviles y sistemas

de videojuegos portátiles lanzada en 1994. Fué la primera en introducir el ISA Thumb 16-bit el cual al trabajar en este estado, se ejecuta un conjunto de instrucciones compactas de 16bits con las cuales se mejora la ejecución y disminuye la densidad de código de programas. Consecutivamente se desarrollaron las versiones ARMv5T que incluía un superset del ISA Thumb añadiendo nuevas instrucciones y el ARMv5TE que añade una extensión de instrucciones para el procesamiento digital.

El procesador ARM926EJ-S (ARM9) [6] de 32bits basado en la versión ARMv5TEJ fué el primero en cambiar de una arquitectura Von Neumann (ARM7) a una arquitectura Harvard diseñada con caches y buses separados de instrucciones y datos, la cual incrementó el desempeño. La MMU del ARM926EJ-S es una v5 MMU es compatible con v4 MMU y requiere de Sistemas Operativos como Symbian OS, WindowsCE, y Linux. Utiliza un coprocesador que mediante los registros especiales (CP15) configuran y asisten en la operación de caches y MMU.

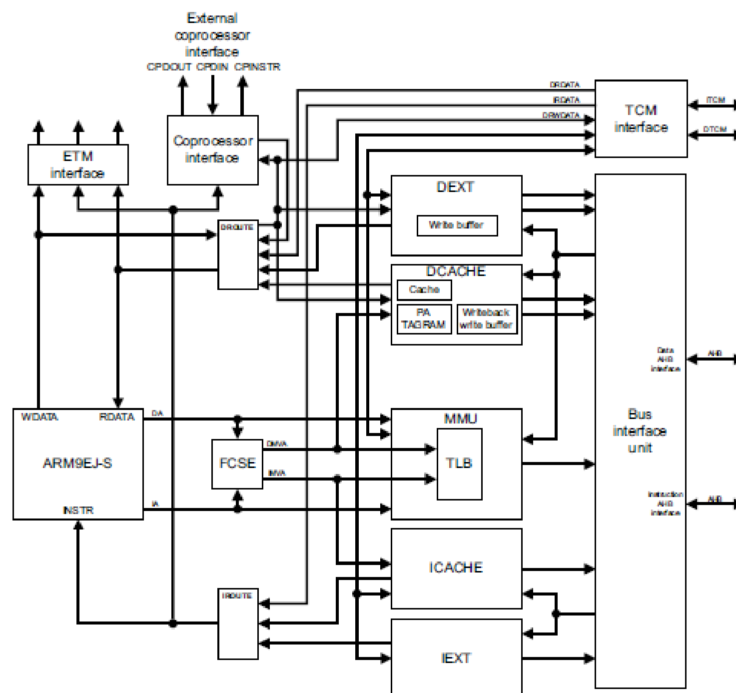


Fig. 2.8 Diagrama de bloques del procesador ARM926EJ-S

Para accesar a los registros CP15 se usan las instrucciones MRC para lectura y MCR para escritura. Para habilitar la MMU se utiliza la instrucción MCR sobre el registro CP15c1.

```
MRC p15, 0, R1, c1, C0, 0 ;Read control register

ORR R1, #0x1 ;Set M bit

MCR p15, 0,R1,C1, C0,0 ;Write control register and enable MMU
```

El núcleo ARM926EJ-S utiliza un rango de direcciones VA de 25bits (0 a 32MB), a la cual se le adhieren 7 bits llamados *Fast Context Switch Extension* (FCSE PID) del registro CP15c13 para permitir multiprocesamiento de 128 x 32MB. La VA + (FCSE PID x 32MB) forman la *Modified Virtual Address* MVA la cual es utilizada por caches y MMU.

Las tablas de página son de 1 o 2 niveles dependiendo de la página: 1MB (*sections*), 64KB (*large pages*), 4KB (*small pages*) y 1KB (*tiny pages*). Para realizar la traducción, la MMU necesita el *Translation Table Base Register TTBR* (registro CP15c2) y la MVA.

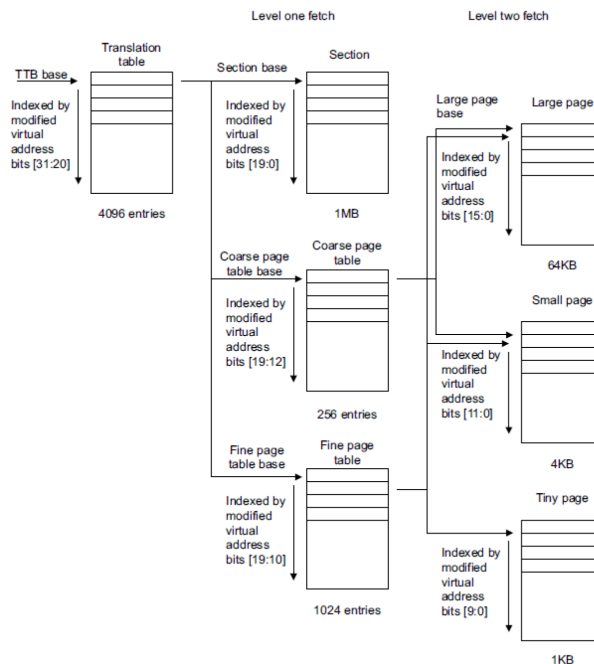


Fig. 2.9 Tablas de traducción de página ARMv5.

La MMU implementa dos modos de acceso cliente y administrador. El modo cliente es identificado con 16 dominios de 2 bits cada uno que definen sus accesos, controlados por el registro CP15c3 *Domain Access Control Register*.

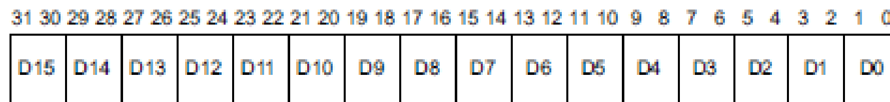


Fig. 2.10 Formato de registro c3 del ARM.

Trabajando en modo cliente los accesos a memoria son comprobados contra los bits de acceso permiso en la traducción de la dirección virtual TLB o tabla de página. En modo administrador los bits no son comprobados, cualquier acceso es permitido.

La traducción a través de las tablas de página de una dirección VA de tamaño de página 4KB *small* se realiza con la siguiente metodología:

- CPU emite la dirección virtual VA.
- VA es identificada con los bits de contexto FCSE PID para formar la MVA.
- Se toman los bits MVA[31:20] para accesar a la entrada de tabla del primer nivel.
- La MMU comprueba los bits de Dominio que especifican uno de los 16 dominios posibles del registro de control de acceso, que indica el acceso permitido. Posteriormente identifica que se trata de un descriptor *Coarse* de primer nivel con los bits [1:0] = 0,1. Si no existe fallo toma la dirección base del segundo nivel.
- Con la dirección base obtenida del segundo y los bits MVA[19:12] se indexa a la entrada de la tabla del segundo nivel.
- La MMU comprueba bits de acceso lectura, escritura en modo usuario, en modo administrador no realiza la comprobación ya que se tiene cualquier tipo de acceso. Los bits [3:2] C y B indican si es *write-back, write-through* o *noncached*. Los bits [1:0]

indican si es una página tipo *Large, small* =1,0 *tiny* o *invalid*. Si no hay algún fallo toma los bits [31:12] que corresponden a la dirección física.

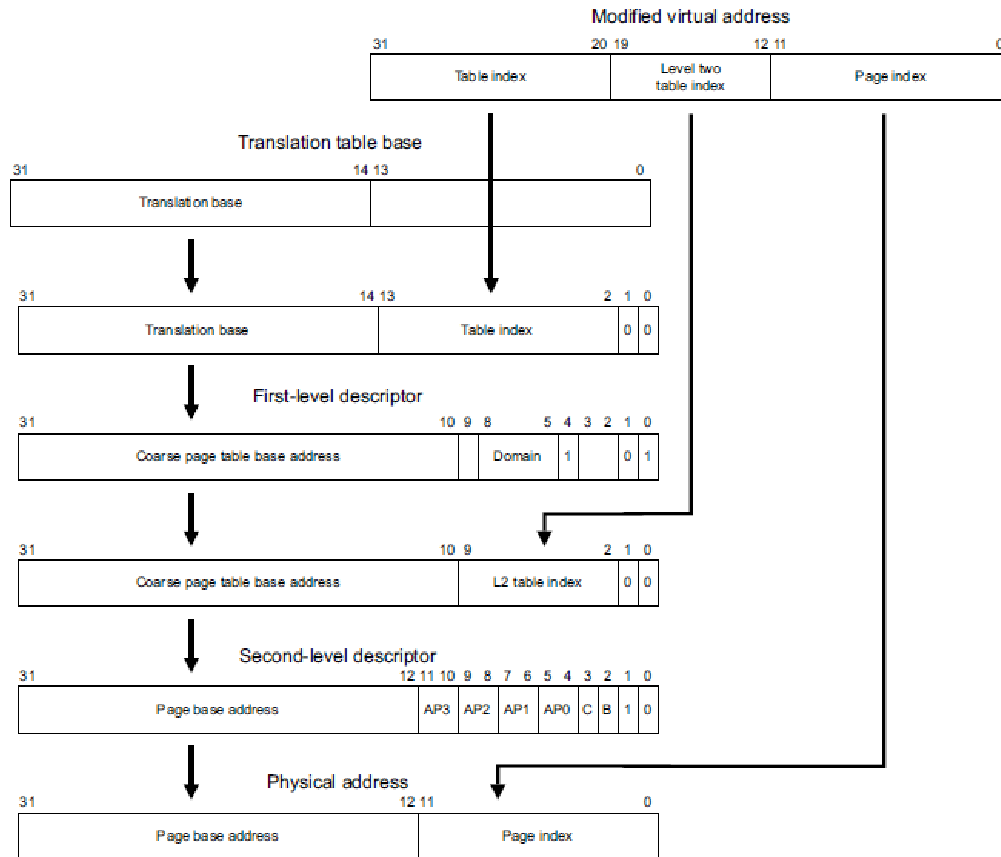


Fig. 2.11 Diagrama de secuencia de traducción completa para una 4KB *small* page.

Se implementa una TLB unificada para instrucciones y datos de 64 entradas para traducir las MVA's, es activada o desactivada usando el registro CP15c8. La TLB es dividida en dos partes:

- 8 entradas totalmente asociativas para almacenar *locked TLB entries*. Las entradas en esta área son preservadas y no reemplazadas, por lo cual no generan un fallo en TLB, son escritas usando el registro *TLB Lockdown Register* c10.
- 32 entradas generales con asociatividad de 2-*way*.

El procesador ARM926EJ-S cuenta con una cache de instrucciones (ICache) una de datos (DCache) y *write buffer*. Son asociativas 4-*way*, 8 palabras por línea (32bytes) y su tamaño puede ser desde 4KB a 128KB en incrementos de potencias de dos. Son activadas independientemente usando los bits I, C, y M del CP15c1. La cache de datos puede trabajar en modo *Write-through* y *Write-back* es activado con el bit B del registro CP15c1.

El *write buffer* es una memoria de alta velocidad FIFO de 16 palabras para datos y de 4 palabras para instrucciones. Entre la cache de datos y la memoria principal, cuyo propósito es optimizar la escritura en la memoria principal. Es usado para escribir en áreas *noncacheable* (periféricos), *write-through* y fallos de escritura en *write-back.*

La cache es de tipo *Virtually indexed, virtually tagged* (VIVT) [7]. Este esquema de almacenamiento de caches realiza búsquedas mucho más rápidas y ahorra energía, ya que no es necesario que se consulte primero la MMU para determinar la dirección física para una determinada dirección virtual y accesar al dato. Sin embargo VIVT es especial y necesita de una administración, de lo contrario sufre de problemas, donde varias diferentes direcciones virtuales pueden referirse a la misma dirección física. El resultado es que éstas direcciones se almacenan en cache por separado a pesar de referirse a la misma memoria, causando problemas de coherencia. En multiprocesamiento es necesario el etiquetado de la dirección virtual con un ID de espacio de direcciones (ASID).

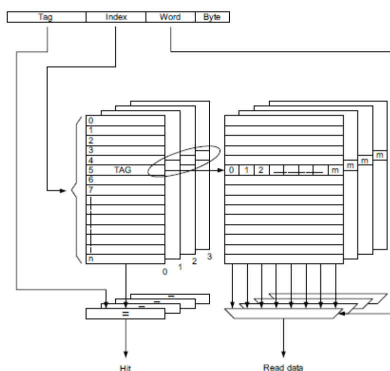


Fig. 2.12 Operación de cache tipo VIVT asociativa de 4-*way*.

2.4 Softcore OpenRISC 1200

El *softcore* OpenRISC es de código abierto, donde el código fuente está en dominio público y distribuido bajo una licencia LGPL que permite que este *softcore* pueda ser descargado, modificado y usado para cualquier implementación.

Una de las características más interesantes del *core* OpenRISC [8] es su capacidad de adaptación y personalización para acomodarse a las diferentes necesidades y tecnologías de implementación que existen como FPGA's y CMOS.

Este *core* está completamente realizado en Lenguaje de Descripción de Hardware (HDL-Verilog) y contempla el conjunto de instrucciones RISC de 32bits llamadas ORBIS32. El OpenRisc 1200 [9] es un procesador escalar segmentado con un pipeline de 5 etapas, bus del sistema WishBone, caches de instrucciones y datos separados basadas en la arquitectura Hardvard. Tiene instrucciones básicas para el procesamiento de señales digitales (DSP) y soporte para gestión de la MV.

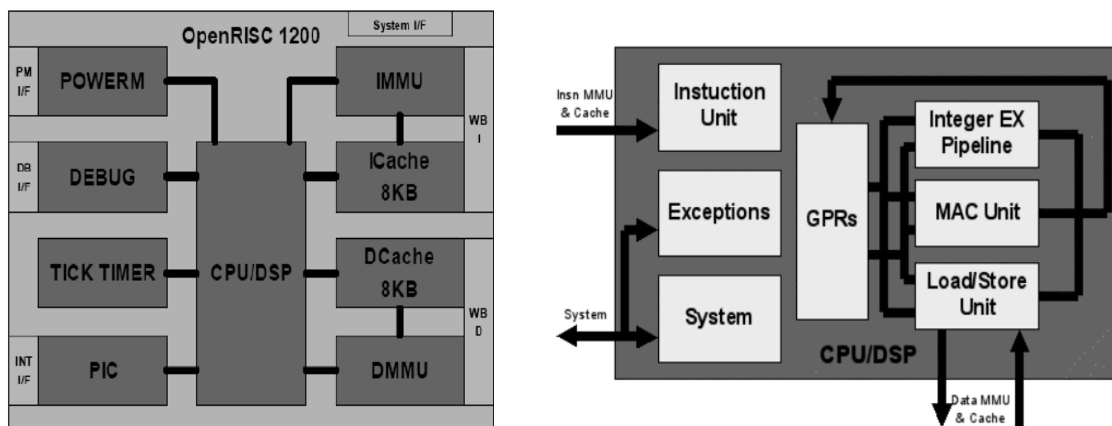


Fig. 2.13 Arquitectura del *softcore* OpenRisc 1200

El OR1200 puede ser integrado en el ORPSoC el cual está compuesto por varios bloques funcionales cuyas características principales son listadas a continuación:

- Arquitectura Harvard de 32 bits, caches separadas de datos e instrucciones.
- 32 registros de propósito general.
- Unidad de Gestión de Memoria MMU (instrucciones y datos).
- Gestión de energía (Sophisticate Power Management Unit (SPMU)).
- Estados de reducción de la potencia de 100x a 2x.
- Frecuencia de reloj controlada por software en modos *slow* e *idle*.
- Unidad de depuración avanzada.
- Acceso y control a la unidad de depuración desde el RISC y el interfaz externo.
- Multiplicador Hardware y unidad MAC.
- Controlador programable de interrupciones.
- *On-chip* RAM.
- Controlador uart serial y paralelo.
- Controlador Ethernet.
- Controlador SPI/FLASH
- Variedad de periféricos con interface Wishbone para usar con el OpenRisc 1200.
- El *pipeline* de ejecución de enteros implementa instrucciones de 32bits (ORBIS32) tipo aritméticas, comparación, lógicas, corrimiento y rotación.

El OR1200 ha sido implementado en tecnología 0.18 μm 6LM y provee 250 DMIPS *dhrystones* a frecuencia de 250MHz y operaciones de 300 DSP MAC 32×32 en una configuración que utiliza MMU y caches aproximadamente solo un millón de transistores.

El procesador cuenta con una cache de mapeo directo (1-*way*) de 8KB para instrucciones y una cache de mapeo directo (1-*way*) de 8KB para datos, cada una con 16 bytes por línea y son físicamente mapeadas (*physically tagged*), por otro lado, el tamaño de cada cache puede ser configurada según los recursos del FPGA. La política de escritura es

write-through, es decir, la escritura también se realiza en el siguiente nivel de memoria. Las operaciones de escritura pueden ser de 1 byte, media palabra (*half-word*) o palabra (*word*).

Las funciones principales de la MMU son traducir las direcciones virtuales en direcciones físicas para accesos a memoria. Además, la MMU proporciona varios niveles de protección de acceso.

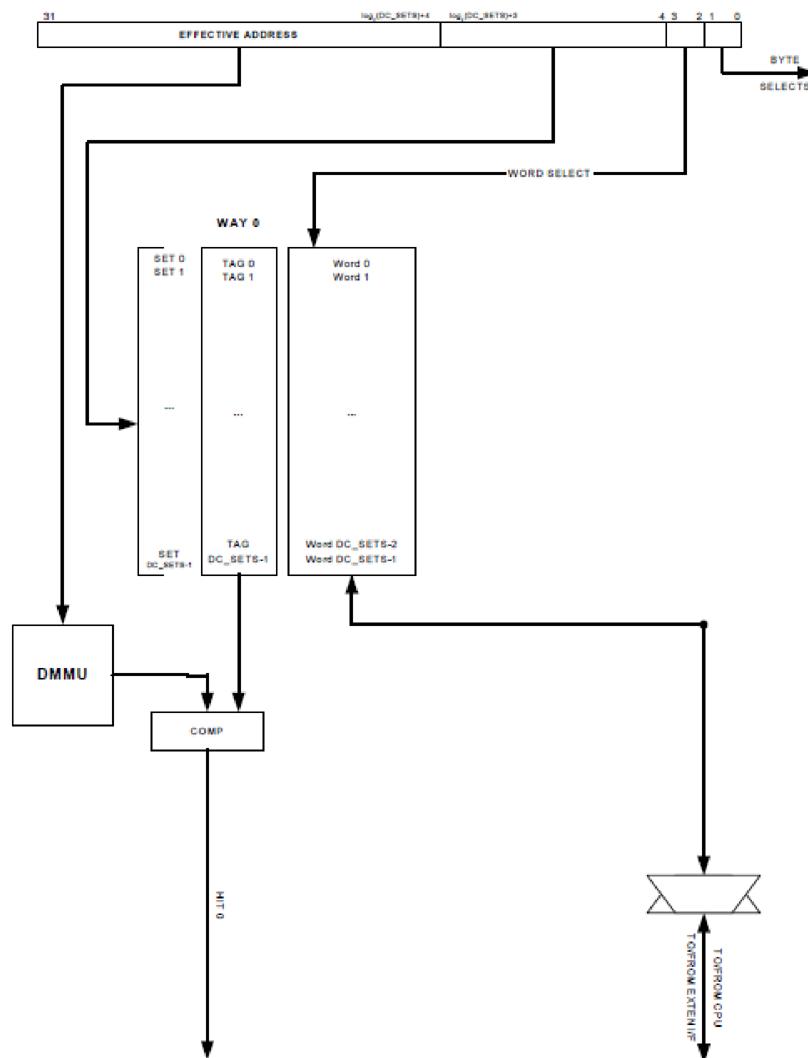


Fig. 2.14 Sistema de memoria OpenRISC 1200.

La MMU del procesador OpenRISC 1200 incluye las siguientes características:

- Soporte para dirección virtual efectiva (EA) de 32 bits.
- Dirección virtual de 36 bits (EA 32 bits + CID 4 bits)
- Implementación de dirección física de 35bits (32 GByte).
- Traducción de dirección de dos niveles empleando tablas de página, para un tamaño de página de 8KB.
- Implementa TLBs de instrucciones y datos configurables hasta 128 entradas de mapeo directo (1-*way*).
- Sistema de protección de páginas.
- Sistema de interrupciones por fallo de página o falta de acceso (*miss and fault*)
- Memoria virtual por demanda (*demand-paged*) al SO.

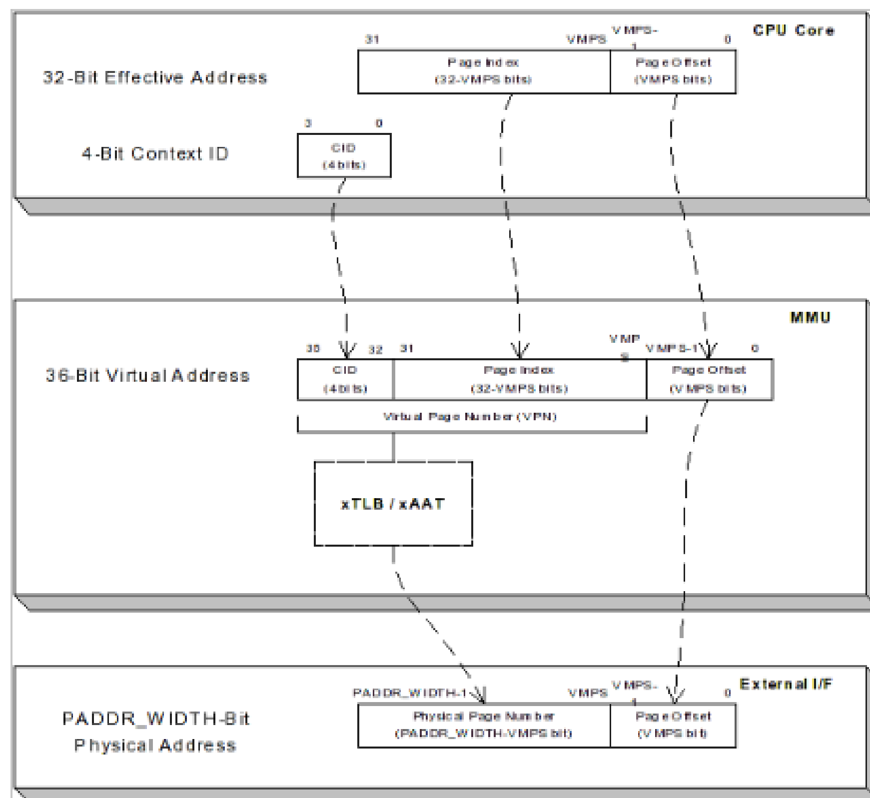


Fig. 2.15 Diagrama de flujo de traducción de DV a VF OpenRISC 1000.

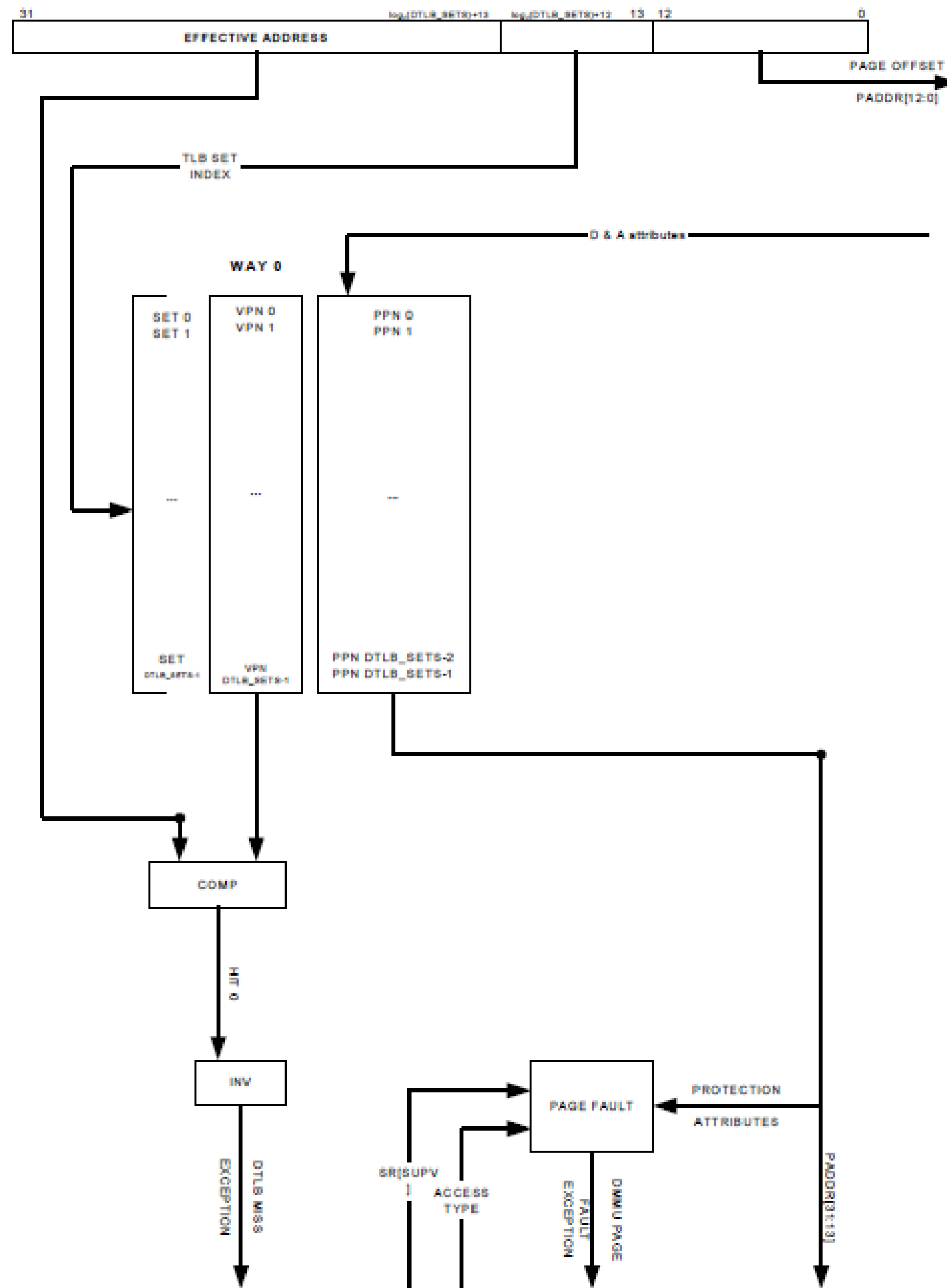


Fig. 2.16 Sistema de MMU/TLB OpenRISC 1200.

