



Contenidos

1. Introducción	5
1.1 ¿Qué se entiende por Arquitectura de un Computador?	6
1.2 Estructura del Procesador	7
1.2.1 Microprocesadores CISC vs. RISC	7
1.2.2 Arquitectura del Procesador: Von Neumann y Harvard	15
1.2.2.1 Máquinas Síncronas	17
1.2.3 Arquitectura básica de un procesador	18
1.2.4 Unidad Aritmético-Lógica (ALU)	20
1.2.5 Anatomía de procesadores modernos	21
1.2.5.1 Procesador “pipelined”	23
1.2.5.2 Riesgos (hazard) en procesadores “pipelined”	29
1.3 Memorias	32
1.3.1 SRAM y DRAM	32
1.4 Diseño de Circuitos Integrados: ASIC y PFGA	35
2. Introducción al ARM	36
2.1 Breve reseña histórica	37
2.2 Evolución Tecnológica	37
2.3 Versiones de la Arquitectura del ARM	39
2.4 Características Generales	40
2.5 Arquitectura del núcleo del ARM8	42
2.6 Arquitectura del núcleo del ARM9	42
2.7 Núcleo (CORE) del ARM8	43
2.8 Núcleo (CORE) del ARM9	44

3. Modelo del programador	45
3.1 Configuración Hardware	46
3.1.1 La señal BIGEND	46
3.1.2 Generación de estados de espera	47
3.2 Modos de Operación	48
3.3 Banco de Registros	49
3.3.1 Registro de Estado del Programa (PSR)	50
3.4 Excepciones	52
3.4.1 FIQ	52
3.4.2 IRQ	53
3.4.3 Fallos de Memoria (Memory ABORT)	53
3.4.4 Interrupción Software (SWI)	53
3.4.5 Instrucciones no reconocidas	54
3.4.6 Sumario de Vectores de Interrupción	54
3.4.7 Prioridades	55
3.5 Reset Global	55
4. Juego de Instrucciones	56
4.1 Sumario del juego de instrucciones del ARM	57
4.1.1 Instrucciones implementadas en el CORE	57
4.1.2 Instrucciones reservadas y restricciones de uso	58
4.2 Ejecución Condicional: el campo de condición	59
4.3 Saltos (B, BL)	60
4.4 Instrucciones de proceso de datos	61
4.5 Instrucciones de transferencia con el PSR (MRS, MSR)	68
4.5.1 Operandos en MSR	68
4.5.2 Restricciones en los operandos	68
4.5.3 Bits Reservados	69
4.6 Multiplicaciones (MUL, MLA)	70
4.7 Multiplicaciones de 64 bits (MULL, MLAL)	71
4.8 Transferencia simple de datos con memoria (LDR, STR)	72
4.9 Transferencia compleja de datos con memoria (halfwords)	76
4.9.1 Direccionamiento autoindexado y offset	76
4.9.2 Modos de funcionamiento	78
4.10 SWP (Simple Data Swap)	80
4.11 Fallos de Memoria (Data Aborts)	81
4.12 SWI (Software Interrupt)	82
4.13 Instrucciones no reconocidas	83
5. Diseñando el CORE	84
5.1 Consideraciones Iniciales	85
5.1.1 Alcance del proyecto	85
5.1.2 Decisiones sobre la arquitectura	87
5.1.3 Algoritmo de diseño	90

5.2	Descomposición previa del diseño	91
5.3	Arquitectura del CORE	94
5.3.1	Reducción de buses en la ruta de datos	99
5.4	Generación de “ <i>bubbles</i> ” y vaciado de la <i>pipeline</i>	100
5.5	ALU	105
5.5.1	Códigos de Operación	113
5.6	Diseñando la “ <i>pipeline</i> ”	115
5.6.1	Fase I	122
5.6.2	Fase II	130
5.6.3	Generación de estados de espera	135
5.7	Interfaz de memoria	141
5.7.1	Creando una interfaz personalizada	143
5.7.2	Accediendo a la memoria de programa	145
5.8	Introducción del control de excepciones	146
5.9	Vista del <i>pin-out</i> final del CORE	149
5.10	Layout de una mega-celda del microprocesador	150
5.10.1	Optimización del diseño	151
5.11	Observaciones	153
5.11.1	Unidad de predicción de saltos	153
6.	Simulaciones	154
6.1	Introducción a la fase de simulaciones	155
6.2	Simulando la ALU	156
6.3	Usando VHDL Simili	157
6.3.1	Generando las entradas: aplicación “Txt2vhdl”	158
6.3.2	Ficheros de salida	159
6.4	Emulando el multiplicador	161
6.5	Simulando la <i>pipeline</i>	166
6.5.1	Verificando la FASE I	167
6.5.2	Verificando la FASE II	169
6.6	Comprobando el control de interrupciones	171
6.7	Ejecutando un programa completo	174
6.8	Simulación del Prototipo de pruebas	189
6.9	Ubicación y descripción de los archivos de test	190
7.	Implementación	192
7.1	Comentarios sobre el tamaño inicial	193
7.2	Planteamiento del prototipo	194
7.3	Diseño hardware del prototipo	197
7.4	Realización del programa	198
7.4.1	Memoria de Datos	199
7.4.2	Calculando los parámetros del programa	199
7.5	Implementación en la Virtex 800	200
7.6	Prueba Final	202