El OR1200 no implementa la recarga de entradas del TLB por hardware, en lugar de ello, se ejecuta una rutina software para buscar la entrada de la tabla de página (PTE) correcta y escribirla en las entradas del TLB registros match xTLBWyMR y translate

xTLBWyTR. El software también es el encargado de gestionar los bits de *accessed* y *dirty* de las entradas, de esa forma es posible para el hardware implementar el reemplazo mediante la política de LRU. En caso de un fallo de traducción en TLB se envía una interrupción para ejecutar una rutina de recarga de entrada.

El procesador puede trabajar con direcciones virtuales o físicas mediante la activación del bit del registro especial *Supervision Register* SR[DME]/[IME]=1-0. La traducción en modo virtual mediante las Tablas de página es administrada por el SO usando dos niveles para la traducción. Las tablas como el apuntador base de las tablas (PTBP) son almacenadas y administradas en variables software con el propósito de reducir el hardware del circuito ya que el objetivo del procesador es ser implementado en sistemas embebidos.

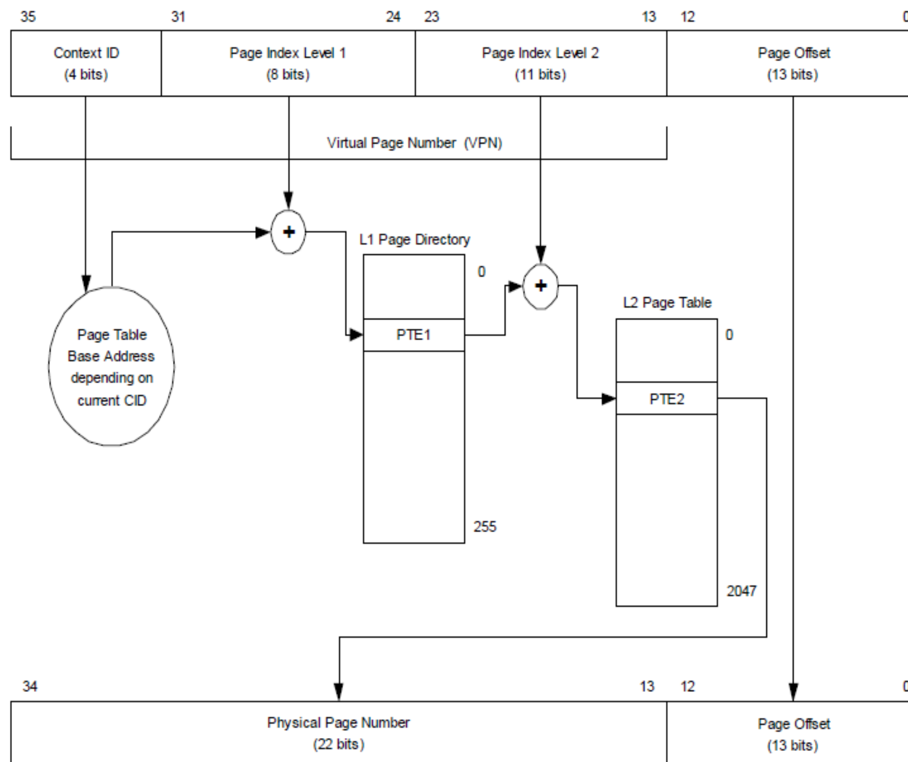


Fig. 2.17 Sistema 32 bits de Tablas de página de dos niveles OpenRISC 1200.

La siguiente tabla resume los registros que el Sistema Operativo utiliza para programar la MMU. Estos registros son de 32 bits de propósito especial en nivel supervisor. Estos registros se encuentran accesibles en el *kernel*-2.4 de Linux con modificaciones para el OpenRISC 2011-2012 accesibles por las instrucciones `l.mtspr/l.mfspr` en modo supervisor. Ambas xTLBs al igual que las TPs son actualizadas por el SO.

Tabla 2.2 Descripción de los registros de control OpenRISC MMU

# Grupo	# Registro	Nombre	Descripción
1	0	DMMUCFGR	Registro de configuración Data MMU
	2	DTLBEIR	Registro entrada inválida de Data TLB
	4-7	DATBMR0-DATBMR3	Registros match Área de datos DATB
	8-11	DATBTR0-DATBTR3	Registros de Área de datos traducción DATB
	1024-1151	DTLBW0MR0- DTLBW0MR127	Registros match de datos en TLB Way0
	1536-1663	DTLBW0TR0- DTLBW0TR127	Registros de datos traducidos en TLB Way0
2	0	IMMUCFGR	Registro de configuración Instruction MMU
	2	ITLBEIR	Registro entrada inválida de Intruction TLB
	4-7	IATBMR0-IATBMR3	Registros match Área de Instrucción IATB
	8-11	IATBTR0-IATBTR3	Registros de Área traducción Instrucción IATB
	1024-1151	ITLBW0MR0- ITLBW0MR127	Registros match de instrucciones en TLB Way0
	1536-1663	ITLBW0TR0- ITLBW0TR127	Registros de inst. traducidos en TLB Way0

Las regiones de memoria mapeables o no mapeables son de direcciones de 30 bits. Las direcciones no mapeables son utilizadas para periféricos externos mapeados a ellas y no a segmentos de memoria del procesador.

Tabla 2.3 Espacio de direcciones OpenRISC

Effective Address	Region
0x00000000 - 0x3FFFFFFF	Cached
0x40000000 - 0x7FFFFFFF	Uncached
0x80000000 - 0xBFFFFFFF	Cached
0xC0000000 - 0xFFFFFFFF	Uncached

Para realizar cualquier acceso a memoria, la dirección efectiva debe ser traducida a una dirección física. Una de la MMU interrupción se produce si esta traducción falla, las cuales pueden ser por fallo en TLB ó por fallo en la Tabla de página PTE.

Tabla 2.4 Vectores de interrupción MMU OpenRISC

Exception name	Vector offset	Causing conditions
Data Page Fault	0x300	No matching PTE found in page tables or page protection violation for load/store operations.
Instruction Page Fault	0x400	No matching PTE found in page tables or page protection violation for instruction fetch.
DTLB Miss	0x900	No matching entry in DTLB.
ITLB Miss	0xA00	No matching entry in ITLB.

Antes que una traducción de dirección virtual sea determinada como válida, el Sistema de Protección de página comprueba el acceso a la PTE mediante los bits de acceso, en caso de que no se cuente con los permisos se prohíbe el acceso y se genera una interrupción de falta de acceso a página (*fault exception*). Los modos de acceso son de lectura y escritura ambos pueden ser en modo usuario y supervisor.

Tabla 2.5 Modos de acceso a memoria OpenRISC 1200

Protection attribute	Meaning
DMMUPR[SREx]	Enable load operations in supervisor mode to the page.
DMMUPR[SWEx]	Enable store operations in supervisor mode to the page.
IMMUPR[SXEx]	Enable execution in supervisor mode of the page.
DMMUPR[UREx]	Enable load operations in user mode to the page.
DMMUPR[UWEx]	Enable store operations in user mode to the page.
IMMUPR[UXEx]	Enable execution in user mode of the page.

2.5 Microprocesador SPARC V8

SPARC Scalable Processor ARChitecture [11] es una arquitectura RISC *big-endian* con ISA escalable, por lo cual cuenta con versiones posteriores de la misma arquitectura con mayor cantidad de características, siempre salvaguardando la compatibilidad de los programas de versiones anteriores. Se han emitido tres grandes versiones de la arquitectura SPARC, la primera de ellas (1986) fue la versión V7 de 32-bits, sucesivamente una versión superior SPARC V8 (1992) que ejecuta operaciones de enteros de 32-bits de multiplicación y división, de punto flotante que es la base para el Estándar IEEE 1754-1994, un estándar IEEE de 32 y 64-bits. La versión V9 de SPARC (1993) posee una arquitectura de 64-bits, la cual se usa en los procesadores actuales como el SPARC64 VIIIfx "Venus" 2.0GHz 8 núcleos 128GFLOP de Fujitsu Laboratories en 2009 usado en la supercomputadora Japonesa "K computer" #1 del Top500 supercomputers en noviembre del 2011 y #3 en noviembre del 2012. Además el SPARC T4 diseñado por Oracle Corporation en el 2011 de 8 núcleos a 2.5GHz.

Desarrollado por Sun Microsystems, SPARC es la primera arquitectura RISC abierta y como tal, las especificaciones de diseño están publicadas, así que pueden ser usadas para propósitos de investigación y desarrollo, sus licencias de fabricación las tienen empresas como Texas Instruments, Atmel, Cypress Semiconductor y Fujitsu.

Una de las ideas innovadoras de esta arquitectura, es la ventana de registros del diseño Berkeley RISC [13]. El procesador posee mucho más que 32 registros enteros, pero presenta a cada instante 32. Una analogía puede ser creada comparando la ventana de registros con una rueda rotativa. Alguna parte de la rueda siempre está en contacto con el suelo; así al girarla tomamos diferentes porciones de la rueda, (el efecto es similar para el *overlap* de la ventana de registros). El resultado de un registro se cambia a operando para la próxima operación, obviando la necesidad de una instrucción Load y Store extras.

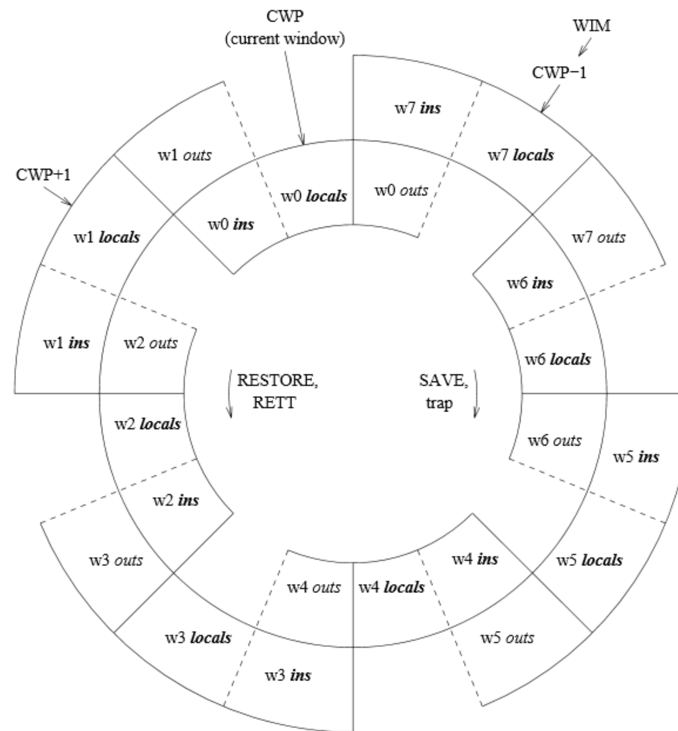


Fig. 2.18 Ventanas de registros [14].

Su diseño permite una significativa reducción de memoria en las instrucciones load/store en relación con otras arquitecturas RISC. Las ventajas se aprecian sobre todo en programas grandes. La cantidad de ventanas es un parámetro de la implementación generalmente 8 a 32 cada una de ellas presenta a cada instante 32 registros divididos en grupos de 8 (globales, salida, locales, entrada) siendo 32 registros enteros de 32 bits, 16 registros de punto flotante de 64 bits.

La Unidad de gestión de memoria SPARC Reference MMU v8 [12] está integrada en el mismo chip del CPU, de igual forma puede ser implementada en un chip con tecnología CMOS ó Bi-CMOS. Tiene las siguientes características: 32 bits para dirección virtual, 36 bits para dirección física para proveer 64GB de espacio de memoria, tamaño de página de 4KB, mapeo de tres niveles, soporte para múltiples procesos, protecciones de nivel página, grandes espacios de mapeo lineal (256KB, 16MB, 4GB).

El modelo de operación de la SRMMU es mediante la obtención de la DV que emite el procesador de ésta, se toma el segmento de bits de la Página virtual y se compara con los *tags* almacenadas de cada una de las entradas del *Page Descriptor Cache PDC* (TLB). Si hay un *hit* se genera la dirección física directamente. De lo contrario es un *miss*, y la MMU genera una *trap instruction_access_MMU_miss* 0x3C (fallo en TLB) para buscar en los Page Table Descriptors hasta encontrar la Page Table Entry buscada, si la entrada es encontrada se envía a la IU y se carga en la PDC para futuras traducciones. En caso de no ser encontrada en las tablas de página se produce un *trap instruction_access_exception* 0x01 (fallo de página) al SO. Además, se checan los permisos y si hay violación, se produce un *trap*. Por ejemplo si un proceso quiere acceder a una página protegida o modificar una de lectura se produce la *trap instruction_access_exception* 0x01.

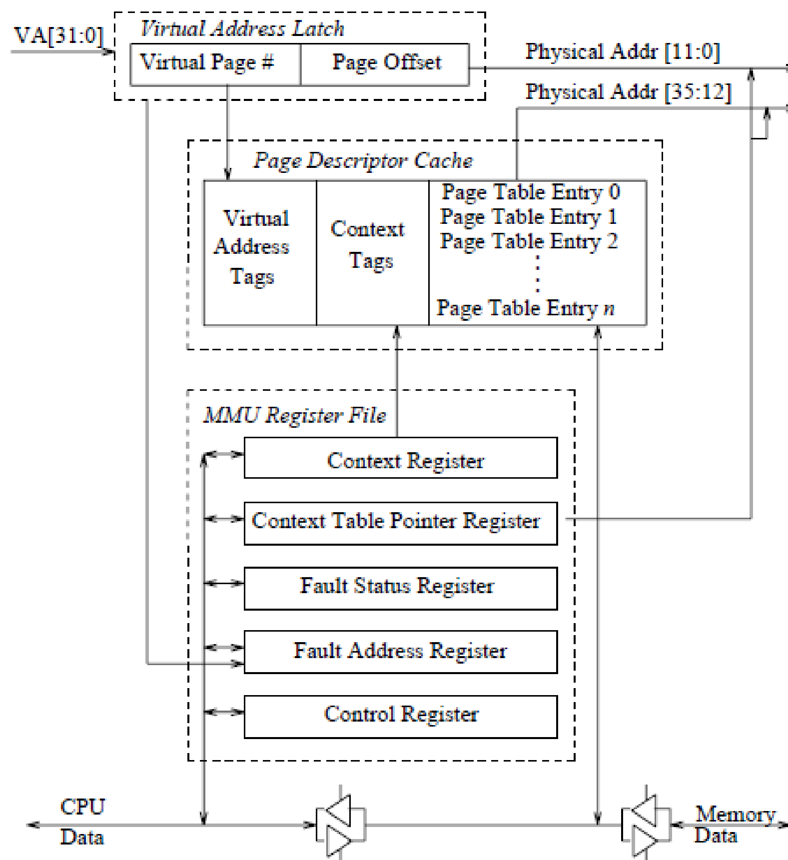


Fig. 2.19 Modelo de operación de la MMU del procesador SPARC

Cinco registros internos definen la configuración y estado de la SRMMU. El Registro de Control contiene información general sobre la operación de la MMU y los indicadores de estado (*status flags*). El identificador del proceso actual se almacena en el Registro de Contexto que es un puntero a la base de la tabla de contexto en memoria llamado Context Table Pointer Register. Si se produce un error de operación de MMU, la dirección que causó el fallo se coloca en el Registro de Dirección de Fallo y la causa de la falla se puede determinar a partir del Registro de Estado de Fallo. A todos los registros internos de MMU se puede acceder directamente por el CPU a través de accesos al mapa de direcciones de periféricos. Mediante el uso de las Alternate Space Instructions (ASI) podemos acceder a estos registros especiales. Estas instrucciones se ejecutan en modo supervisor y tienen la siguiente estructura: "`lda [addr_mmureg] asi,%r`" and "`sta %r,[addr_mmureg] asi`". Para más información consultar el Apéndice H del SPARCv8 Manual.

Tabla 2.6 Direcciones de registros internos de la SRMMU

VA[31:0]	Register
0X000000xx	Control Register
0X000001xx	Context Table Pointer Register
0X000002xx	Context Register
0X000003xx	Fault Status Register
0X000004xx	Fault Address Register
0X000005xx to 0X00000Fxx	Reserved
0X000010xx to 0XFFFFFFxx	Unassigned

Fig. 2.20 **Registro de Control:**

IMPL	Ver	Custom	PSO	resvd	NF	E
31 28	27 24	23 8	7	6 2	1	0

– IMPL: MMU Implementada.
– VER: Versión de MMU.
– CUS: Usados para aplicaciones especiales.
– PSO: *Partial Store Order*, modo de trabajo con el sistema de memoria.
– NF: *No Fault* bit, desactivar *traps* para el procesador.
– E: Activar MMU= 1, Desactivar MMU=0.

Fig. 2.21 **Registro Apuntador de Contexto:** Contiene la DF base de las Tablas de contexto.

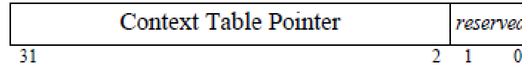


Fig. 2.22 **Registro de Numero de Contexto:** Almacena el número de contexto del proceso en ejecución, con el se realiza el desplazamiento (*offset*) dentro de la tabla de contexto.

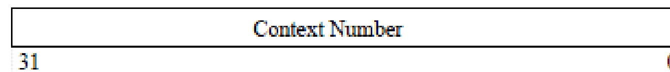
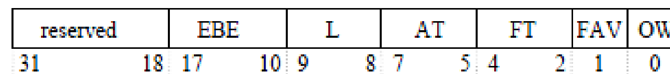


Fig. 2.23 **Registro de Estado de Fallo:** Contiene el estado de la MMU en un *trap* por fallo.



– EBE: sin usar.
– L: Nivel de la Tabla de página que causó el fallo.

L	Level
0	Entry in Context Table
1	Entry in Level-1 Page
2	Entry in Level-2 Page
3	Entry in Level-3 Page

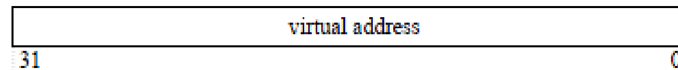
– AT: Define el tipo de acceso que causó el fallo.

AT	Access Type
0	Load from User Data Space
1	Load from Supervisor Data Space
2	Load/Execute from User Instruction Space
3	Load/Execute from Supervisor Instruction Space
4	Store to User Data Space
5	Store to Supervisor Data Space
6	Store to User Instruction Space
7	Store to Supervisor Instruction Space

– FT: Define el tipo de fallo.

FT	Fault type
0	None
1	Invalid address error
2	Protection error
3	Privilege violation error
4	Translation error
5	Access bus error
6	Internal error
7	Reserved

Fig. 2.24 **Registro de Dirección de Fallo:** Contiene la DV que provocó la interrupción por fallo.



La SRMMU puede retener traducciones del espacio de direcciones de varios procesos al mismo tiempo. Ésto se reduce a cambios de contexto entre procesos, actividades como el llenado de tablas de página, TLB, entre otras. Cada espacio de direcciones es identificado por bits de contexto que son utilizados para mantener diferenciadas las direcciones virtuales de los diferentes procesos en el *Page Descriptor Cache PDC* (TLB), además, estos bits de contexto son usados para indexar a la Tabla de contexto (*Context Table*) en la memoria principal para encontrar el apuntador base del directorio de página del proceso (*Root Pointer Page Table Descriptor*). La gestión y asignación de contextos es responsabilidad del software de gestión de memoria del SO.

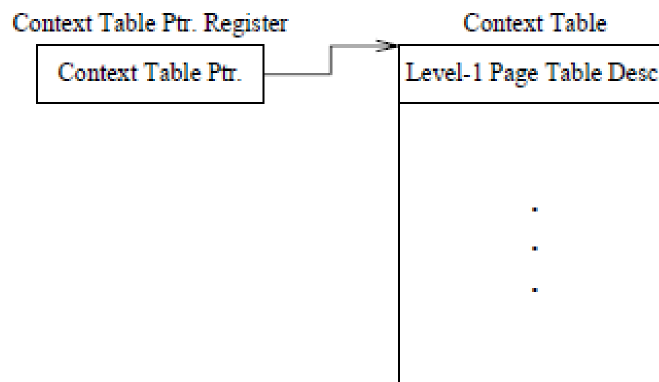


Fig. 2.25 Tabla de contexto (*Context Table*)

La figura 2.26 dá una descripción detallada del proceso de traducción y de las estructuras que están involucradas para dicho proceso. El primer nivel de la jerarquía de la tabla de páginas es el de Context Table (1) Está indexado por el Context Number (CTXNR) un registro que se inicializa con un número único que está asociado a cada proceso. En un cambio de proceso el registro debe ser actualizado. Los niveles 1-4 de la jerarquía de la tabla de página (2) son indexados por los diferentes segmentos de la DV. Si un Page Table Descriptor (PTD) es encontrado este apuntará a una Page Table Entry (PTE) para

completar la traducción. El PTD y PTE son diferenciados entre sí por el campo ET (3), donde ET=1 indica PTD and ET=2 indica PTE, ET=0 indica que no hay entrada (*page fault*). El nivel 1 puede mapear a 4GB, el nivel 2 a 16MB, el nivel 3 a 256K y el nivel 4 a 4K. El PTE contiene El número de la página física (*Physical Page Number*) y algunos campos adicionales (*flags*) que indican el tipo de acceso, protección y política de reemplazo. La dirección física (4) obtenida como resultado de la traducción de la MMU está formada por el número de página física y el desplazamiento, (los tamaños varían dependiendo de las tablas de página del nivel jerárquico).

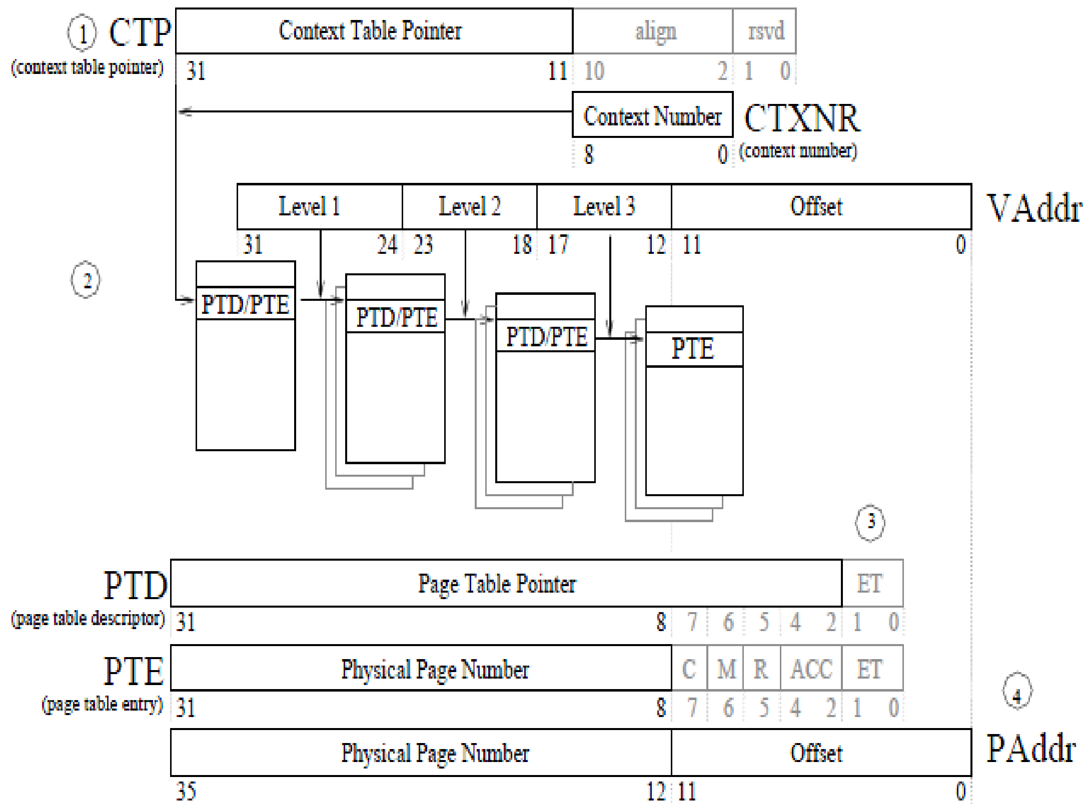


Fig. 2.26 Proceso de traducción de la SRMMU

2.6 Artículos

En esta sección del Estado del arte, se examinan algunas propuestas encontradas en la literatura reciente, donde se discuten proyectos en los que implementan técnicas de arquitectura de computadoras para mejorar el Sistema de memoria y la Unidad de gestión de memoria MMU.

2.6.1 Mejora del desempeño de Tablas de Página

Cuando ocurre un fallo en TLB, por lo menos dos accesos a memoria son necesarios para completar la traducción en la tabla de página. Sin embargo, los espacios de direcciones físicas y virtuales ha ido creciendo en tamaño, por lo cual la profundidad del árbol de tablas de páginas se ha ido incrementando, en el Pentium Pro necesita de tres niveles para una dirección de 32 bits y más recientemente de cuatro niveles en el AMD Opteron para direcciones virtuales de 48bits. De hecho, cada década desde la introducción del 80386, la profundidad del árbol de tablas de página ha ido incrementando un nivel.

La arquitectura estándar x86-64 usa un tamaño de página de 4KB, por lo cual son 12 bits para el offset de página. Los 36 bits restantes de la dirección virtual son divididos en cuatro grupos de 9 bits, los cuales son usados para direccionar una de las entradas de los niveles de las tablas de página. Los cuatro niveles son llamados PML4 (*Page Map Level 4*) (L4), PDP (*Page Directory Pointer*) (L3), PD (*Page Directory*) (L2) y PT (*Page Table*) (L1). Finalmente la dirección virtual de 48 bits usa extensión de signo (*sign extended*) hasta los 64bits.

63:48	47:39	38:30	29:21	20:12	11:0
se	L4 idx	L3 idx	L2 idx	L1 idx	*page offset*

Fig. 2.27 Segmentación de dirección virtual x86_64

En consecuencia si la dirección virtual crece también lo harán los niveles de traducción de la tabla de página y cada entrada de la tabla utiliza 8 bytes para su almacenamiento, en total 4KB por nivel. Pero el factor principal en el rendimiento es que cada uno de los 4 niveles necesita de un acceso a la memoria principal. Es donde el TLB ayuda, pero investigaciones muestran que el porcentaje de fallos en TLB en general ronda entre el 5-14% en aplicaciones de tamaño promedio [15] el cual aumenta en otras de mayor cantidad de transacciones de memoria afectando el rendimiento del sistema. El uso de páginas de mayor tamaño puede reducir este impacto, pero con aumentos de transacciones de memoria su eficacia disminuye.

En años recientes, los procesadores Intel y AMD de arquitectura x86_64 implementan técnicas para acelerar el proceso de traducción de direcciones virtuales a físicas de la MMU una de ellas son las caches de traducción (*translation caches*) [16] las cuales almacenan traducciones parciales y permiten saltarse uno o más niveles de las tablas de página. Las caches Intel y AMD tienen diferencias en diseño y operación pero con el objetivo de almacenar entradas de tablas de página de los niveles superiores del árbol.

El Amd Page Walk Cache [17] almacena entradas de las tablas de página de cualquier nivel del árbol, además de indexar cada una de estas entradas con la dirección física. Por otro lado Intel Paging-Structure Caches implementa diferentes caches para cada nivel del árbol de tablas de página y las entradas son indexadas por fragmentos de la dirección virtual a traducir.

Base Location	Index	Next Page
125	0ae	508
042	00c	125
613	0b9	042
...	...	...

Fig. 2.28 Ejemplo de contexto de UPTC de AMD.

El Amd Page Walk Cache usa una cache de tabla de página llamada Unified Page Table Cache UPTC que se almacena en la cache de datos L2. Cuando la MMU accesa a la tabla de página para la traducción de una dirección virtual (0b9, 00c, 0ae, 0c2, 016) y si posteriormente la MMU trata de traducir la dirección virtual (0b9, 00c, 0ae, 0c3, 103), la MMU comenzará la traducción buscando el 0B9 del nivel L4 (ubicado en la dirección 613 y que hace referencia en el registro CR3). Puesto que ésta entrada de la tabla página está presente en la UPTC, no necesita accesar hasta la tabla de página para obtener la entrada L3 que tiene la dirección física 042. El mismo proceso se repite entonces para localizar el L2 con L3 y L1 con L2 de las tablas de página. Una vez que la dirección de la página L1 se encuentra, en la entrada correspondiente se carga desde la memoria para determinar la dirección de la página física de los datos deseados. Sin esta cache de tabla de página, todos los cuatro niveles deben ser referenciados hasta la tabla de página en memoria principal.

Intel Paging-Structure Caches [18] hacen uso de Split Translation Cache STC que almacenan los diferentes niveles de entradas de las tablas de página en caches separadas, además de ser indexadas por fragmentos de la dirección virtual. Cuando la MMU accesa a la tabla de página para la traducción de una dirección virtual (0b9, 00c, 0ae, 0c2, 016) y posteriormente desea traducir la dirección virtual (0b9, 00c, 0dd, 0c3, 929) intentará encontrar L1, L2, L3 y L4 en sus caches correspondientes usando los fragmentos de la dirección virtual. La ubicación de L3 se almacena en la cache de entrada L4 e indexada por el índice de L4, (0B9). De manera similar, la ubicación de L2 se almacena en la cache L3 y etiquetados por la L4 y L3, los índices (0B9, 00c). Finalmente L1 se almacenan en la cache L2 y es indexada por L4, L3 y L2, (0B9, 00c, 0DD).

	L4 index	L3 index	L2 index	Next Page
L2 entries	0b9 ...	00c ...	0ae ...	508 ...
L3 entries	0b9 ...	00c ...		125 ...
L4 entries	0b9 ...			042 ...

Fig. 2.29 Ejemplo de contexto de STC de Intel.

Las caches de tablas en MMU se han convertido en componentes críticos en los procesadores x86 actuales y futuros. El uso de la cache L2 para almacenar estas caches de entradas de tablas de página, reducen significativamente los accesos a la DRAM.

2.6.2 Mejora en TLB

Para acelerar el proceso de traducción de direcciones virtuales a físicas, los procesadores actuales hacen uso de un Translation Look-aside Buffer (TLB) [19] el cual almacena las últimas traducciones realizadas para acceder al sistema de memoria. En el sistema de memoria se hace uso de la memoria cache para reducir el tiempo de acceso a la memoria. El cache es una memoria pequeña y rápida (SRAM), la cual almacena copias de datos ubicados en la memoria principal que se utilizan con más frecuencia. Con ellas se puede optimizar el desempeño de los principios de localidad espacial y temporal de Donald Knuth (1971): "menos del 4 por ciento de un programa, generalmente, representa más de la mitad de su tiempo de ejecución".

Las caches más usadas por efectividad y costo son las Physically indexed, physically tagged PIPT que solo son accesadas con los bits de la dirección física. La solución más simple es primero accesar al TLB y después a la cache (a). Entonces, cada acceso a datos e instrucciones lleva a un ciclo adicional y en los procesadores RISC se produce una mayor penalización en bifurcaciones mal previstas (*mispredicted branches*).

Para evitar estos retrasos el TLB y cache deben ser accesados en paralelo. Esto es posible mediante proveer con los VPN bits a la TLB para obtener la dirección física y los bits de desplazamiento a la cache para que pueda indexar a la localidad donde se encuentra el posible dato, al mismo tiempo o en paralelo (b). Posteriormente se comparan la dirección física con el *tag* de la localidad de la cache, de acertar se obtiene el dato o de lo contrario se produce un cache *miss*.

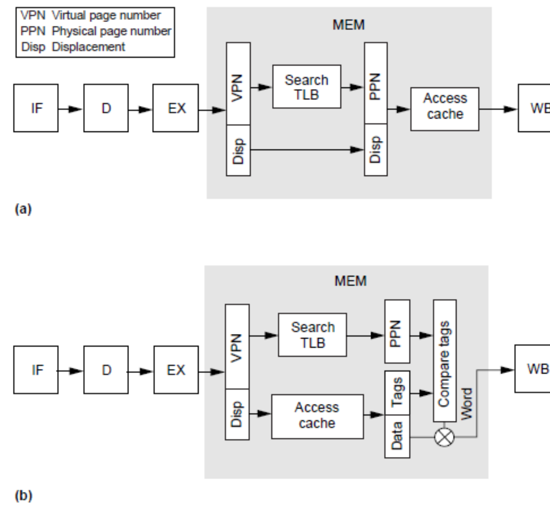


Fig. 2.30 (a)Acceso secuencial TLB/cache. (b) Acceso en paralelo a TLB/cache.

2.6.3 Mejora en jerarquía de memoria.

Los procesadores actuales logran un performance de hasta 100,000 MIPS. Estos procesadores fácilmente podrían perder la mitad o más de su desempeño en la jerarquía de memoria si el diseño de la jerarquía se basa en las técnicas convencionales de almacenamiento en cache.

Diversas técnicas hardware se han desarrollado para mejorar el rendimiento del sistema de cache [20] una de ellas es la implementación de una Cache víctima (*victim cache*) su objetivo es, el de reducir la penalización de tasa de fallos. Es un pequeño buffer asociativo de alta velocidad que almacena bloques recientemente eliminados del cache superior por lo general L1, una cache víctima de 4 entradas elimina entre el 20%-95% de fallos por conflicto en cache de mapeo directo manteniendo el acceso rápido. Reduce la tasa de fallos ya que se requiere solo de un ciclo extra para obtener el dato de esta pequeña cache en lugar de accesar al siguiente nivel de la jerarquía de cache L2 que tomaría de 7 a 16 ciclos obtener el dato.

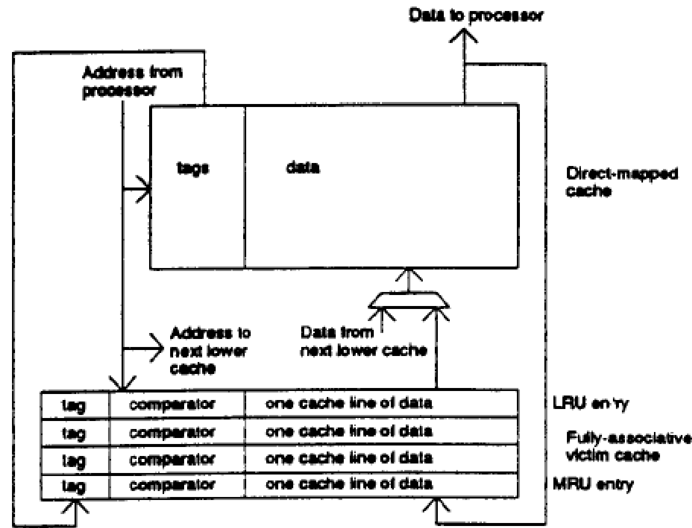


Fig. 2.31 Diagrama de operación de cache con la Cache Víctima

Con acierto en L1 no se necesita accesar a la cache víctima. Un fallo en L1 por bloque en ubicación b, acierto en cache víctima en ubicación v, intercambia contenido de b y v, esto toma un ciclo extra. Fallo en L1, falló en cache víctima, se carga bloque del siguiente nivel de la jerarquía y se escribe en L1 y escribir el bloque reemplazado en L1 en cache víctima, si la cache víctima está llena, reemplazar una de las entradas.

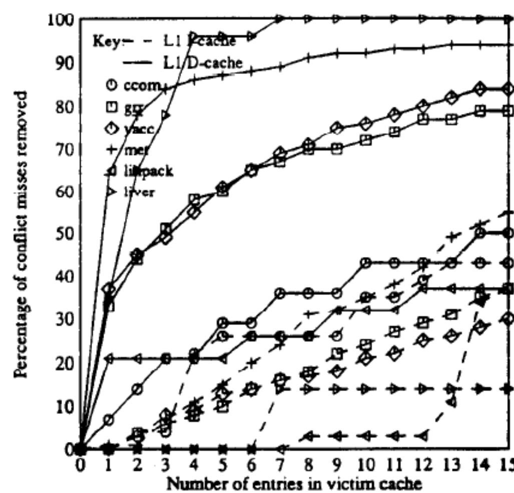


Fig. 2.32 Fallos por conflicto removidos por la cache víctima.

El desempeño de una cache víctima depende del tamaño y número de líneas de la cache L1. En general dos factores principales deben ser tomados en cuenta para un buen desempeño del sistema, el primero es que al incrementar el número de líneas en L1, la cache víctima debe reducir su tamaño, el segundo es que al incrementar el tamaño de línea en la cache L1 se incrementa el número de fallos por conflicto que puede eliminar la cache víctima.

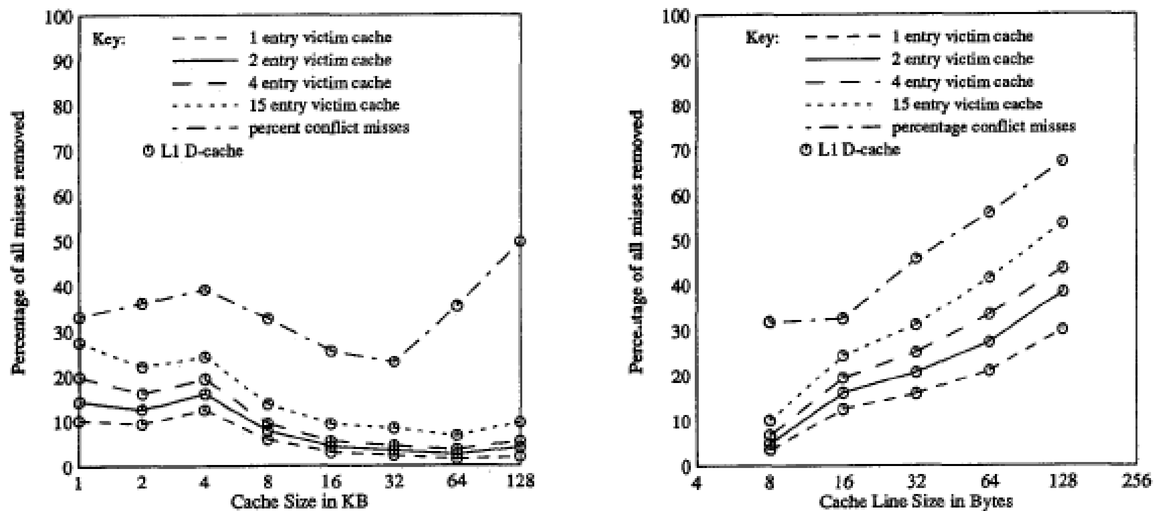


Fig. 2.33 Desempeño de cache víctima con variaciones de cache L1.

2.7 Resumen del capítulo

En este segundo capítulo se expone lo distintivo de los diferentes diseños de arquitecturas MMU que presentan algunos de los procesadores más populares, así como también las estructuras funcionales (diagramas de bloques) implementadas, sus características y operación para la gestión de la MV y accesos a memoria. Se analizan algunos artículos relevantes para mejorar el diseño del sistema de gestión de memoria y la MMU. Principalmente se explica de manera detallada la estructura, operación y algunos conceptos importantes de la SRMMU del procesador SPARCv8 que es la arquitectura base para el desarrollo de este trabajo de Tesis.

CAPÍTULO 3.

MARCO TEÓRICO

Este capítulo presenta dos partes, la primera consta de la sección 3.1 contiene información característica de la arquitectura de los procesadores superescalares y de la sección 3.2 sobre la función, especificaciones de elementos y caracterización de la Unidad de gestión de memoria MMU que implementan estos procesadores, la cual centra el estudio de esta Tesis. En la segunda parte, se describen las características de los elementos usados en este proyecto FPGAs, *System on Chip* (SoC), entorno software para SoC, administración de memoria del SO Linux, software para la simulación y evaluación.

3.1 Procesadores Superescalares

3.2 Unidad de gestión de memoria MMU

3.3 Elementos del proyecto

- *3.3.1 FPGA y Tarjeta de desarrollo*
- *3.3.2 System on Chip (SoC)*
- *3.3.3 Entorno Software para SoC*
- *3.3.4 Administración de memoria del SO Linux*
- *3.3.5 Software para la simulación y evaluación*

3.4 Resumen del Capítulo

CAPÍTULO 3 Marco teórico

3.1 Procesadores Superescalares

Los microprocesadores actuales son producto de la evolución de distintas arquitecturas y diseños predecesores, un gran avance en lo que a arquitectura de computadoras fue en sus comienzos, el de obtener un IPC igual a uno; es decir, que se ejecuta sólo una instrucción en cada ciclo: esta es la microarquitectura del procesador escalar.

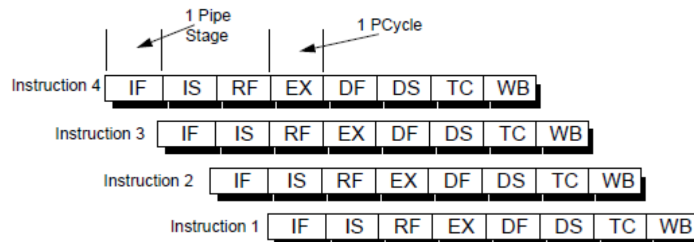


Fig. 3.1 Pipeline de una arquitectura escalar.

Con investigaciones posteriores en 1990 se desarrolló la microarquitectura superescalar [21] que efectúa el paralelismo de instrucciones además del paralelismo de flujo, esto, gracias a su estructura en pipeline que consta con las siguientes etapas: Lectura (*fetch*), Decodificación (*decode*), Lanzamiento (*dispatch*), Ejecución (*execute*), Escritura (*writeback*), Finalización (*commit*). El paralelismo de la máquina no se logra simplemente replicando varias veces cada etapa del cauce. El procesador debe ser capaz de analizar el flujo de instrucciones del programa e identificar el paralelismo a nivel de instrucción. También tiene que organizar la captación, decodificación y ejecución de las instrucciones, como el proceso de iniciar la ejecución de la ejecución de instrucciones en las unidades funcionales (emisión de instrucciones)

Para incrementar el Paralelismo a Nivel de Instrucción (ILP, por sus siglas en inglés) que mejore el desempeño de los procesadores superescalares es por medio de una

ejecución fuera de orden (*out-of-order*) que elimine paros por dependencias e incremente la velocidad de ejecución, para llevarse a cabo se necesita de técnicas hardware de renombrado de registros, predictor de saltos y ejecución especulativa.

Un procesador superescalar maneja más de una instrucción en cada etapa *n-ways*. El número máximo de ejecución de instrucciones en cada etapa depende del número y del tipo de las unidades funcionales independientes de los tipos enteros y flotantes como Unidad aritmético lógica (ALU) Unidad de lectura/escritura en memoria (Load/Store Unit) Unidad de coma flotante (Floating Point Unit) y Unidad de salto (Branch Unit).

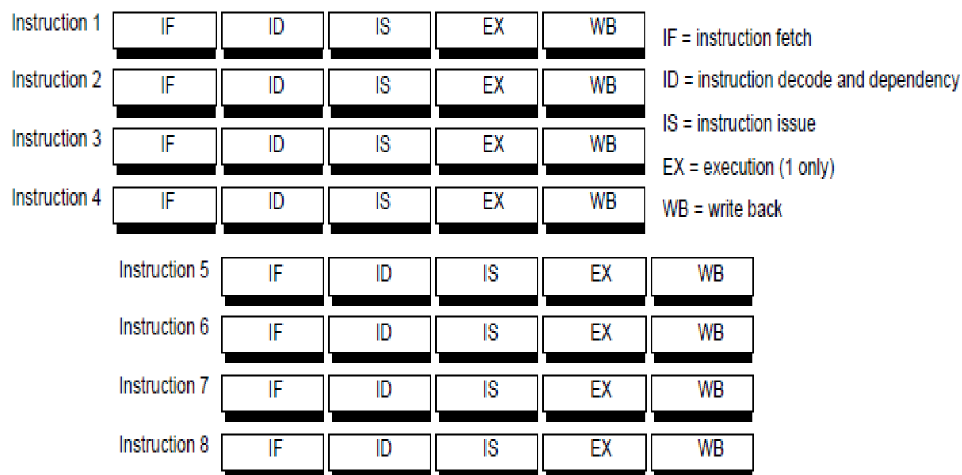


Fig. 3.2 Pipeline de un procesador superescalar de 4-*way*.

El hecho de permitir la emisión desordenada de instrucciones y la finalización desordenada puede originar dependencias de salida y anti-dependencias. La naturaleza de estas dependencias es diferente a la de las dependencias de datos verdaderas, que reflejan el flujo de datos a través de un programa y su secuencia de ejecución. Las dependencias de salida y las anti-dependencias, por otra parte, no son verdaderas dependencias, surgen porque los valores de los registros no pueden reflejar ya la secuencia de valores establecida por el flujo del programa.

Las anti-dependencias y las dependencias de salida son realmente conflictos de almacenamiento. Son un tipo de conflicto por los recursos en el que varias instrucciones compiten por los mismos registros. En cuyo caso el procesador debe resolver el conflicto deteniendo temporalmente alguna etapa del cauce. Se puede comprender entonces que la frecuencia de aparición de este tipo de instrucciones aumenta con el uso de las técnicas de optimización de registros, que intentan maximizar el uso de los registros.

Como se mencionó, un método para resolver tales conflictos es usar la técnica de renombramiento de registros. Consiste en que el hardware del procesador asigne dinámicamente los registros, que están asociados con los valores que necesitan las instrucciones en diversos instantes de tiempo. Cuando se ejecuta una instrucción, donde su operando destino es un registro, se le asigna un nuevo registro físico para almacenar el resultado, y las instrucciones posteriores que accedan a ese valor como operando fuente en ese registro, tienen que atravesar un proceso de renombramiento, donde se revisan las referencias a registros, para que definitivamente hagan referencia al registro físico que contiene el valor que se necesita. De este modo, diferentes instrucciones que tienen referencias a un único registro de la arquitectura (registro lógico), pueden referirse a diferentes registros reales (registros físicos), con valores diferentes. Por ejemplo:

```
I1: r3 ← r3 op r5
I2: r4 ← r3 + 1
I3: r3 ← r5 + 1
I4: r7 ← r3 op r4
```

La anterior secuencia de instrucciones, tiene varias dependencias [22], incluidas una dependencia de salida y una anti-dependencia. Si aplicamos el renombramiento de registros a dicho código tendremos:

```
I1: r3b ← r3a op r5a
I2: r4b ← r3b + 1
I3: r3c ← r5a + 1
I4: r7b ← r3c op r4b
```

Las referencias a un registro sin la letra del subíndice hacen referencia a un registro lógico, un registro de la arquitectura. Las referencias a un registro con la letra del subíndice hacen referencia a un registro físico, un registro hardware.

Nótese que en el ejemplo la creación del registro `r3b` en la instrucción `I3` evita la antidependencia (entre `I2` e `I3`) y la dependencia de salida (entre `I1` e `I3`). El resultado es que utilizando el renombramiento de registros `I1` e `I3` pueden ejecutarse en paralelo.

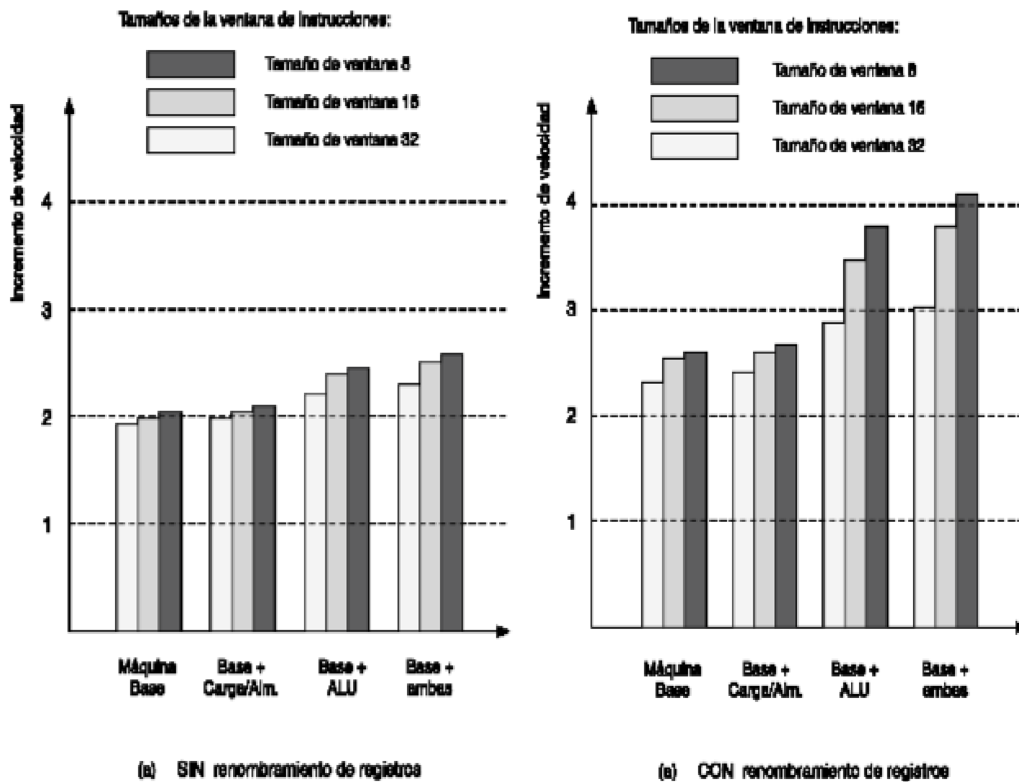


Fig. 3.3 Benchmark enfocado al renombrado de registros.

Esta arquitectura implementa un *set* de instrucciones RISC (*Reduced Instruction Set Computer*) con la característica de ser instrucciones de tamaño fijo y presentadas en un reducido número de formatos y sólo las instrucciones de carga y almacenamiento acceden a la memoria de datos.

Formatos de instrucción simples y uniformes que facilitan la extracción/decodificación de los diversos campos y permiten traslape máquina entre interpretación *opcode* y lectura de registro, cuya meta es ejecutar las operaciones de uso más común tan rápido como sea posible [23].

Fig. 3.4 Formatos de instrucciones tipo RISC del procesador MIPS.

Esta mejora en la arquitectura *load/store* confina los cálculos de dirección de memoria y demoras de acceso a un pequeño conjunto de instrucciones para lectura y escritura y todas las otras obtienen sus operandos de registro más rápidos y compactamente direccionables.

Un chip RISC típicamente tendrá menos transistores dedicados a la lógica principal. Esto permite a los diseñadores una flexibilidad considerable; así pueden, por ejemplo: incrementar el tamaño del conjunto de registros, tener mayor velocidad en la ejecución de instrucciones, implementar medidas para aumentar el paralelismo interno, añadir caches enormes, bajo consumo de energía o de tamaño limitado.

Hoy en día se implementa en la mayoría de los procesadores modernos en la arquitectura x86 las instrucciones para el usuario son CISC (Complex instruction set computing) las cuales se traducen a instrucciones más simples basadas en RISC en hardware, ya que una instrucción RISC corresponde a una microinstrucción, que se implementa de una secuencia de pasos en la ejecución de instrucciones más complejas.

3.2 Unidad de gestión de memoria MMU

La gestión de memoria es una de las partes más importantes del SO [24]. Desde los tiempos de los primeros ordenadores, existió la necesidad de disponer de más memoria de la que físicamente existía en el sistema. Entre las diversas estrategias desarrolladas para resolver este problema, la de mayor éxito ha sido la MV. La MV hace que el sistema parezca disponer de más memoria de la que realmente tiene, compartiéndola entre los distintos procesos conforme la necesitan.

La Unidad de Gestión de Memoria o unidad de manejo de memoria (MMU) es un dispositivo de Hardware formado por un grupo de bloques funcionales, responsable de la gestión de MV para un mayor espacio de memoria mediante la traducción de las direcciones virtuales a direcciones físicas, permite el multiprocesamiento en la memoria física del sistema, además de la protección de la memoria para dar a cada proceso un espacio de dirección virtual y su protección contra escritura. Junto con el SO permite el mapeo de imágenes y archivos dentro de un espacio de memoria, además efectúan la *shared virtual memory* que permite a los procesos compartir espacios de memoria.