A. Ficheros	203
A.1 Ficheros de Simulación del CORE	204
A.2 Ficheros de Simulación del Prototipo	206
A.3 Ficheros para Implementación	207
A.4 Ficheros VHDL	209
A.5 Organización del CD-ROM	210
 B. Uso del diseño	 211
B.1 Entradas a Configurar	212
B.2 Alineación de direcciones	215
B.3 Señal de salida <i>Privileged</i>	216
 C. Bibliografía	 217
C.1 Información referente al ARM	218
C.2 Manuales de Xilinx	218
C.3 Artículos y Publicaciones on-line	218
C.4 Otras obras relacionadas	219
 D. Código VHDL	 220

1

Introducción

En este apartado se pretende abordar cualitativamente el estudio teórico del diseño de computadores, sobre todo en lo que a máquinas RISC se refiere, mostrando las distintas tendencias y arquitecturas de procesadores, para posteriormente justificar las decisiones adoptadas en la elaboración de este proyecto.

1.1 ¿Qué se entiende por Arquitectura de un Computador?

1.2 Estructura del Procesador

1.2.1 Microprocesadores CISC vs. RISC

- Introducción
- Arquitecturas CISC
- Arquitecturas RISC
- Principios de diseño de las máquinas RISC
- El papel de los compiladores en sistemas RISC
- Capacidad de procesamiento de los sistemas desde el punto de vista del usuario
- Aplicaciones de los procesadores RISC
- Conclusiones

1.2.2 Arquitectura del Procesador: Von Neumann y Harvard

1.2.2.1 Máquinas Síncronas

1.2.3 Arquitectura básica de un procesador

1.2.4 Unidad Aritmético-Lógica (ALU)

1.2.5 Anatomía de procesadores modernos

1.2.5.1 Procesador “pipelined”

1.2.5.2 Riesgos (Hazard) en procesadores “pipelined”

1.3 Memorias

1.3.1 SRAM y DRAM

1.4 Diseño de Circuitos Integrados: ASIC y FPGA

1.1 ¿Qué se entiende por Arquitectura de un Computador?

$$\begin{array}{ccccc} \text{Arquitectura} & & \text{Arquitectura del Juego de Instrucciones} \\ \text{de un} & = & + \\ \text{Computador} & & \text{Organización de la Máquina} \end{array}$$

Arquitectura del Juego de Instrucciones (ISA)

La Arquitectura del Juego de Instrucciones (Instruction Set Architecture ó ISA) describe la estructura de un computador desde el punto de vista del programador. Cuando una familia de procesadores ejecuta el mismo código binario, se dice que tienen la misma arquitectura, refiriéndose éste termino más concretamente al ISA.

Estos procesadores pueden contar con una organización interna totalmente distinta, pero todos ellos aparecerán como idénticos al programador, porque sus juegos de instrucciones son el mismo.

Es posible construir lo que se denomina un “procesador de microcódigo”, que puede ejecutar múltiples ISAs. El término microcódigo (microcode) se refiere al más bajo nivel programable de una máquina, y normalmente se considera parte del hardware.

Organización de la Máquina

Con esta terminología nos referimos al *layout* e interconexiones de varias unidades funcionales. Procesadores con una amplia gama de prestaciones, pero el mismo ISA; pueden ser contruidos simplemente alterando detalles tales como:

- Longitud de la estructura *pipeline*
- Número de unidades funcionales
- Tamaño de *cache* y organización
- Sofisticación de la unidad de selección de instrucción a cursar

La Arquitectura del Juego de Instrucciones y la Organización de la Máquina no tienen por qué estar necesariamente relacionadas. De hecho, detalles como la organización de una máquina son normalmente transparentes al *software*. Los programas compilados para un mismo ISA se ejecutarán de la misma forma en máquinas con diferente estructura interna, aunque probablemente la velocidad de ejecución varíe de una arquitectura a otra.

Un ejemplo claro de este hecho se pone de manifiesto en el uso de *cache*, que es totalmente transparente al software: un programa se ejecutará del mismo modo en sistemas con o sin *cache*.

1.2 Estructura del Procesador

1.2.1 Microprocesadores CISC vs. RISC

Hoy en día, los programas cada vez más grandes y complejos demandan mayor velocidad en el procesamiento de la información, lo que implica la búsqueda de microprocesadores más rápidos y eficientes.

Los avances y progresos en la tecnología de semiconductores, han reducido las diferencias en la velocidades de procesamiento de los microprocesadores con las velocidades de las memorias, lo que ha repercutido en nuevas tecnologías en el desarrollo de microprocesadores. Hay quienes consideran que en breve los microprocesadores RISC (reduced instruction set computer) sustituirán a los CISC (complex instruction set computer), pero existe el hecho de que los microprocesadores CISC tienen un mercado de software muy difundido, aunque tampoco tendrán ya que establecer nuevas familias en comparación con el desarrollo de nuevos proyectos con microcódigo en tecnología RISC.

La arquitectura RISC plantea en su filosofía de diseño una relación muy estrecha entre los compiladores y la misma arquitectura, como se verá más adelante.

Introducción

Veamos primero cuál es el significado de los términos CISC y RISC:

- CISC (complex instruction set computer): computadoras con un conjunto o juego de instrucciones complejo
- RISC (reduced instruction set computer): computadoras con un conjunto o juego de instrucciones reducido.

Los atributos complejo y reducido describen las diferencias entre los dos modelos de arquitectura para microprocesadores sólo de forma superficial. Se requiere de muchas otras características esenciales para definir los RISC y los CISC típicos. Aún más, existen diversos procesadores que no se pueden asignar con facilidad a ninguna categoría determinada.

Así, los términos complejo y reducido, expresan muy bien una importante característica definitiva, siempre que no se tomen sólo como referencia las instrucciones, sino que se considere también la complejidad del hardware del procesador.

Con tecnologías de semiconductores comparables e igual frecuencia de reloj, un procesador RISC típico tiene una capacidad de procesamiento de instrucciones de dos a cuatro veces mayor que la de un CISC, pero su estructura de hardware es tan simple, que se puede realizar en una fracción de la superficie ocupada por el circuito integrado de un procesador CISC.

Esto hace suponer que RISC reemplazará al CISC, pero la respuesta a esta cuestión no es tan simple, ya que:

- Para aplicar una determinada arquitectura de microprocesador son decisivas las condiciones de realización técnica y sobre todo la rentabilidad, incluyendo los costos de software.
- Existían y existen razones de compatibilidad para desarrollar y utilizar procesadores de estructura compleja, así como un extenso conjunto de instrucciones.

La meta principal es incrementar el rendimiento del procesador, ya sea optimizando alguno existente, o se desee crear uno nuevo. Para esto se deben considerar tres áreas principales a cubrir en el diseño del procesador, y estas son:

- La arquitectura.
- La tecnología del proceso: se refiere a los materiales y técnicas utilizadas en la fabricación del circuito integrado
- El encapsulado: se refiere a cómo se integra un procesador con lo que le rodea en un sistema funcional, que de alguna manera determina la velocidad total del sistema.

Aunque la tecnología de proceso y de encapsulado son vitales en la elaboración de procesadores más rápidos, es la arquitectura del procesador lo que marca la diferencia entre el rendimiento de una CPU y otra. Y es en la evaluación de las arquitecturas RISC y CISC donde centraremos nuestra atención.

Dependiendo de cómo el procesador almacene los operandos de las instrucciones de la CPU, existen tres tipos de juegos de instrucciones:

1. Juego de instrucciones para arquitecturas basadas en pilas.
2. Juego de instrucciones para arquitecturas basadas en acumulador.
3. Juego de instrucciones para arquitecturas basadas en registros.

Perspectiva Histórica

Es importante tener en cuenta las condiciones tecnológicas existentes en el preciso marco temporal en el que se desarrollan ambas estrategias de diseño. Cada una de ellas intenta aprovechar de la forma más eficiente posible los recursos tecnológicos disponibles a la hora de abordar el diseño de una máquina.

Para comprender el contexto histórico y tecnológico en el que RISC y CISC se desarrollan, es necesario tener en cuenta en primer lugar el estado de la técnica en VLSI, memorias y compiladores, a finales de los setenta y principios de los ochenta.