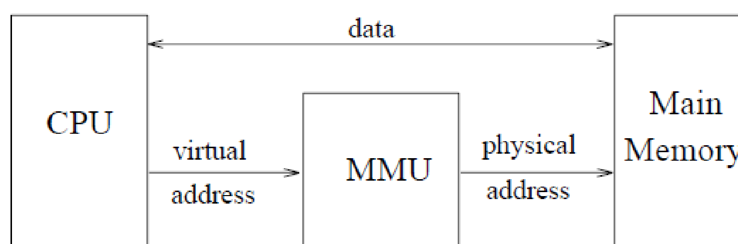


Fig. 3.5 Diagrama de bloques de un sistema con MMU.

Los procesos tienen código que sólo se usa en ciertas ocasiones, como en la inicialización o para procesar un evento particular, además no se hace uso del 100% de los datos contenidos dentro de un período de tiempo determinado. Sería superfluo cargar todo su código y datos en la memoria física donde podría terminar sin usarse. El sistema

no funcionaría eficientemente si multiplicamos ese gasto de memoria por el número de procesos en el sistema. Para solventar el problema, el SO usa una técnica llamada Paginación por demanda (*demand paging*) que sólo copia un marco de página de memoria de un proceso en la memoria física del sistema cuando el proceso trata de usarla. De esta manera, en vez de cargar el código y los datos en la memoria física de inmediato, el núcleo del SO gestiona las Tablas de página del proceso y designa las áreas virtuales como existentes, pero no en memoria y se traen los marcos de página de la memoria secundaria a la memoria principal conforme son solicitados.

Entonces, la traducción de la memoria virtual como la física están divididas en trozos de un tamaño manejable llamados marcos de páginas, estos son del mismo tamaño, en principio no necesitarían serlo pero de no ser así, la administración del sistema se complicaría. En un sistema operativo como Linux en un sistema Alpha AXP utiliza marcos de página de 8KB, y en un sistema Intel x86 utiliza marcos de página de 4 KB.

Un sistema actual puede trabajar en modo real o con memoria virtual, cuando es el segundo caso la CPU genera una dirección de memoria virtual, que puede ser traducida a una dirección física mediante la MMU realiza una búsqueda en la TLB o en las Tablas de marcos de página que son mantenidas por el Sistema Operativo.

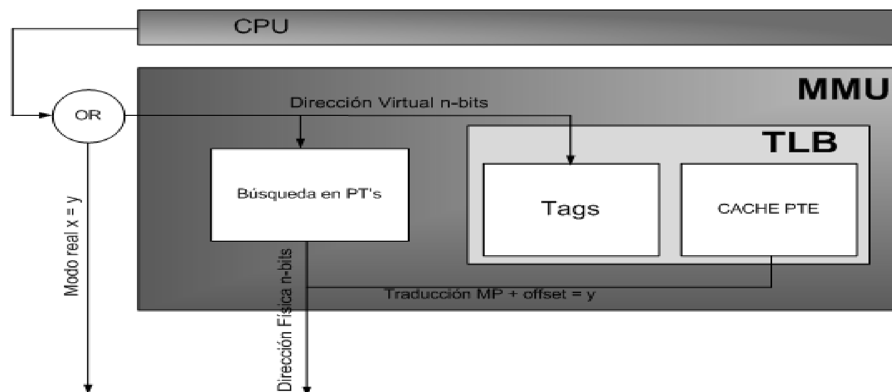


Fig. 3.6 Diagrama de bloques de la operación de MMU.

El TLB (Translation Lookaside Buffer) es una pequeña cache de alta velocidad por lo general *n-way* o totalmente asociativa que mantiene entradas de las traducciones usadas recientemente. Cuando la dirección requerida por la CPU se encuentra en el TLB y los *tags* de acceso son permitidos, su traducción a dirección física es entregada, en lo que se conoce como acierto en el TLB (TLB *hit*). En otro caso, cuando la dirección buscada no se encuentra en el TLB (TLB *miss*) genera una interrupción al SO para traer la entrada de la traducción que generó el fallo en alguna de sus entradas (TLB *entries*) mediante un algoritmo de reemplazo o selección.

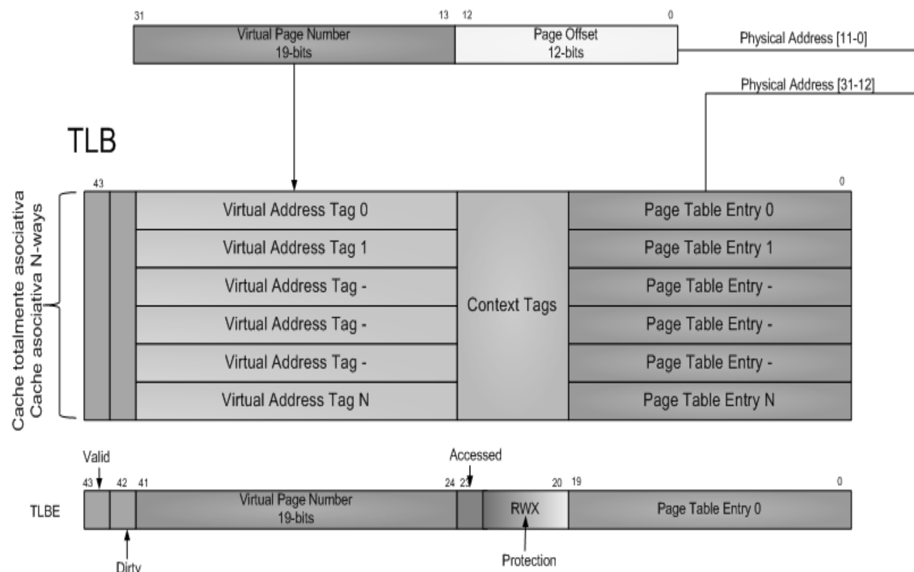


Fig. 3.7 Traducción de dirección virtual a dirección física mediante el TLB.

De no encontrarse, no activar o no contar con el TLB, la MMU realiza la búsqueda en las Tablas de página de la DV hasta encontrar la entrada que contiene la traducción o se produce un error de fallo de página que genera una interrupción al SO para que traiga la página a la memoria principal y actualice la entrada en la Tabla de páginas. La forma de obtener la dirección física en las tablas es mediante la segmentación en grupos bits de la dirección virtual con los cuales se realiza el desplazamiento dentro del directorio y el desplazamiento dentro de las tablas. Los bits menos significativos de la dirección virtual

representan el offset de la página, los bits más significativos representan el número de página virtual. De esta forma el sistema de tablas para 32bits representa 4GB de direcciones de memoria virtual con páginas de 4KB y 4MB.

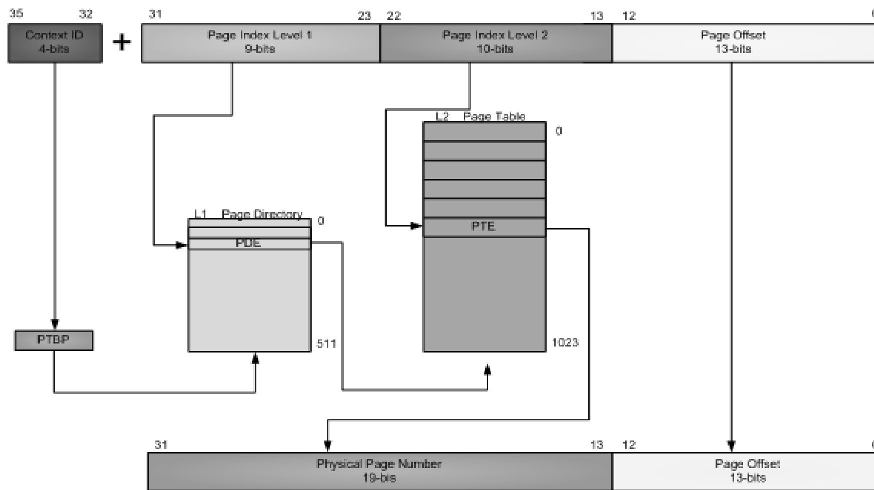


Fig. 3.8 Traducción de DV a DF mediante Tablas de marco de página.

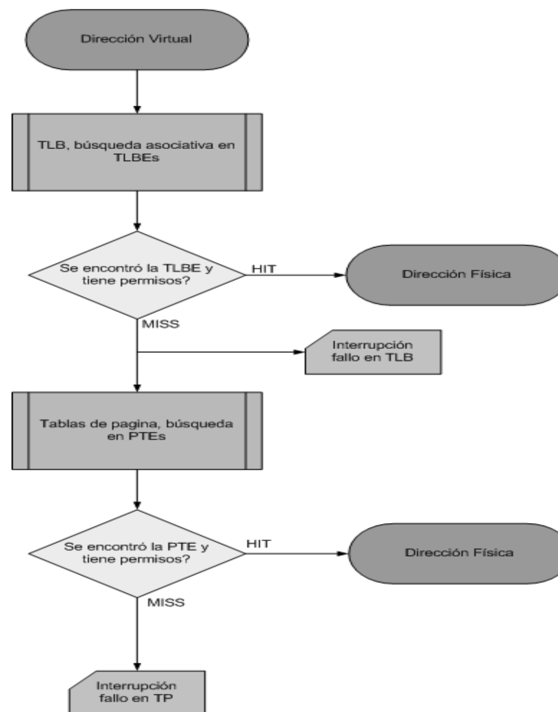


Fig. 3.9 Diagrama de flujo de la operación de MMU.

3.3 Elementos del proyecto

3.3.1 FPGA y Tarjeta de desarrollo

Un FPGA (*Field Programmable Gate Array*) es un dispositivo semiconductor que contiene bloques de lógica cuya interconexión y funcionalidad se puede programar. La lógica programable puede reproducir desde funciones tan sencillas como las llevadas a cabo por una puerta lógica o un sistema combinacional y hoy en día con la evolución de la integración se implementan hasta complejos sistemas en un chip SoPC, (*System on Programmable Chip*), sistemas embebidos, aplicaciones de procesamiento digital de señales, entre otros.

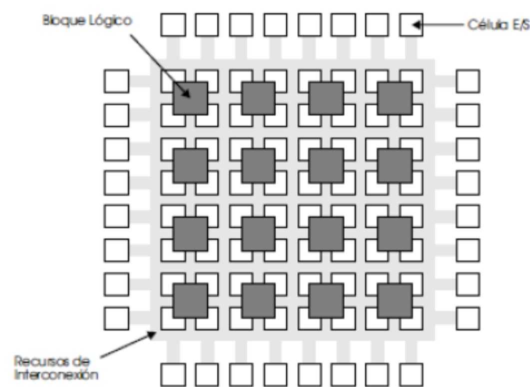


Fig. 3.10 Estructura interna de una FPGA.

Los FPGAs contienen componentes lógicos programables llamados elementos lógicos (LE) y una jerarquía de interconexiones reconfigurables que permiten a los LEs conectarse físicamente. Con esto configurar los LEs para realizar funciones complejas o puertas lógicas meramente sencillas como AND y XOR. En los FPGAs, los bloques lógicos incluyen elementos de memoria, que pueden ser simples flipflops o más bloques completos de memoria. El FPGA usado en esta Tesis es el Altera Cyclone IV de 115K LE de la tarjeta de desarrollo Terasic DE2-115, cada uno de ellos se compone principalmente de una LUT(*lookup table*) de 4 entradas, un mux y un flipflop.

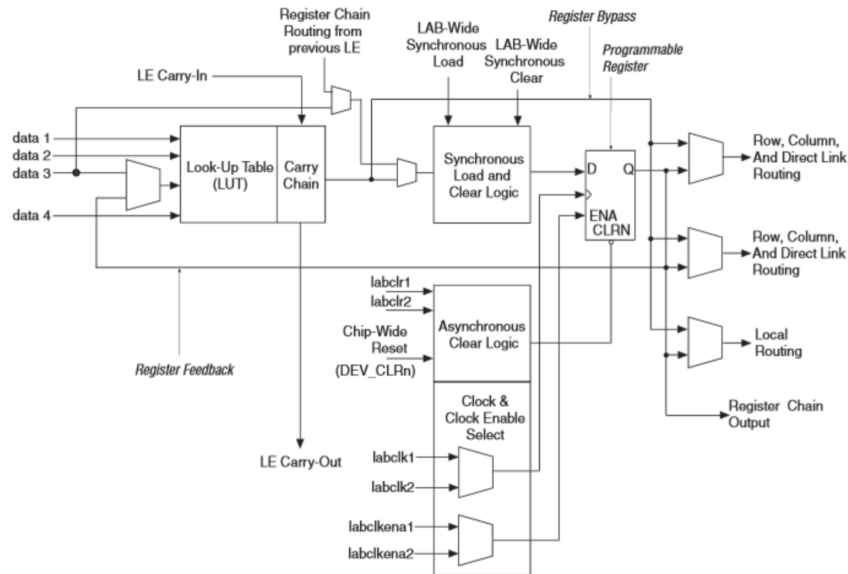


Fig. 3.11 Elemento lógico del FPGA Cyclone IV.

A diferencia de generaciones anteriores de FPGAs usando I/Os con lógica programable e interconexiones, los FPGAs de hoy incorporan SRAM, PLLs, multiplicadores, I/O de alta velocidad, bloques lógicos dedicados y otros más.

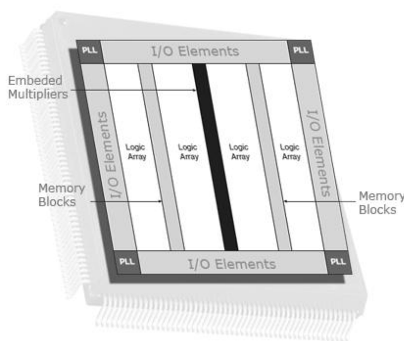


Fig. 3.12 Diagrama de recursos del FPGA Cyclone IV.

Los recursos principales del FPGA Cyclone IV EP4CE115F29 son: 114,480 LEs, 432 M9K bloques de memoria que es igual a 3,888 Kbits de memoria embebida y 4 PLLs.

Los componentes de la tarjeta DE2-115 [25] que son utilizados en esta tesis son: el JTAG para la configuración y depuración del FPGA, memoria SDRAM 128MB (32Mx32bit), *push- buttons*, indicadores de estado LED, el reloj 50MHz y uso de PLL’s para otras frecuencias.

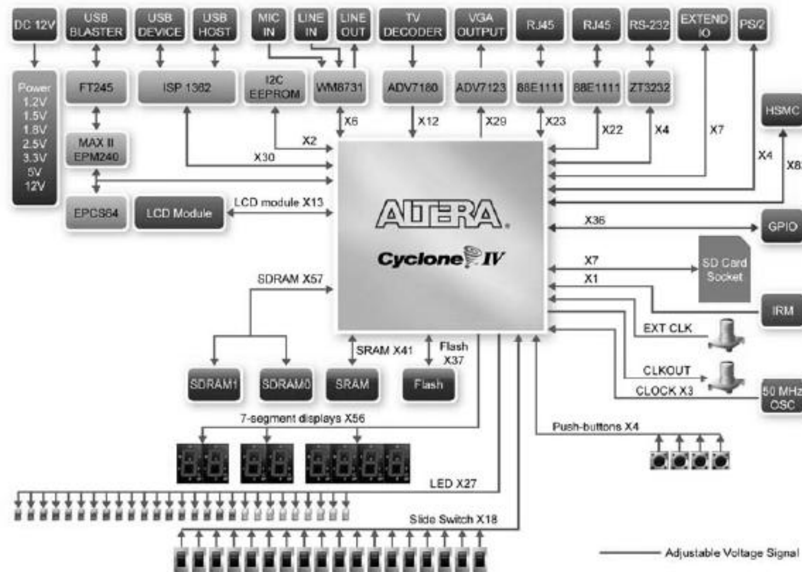


Fig. 3.13 Diagrama de componentes de la DE2-115.

Entonces, con el diseño HDL y usando una herramienta EDA (*Electronic Design Automation*), se genera un *netlist* de la tecnología mapeada. Mediante el uso de software para el *place&route* se corresponde el *netlist* generado con la arquitectura FPGA usada. Posteriormente se valida el *layout* resultado del *place&route*, mediante la simulación de funcionalidad *testbench* para estimular el sistema y observar los resultados, además de simulaciones como *timing analysis*, *power analysis* y otras metodologías de verificación. Una vez completados los procesos de diseño y validación, se genera el fichero binario usado para programar la FPGA con una herramienta existente en el Quartus II (Programmer), mediante el conector hardware (JTAG de Altera).

3.3.2 System on Chip (SoC)

En las últimas décadas hemos asistido a una revolución en el mundo de la informática debida al auge que ha cobrado el uso de software libre. Desde sus comienzos hasta hoy, se han producido profundas transformaciones en el modo de desarrollar software y en los modelos de negocio, moviendo no sólo al mundo de la informática si no a la industria en general. Otro ejemplo muy significativo y relacionado con el mundo del software es el mundo del hardware.

Quizás, debido a su carácter tangible y a su mayor costo de producción y replicación, esta revolución ha tardado algunos años más en extenderse a este mundo. Por otro lado, desde hace algunos años con la popularización de los dispositivos programables FPGA a la par que aumentaron sus prestaciones, se han gestado multitud de comunidades dedicadas a la generación de módulos funcionales bajo licencias libres generalmente aceptando el modelo de licencia GPL/LGPL (*Lesser General Public License*).

De estas comunidades de usuarios, han surgido iniciativas empresariales como Gaisler Research o Beyond Semiconductor, y en sentido inverso, empresas establecidas en el antiguo modelo han liberado sus desarrollos, en torno a los cuales se han creado comunidades abiertas. Ejemplos recientes de estos últimos encontramos empresas como Sun Microsystems con su serie de procesadores UltraSparc.

Paralelamente al movimiento del hardware libre y como factor catalizador de éste, el diseño electrónico se ha hecho accesible a nuevos sectores. El desarrollo de chips programables de altas prestaciones y gran nivel de integración ha permitido que en una pastilla de silicio se pueda desarrollar un sistema casi completo, actualmente es llamado *System on Chip* o por sus iniciales SoC [26]. Los sistemas embebidos SoC se les integra a modo de bloques constructivos o módulos, los cuales permiten coexistir con otras metodologías subyacentes, para así aprovechar el *know-how* de los equipos de diseño a la

par que da como resultado sistemas más rápidos, energéticamente eficientes y con menor ocupación de área. Estos módulos reutilizables se denominan IPs (*Intellectual Propierty*). Todas estas ventajas han hecho que la metodología SoC sea hoy en día la más aceptada dentro del desarrollo de sistemas embebidos. Los *IPcores* pueden ser muy complejos, como un microprocesador *Soft-Core*, o muy simples, como en el caso de un *IPcore* para controlar un GPIO.

El término embebido se refiere al hecho de que la microcomputadora es encapsulada en un solo circuito integrado, estos sistemas embebidos en sistemas electrónicos son usados para aplicaciones específicas y se les conoce como sistemas embebidos los cuales están compuestos por microprocesador, memoria, entradas y salidas a periféricos y un programa de aplicación con o sin necesidad de un SO.

Las características de los SoC son:

- Dimensión reducida: más componentes en el mismo dado de silicio.
- Mejor rendimiento: las comunicaciones entre los componentes del mismo dado de silicio son más rápidas que comunicaciones entre elementos de diferentes chips.
- Reducción del consumo: puesto que se reduce la utilización de los *pads* E/S y la longitud de las líneas de transmisión. Porque los *pads* consumen mucha energía.
- Reducción de los costos: puede ser reconfigurado para futuras aplicaciones.

Los *Soft-Core* que son microprocesadores completamente descritos en un lenguaje de descripción de hardware como VHDL o Verilog, y orientados a ser integrados dentro de un diseño SoC sintetizado en un FPGA. Actualmente podemos encontrar multitud de opciones en lo que a *softcore* se refiere, tanto libres como no libres. Dentro de las alternativas libres podemos optar por LEON, OpenRISC, OpenSPARC, UltraSparc, entre otras. De las opciones no libres, que normalmente requieren la adquisición de licencias e incluso pueden vetar su implementación fuera de productos ajenos a la marca que los

distribuye, las opciones más populares son MicroBlaze y PowerPC de Xilinx y el NIOS y MP32 (MIPS) de Altera. Estos procesadores son altamente integrables en un SoC.

Para el desarrollo de esta Tesis se seleccionó el *softcore* Leon3 de 32 bits de alto performance, basado en la arquitectura y conjunto de instrucciones SPARCv8, principalmente porque es de código abierto ya que tiene un modelo de licencia LGPL/GPL e implementa la SRMMU con la cual puede gestionar la MV del SOLinux SPARC. Los elementos principales del Leon3 que lo hacen nuestra mejor opción son los siguientes:

- Arquitectura SPARC v8
- SPARC MMU-Reference (SRMMU)
- Modelo de memoria (caches configurables, asociatividad, políticas de reemplazo)
- SPARC v8 ISA (Ensamblador)
- Librería GRLIB (IP's cores periféricos) VHDL
- Bus Amba2.0
- *Toolchain (Crosscompiler)*
- SOLinux

A continuación se exponen las características destacadas de la SRMMU del Leon3 comparándolo con el OpenRISC1200:

Tabla 3.1 Tabla comparativa entre Leon3 SPARC-MMU y OpenRisc 1200 MMU

LEON3 SPARC-MMU	OpenRisc1200 MMU
-TLB configuración entradas	-TLB configuración entradas
-TLB unificada, Harvard	-TLB Harvard
-Tamaño de pagina 4K, 256K, (diferentes niveles de traducción)	-Tamaño de pagina 8K,
-Politica de reemplazo LRU, Aleatoria	-Sin política de reemplazo (mapeo directo)

Por lo anterior el Leon3 nos permite obtener una MMU más completa y una plataforma de evaluación (*softcore*-MMU + SoC + SO) para el desarrollo de esta Tesis.

Los SoC están principalmente compuestos por un microprocesador digital, en torno al cual se configura el resto del sistema atendiendo a las características concretas de éste. La elección del Leon3, su configuración y aplicación determina en gran medida la arquitectura del resto del sistema y por tanto la metodología de diseño a emplear. Además dentro del diseño del SoC, el bus juega un papel fundamental el modo en que vamos a interconectar los distintos componentes. Dentro de la arquitectura de interconexión, podemos diferenciar dos conceptos: la topología y el protocolo lógico. La topología se refiere a la forma, física o lógica, de los caminos de datos entre los distintos componentes. Por otro lado, el protocolo lógico da las reglas de comunicación para que el sistema funcione a través del bus físico. Las arquitecturas más utilizadas en el mercado actualmente son: ARM AMBA, IBM CoreConnect, Altera Avalon y Wishbone. En esta tesis hacemos uso del bus estándar AMBA 2.0 cuya elección viene determinada por el Leon3 y los periféricos, ya que han de ser compatibles y fundamentalmente porque están libres de patentes, royalties y copyright.

Algunos de los diferentes IPcores que podemos implementar en el SoC son el *softcore* Leon3-MMU, Bus AMBA 2.0, Advanced High-performance Bus (AHB), Advanced Peripheral Bus (APB) Bridge, UART, GPIO, USB, Ethernet, entre otros. Los diversos componentes disponibles para el sistema se agrupan por funcionalidad y por velocidad.

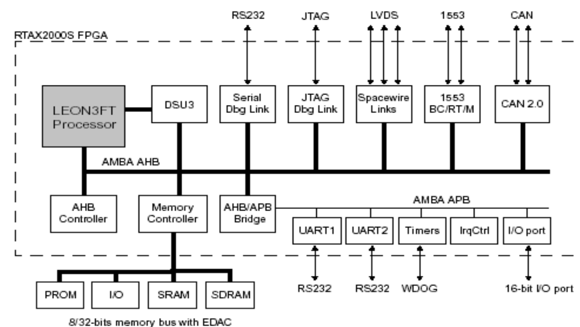


Fig. 3.14 SoC basado en el procesador LEON3

La motivación y objetivos de esta tesis es el de diseñar y evaluar una Unidad de Gestión de Memoria (MMU) de un procesador superescalar, utilizando lenguajes de descripción de hardware (HDL), con el objetivo de ser implementada en dispositivos lógicos programables (FPGA) con ayuda de un *softcore* probar su caracterización y eficiencia.

3.3.3 Entorno Software para SoC

Para compilar nuestro software disponemos del conjunto de herramientas de GNU, llamado *toolchain*. La GNU *toolchain* ha sido portada al *sofcore* LEON [27] para permitir el desarrollo de aplicaciones y de SO.

El GNU *toolchain* es un término que agrupa a una serie de proyectos que contienen las herramientas de programación producidas por el proyecto GNU. Estos proyectos forman un sistema integrado para lograr programar en distintos lenguajes, crear y configurar SO para embebidos y arquitecturas. Es instalado y usado en un SOLinux *host* con paquetes y librerías necesarias, además puede ser instalado en Microsoft Windows con Cygwin.

Aeroflex Gaisler proporciona y mantiene un conjunto de herramientas de compilación cruzada (*toolchain*) para el *softcore* Leon3. Una de ellas es BCC (Bare-C Cross Compilation System) [28] para la compilación de aplicaciones estáticas C/C++ que corren sobre hardware sin un SO enfocado a los LEON-SoC. Las herramientas incluidas son:

- GNU binutils: una colección de herramientas binarias (assembler, linker ...).
- GNU GCC 4.4.2: colección de la compilación que incluye el ANSI C, C++.
- GNU Make: automatización de la estructura y de la compilación.
- GNU GDB: depurador interactivo.
- uClibc y NewlibC: librerías destinadas al uso en sistemas embebidos.

Otra de las herramientas usadas es linuxbuild [29] que es un paquete que incluye Makefiles *scripts* para la creación y configuración del SO-Linux, los componentes necesarios son:

- SO Linux 2.6 para el LEON-SoC.
- mklinuximg: crea la imagen RAM de Linux para ser ejecutada en la memoria principal del LEON-SoC.
- U-Boot: *bootloader* para el SO.
- Buildroot: distribución que incluye drivers para el LEON-SoC.
- BusyBox 1.7.5: Es un paquete que combina todos los comandos comunes de Linux en un solo ejecutable.

Para comunicarnos con el LEON-SoC, hacemos uso de la herramienta GRMON que se comunica con la Unidad de depuración LEON (*Debug System Unit* DSU) que permite la depuración del sistema destino, acceso de lectura/escritura a los registros y localidades de memoria, además permite la descarga y ejecución de software y SO a la memoria del sistema. GRMON se instala en nuestro *host* y la interfaz de comunicación usada es el usb-blaster a la Tarjeta DE2-115.

Por otra parte, otra forma de obtener nuestro entorno de desarrollo, las empresas desarrolladoras de *softcores*, para mejorar el uso del compilador para la arquitectura, brindan imágenes Linux para máquinas virtuales como VMware Player las cuales contienen todo el entorno de desarrollo instalado para ser más eficiente el proceso.

3.3.4 Administración de memoria del SO-Linux

El *kernel* de Linux gestiona todas las áreas de memoria virtual y el contenido de la memoria virtual de cada proceso se describe mediante una estructura mm_struct a la cual se apunta desde la estructura task_struct del proceso. La estructura mm_struct del proceso también contiene información sobre la imagen ejecutable cargada y un puntero a las tablas

de páginas del proceso. Contiene punteros a una lista de estructuras vm_area_struct, cada una de las cuales representa un área de memoria virtual dentro del proceso. Esta lista enlazada está organizada en orden ascendente.

Como estas áreas de memoria virtual vienen de varias fuentes, Linux introduce un nivel de abstracción en la interfaz haciendo que la estructura vm_area_struct apunte a un grupo de rutinas de manejo de memoria virtual (via vm_ops).

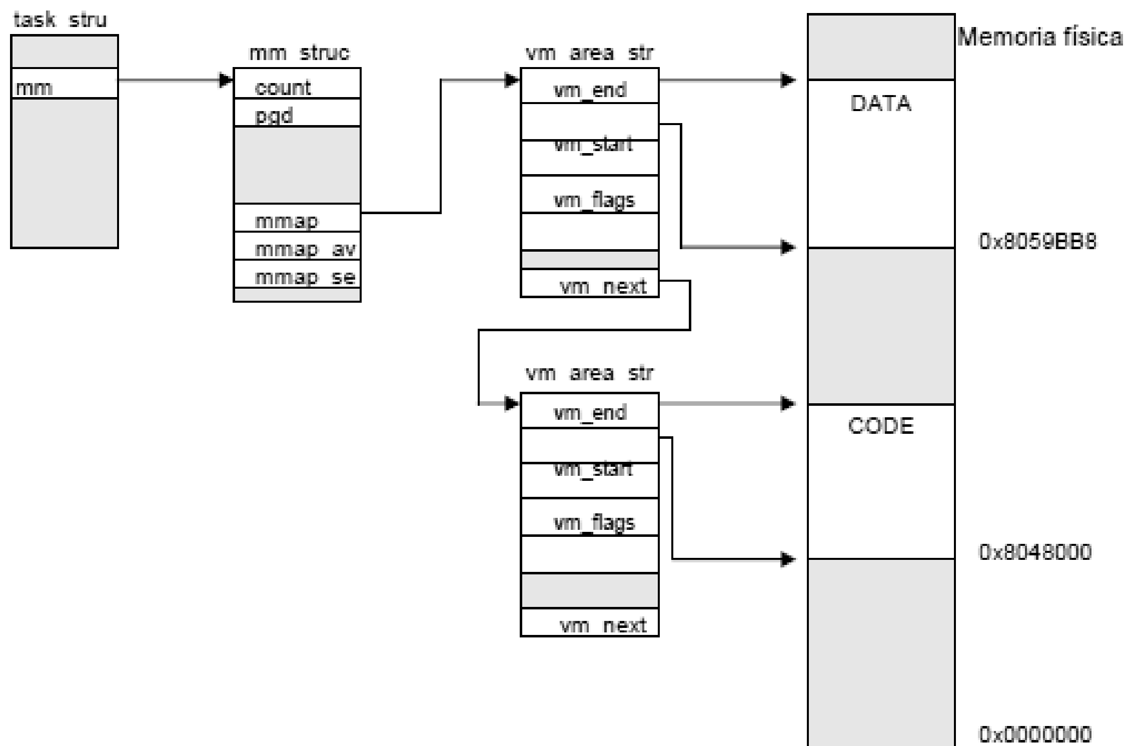


Fig. 3.15 Manejo de la memoria virtual Linux.

El núcleo de Linux accede repetidamente al grupo de estructuras vm_area_struct del proceso según crea nuevas áreas de memoria virtual para el proceso y según corrige las referencias a la memoria virtual que no está en la memoria física del sistema. Por esta razón, el tiempo que se tarda en encontrar la estructura vm_area_struct correcta es un

punto crítico para el rendimiento del sistema. Para acelerar este acceso, Linux también organiza las estructuras vm_area_struct en un árbol Rojo-Negro, en versiones anteriores en un árbol AVL.

Cuando a un proceso se reserva memoria virtual, en realidad Linux no reserva memoria física para el proceso. Lo que hace es describir la memoria virtual creando una nueva estructura vm_area_struct. Esta se une a la lista de memoria virtual del proceso. Cuando el proceso intenta escribir en una dirección virtual dentro de la nueva región de memoria virtual, el sistema creará un fallo de página. El procesador (MMU) tratará de decodificar la dirección virtual, pero dado que no existe ninguna entrada de tabla de páginas para esta memoria, no lo intentará más, y creará una interrupción de fallo de página, dejando al núcleo de Linux la tarea de reparar el fallo. Linux mira a ver si la dirección virtual que se trató de usar está en el espacio de direccionamiento virtual del proceso en curso. Si así es, Linux crea los PTEs (entrada en la tabla de páginas) apropiados y reserva una página de memoria física para este proceso. Puede que sea necesario cargar el código o los datos del sistema de ficheros o desde el disco de intercambio dentro de ese intervalo de memoria física. El proceso se puede reiniciar entonces a partir de la instrucción que causó el fallo de página y esta vez puede continuar, dado que memoria física existe en esta ocasión.

Linux mantiene las Tablas de página y sus tres niveles existentes: El directorio de tablas de páginas (punteros a tablas intermedias). La tabla de páginas intermedia (punteros a tablas de páginas). Las tablas de páginas (punteros a páginas).

```
typedef struct { unsigned long pte_low, pte_high; } pte_t;

typedef struct { unsigned long long pmd; } pmd_t;

typedef struct { unsigned long long pgd; } pgd_t;

typedef struct { unsigned long long pgprot; } pgprot_t;
```

Cada proceso (task_struct) tiene un campo mm de tipo mm_struct que almacena la información de su espacio de direcciones. El campo pgd mantiene el directorio de páginas utilizado para resolver la dirección física dada una dirección virtual.

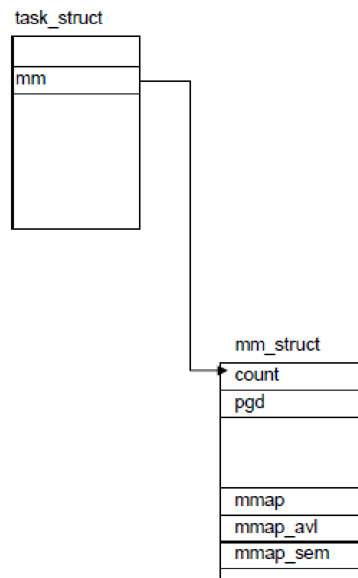


Fig. 3.16 Estructura de mm_struct.

Se producen constantes asignaciones y liberaciones de páginas físicas. Por ejemplo, cuando una imagen se carga a memoria, el SO necesita asignar páginas. Éstas serán liberadas cuando la imagen concluya su ejecución y se descargue. Otro uso de páginas físicas es para contener estructuras de datos específicas tales como las propias tablas de páginas. Los programas y las estructuras de datos relacionados con la asignación y liberación de páginas son quizás los más críticos para obtener un subsistema de memoria virtual eficiente.

El *kernel* solicita memoria de tres formas:

- Directamente al Buddy System, para asignaciones genéricas de grupos de marcos de página contiguos y potencia de 2.

- Al Slab Allocator (asignación de memoria por fragmentos dentro de una página, para evitar así la fragmentación interna), para objetos frecuentemente usados kmalloc().
- Usando vmalloc() para obtener áreas de memoria virtual contigua sin garantía de que también lo sea físicamente.

A los procesos de usuario no se les asigna realmente páginas, sino áreas de memoria (memory descriptor (mm_struct) + memory areas (vm_area_struct)), en otras palabras se les da, rangos de direcciones lineales válidos que se asignarán en el momento en que se vayan a usar.

3.3.5 Software para la simulación y evaluación

Para evaluar el diseño se ejecutaron *benchmarks* que son una técnica utilizada para medir el rendimiento de un sistema o componente del mismo mediante la comparativa. Los *benchmarks* aplicados tratan de incluir fragmentos de aplicaciones reales, o algoritmos comparables a algoritmos de aplicaciones reales, como un intento de comportarse comparables a las aplicaciones reales. Los *test* ejecutados sobre el diseño son Dhrystone, Stanford, Whetstone y lat_mem/bw_mem del paquete LMbench para estimar latencias y ancho de banda del circuito, los cuales son discutidos e implementados en el Capitulo 5 de pruebas y resultados.

3.4 Resumen del Capítulo

En este tercer capítulo se explicaron las características de los procesadores superescalares, así como las técnicas que se implementan para incrementar su desempeño, como el renombramiento de registros, la predicción de saltos condicionales y la ejecución especulativa de instrucciones. Posteriormente se discutió la arquitectura y función que tiene la estructura conocida como Unidad de gestión de memoria MMU y el conjunto de elementos que son utilizados en el desarrollo de esta tesis.

CAPÍTULO 4.

DISEÑO DE LA ARQUITECTURA PROPUESTA

En este capítulo se presenta el proceso que se sigue para realizar el diseño de la arquitectura de la Unidad de Gestión de Memoria (MMU) para un Procesador superescalar. En la sección 4.1 se explica la metodología que se aplica para tal fin. En la sección 4.2 se describen los elementos y características de diseño del LEON-SoC utilizado para el desarrollo de esta Tesis. Posteriormente, en la sección 4.3 se presenta el diseño y operación de los elementos del Sistema de memoria que comprende la MMU, Caches y RAM. Por último en la sección 4.4 se genera y analiza la arquitectura propuesta de la MMU de un Procesador superescalar con ejecución de instrucciones fuera de orden.

4.1 Metodología de diseño
4.2 Diseño del LEON-SoC.
4.2.1 Softcore Leon3
4.2.2 Bus AMBA 2.0
4.2.3 Debug Support Unit DSU
4.2.4 JTAG Debug Link
4.2.5 UART (RS232)
4.2.6 Controlador SDRAM
4.3 Sistema de gestión de memoria.
4.3.1 Sistema de caches
4.3.2 Operación SRMMU/Cache
4.3.3 Registro de control LEON/SRMMU
4.3.4 Mapeo SRMMU/ASI
4.3.5 Modelo de Interrupciones
4.4 Arquitectura del diseño propuesto de MMU superescalar.
4.5 Resumen del Capítulo

CAPÍTULO 4 Diseño de la Arquitectura propuesta

4.1 Metodología de diseño

La metodología que se sigue para realizar el diseño de la Unidad de Gestión de Memoria (MMU), permite que al final de este proceso se obtenga el modelo de la arquitectura que tendrá las características necesarias para que realice adecuadamente las funciones de las que está encargada y que se han descrito en la sección 3.2 y permite además, que sea posible introducir mejoras en el modelo de su arquitectura con el objetivo de incrementar el desempeño general del procesador.

La metodología inicia realizando un estudio de las características de las arquitecturas implementadas por los procesadores modernos, así como de las propuestas recientes encontradas en la literatura, expuestas en el Capitulo 2. En base a esta información se obtiene el *background* de operación, caracterización y del diseño de la MMU, para entonces realizar nuestra propuesta de diseño tomando como base la arquitectura de la MMU del *softcore* que es utilizado para el desarrollo de esta Tesis.

El proceso de selección del *softcore* de entre múltiples, fue delineado principalmente por opciones *open-source* que permite estudiar cómo funciona, diseñar, evaluar y no requieren la adquisición de licencias para su uso. Dentro de las alternativas libres se optó por el *softcore* Leon3 de 32 bits de arquitectura SPARCv8, especialmente porque implementa la SRMMU que es una arquitectura MMU contemporánea de alto performance con la cual puede gestionar la MV del SOLinux. Otro factor de selección fue por la gran cantidad de IPcores disponibles para diseñar un LEON-SoC, además del *toolchain* disponible para generar software y el SO para nuestro sistema, que lo hacen nuestra mejor opción.

Enseguida se elabora el SoC el cual es necesario para analizar y evaluar la operación de la MMU y el sistema de memoria del *softcore*. El SoC está compuesto por los

elementos mínimos para obtener el sistema embebido que son el *softprocessor* Leon3, el bus Amba, el controlador de memoria SDRAM, el JTAG link, la DSU y el UART.

El flujo de diseño que describe la metodología que se utiliza para el diseño de la MMU se muestra en el diagrama de la figura.

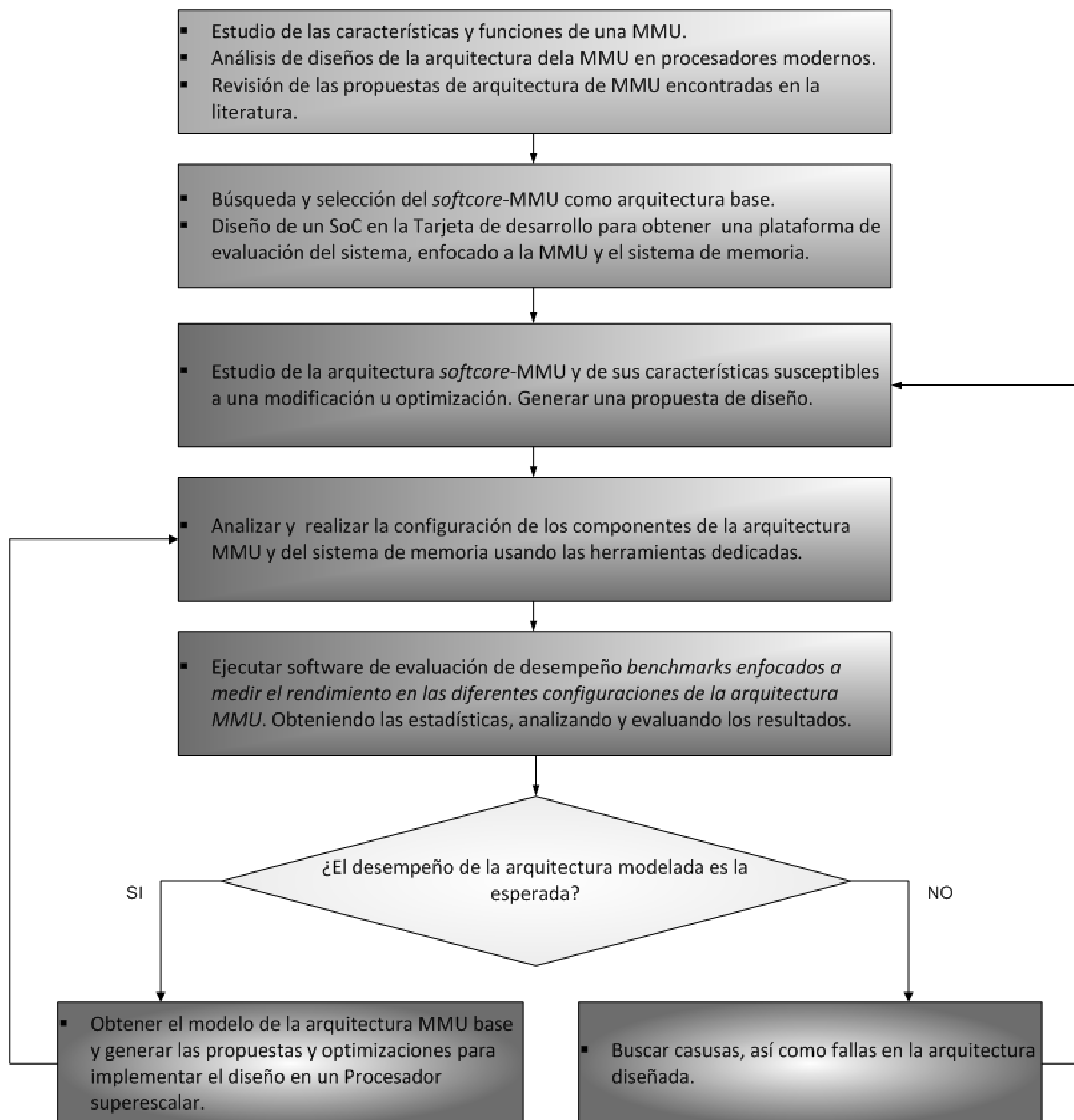


Fig. 4.1 Diagrama de flujo para el diseño propuesto.

4.2 Diseño del LEON-SoC

En esta sección se ofrece una visión general de la arquitectura del LEON-SoC utilizado, sus componentes y características. La Figura 4.2 muestra el diagrama de bloques de la arquitectura del SoC implementado.

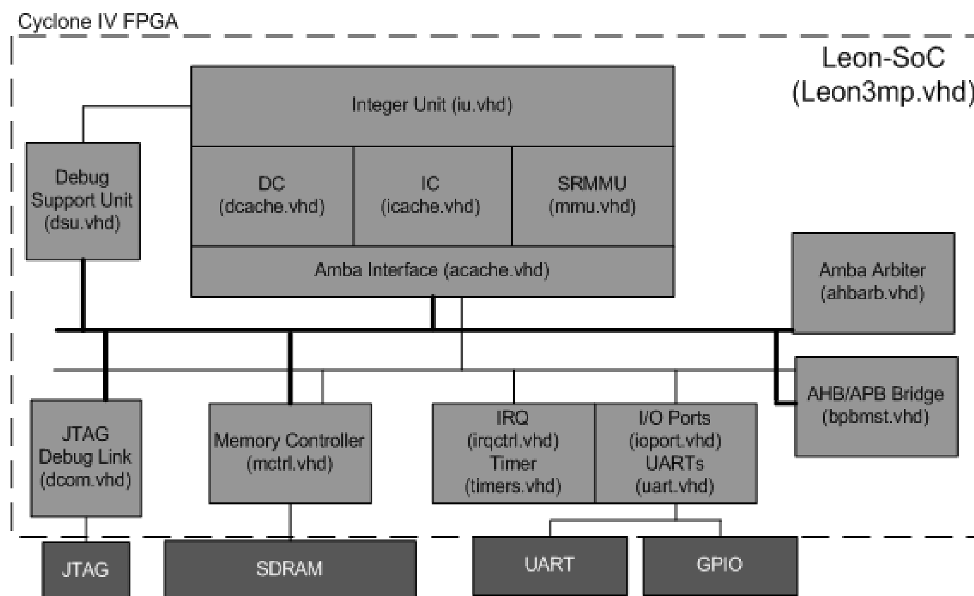


Fig. 4.2 Diagrama de la arquitectura del SoC

Los elementos del sistema son:

- Leon3 (IC/DC/SRMMU)
- Bus Amba 2.0 (AHB/APB)
- DSU
- JTAG Debug link
- UART
- Controlador SDRAM

4.2.1 Softcore Leon3

El Leon3 [31] es un procesador VHDL sintetizable de 32 bits RISC, *big-endian* con una arquitectura SPARCv8, desarrollado por la Agencia Espacial Europea (ESA) y por la empresa Gaisler Research. Todo su código está disponible bajo la licencia GPL, aunque también está bajo la comercial. Es altamente configurable lo que lo hace idóneo para el diseño del SoC.

Tiene una IU que ejecuta instrucciones enteras SPARCv8. Su implementación se centra en la portabilidad y baja complejidad, sin embargo, su descripción está severamente relacionada, por lo que es difícil de integrar nuevas características (y entender el código fuente). El número de ventanas de registro se puede configurar dentro del límite estándar SPARC (2-32), dejamos un valor predeterminado de 8 ventanas. El *pipeline* LEON es de 7 niveles: *fetch, decode, register access, execute, memory, exception* y *write back stage.*

FE (*Instruction Fetch*): Si la cache de instrucciones está activada, la instrucción se extrae de la cache de instrucciones. De lo contrario, es extraída directamente desde al controlador de memoria. La instrucción es válida al final de esta etapa y es captada dentro de la IU.

DE (*Decode*): La instrucción se decodifica y las direcciones destino del tipo salto CALL y branch se generan.

RA (*Register access*): Los operandos son leídos desde los registros o de inmediatos.

EX (*Execute*): Es la etapa donde ejecutan operaciones ALU, lógicas y corrimiento. Para las operaciones de memoria (por ejemplo, LD) y para las de salto JMPL / RETT, se genera la dirección.

ME (*Memory*): En ella se accede a la cache de datos para realizar operaciones de lectura y escritura (LD/ST)

XC (*Exception*) Interrupciones son ejecutadas.

WR (*Write*): Los resultados ALU, lógica, corrimiento o las operaciones de lectura de cache, se vuelven a escribir en los registros.

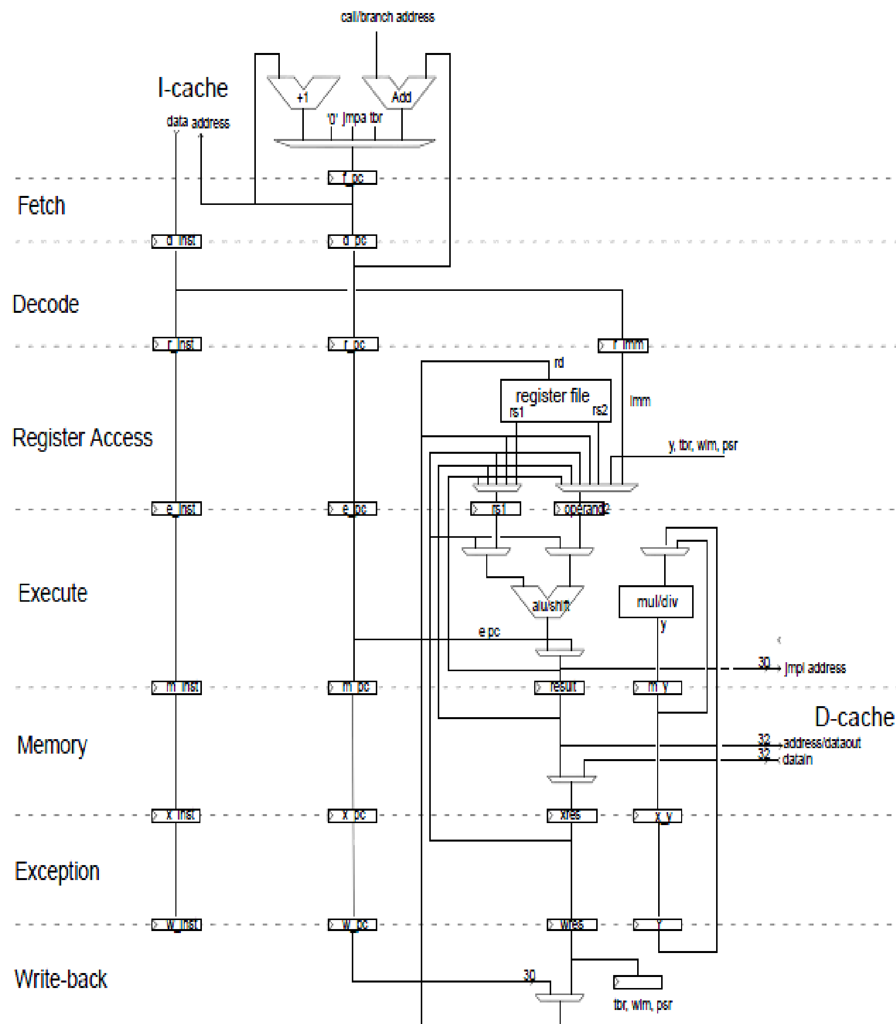


Fig. 4.3 Diagrama del pipeline de la Unidad de ejecución de enteros (IU) del Leon3

El *pipeline* del Leon3 implementa un predictor de saltos estático (branch prediction) usando una estrategia de *branch-always*, y comienza haciendo *fetch* de la dirección de la instrucción tipo branch. En una predicción correcta, 1 o 2 ciclos son ahorrados.

4.2.2 Bus AMBA 2.0

Es un bus ampliamente utilizado (procesadores ARM lo usan) como *bus-on-chip.* La especificación AMBA 2.0 [32] es ampliamente utilizada en procesadores embebidos y es libre de licencias. Sigue una estrategia de gestión e interconexión de los bloques funcionales que componen al SoC. Esta especificación define 2 tipos de buses:

AHB: Este bus es de alto rendimiento y es para una interconexión tipo múltiples maestros a múltiples esclavos. En este bus conectamos los *ipcores* maestros:

- LEON3 SPARC V8 Processor (Master 0)
- JTAG Debug Link (Master 1)

Los *ipcores* esclavos:

- LEON2 Memory Controller PROM/IO/SRAM/SDRAM AHB: 00000000 - 20000000 APB: 80000000 – 80000100 , sdram: 32-bit 128 Mbyte @ 0x40000000
- AHB/APB Bridge AHB: 80000000 – 80100000
- LEON3 Debug Support Unit (DSU) AHB: 90000000 - A0000000

La operación del AHB comienza por el maestro al cual se le debe conceder acceso al bus. Este proceso es iniciado por el maestro al enviar una señal de petición y dirección del esclavo para transferencia al *IPcore* AHB que indica cuando se le concederá el uso del bus. Una vez aceptado comienza la transferencia de datos entre el maestro y el esclavo.

APB: Es el bus que se usa para periféricos lentos, con un costo energético bajo (*low power*) y adecuado para un gran número de ellos. Es del tipo *single-master* y se comunica con el bus AHB mediante un *bridge* (bridge AHB-APB), lo que hace que él se comunique con los maestros del AHB. Esto otorga cierta independencia al bus APB ya que puede estar realizando transacciones mientras en el AHB se realizan otras tareas.

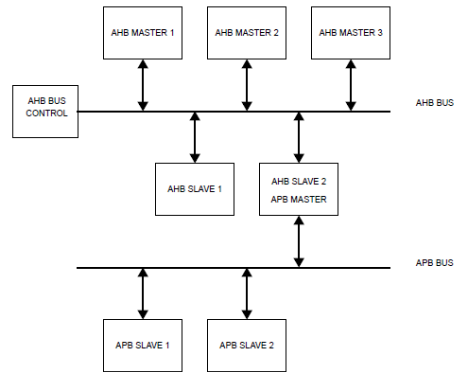


Fig. 4.4 Vista conceptual del bus AMBA AHB/APB

Aquí conectamos los *IPcore* periféricos esclavos:

- Generic UART APB: 80000100 – 80000200
- General Purpose I/O port APB: 80000900 - 80000A00

Las estructuras internas del AHB y APB se componen de diferentes elementos. En el caso del bus AHB están el Decoder y el Arbiter. El Arbiter lo que hace es asegurarse de que solamente un maestro esté usando el bus al mismo tiempo, esto lo realiza mediante un multiplexor, que únicamente permite pasar las señales del maestro que tiene el control del bus, mientras que el Decoder, genera las señales de selección de cada esclavo en función de la dirección de transferencia.

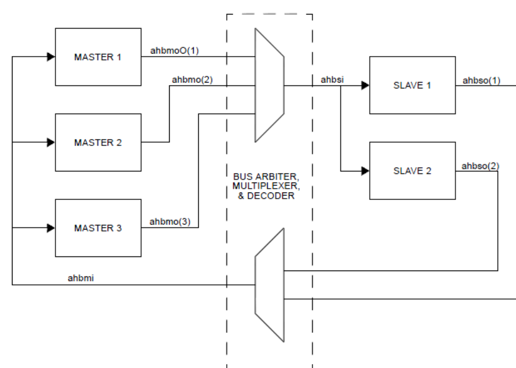


Fig. 4.5 Resumen de interconexión del bus AHB

Las entradas y salidas de un maestro AHB son definidas como record types del lenguaje VHDL y se encuentran en un paquete en la librería GRLIB AMBA:

```
-- AHB master inputs
type ahb_mst_in_type is record
hgrant : std_logic_vector(0 to NAHBMST-1); -- bus grant
hready : std_ulogic; -- transfer done
hresp : std_logic_vector(1 downto 0); -- response type
hrdata : std_logic_vector(31 downto 0); -- read data bus
hirq : std_logic_vector(NAHBIRQ-1 downto 0); -- interrupt result bus
end record;
-- AHB master outputs
type ahb_mst_out_type is record
hbusreq : std_ulogic; -- bus request
hlock : std_ulogic; -- lock request
htrans : std_logic_vector(1 downto 0); -- transfer type
haddr : std_logic_vector(31 downto 0); -- address bus (byte)
hwrite : std_ulogic; -- read/write
hsize : std_logic_vector(2 downto 0); -- transfer size
hburst : std_logic_vector(2 downto 0); -- burst type
hprot : std_logic_vector(3 downto 0); -- protection control
hwdata : std_logic_vector(31 downto 0); -- write data bus
hirq : std_logic_vector(NAHBIRQ-1 downto 0);-- interrupt bus
hconfig : ahb_config_type; -- memory access reg.
hindex : integer range 0 to NAHBMST-1; -- diagnostic use only
end record;
```

Un *IPcore* maestro AHB es definido de la siguiente manera:

```
library grlib;
use grlib.amba.all;
library ieee;
use ieee.std_logic.all;
entity ahbmaster is generic (
        hindex : integer := 0); -- master bus index
port (
        reset : in std_ulogic;
        clk : in std_ulogic;
        ahbmi : in ahb_mst_in_type; -- AHB master inputs
        ahbmo : out ahb_mst_out_type -- AHB master outputs);
end entity;
```

De la misma manera son definidas en la librería GRLIB AMBA las entradas y salidas de un esclavo AHB:

```
-- AHB slave inputs
type ahb_slv_in_type is record
hsel : std_logic_vector(0 to NAHBSLV-1); -- slave select
haddr : std_logic_vector(31 downto 0); -- address bus (byte)
hwrite : std_ulogic; -- read/write
htrans : std_logic_vector(1 downto 0); -- transfer type
hsize : std_logic_vector(2 downto 0); -- transfer size
hburst : std_logic_vector(2 downto 0); -- burst type
hwdata : std_logic_vector(31 downto 0); -- write data bus
hprot : std_logic_vector(3 downto 0); -- protection control
hready : std_ulogic; -- transfer done
```

```
hmaster : std_logic_vector(3 downto 0); -- current master
hmastlock : std_ulogic; -- locked access
hbsel : std_logic_vector(0 to NAHBCFG-1); -- bank select
hirq : std_logic_vector(NAHBIRQ-1 downto 0); -- interrupt result bus
end record;
-- AHB slave outputs
type ahb_slv_out_type is record
hready : std_ulogic; -- transfer done
hresp : std_logic_vector(1 downto 0); -- response type
hrdata : std_logic_vector(31 downto 0); -- read data bus
hsplit : std_logic_vector(15 downto 0); -- split completion
hirq : std_logic_vector(NAHBIRQ-1 downto 0); -- interrupt bus
hconfig : ahb_config_type; -- memory access reg.
hindex : integer range 0 to NAHBSLV-1; -- diagnostic use only
end record;
```

El *IPpcore* esclavo AHB es definido a continuación:

```
library grlib;
use grlib.amba.all;
library ieee;
use ieee.std_logic.all;
entity ahbslave is
generic (
hindex : integer := 0); -- slave bus index
port (
reset : in std_ulogic;
clk : in std_ulogic;
abhsi : in ahb_slv_in_type; -- AHB slave inputs
ahbso : out ahb_slv_out_type -- AHB slave outputs);
end entity;
```

El bus APB hace uso de un decodificador de dirección y de un multiplexor con los cuales selecciona el esclavo. Y de la misma manera que el AHB las señales de entrada y salida son agrupadas (APBI-APBO) y definidas en la librería GRLIB AMBA [33].

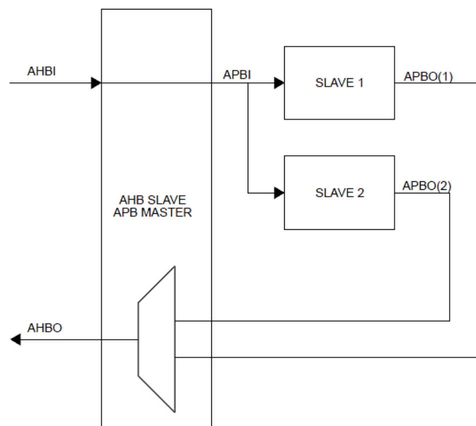


Fig. 4.6 Resumen de interconexión del bus APB

4.2.3 Debug Support Unit DSU

Se usa para la depuración del SoC, y accesar a los registros de propósito general y especial del procesador, contenido de memoria principal y cache. El procesador Leon3 puede usarse en modo depuración (*debug mode*) durante el cual el pipeline esta en modo idle y es controlado a través de la DSU. Para acceder a la DSU desde nuestro host lo hacemos con el software GRMON a través de la interface JTAG del FPGA.

4.2.4 JTAG Debug Link

Provee acceso al sistema a través de JTAG. Su función es traducir las señales del protocolo JTAG a instrucciones de lectura y escritura para el bus AHB. En este sistema la DSU se comunica con el exterior mediante el JTAG Debug Link.

4.2.5 UART (RS232)

El *ipcore* UART es un medio de I/O que nos permite la comunicación del LEON-SoC a una terminal en nuestro *host*. Como se especificó, se hace uso de la librería Newlib que en *System Calls* del tipo printf envía la salida de las mismas al puerto serial. La salida del UART está conectada a la entrada del JTAG. Su aplicación sería la siguiente: se abre una terminal modem se ejecuta GRMON para comunicarse con el puerto JTAG USB_BLASTER del SoC, en esta ventana se imprimen los datos enviados por el UART_stx que es la salida de impresión de nuestro firmware a ejecutar.

4.2.6 Controlador SDRAM

El controlador maneja memoria PC133 SDRAM [34] de bus de datos de 32 bits y ocupa un espacio de direcciones de acuerdo a la cantidad de memoria de la SDRAM. La SDRAM controlada es de 128MB (64Mx2), 13 bits de dirección (13 *row* / 10 *column*) y de 4 bancos. Implementa el modo de operación *burst transfer* para accesar a direcciones consecutivas para el caso de una escritura de línea en la IC y para DL y DST en la DC.

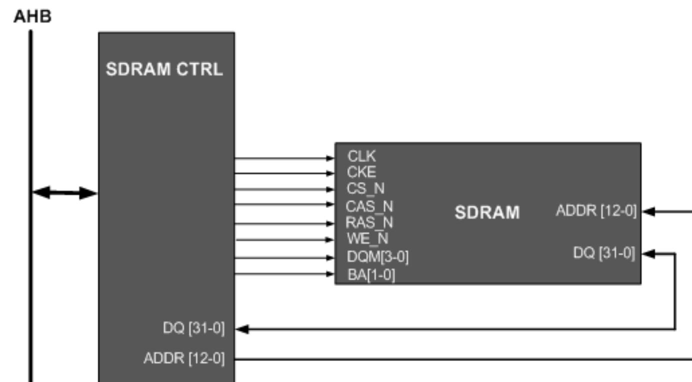


Fig. 4.7 Controlador SDRAM conectado al bus AHB y a la memoria SDRAM

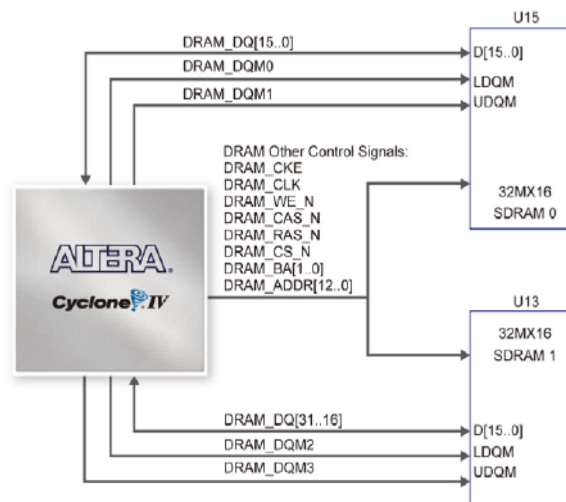


Fig. 4.8 Conexión de las SDRAM y el FPGA de la DE2-115

El controlador es programado a través del registro de configuración mapeado en el espacio de direcciones definido en el AHB. La operación del controlador SDRAM es mediante el registro de configuración (SDCFG).

31	30	29 27	26	25 23	22 21	20 18	17	16	15	14 0
Refresh	tRP	tRFC	tCD	SDRAM bank size	SDRAM col. size	SDRAM command	Page-Burst	MS	D64	SDRAM refresh load value

Fig. 4.9 Registro de configuración (SDCFG).

- Refresh: si se activa realiza el *refresh* de la memoria.
- tRP, tRFC, tCD: especifican los tiempos en ciclos de reloj.
- bank size: define el tamaño del banco por cada uno de ellos. “001”=8MB.
- colum size: Tamaño de la columna. “10”=1024.
- command: se escribe para ejecutar un comando de operación. “100”=AUTO-REFRESH, “110”=LOAD-COMMAND-REGISTER, “111”=LOADEXTENDED-COMMAND-REGISTER. Este campo es reseteado cada vez que termina la ejecución del comando.
- Page Burst: si es activado a 1, realiza operaciones de lectura en modo page burst.
- MS: Mobil SDRAM, siempre 0.
- D64: Data 64bits, no activamos este bit ya que nuestro bus es de 32bits.
- refresh load value: Indica el período entre cada comando AUTO-REFRESH.

Las direcciones mapeadas a la memoria principal (SDRAM) comprende el espacio de direcciones: 0x40000000 – 0x80000000.

4.3 Sistema de gestión de memoria

Después de obtener una visión general de la arquitectura del SoC, a continuación se detallan los elementos del Sistema de gestión de memoria.

4.3.1 Sistema de caches

El Sistema de caches del Leon3 es Harvard y configurable. Las dos caches pueden ser configuradas con 1-4 sets (asociatividad), 1-256 KB / set, 16 o 32 bytes por línea. Las áreas que son mapeadas a las caches son determinadas por la configuración de direcciones en el bus AHB. Las políticas de reemplazo pueden ser LRU o *random*. Las caches operan del modo VIVT al hacer uso del SO y la MMU como en un procesador SPARCv8.

Operación de la IC: Cuando ocurre un *cache miss*, la instrucción es extraída de memoria principal y su correspondiente *tag* y contenido es escrito en una localidad de la cache de acuerdo a la configuración (asociatividad y política de reemplazo) Si el bit de *instruction burst fetch* esta activado en el registro de control de cache (CCR) la línea de cache es escrita desde el punto inicial de la dirección solicitada hasta el final de la línea. Al mismo tiempo la instrucción es enviada a la IU.

Operación de la DC: De la misma manera que la IC cada línea de la cache contiene su *tag* asociada y un bit válido por cada sub-bloque (4-bytes). En un cache *miss* de lectura, el dato es extraído de memoria principal y es escrito en una localidad de la cache de acuerdo a la configuración y política de reemplazo. Si se produce un error de acceso a la memoria durante un Load se generará una interrupción de error de acceso a datos (tt = 0x9). La DC utiliza la política de escritura *write-through* e implementa un *buffer* de escritura (WRB) que consta de tres registros de 32 bits utilizado para retener temporalmente datos a almacenar hasta que se envía a la memoria principal. La WRB se debe vaciar antes de una secuencia *load-miss/cache-fill* para evitar que datos obsoletos puedan ser leídos de la memoria. Si se produce un error de escritura en el *buffer*, se lanzará la interrupción 0x2b. Dependiendo de la actividad de la memoria y cache, el ciclo de escritura puede no ocurrir hasta varios ciclos de reloj después de que las instrucciones de almacenamiento hayan sido completadas. Usa el modo de acceso *burst* para las instrucciones DL y DST (*Double Load y Double Store*).

Una *Tag* **de la IC y DC** está compuesta de los siguientes campos:

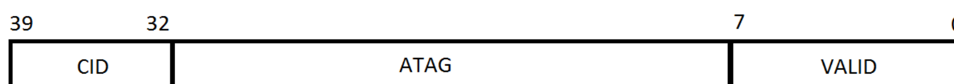


Fig. 4.10 Estructura de una *Tag* de la IC y DC

- CID: Identificador de contexto (proceso).
- ATAG: contiene la *tag address* de la línea.
- VALID: valida los sub-bloques o palabras (4-bytes) de la línea. Son activados si se concreta correctamente una escritura de línea en un *cache miss* y si están activados pueden ser referenciados, de lo contrario no, y esto debido a un error en memoria principal por el cual no se pudo escribir en la línea de cache correspondiente a un *cache miss*.

Solo los bits necesarios son usados en la *tag*, dependiendo de la configuración de la cache.

Cache *flushing*: Ambas IC y DC son vaciadas por la ejecución de una instrucción FLUSH ASI=0x10 para IC y 0x11 para DC. La IC es también vaciada al activar el bit FI del registro de control de cache (CCR) o mediante una instrucción de escritura con ASI=0x15. La DC es también vaciada al activar el bit FD del CCR o mediante una instrucción de escritura con ASI=0x16. El *cache flushing* toma 1 ciclo por cada línea.

Registro de control de caches (CCR): Indica la operación de las caches de IC y DC.

31	23	22	21		17	16	15	14	6	5	4 3	2 1 0
	DS	FD	FI		ST	IB	IP	DP		DF	IF	DCS	ICS

Fig. 4.11 Registro de control de cache (CCR)

- DS, ST, solo se implementan en modo *cache snonping*.
- FD: activado hace FLUSH a la DC.
- FI: activado hace FLUSH a la IC.
- IB: *Instruction burst fetch.*
- IP: *IC flush pending*, se activa cuando se está realizando FLUSH.
- DP: *DC flush pending*, se activa cuando se está realizando FLUSH.

- DF, IF: *Cache Freeze,* activado la cache esta congelada cuando una interrupción es ejecutada.
- DCS: Indica el estado de la DC, congelada, activada o desactivada.
- ICS: Indica el estado de la IC, congelada, activada o desactivada.

Registros de configuración de cache: Indican configuración de la IC y DC uno por cache.

31	30	29 28	27	26 25 24	23 20	19	18 16	15 12	11 4	3	0
CL		REPL	SN	SETS	SSIZE	LR	LSIZE	LRSIZE	LRSTART	M	

Fig. 4.12 Registros de configuración de cache.

- CL: activado si implementa *cache locking.*
- REPL: política de reemplazo, 00-mapeo directo, 01-LRU, 10-LRR, 11-rand.
- SN: activado si implementa *cache snooping.*
- SETS: número sets (asociatividad) 000-MD, 001-2way, 010-3way, 011-4way.
- SSIZE: indica el tamaño en KB del set.
- LSIZE: indica el tamaño de la línea, 16 o 32 bytes.
- LRSTART: indica la dirección inicial (8-MSB) de la memoria ram.
- M: Activado si la MMU es activada.

Los registros de control y configuración pueden accederse a través de las instrucciones LSA/STA usando un ASI=2 y la dirección correspondiente del registro más información en el manual SPARCv8.

Tabla 4.1 Mapeo ASI = 2 a registros de Control y Configuración de Caches

Address	Register
0x00	Cache control register
0x04	Reserved
0x08	Instruction cache configuration register
0x0C	Data cache configuration register

Después de iniciar o por un *reset*, las caches deben deshabilitarse y el registro de control de caches (CCR) se pone 0. Antes de que las caches sean habilitadas nuevamente debe realizarse un flush para limpiar *tags* y bits válidos. La secuencia en ensamblador sería la siguiente:

```
flush
set 0x81000f, %g1
sta %g1, [%g0] 2
```

Ciclos de operación del *pipeline*: Para generar las instrucciones SL y SST (*Single Load y Store*) (4 bytes) son necesarios 1 y 2 ciclos respectivamente en la etapa de Execute. En el primer ciclo se genera la dirección que es pasada a la etapa de Memory para el correspondiente acceso, aquí acaba una SL. Cuando es un SST se realiza un segundo ciclo en Execute para enviar a la etapa de Memory el dato a escribir en la dirección. Para un DL y DST (*Double Load y Store*) son necesarios 2 y 3 ciclos para concretar la instrucción.

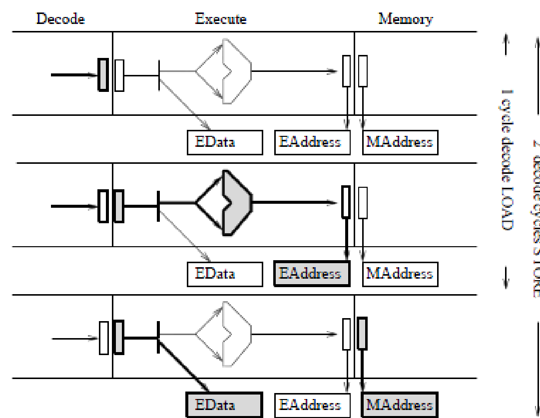


Fig. 4.13 Ciclos de operación en la etapa de EX (*Execute*)

En caso de un *miss cache* en la etapa de Memory, el *pipeline* se detendrá (*stall the pipeline*), la DC cambiara su estado y se enviara la DV a la MMU para traducirla a DF (1 ciclo más si se encuentra en TLB).

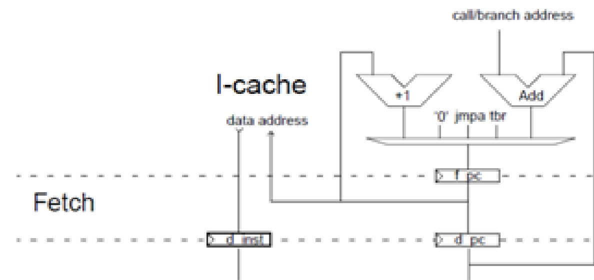


Fig. 4.14 Ciclos de operación en la etapa de FE (*Fetch*)

La dirección de una instrucción se obtiene al incrementar el PC en la etapa de Fetch. Además esta dirección puede ser generada en la etapa de Execute por una instrucción branch o jump. Al ocurrir un *miss* en la IC cambiará su estado y se enviará la DV a la MMU para traducirla a DF. Después de traducida se hará el *fetch* completo de la línea de cache desde la memoria principal, mientras el *pipeline* se detendrá en la etapa de Decode esperando por la dirección.

Implementación de Cache: Bloques RAM se utilizan para implementar los *tags* y datos contenidos en las memorias cache. Dependiendo de la configuración de la cache, diferentes tipos y tamaños de bloques de RAM se utilizan. El *tag* se implementa con una *syncram (single port RAM)* por cache *way*. La parte de la cache que contiene los datos (instrucciones o datos) es también una memoria tipo *syncram*. Se necesita de 1 ciclo para comparar el *tag* y de 1 ciclo para obtener o escribir el dato.

4.3.2 Operación SRMMU/Cache

La MMU es una SPARC V8 Reference Memory Management Unit (SRMMU) los detalles de operación están descritos en el Capitulo 2. Cuando la MMU está activada, los *tags* de caches almacenan la DV y también incluyen un campo de contexto de 8 bits. Tanto la dirección del *tag* y el campo de contexto deben coincidir para generar un cache *hit*.

Debido a que la cache es *Virtually Tagged* VT, no se necesitan ciclos de reloj adicionales en caso de *hit* en cache en un load. En caso de un *miss* load en cache o store *hit* (*write-through cache*), 2 ciclos de reloj adicionales se utilizan para traducir la dirección física si hay un *hit* en el TLB. Si hay un *miss* en TLB la tabla de páginas debe ser referenciada, lo que resulta en un máximo de cuatro accesos de lectura AMBA y una posible operación de reescritura.

En un fallo de página la MMU genera la interrupción 0x09, y actualiza sus registros de estado (MMU s*tatus registers*) de acuerdo a la operación de la SRMMU.

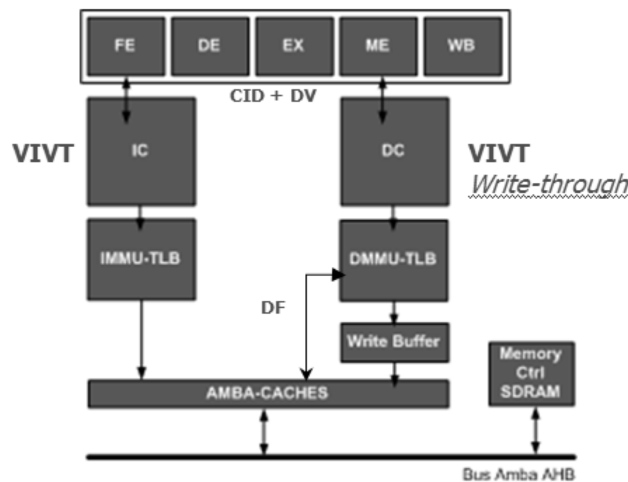


Fig. 4.15 Estructura del Sistema de memoria, Caches, MMU y controlador SDRAM

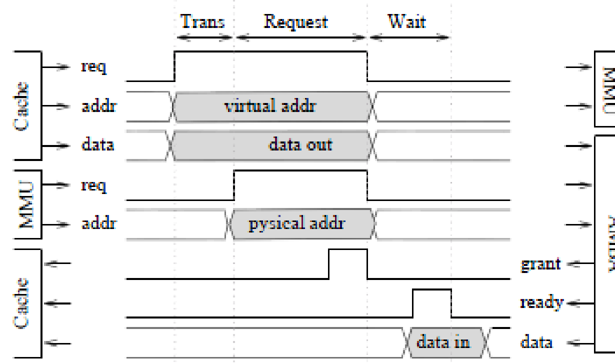


Fig. 4.16 Protocolo de la operación del Sistema de memoria

Para el *write buffer* de la DC, la traducción de la DV se debe realizar antes de escribir sobre él, ya que este enviará la DF a escribir en la memoria principal.

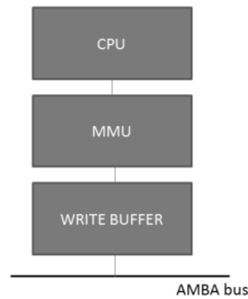


Fig. 4.17 Write Buffer de la DC

4.3.3 Registro de control LEON/SRMMU

El *layout* del registro de control de la MMU del Leon3 es mostrado a continuación, la definición de los demás registros de la SRMMU son expuestos en el Capitulo 2, ya que son exactos a la arquitectura SRMMU SPARCv8.

31 28	27 24	23 21	20 18	17 16	15	14	2	1	0
IMPL	VER	ITLB	DTLB	PSZ	TD	ST	RESERVED	NF	E

Fig. 4.18 Registro de control LEON/SRMMU

- IMPL: ID de la implementación de MMU.
- VER: versión de la ID implementada.
- ITLB, DTLB: número de entradas de TLB.
- PSZ: tamaño de página. 0=4KB, 1=8KB, 2=16KB, 3=32KB,
- TD: TLB *Disable*, cuando TD=1 cada traducción es con las Tablas de página.
- ST: *Separate* TLB, instrucciones y datos.
- E: *Enable* MMU.

4.3.4 Mapeo SRMMU/ASI

Con las instrucciones alternativas LDA / STA, además de poder accesar a registros especiales del hardware, son utilizadas como instrucciones para la operación de la MMU, las cuales son usadas por Linux.

Tabla 4.2 Bits de dirección ASI y descripción de uso

ASI	Usage
0x10	Flush I and D cache
0x14	MMU diagnostic dcache context access
0x15	MMU diagnostic icache context access
0x18	Flush TLB and I/D cache
0x19	MMU registers
0x1C	MMU bypass
0x1D	MMU diagnostic access
0x1E	MMU snoop tags diagnostic access

Con las LDA / STA con ASI 1C "MMU bypass" evitamos la traducción en la MMU, esto es de utilidad para el Kernel que se ejecuta en modo real como la gestión de las Tablas de página. Para más información acerca del uso de las ASI, hacer referencia al SPARCv8 manual Appendix I.

4.3.5 Modelo de Interrupciones de memoria

La tabla a continuación presenta el listado de las interrupciones mencionadas generadas por el Sistema de memoria las cuales son las del modelo SPARC Trap Model y son atendidas por el procesador de acuerdo a la prioridad establecida.

Tabla 4.3 Tabla de interrupciones del Sistema de memoria

Trap	TT	Pri	Description
write error	0x2b	2	write buffer error during data store
instruction_access_error	0x01	3	Error during instruction fetch
data_access_exception	0x09	13	Access error during data load, MMU page fault

4.4 Arquitectura del diseño propuesto de MMU superescalar

La figura 4.8 muestra un resumen de los componentes implementados para el diseño propuesto de la MMU superescalar. Además, él diagrama trata de dar una visión general del proceso de traducción y operación de la arquitectura, que comienza de la siguiente forma: las IC y DC reciben la DV para enviarla a la MMU para su traducción correspondiente (1), además, la DC recibe las direcciones ASI's de la MMU (2) para hacer *flush* o *bypass* estas últimas son enviadas al controlador de memoria sin traducción ya que son emitidas como DF, en este caso *writebuffer* de la DC recibirá la dirección y realizara la solicitud para enviarla por el bus AMBA hacia el controlador (3).

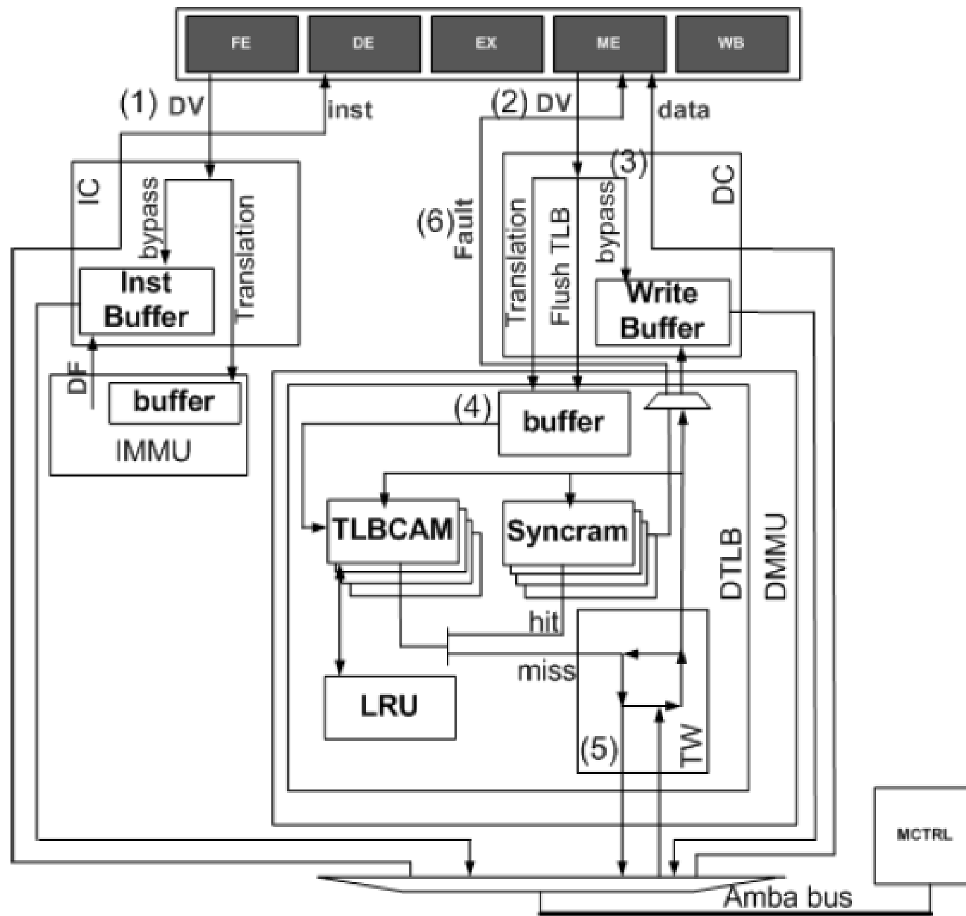


Fig. 4.19 Diseño de componentes para obtener la MMU superescalar

Las caches son *dual_port* que permiten realizar hasta dos operaciones simultáneas de escritura o lectura (arquitectura superescalar), por lo cual puede emitir hasta dos direcciones por ciclo a la MMU, por lo cual, la MMU cuenta con un *buffer* para almacenarlas antes de que se les permita su traducción (4) en el TLB. En caso de que la operación de traducción en el TLB falle, se hará un recorrido en las Tablas de página (5) para traducir o en caso contrario enviar la interrupción de fallo (6). En cada fallo de traducción en TLB, una de sus entradas es removida por el algoritmo de reemplazo LRU, la nueva traducción es almacenada en la entrada y la DF es enviada al *buffer* de Instrucciones o al de escritura los cuales harán la solicitud para enviar la DF por el bus AMBA y acceder a la localidad de la MP.

La implementación de las caches es mediante dos memorias una para comparar *tags* y otra para acceder a los datos, el tipo de memoria es *dual_port* ya que la arquitectura superescalar de dos instrucciones por ciclo. Ambas caches son asociativas de 4 vías, en el primer ciclo se emitirá la DV al controlador de memoria para acceder a la *tagmemory* y comparar el *tag* de las localidades mapeadas, en caso de *hit* en el segundo ciclo se emitirá la dirección para accesar a la *datamemory* para obtener el dato. En caso de *miss* se emitirá una señal al procesador para detener la ejecución (etapa de WB) de esa instrucción mientras es traída de la memoria principal, en paralelo a este proceso el controlador de la cache ejecutara la política de reemplazo random para la IC y LRU para la DC para escribir el nuevo dato.

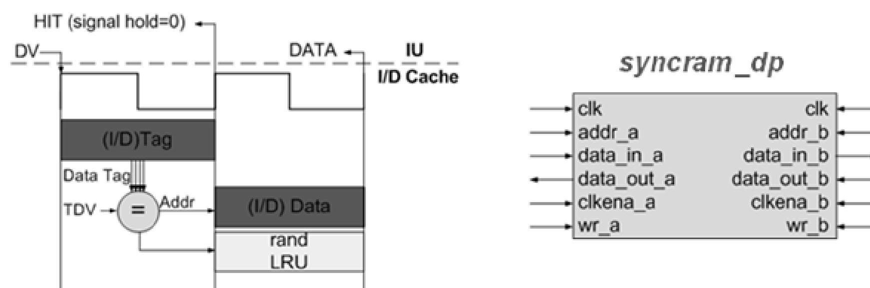


Fig. 4.20 Implementación de caches *dual_port*

El TLB es implementado con una memoria direccionable por contenido (TLBCAM) que contiene los *tags* y una memoria síncrona (*syncram*) que contiene los datos (las DFs).

En cada ciclo de reloj el *buffer* de la MMU envía una DV que se comparará con todas las entradas *tags* de la TLBCAM en un ciclo de reloj, en caso de un hit se actualizara los bits de acceso (LRU) y se accederá a la *syncram* para obtener la DF en un ciclo más, por lo cual la operación de traducción mediante el TLB es mediante 2 ciclos de reloj.

En caso de un TLB *miss* se accederá a las Tablas de página mediante TW que está conectado al bus AMBA para acceder a la MP al igual que el Inst. buffer y el Write buffer. Este proceso requiere de hasta 4 accesos a las Tablas de página en la MP hasta obtener la traducción o en caso contrario enviará la interrupción al SO de fallo de página.

La operación de flush en TLB se lleva a cabo mediante una instrucción ASI hacia la MMU. La operación toma por lo menos un ciclo por entrada del TLB en invalidar cada una de las entradas que hacen *match* con el flush, en caso de que bits de acceso (*referenced/modified*) estén activados la entrada será enviada y escrita a las tablas de pagina para en la memoria principal para actualizar.

4.5 Resumen del Capítulo

En este capítulo se describió la metodología utilizada para realizar el diseño de la arquitectura del MMU, la cual consistió en un análisis de arquitecturas y propuestas recientes, selección de la arquitectura base que fue la SRMMU del procesador Leon3 ya que es de alto performance y están disponibles una gran cantidad de IPcores que nos permitieron desarrollar un SoC como plataforma de desarrollo y evaluación del procesador y su sistema de memoria. Enseguida se presentaron las características y se detallaron los elementos que componen la arquitectura propuesta para el Sistema de memoria y la MMU de un procesador superescalar.

CAPÍTULO 5.

PRUEBAS Y RESULTADOS

El presente capítulo presenta el análisis de las estadísticas obtenidas tras realizar *benchmarks* a la arquitectura diseñada mediante la evaluación de diferentes esquemas y configuraciones de los elementos que conforman el sistema de memoria (MMU y Caches). Una vez que se tienen los resultados de las evaluaciones correspondientes de cada una de las configuraciones simuladas, se realizó la selección de las estadísticas de mayor interés que reflejen mejor el impacto que tienen las modificaciones implementadas en el modelo propuesto respecto a una arquitectura convencional.

5.1 Descripción de los factores a evaluar
5.2 Descripción del Procesador
5.3 Configuración de Caches
5.4 Configuración de la MMU
5.5 Benchmarks
5.5.1 Dhrystone 2.1
5.5.2 Stanford
5.5.3 Whetstone
5.5.4 LMbench
5.6 Resultados
5.7 Resumen del Capítulo

CAPÍTULO 5 Pruebas y resultados

5.1 Descripción de los factores a evaluar

Para evaluar el desempeño de la arquitectura propuesta que se ha modelado, es necesario seleccionar del conjunto de estadísticas arrojadas tras la simulación, las que mejor sirvan para propósitos definidos. Para el presente trabajo, se analiza en primer lugar la métrica correspondiente al número de instrucciones ejecutadas por segundo (MIPS). Esta estadística es considerada como una de las más importantes para evaluar el desempeño del procesador y su sistema de memoria ya que los *benchmarks* ejecutados contienen múltiples accesos a memoria.

Otra de las métricas de importancia son aspectos de implementación del circuito, como son la frecuencia y área utilizada (recursos usados del FPGA).

El trabajo consiste en evaluar diferentes configuraciones del sistema de memoria (MMU y caches), cada configuración se sintetiza e implementa en la tarjeta DE2-155 y su desempeño es medido por medio de la ejecución de *benchmarks* estándar, al final se produce el reporte que en el siguiente capítulo es analizado y detalla conclusiones de la arquitectura.

5.2 Descripción del Procesador

Las configuraciones de la arquitectura del sistema de memoria (MMU y caches) serán evaluadas especialmente en performance y aspectos de implementación siendo los siguientes:

- Performance (Resultados de *benchmarks* aplicados).
- Recursos utilizados por el circuito.
- Frecuencia del sistema.

Con los anteriores podemos sacar una medida de rendimiento ya que ha sido la fuerza impulsora clave detrás de los avances en la arquitectura de computadoras. Es erróneo esperar que una arquitectura sea mejor porque ejecute más rápidamente aplicaciones.

A continuación se presenta el resumen de los parámetros y configuración del procesador que permanecen para todas las configuraciones a evaluar.

Tabla 5.1 Configuración base del *softcore* Leon3 y controlador de memoria

LEON3	
Tipo de arquitectura	*Escalar*
Etapas de pipeline	*7*
ISA	*SPARC V8*
Tipo	*32-bits / Big endian*
Modos de direccionamiento	*Inmediato, desplazamiento e indexado*
Registros	
Tipo	*Ventana de registros*
Número de ventanas	*8*
Registros globales	*8*
Registros generales/ventana	*16*
Total de GPR	*136*
SPARC v8	
Multiplicador	*1 - 35 ciclos*
Divisor	*35 ciclos*
FPU IEEE-754	*4 - 25 ciclos*
Cache	
Tipo	*Harvard*
Tamaño de sub-bloque	*4 bytes (palabra)*
Política de escritura	*Write-through*
Write buffer	*3 entradas / 32 bits*
Valid bits	*1 por sub-bloque*
MMU	
Arquitectura	*Harvard (ITLB / DTLB)*
Tipo de TLB	*Totalmente Asociativa*
Tamaño de página	*4K*
Controlador de memoria	***SDRAM***
Tamaño	*128M*
Lacia de lectura	*2 ciclos*
Latencia de escritura	*3 ciclos*
Read burts	*Si*

5.3 Configuración de Caches

Esta sección tiene como objetivo presentar el resumen de las configuraciones implementadas en las Caches Harvard del procesador a evaluar. Las configuraciones Conf-1, Conf-2 y Conf-3 que se presentan fueron seleccionadas por parámetros de diseño que influyen en su rendimiento (tasa de impacto) y costo de implementación. Los parámetros más importantes de cache son:

- Tamaño de cache: en KB. Una cache más grande puede retener más datos útiles, pero es más costosa y quizá más lenta.
- Tamaño de bloque o ancho de línea: es la unidad de transferencia de datos entre la cache y memoria principal. Con una línea más grande se llevan más datos a la cache con cada fallo. Lo anterior puede mejorar la tasa de impacto, pero también tiende a almacenar datos de menor utilidad que pueden generar el reemplazo.
- Política de mapeo: puede generar más costos de hardware y puede o no tener beneficios de rendimiento como consecuencia, una operación más compleja y lenta.
- Política de reemplazo: Determina en cuál de los bloques (en los que se puede mapear) se debe sobreescribir.

Tabla 5.2 Configuraciones de caches a evaluar

Conf-1	
Ancho de línea	*16 bytes*
Tipo de Mapeo	*Directo*
Tamaño cache/way	*4K*
Política de reemplazo	
IC	*S/N*
DC	*S/N*
Tamaño de cache	***4K***

Conf-2	
Ancho de línea	*16 bytes*
Tipo de Mapeo	*4-ways*

Tamaño cache/way	*4K*
Política de reemplazo	
IC	*LRU*
DC	*LRU*
Tamaño de cache	***16K***

Conf-3	
Ancho de línea	*32 bytes*
Tipo de Mapeo	*4-ways*
Tamaño cache/way	*4K*
Política de reemplazo	
IC	*random*
DC	*LRU*
Tamaño de cache	***16K***

5.4 Configuración de la MMU

De igual forma se muestra el resumen de las configuraciones implementadas en la MMU. Configuraciones MMU-1, MMU-2 y MMU-3 que se presentan fueron seleccionadas por parámetros de diseño que influyen en su rendimiento de la memoria virtual (tasa de impacto de traducción) y costo de implementación:

Tabla 5.3 Configuraciones de la MMU a evaluar

MMU1	
Política de reemplazo	*LRU*
Entradas en ITLB	*32*
Entradas en DTLB	*32*
Tamaño de página	*4K*

MMU2	
Política de reemplazo	*LRU*
Entradas en ITLB	*64*
Entradas en DTLB	*64*
Tamaño de página	*4K*

MMU3	
Política de reemplazo	*Increment*
Entradas en ITLB	*64*
Entradas en DTLB	*64*
Tamaño de página	*4K*

5.5 Benchmarks

Esta sección contiene los diferentes *benchmarks* usados y además son discutidos con pros y contras con respecto a su uso en la evaluación.

El *benchmark* ideal mide el rendimiento del tipo de aplicaciones que el sistema ejecutará, pero este tipo de *benchmarks* son muy difíciles de construir. Los *benchmarks* actuales tratan de incluir fragmentos de las aplicaciones reales, o algoritmos comparables a algoritmos de aplicaciones reales, como un intento de comportarse comparables a las aplicaciones reales.

5.5.1 Dhrystone 2.1

Por su sencillez y tamaño, actualmente es comúnmente aplicado para evaluar sistemas embebidos. Dhrystone [35] es un *benchmark* sintético el cual está diseñado para medir el rendimiento del pipeline de enteros. Está compuesto por operaciones aritméticas de enteros, operaciones de cadena, lógicas y accesos a memoria que reflejan las actividades de aplicaciones generales que procesa CPU. El *benchmark* está escrito en lenguaje C lo que lo hace muy portable, pero tiene algunos inconvenientes de precisión:

- El tamaño del código es muy pequeño, por lo cual no es bueno para evaluar el desempeño del sistema de memoria.
- El código puede ser compilado más óptimamente o no, por diferentes compiladores.

5.5.2 Stanford

Stanford es un pequeño conjunto de pruebas reunidas, diseñadas por John Hennessy. La suite incluye los siguientes programas:

- Perm: programa de permutación muy recursivo.
- Towers: programa para resolver el problema de Torres de Hanoi.
- Queens: programa para resolver el problema de ocho reinas 50 veces.
- Intmm: programa de multiplicación de dos matrices de enteros.
- Mm: programa de multiplicación de dos matrices de punto flotante.
- Puzzle: Cómputo atado.
- Quick: ordenar una matriz utilizando Quicksort.
- Bubble: ordenar una matriz utilizando BubbleSort.
- Tree: ordenar una matriz utilizando Treesort.
- FFT: programa que calcula la transformada rápida de Fourier.

Stanford mide el tiempo de ejecución en milisegundos para cada uno de los diez pequeños programas incluidos en el conjunto de pruebas. Dos sumas ponderadas también son calculadas como resultado. Uno que refleja los tiempos de ejecución de los programas de punto fijo y otro refleja los tiempos de ejecución de los programas de punto flotante. Las sumas ponderadas se calculan en base a los tiempos de ejecución de los programas, entre más pequeño mejor. La suma ponderada de punto fijo incluye los tiempos de ejecución para todos los programas excepto Mm y FFT y la suma ponderada de punto flotante incluye todos los tiempos de ejecución.

5.5.3 Whetstone

En sus comienzos demostró ser una buena medida de rendimiento, sin embargo con el tiempo su funcionalidad se vió reducida por su alta sensibilidad a optimizadores. A finales de la década de 1980 y comienzos de la década de 1990 se reconoció que el

Whetstone no funcionaría adecuadamente para medir el rendimiento de supercomputadoras con multiprocesadores paralelos. Sin embargo el Whetstone aún se utiliza ampliamente, porque provee una medida muy razonable de rendimiento de monoprocesadores de aritmética flotante.

Los resultados son provistos en MWIPS (Millones de Instrucciones Whetstone Por Segundo). El significado de la expresión "Instrucciones Whetstone" no queda claro, salvo que uno examine cuidadosamente el código fuente. Sin embargo, se puede definir a una instrucción Whetstone como una instrucción de punto flotante promedio. Se calcula como (100 * cantidad de iteraciones * cantidad de *WIPS* por iteración / tiempo de ejecución).

5.5.4 LMbench

Es una suite portable de *benchmarks* para UNIX, se hará uso de los *benchmarks* lat_mem_rd y bw_mem enfocados a medir latencias y ancho de banda del sistema de memoria.

En el uso de sistemas reales SW/HW, las latencias de memoria aparte de ser resultados generados por el hardware (tipo de memoria y bus) también lo son por la secuencia de instrucciones ejecutadas, aquellas que ocasionan paros en el *pipeline* por dependencia de datos (*blocking loads*) que incrementan la latencia. Por lo anterior, las instrucciones load y store que ejecuta una aplicación puede generar un impacto, sobre todo las instrucciones store que generan una mayor latencia ya que para actualizar la memoria se genera una escritura en los niveles siguientes de la jerarquía (*written back to storage*). El test lat_mem_rd mide latencias solo del tipo load ya que se trata de conocer las latencias "puras" de acceso. Consiste en crear una matriz y avanzar N bytes a través de toda la matriz, la evaluación mide en nanosegundos la latencia de lectura en el sistema de memoria. Los argumentos para su ejecución son: el tamaño de la matriz en MB y el avance en bytes (*stride size*). Nota: este *benchmark* solo mide accesos de datos no de instrucciones.

Otro parámetro importante a conocer de nuestro sistema es el ancho de banda MB/s. En *benchmark* utilizado es bw_mem el ancho de banda de sistema de memoria. Consiste en asignar el doble de la cantidad especificada de memoria y luego tomar el tiempo de copiado del primer espacio de memoria al segundo espacio de memoria. Este *benchmark* permite medir los tiempos de operaciones de las lectura, escritura y lectura-escritura en el sistema de memoria. Su ejecución y parámetros son de la siguiente forma: bw_mem_cp [size] [rd|wr|rdwr].

Para más información sobre estos dos *benchmarks* hacer referencia al documento *Measuring Cache and Memory Latency and CPU to Memory Bandwidth For use with Intel® Architecture.* [36]

5.6 Resultados

En este apartado se presentan los resultados obtenidos a partir de las configuraciones previas establecidas.

El primer *benchmark* aplicado es el de Dhrystone, a continuación un ejemplo de su carga y ejecución sobre el SoC implementado:

```
grmon2> load dhry
  40000000 .text                      55.2kB /  55.2kB   [===============>] 100%
  4000DCA0 .data                       2.7kB /   2.7kB   [===============>] 100%
  Total size: 57.86kB (2.16Mbit/s)
  Entry point 0x40000000
  Image /home/fernandw/Downloads/bencharks/dhry loaded

grmon2> run
Execution starts, 400000 runs through Dhrystone
Microseconds for one run through Dhrystone:    8.2
Dhrystones per Second:                      121951.2
Dhrystones MIPS      :                        69.4
Program exited normally.
```

Este *benchmark* se ejecutó sin SO con el fin de realizar una evaluación enfocada solo a testear las caches, se realizó con 400,000 corridas a las configuraciones establecidas

previamente, con el fin de revelar la que será implementada en el diseño final del Sistema de memoria.

Tabla 5.4 Tabla de resultados Dhrystone sobre caches

DHRYSTONE	Conf-1		**Conf-2**		**Conf-3**	
Frecuencia (MHz)	*50*	*80*	*50*	***80***	*50*	***80***
Tiempo por un Dhrystone (iteración)	13.4	9.5	8.1	5.1	8.1	5.1
Dhrystones por segundo	74487.9	104986.9	121951.2	198019.8	121951.2	198019.8
Dhrystones MIPS	42.4	59.8	69.4	**112.7**	70.3	**112.7**

Los resultados muestran que tanto Conf-2 y Conf-3 a la frecuencia máxima (80MHz) del sistema son las que mejores resultados ofrecen. Pero como se indicó previamente al ser un programa pequeño y de pocos accesos a memoria, por lo que es necesario realizar más pruebas.

El siguiente *benchmark* aplicado es el de Stanford a continuación un ejemplo de su carga y ejecución sobre el SoC implementado:

```
grmon2> load stanford
  40000000 .text                    85.5kB /  85.5kB   [===============>] 100%
  40012E20 .data                     2.7kB /   2.7kB   [===============>] 100%
  Total size: 88.23kB (2.14Mbit/s)
  Entry point 0x40000000
  Image /home/fernandw/Downloads/benchmarks/stanford loaded


grmon2> run
Starting
    Perm  Towers  Queens   Intmm      Mm  Puzzle   Quick  Bubble    Tree     FFT
     150     150     100     117    1317     800     116     184     383    1566
Nonfloating point composite is       285
Floating point composite is      1365

Program exited normally.
```

De igual forma este *benchmark* se ejecutó sin SO con el fin de realizar una evaluación enfocada solo a caches, se aplicó a las configuraciones establecidas a 80MHz que es nuestra mejor frecuencia de operación obtenida.

Tabla 5.5 Tabla de resultados Stanford, evaluación caches

Bench	Conf-1	Conf-2	Conf-3
Run time (ms)			
Perm	83	83	83
Towers	100	83	100
Queens	67	50	50
Intmm	84	67	84
Mm	100	100	100
Puzzle	550	500	500
Quick	84	67	67
Bubble	133	116	117
Tree	300	233	233
FFT	150	133	133
Nonfloating	200	**175**	**178**
Floating point	296	**264**	**266**

El Stanford es un *benchmark* más largo y con múltiples accesos a memoria (más que el Dhrystone 2.1) por lo que nos permite estresar el sistema de memoria. Los resultados obtenidos nos exponen que nuevamente Conf-2 y Conf-3 son las configuraciones con mejores tiempos de ejecución (el menor tiempo es mejor).

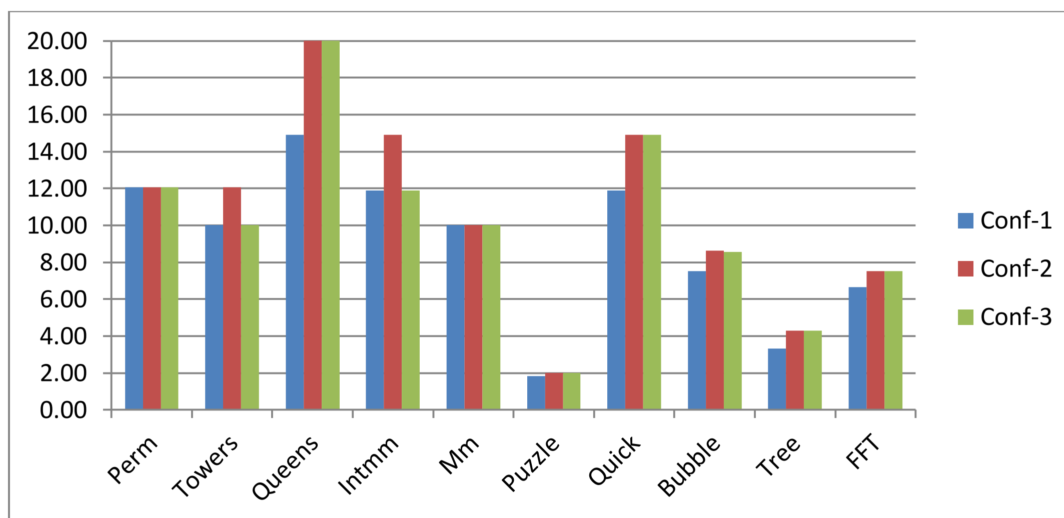


Fig. 5.1 Gráfica de resultados Stanford de comparación de configuraciones de caches. Los resultados son iteraciones/seg.

Otro de los objetivos de selección es lograr una configuración que muestre el mejor rendimiento con un bajo costo de elementos lógicos y memoria RAM. Por lo anterior en aspectos de implementación los recursos utilizados en Conf-3 son menores y nos dan una directiva por esta selección y esta es implementada en las siguientes evaluaciones.

Tabla 5.6 Tabla de recursos implementados enfocados a caches

	Cong-2		**Conf-3**	
	%	Total	%	Total
Total Logic Elements	43	49,038	39	**44,261**
Total memory bits (KB)	10	47.72	8	**41.21**

La segunda parte de la evaluación es la MMU la cual gestiona la MV del SO, a continuación se muestra un ejemplo de la carga de SO Linux 2.6 y su ejecución sobre el SoC implementado:

```
grmon2> load image.ram
 40000000 .text              4.2kB /   4.2kB [===============>] 100%
 400010B0 .data              80B              [===============>] 100%
 40004000 .vmlinux           4.7MB /  4.7MB   [===============>] 100%
 404B6EB0 .startup_prom      31.5kB / 31.5kB   [===============>] 100%
 Total size: 4.73MB (2.26Mbit/s)
 Entry point 0x40000000
 Image /home/fernandw/Downloads/linuxbuild/out/images/image.ram loaded

grmon2> run
OF stdout device is: /a::a
Booting Linux...
Linux version 2.6.36.4 (fernandw@fernandw) (gcc version 4.4.2 (crosstool-NG-) ) #2 Tue
May 14 18:58:03 CDT 2013
ARCH: LEON
TYPE: Leon3 System-on-a-Chip
Ethernet address: 00:00:7c:cc:01:45
CACHE: 4-way associative cache, set size 4k
Boot time fixup v1.6. 4/Mar/98 Jakub Jelinek (jj@ultra.linux.cz). Patching kernel for
srmmu[LEON]/iommu
OF stdout device is: /a::a
PROM: Built device tree with 10396 bytes of memory.
Built 1 zonelists in Zone order, mobility grouping on.  Total pages: 31277
Kernel command line: console=ttyS0,38400 init=/sbin/init
PID hash table entries: 512 (order: -1, 2048 bytes)
Dentry cache hash table entries: 16384 (order: 4, 65536 bytes)
Inode-cache hash table entries: 8192 (order: 3, 32768 bytes)
Memory: 122792k/131048k available (1532k kernel code, 8256k reserved, 612k data, 2652k
init, 0k highmem)
Hierarchical RCU implementation.
       RCU-based detection of stalled CPUs is disabled.
       Verbose stalled-CPUs detection is disabled.
```

```
Console: colour dummy device 80x25
console [ttyS0] enabled
Calibrating delay loop... 49.66 BogoMIPS (lpj=248320)
pid_max: default: 32768 minimum: 301
Mount-cache hash table entries: 512
bio: create slab <bio-0> at 0
ROMFS MTD (C) 2007 Red Hat, Inc.
msgmni has been set to 239
io scheduler noop registered
io scheduler deadline registered
io scheduler cfq registered (default)
Serial: GRLIB APBUART driver
ffd0dbc8: ttyS0 at MMIO 0x80000100 (irq = 2) is a GRLIB/APBUART
grlib-apbuart at 0x80000100, irq 2
leon: power management initialized
/home/fernandw/Downloads/Linux_FPU/linuxbuild-1.0.6/linux/linux-2.6-
git/drivers/rtc/hctosys.c: unable to open rtc device (rtc0)
Freeing unused kernel memory: 2652k freed
Starting logging: OK
Initializing random number generator... done.
Starting network...
ip: socket: Function not implemented
ip: socket: Function not implemented

Welcome to Buildroot
buildroot login: root
login[67]: root login on 'ttyS0'
# dhrystone

Dhrystone Benchmark, Version 2.1 (Language: C)
Program compiled without 'register' attribute
Please give the number of runs through the benchmark: 400000
Execution starts, 400000 runs through Dhrystone
Execution ends
Final values of the variables used in the benchmark:
Int_Glob:            5
        should be:   5
Bool_Glob:           1
        should be:   1
Ch_1_Glob:           A
        should be:   A
Ch_2_Glob:           B
        should be:   B
Arr_1_Glob[8]:       7
        should be:   7
Arr_2_Glob[8][7]:    400010
        should be:   Number_Of_Runs + 10
Ptr_Glob->
  Ptr_Comp:          151560
        should be:   (implementation-dependent)
  Discr:             0
        should be:   0
  Enum_Comp:         2
        should be:   2
  Int_Comp:          17
        should be:   17
  Str_Comp:          DHRYSTONE PROGRAM, SOME STRING
        should be:   DHRYSTONE PROGRAM, SOME STRING
Next_Ptr_Glob->
  Ptr_Comp:          151560
        should be:   (implementation-dependent), same as above
  Discr:             0
        should be:   0
```

```
  Enum_Comp:          1
        should be:   1
  Int_Comp:           18
        should be:   18
  Str_Comp:           DHRYSTONE PROGRAM, SOME STRING
        should be:   DHRYSTONE PROGRAM, SOME STRING
Int_1_Loc:            5
        should be:   5
Int_2_Loc:            13
        should be:   13
Int_3_Loc:            7
        should be:   7
Enum_Loc:             1
        should be:   1
Str_1_Loc:            DHRYSTONE PROGRAM, 1'ST STRING
        should be:   DHRYSTONE PROGRAM, 1'ST STRING
Str_2_Loc:            DHRYSTONE PROGRAM, 2'ND STRING
        should be:   DHRYSTONE PROGRAM, 2'ND STRING

Microseconds for one run through Dhrystone:   14.2
Dhrystones per Second:                      70298.8

# whetstone

Loops: 1000, Iterations: 1, Duration: 4 sec.
C Converted Double Precision Whetstones: 25.0 MIPS
```

Con el fin de evaluar el Sistema con la MMU se ejecutaron sobre el SO los *benchmark* de Dhrystone y Whetstone a la frecuencia de operación máxima lograda que es de 80MHz. Los resultados obtenidos de estos *benchmarks* y de síntesis son mostrados a continuación:

Tabla 5.7 Tabla de resultados de *benchmarks* enfocados a MMU/SO

DHRYSTONE	MMU-1	**MMU-2**	MMU-3
Frecuencia	*80*	*80*	*80*
Tiempo por un Dhrystone (iteración)	8.9	8.9	14.2
Dhrystones por segundo	112676.1	112676.1	70298.8
Dhrystones MIPS	**64.12**	**64.12**	40.01
WHETSTONE			
Converted Double Precision Whetstones:	**50.0**	**50.0**	50.0

Es de observarse que con el SO Linux 2.6 el mejor resultado de ejecución es de 64.12 MIPS y en esta misma arquitectura y frecuencia, sin SO el mejor resultado es de 112.7 MIPS, esto es debido al proceso de traducción de DF a DV y los múltiples procesos que se ejecutan en las caches VIVT.

Tabla 5.8 Tabla de recursos implementados por la arquitectura SoC/MMU

	MMU-1		**MMU-2**		MMU-3		Ciclone IV-115	
	%	Total	%	Total	%	Total	%	Total
Total Logic Elements	35	39,149	40	45,826	38	43,071	100	114480
Total memory bits (KB)	8.5	40.98	8.5	41.22	8.5	41.22	100	486

De acuerdo a estos resultados obtenidos las MMU-1 y MMU-2 son las de mejor rendimiento. Se selecciona MMU-2 ya que implementa los TLBs de mayor número de entradas (64 entradas) y los recursos usados son aceptables y comparables con las demás arquitecturas, además de que es fácilmente sintetizable, debido a la capacidad del FPGA.

Después de seleccionar nuestro Sistema de memoria (MMU y caches) ejecutamos el *benchmark* lat_mem_rd en nuestra plataforma (SoC + SOLinux) con el objetivo de conocer las latencias reales del sistema de memoria. Como se expresó anteriormente en la sección 5.5.4 el *benchmark* consiste en crear una matriz en memoria y después accesar a ella (múltiples loads) del último al primer dato avanzando N bytes a través de toda la matriz, el test mide en nanosegundos las latencias del sistema según el área de memoria a la que se accesa. Para esta evaluación la matriz generada es de 1MB y el avance es de 32B que es el tamaño de línea en la cache.

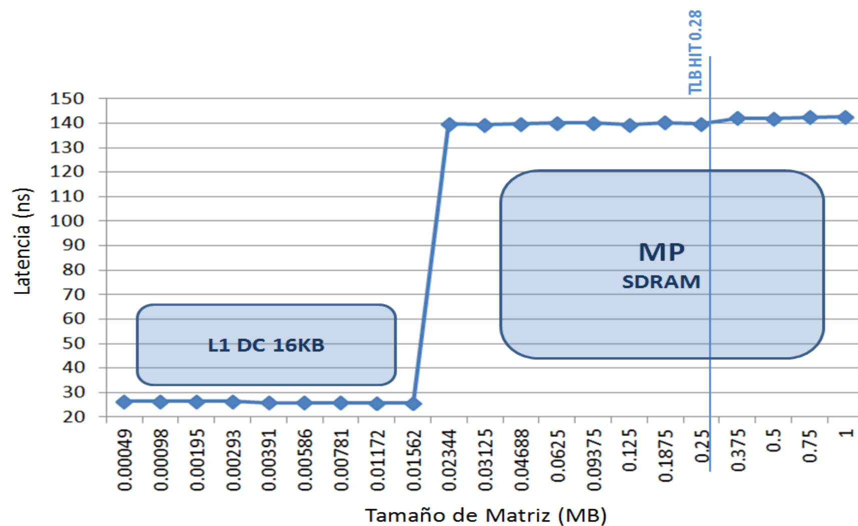


Fig. 5.2 Gráfica de resultados *benchmark* lam_mem_rd latencias (ns).

De la gráfica anterior podemos concluir que los accesos a la DC-16KB son en promedio de 26ns (2 ciclos) con el sistema operando a 80MHz. Las latencias de acceso a la memoria principal son mucho mayores en promedio 140ns (11 ciclos), debido a que es una memoria más lenta y antes de accesar a ella se debe traducir la DV a DF a través de la MMU. La MMU cuenta con una TLB de 64 entradas las cuales mapean a páginas de 4KB, lo que nos daría un total aproximado de 262KB de *hits* en TLB. En los 768KB restantes se incrementa un poco la latencia debido a que cada 4KB hay un *miss* o fallo en TLB y se procede a buscar en la TP (ciclos extra en cada acceso) hasta completar la matriz de 1MB.

Actualmente el performance de este tipo de sistemas embebidos en FPGA por su tecnología y frecuencia máxima de operación, se sigue comparando con los procesadores de antaño 90's, en el sitio web de LMbench podemos encontrar estas evaluaciones.

Tabla 5.9 Tabla comparativa de latencias (ns) del sistema de memoria.

CPU	OS	Mhz	L1	L2	MP
6000-990	AIX 3.x	70	13	-	141
Powerpc	AIX 4.x	133	6	164	394
K210	HP-UX B.10.01	119	8	-	349
R10K	IRIX64 6.2	200	5	55	1115
P5-133	FreeBSD 2.2-C	132	7	81	182
P6	Linux 1.3.37	200	10	53	179
Alpha	Linux 1.3.57	136	3	83	357
8400	OSF1 V3.2	302	3	42	396
alpha	OSF1 V3.0	147	12	67	291
ultraspar	SunOS 5.5	166	6	42	270
Leon3/SoC	**Linux 2.6**	**80**	**26**	**-**	**140**

Como ya se dijo anteriormente el rendimiento de acceso a memoria es un factor muy importante del performance general del sistema. Otro factor es el ancho de banda de datos en la memoria, que se considera la cantidad de datos que pasa a través del sistema por unidad de tiempo. Para conocer esta característica, hacemos uso de la evaluación

bw_mem el cual crea una cantidad especificada de memoria y luego mide el tiempo de lectura y copiado en otra área de memoria. Nuestro promedio de ancho de banda es medido en Megabytes por segundo (MB/s) y los resultados obtenidos (lectura, escritura y lectura-escritura) se muestran en la siguiente grafica.

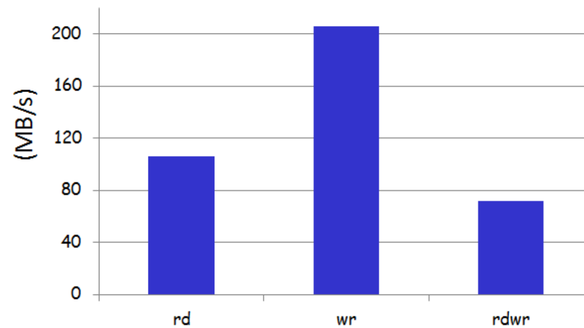


Fig. 5.3 Gráfica de resultados *benchmark* bw_mem ancho de banda (MB/s).

Tabla 5.10 Tabla comparativa de ancho de banda (MB/s) del sistema de memoria.

CPU	OS	RD	WR
6000-990	AIX 3.x	205	364
K210	HP-UX B.10.01	117	126
8400	OSF1 V3.2	120	123
ultraspar	SunOS 5.5	129	152
Leon/SoC	**Linux 2.6**	**106**	**206**

Conociendo estos valores logramos una mejor comprensión del funcionamiento del sistema. Además de ser útil para comparar entre diferentes sistemas de memoria o adecuar una posible configuración específica para cierto uso o aplicación.

5.7 Resumen del Capítulo

En este capítulo se presentaron los resultados obtenidos después de realizar una serie de simulaciones y evaluaciones correspondientes a las arquitecturas que se deseaban evaluar. Las conclusiones finales que se obtiene después de que se han analizado dichos resultados se presenta en el siguiente Capítulo.

CAPÍTULO 6.

CONCLUSIONES Y TRABAJO FUTURO

En este capítulo se exponen los resultados obtenidos a través de las evaluaciones, justificando cada uno de los objetivos propuestos en la tesis. Además se mencionan los trabajos futuros en relación al presente trabajo de tesis.

6.1 Conclusiones
6.2 Trabajo Futuro

CAPÍTULO 6 Conclusiones y Trabajo futuro

6.1 Conclusiones

En el presente trabajo se ha realizado el diseño de la arquitectura de una Unidad de Gestión de Memoria (MMU) para un procesador superescalar con ejecución fuera de orden. En este trabajo, la MMU tiene la característica de estar diseñada para una operación de hasta dos *fetch* de instrucciones y dos accesos a memoria (load/store) por ciclo de reloj.

Para este diseño se implemento satisfactoriamente un TLB hardware, su administración y manejo de fallas se realiza por completo mediante circuitos HW de la MMU. Las interrupciones, para el SO ocurren sólo cuando una página no se encuentra en memoria. Otros diseños MMU de softcores la administración del TLB es por SW, las entradas son cargadas y gestionadas por el SO, todo esto con interrupciones que al

ejecutarlas ocasionan cambios de contexto y hacen uso del pipeline del procesador reduciendo la velocidad del sistema.

Al haber evaluado las latencias del sistema de memoria podemos concluir que los TLBs de 64 entradas ayudan a obtener un buen performance, ya que con TLBs de menor número de entradas generan más accesos a las TP en MP y consecuencia de esto aumentan la latencia de acceso a la MP. Por otro lado, con TLBs de mayor número de entradas implementamos un gran número de recursos hardware, además incrementa la latencia en cada acceso a la MP ya que el circuito se vuelve más complejo y lento.

EL ITLB y el DTLB son circuitos totalmente asociativos sintetizados con memorias síncronas de alta velocidad y se implementa un sub-circuito para realizar un remplazo LRU. Ambos brindaron el mejor performance de ejecución en las evaluaciones, y que con ambas se logra mantener las DF (traducciones) más usadas, eliminando fallos de conflicto y por su configuración (número de entradas) su operación sigue siendo de alta velocidad.

Resulta muy costoso traer datos innecesarios a la MP, principalmente el desperdicio en ancho de banda. Por lo cual se utiliza paginación por demanda ya que con esta técnica solo traemos los marcos de página conforme son solicitados. De igual forma resulta eficiente ya que al ejecutar una imagen, el SO-Linux trae la primera mitad de datos a la MP y el resto es demandando en el proceso.

El uso de caches de tipo VIVT disminuye las latencias y ahorran energía del sistema de memoria, ya que no es necesario que se consulte primero la MMU para determinar la DF de una determinada DV y accesar a la memoria cache. Los procesadores de bajo consumo ARM las implementan.

Los diseños de estas caches implementadas en el sistema, fueron seleccionados por parámetros de diseño que influyen en su rendimiento y costo de implementación (tasa de

impacto y capacidad de almacenamiento). El tamaño de caches en SoC sobre FPGA suele ser entre 4K, 8K, 16K y 32K. Las evaluaciones realizadas determinaron el tamaño de las caches que es de 16K para ambas ya que puede retener más datos útiles sobre los tamaños de 4K y 8K, por otro lado, diseñar una cache igual o mayor a 32K es demasiada costosa e ineficiente es muy grande para el tipo de aplicaciones que se ejecutan en sistemas embebidos de esta clase. El ancho de línea seleccionado es de 32 bytes sobre 16 bytes, por tres razones: la primera de ellas es que con una línea más grande puede mejorar la tasa de impacto, la segunda razón es que el ancho es la unidad de transferencia de datos entre la cache y la memoria principal, por lo que con una línea más grande se llevan más datos a la cache con cada fallo y la razón principal es que con un tamaño de línea de 16 bytes los bloques de memoria implementados son mayores, ya que son necesarias mas líneas que contienen *tags* y bits de contexto por cada una de ellas, generando 47.72KB de memoria implementada contra 41.21KB de un ancho de 32 bytes. Se aplicaron distintas políticas de mapeo: mapeo directo (MD) y asociativo (*n-ways*), la de mejor desempeño en los *benchmarks* ejecutados es la asociativa de 4-*ways*, aplicar una cache de mayor asociatividad genera más costos de hardware casi 800LE por vía, además no brindó un mayor rendimiento como consecuencia de una operación más compleja y lenta. La política de reemplazo determina en cuál de los bloques en los que se puede mapear se debe sobreescribir. La política *Least Recently Used* (LRU) es actualmente una de las políticas de reemplazo más precisas (por encima de rand) pero a su vez, es una de las más costosas ya que requiere mantener un registro de lo que utiliza con bits extras para identificar el sub-bloque (palabra) menos usado de cada una de las líneas. Se implementaron las políticas de rand y LRU en la IC y DC respectivamente, la IC mostró el mismo rendimiento aplicando los *benchmarks* Dhrystone y Stanford con ambas políticas, LRU obtuvo 64.16MIPS y 64.12MIPS para rand esto, debido a que en la IC es menor la tasa de fallo ya que la cantidad de código del software que se ejecuta en este tipo de sistemas no suele ocasionar problemas por capacidad, además la mayoría de instrucciones son serializadas por lo cual

existen menores fallos por conflicto, haciendo no necesario un algoritmo muy preciso ya que raramente se recurre a él, seleccionando rand porque es la que menos recursos hardware utiliza.

El buen ancho de banda del sistema de memoria es resultado de su arquitectura Harvard, jerarquía que implementa (Caches virtuales L1 y MP SDRAM), además del tipo de transferencia de datos usado. Las transferencias entre las caches y MP es *burts* que completa la línea de cache de 32 bytes, la cual aumenta los *hits* de localidad espacial elevando potencialmente el ancho de banda del sistema.

Por lo anterior el Sistema de gestión de memoria demanda más investigación ya que son varios los aspectos que influyen en el desempeño del sistema, como son: el diseño, su configuración, operación y técnicas de arquitectura de computadoras.

La evaluación e investigación en el área de arquitectura de computadoras, se lleva a cabo por simulación en simuladores software, sin embargo, además de ser lentos, en un diseño digital HDL se logran analizar restricciones reales de comportamiento, área y consumo de energía, donde más tarde nuevas ideas puedan ser incorporadas al diseño y puedan ser sintetizadas a un dispositivo lógico programable FPGA de alta velocidad y capacidad. Sin embargo, los FPGA aún no son tan rápidos como los procesadores actuales y sus restricciones físicas determinan en gran medida el área y velocidad del circuito a simular y el prototipo del diseño y su depuración, demandan mayor tiempo que en un simulador software.

6.2 Trabajo Futuro

Como trabajo futuro se propone realizar un análisis para la elección de un tamaño de página óptimo en función del comportamiento típico de la aplicación, así como de la diferencia en las latencias y tasas de datos de la memoria principal y disco duro. En función de la aplicación se debe tomar en cuenta que cuando los accesos a memoria tienen gran cantidad de localidad espacial, un tamaño de página grande tiende a reducir el número de fallos obligatorios, debido al efecto de *prefetching.* No obstante, las mismas páginas más grandes desperdiciarán espacio y ancho de banda de memoria cuando haya poca o ninguna localidad espacial. Para este trabajo es necesario un *kernel* que cuente con un sistema de gestión de memoria con un tamaño de página configurable.

Continuar con la investigación de posibles mejoras en la microarquitectura de la MMU, así como la búsqueda de nuevas ideas de diseño a fin de mejorar el rendimiento de traducción, la reducción de bloques de memoria y recursos utilizados por el circuito, ampliar la jerarquía de memoria e implementar técnicas de arquitectura para lograr un sistema de memoria más eficiente del procesador, además de comenzar el diseño de la MMU y su sistema de memoria para arquitecturas multicore, lo anterior verificando mediante simulaciones que en conjunto logren un mayor incremento en el desempeño de esta área de la arquitectura del procesador.

Proponer y evaluar nuevas ideas en el diseño del software relacionado con el performance y escalabilidad del sistema de gestión de memoria del kernel de Linux, de diferentes arquitecturas de procesadores y de sistemas embebidos teniendo en cuenta la eficiencia de la transferencia de datos entre el software y el hardware, las arquitecturas multicore y la computación centrada en memoria compartida de los sistemas operativos de la próxima generación.

En este trabajo de tesis se generó el diseño HDL de la arquitectura de la MMU propuesta, otro objetivo propuesto sería el de desarrollar el diseño a nivel VLSI con tecnología CMOS y evaluar mediante simulación el circuito SPICE enfocado en áreas de interés como velocidad, área y el consumo de potencia para el diseño de procesadores modernos.

Posteriormente elaborar su correspondiente *Layout* usando un grupo de celdas digitales estándar que cumplan con las *Design Rules Checker* (DRC) para una tecnología CMOS (μm) del proceso de fabricación SUBM, utilizando herramientas Tanner / Mentor Graphics, el cual podrá unirse a diseños posteriores de otros módulos del procesador siguiendo esta misma metodología de diseño, hasta lograr obtener eventualmente un diseño completo de un procesador superescalar.

Referencias

[1] MIPS R10000 Microprocessor User's Manual Version 2.0, MIPS Technologies, Inc. 2011 North Shoreline.

[2] Pentium Processor Family Developer's Manual, Volume 3, COPYRIGHT © INTEL CORPORATION.

[3] Understanding 4M Page Size Extensions on the Pentium Processor, Robert R. Collins, http://www.rcollins.org/ddj/May96/

[4] Paging Extensions for the Pentium Pro Processor, Robert R. Collins, http://www.rcollins.org/ddj/Jul96/Jul96.html

[5] ARM architecture, http://en.wikipedia.org/wiki/ARM_architecture. September 7, 2012

[6] ARM926EJ-S Technical Reference Manual, Revision: r0p5, Copyright © 2008 ARM Limited.

[7] VIRTUAL-ADDRESS CACHES Part 1: Problems and Solutions in Uniprocessors, Michel Cekleov Sun Microsystems Michel Dubois University of Southern California. 1997 IEEE

[8] OpenCores the #1 community within open source hardware IP-cores. © copyright 1999-2012 OpenCores.org, equivalent to ORSoC. http://opencores.org/

[9] OpenRISC 1200 IP Core Specification, Damjan Lampret Rev. 0.7

[10] Memory management within the OpenRISC 1000 architecture.opyright 1999-2012 OpenCores.org http://opencores.org/or2k/Memory_management

[11] SPARC International, Inc. Copyright 1994-2012. http://www.sparc.org/specificationsDocuments.html

[12] The SPARC Architecture Manual Version 8, Revision SAV080SI9308, SPARC International, Inc. - Printed in U.S.A.

[13] Design and Implementation of RISC I, C.E. Sequin and D.A.Patterson, Electrical Engineering and Computer Sciencies, University of California, Berkeley.

[14] Understanding stacks and registers in the Sparc architecture, http://www.sics.se/~psm/sparcstack.html

[15] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne. Accelerating two-dimensional page walks for virtualized system. In ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems, pages 26–35, New York, NY, USA, 2008. ACM.

[16] Translation Caching: Skip, Don't Walk (the Page Table) Thomas W. Barr, Alan L. Cox, Scott Rixner. Rice University, Houston, TX. ISCA'10, June 2010, Saint-Malo, France.

[17] AMD x86-64 Architecture Programmer's Manual, Volume 2.

[18] Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide Part 1.

[19] Virtual mermory in contemporary microprocessors Bruce Jacob, University of Maryland, Trevor Mudge, University of Michigan, 1998 IEEE.

[20] Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers, Norman P. Jouppi, Digital Equipment Corporation Westem Research Lab, Palo Alto, CA. IEEE 1990.

[21] Introduction to the MIPS32 Architecture, Volume I-A. MIPS Technologies Inc.

[22] Data dependencies MIPS processor.
http://bnrg.eecs.berkeley.edu/~randy/Courses/CS252.S96/Lecture09.pdf

[23] MIPS32 Architecture For Programmers Volumen II, Silicon Graphics.

[24] Sistemas Operativos Modernos, 3era Edición Andrew S. Tanenbaum Editorial Pearson Prentice Hall.

[25] DE2-155 User Manual, Terasic-Altera 2010

[26] SoC architecture and design
https://www.doc.ic.ac.uk/~wl/teachlocal/cuscomp/notes/cc10.pdf

[27] Cross Compiler System AeroFLEX Gaisler,
http://gaisler.com/index.php/downloads/compilers?task=view&id=161

[28] BCC - Bare-C Cross-Compiler User's Manual, Version 1.0.41, July 2012, Authors: Jiri Gaisler, Konrad Eisele.

[29] Building the LINUX kernel for LEON, Written by Konrad Eisele, Daniel Hellstrom, Kungsgatan 12, 413 11 Gothenburg, Sweden.

[30] The Linux Kernel Sources Chapter 14, http://tldp.org/LDP/tlk/sources/sources.html

[31] GRLIB IP Core User's Manual, Version 1.1.0 - B4113, January 2012, Copyright Aeroflex Gaisler.

[32] AMBA Open Specifications, http://www.arm.com/products/system-ip/amba/amba-open-specifications.php

[33] GRLIB IP Library User's Manual, Version 1.2.2 - B4123, January 2013, Copyright Aeroflex Gaisler.

[34] SDRAM operation and characteristics, http://en.qi-hardware.com/wiki/SDRAM.

[35] Benchmarking in context: Dhrystone, By Richard York, ARM Ltd. March 2002

[36] Measuring Cache and Memory Latency and CPU to Memory Bandwidth For use with Intel® Architecture, Joshua Ruggiero, December 2008

Anexos

Anexo A Código del diseño MMU

Anexo B The SPARC Architecture Manual Version 8, Revision SAV080SI9308

Anexo C LEON/GRLIB, Configuration and Development Guide, December 2012.

Anexo D AMBA™ Specification, Rev. 2.0 ARM ARM IHI 0011A Limited 1999