- ❑ Almacenamiento / Memorias: en los setenta, los computadores usaban memorias de núcleo magnético para almacenar el código de un programa. La memoria no sólo era cara, sino que además era terriblemente lenta. El alto coste de la memoria principal, y la lentitud del almacenamiento secundario, provocaron que un código fuera bueno si era suficientemente *compacto*. La aparición de la RAM mejoró la velocidad de almacenamiento en memoria, y redujo sensiblemente el coste total del sistema.
- ❑ Compiladores: el trabajo de los compiladores era bastante simple, y consistía en traducir código escrito en lenguajes de alto nivel, como C ó PASCAL, a lenguaje ensamblador. Los ensambladores se encargaban después de convertir el lenguaje ensamblador a código máquina. La compilación suponía un tiempo elevado, y el resultado de la operación difícilmente era óptimo. Si realmente se quería un código compacto y optimizado, la única alternativa era codificar directamente en ensamblador (hoy en día esta afirmación aún sigue siendo válida).
- ❑ VLSI: la densidad de transistores en el estado en el que se encontraba aún la tecnología VLSI (Very Large Scale Integration) era bastante baja con respecto a los estándares actuales.

Arquitecturas CISC

La microprogramación es una característica importante y esencial en casi todas las arquitecturas CISC, como por ejemplo:

- Intel 8086, 8088, 80286, 80386, 80486
- Motorola 68000, 68010, 68020, 68030, 68040

La microprogramación significa que cada instrucción de máquina es interpretada por un microprograma localizado en una memoria dentro del circuito integrado del procesador.

En la década de los sesenta, la microprogramación, por sus características, era la técnica más apropiada para las tecnologías de memorias existentes en esa época, y permitía desarrollar también procesadores con compatibilidad ascendente. En consecuencia, los procesadores se dotaron de poderosos conjuntos de instrucciones.

Las instrucciones compuestas son decodificadas internamente y ejecutadas con una serie de microinstrucciones almacenadas en una ROM interna. Para esto, se requieren de varios ciclos de reloj (al menos uno por microinstrucción).

Arquitecturas RISC

Buscando aumentar la velocidad del procesamiento se descubrió en base a la experiencia que, con una determinada arquitectura de base, la ejecución de programas compilados directamente con microinstrucciones y residentes en memoria externa al circuito integrado, resultaban ser más eficientes gracias a que el tiempo de acceso de las memorias se fue decrementando conforme se mejoraba su tecnología.

Debido a que se tiene un conjunto de instrucciones simplificado, éstas se pueden implantar por hardware directamente en la CPU, lo cual elimina el microcódigo y la necesidad de decodificar instrucciones complejas.

En investigaciones hechas a mediados de la década de los setentas, con respecto a la frecuencia de utilización de una instrucción en un CISC y al tiempo para su ejecución, se observó lo siguiente:

- Alrededor del 20% de las instrucciones ocupa el 80% del tiempo total de ejecución de un programa
- Existen secuencias de instrucciones simples que obtienen el mismo resultado que secuencias complejas predeterminadas, pero requieren tiempos de ejecución más cortos.

Las características esenciales de una arquitectura RISC pueden resumirse como sigue:

- Estos microprocesadores siguen tomando como base el esquema moderno de Von Neumann.
- Las instrucciones, aunque con otras características, siguen divididas en tres grupos:
 - a) Trasferencia
 - b) Operaciones
 - c) Control de flujo
- Reducción del conjunto de instrucciones a instrucciones básicas, con las que pueden implantarse todas las operaciones complejas.
- Arquitectura del tipo *load-store* (carga y almacena). Las únicas instrucciones que tienen acceso a memoria son “load” y “store”, registro a registro, con un menor número de accesos a memoria.
- Casi todas las instrucciones pueden ejecutarse en un ciclo de reloj, base importante para la reorganización de la ejecución de instrucciones por medio de un compilador.
- *Pipeline* (ejecución simultánea de varias instrucciones): posibilidad de reducir el número de ciclos de máquina necesarios para la ejecución de la instrucción, ya que esta técnica permite que una instrucción pueda empezar a ejecutarse antes de que haya terminado la anterior.

El hecho de que la estructura simple de un procesador RISC conduzca a una notable reducción de la superficie del circuito integrado, se aprovecha con frecuencia para ubicar en el mismo funciones adicionales.

- Unidad para el procesamiento aritmético de punto flotante
- Funciones de control de memoria *cache*
- Unidad de administración de memoria
- Implantación de un conjunto de registros adicionales

La relativa sencillez de la arquitectura de los procesadores RISC conduce a ciclos de diseño más cortos cuando se desarrollan nuevas versiones, lo que posibilita siempre la aplicación de las más recientes tecnologías de semiconductores. Por ello, los procesadores RISC no solo tienden a ofrecer una capacidad de procesamiento del sistema de dos a cuatro veces mayor, sino que los saltos de capacidad que se producen de generación en generación son mucho mayores que en los CISC.

Por otra parte, es necesario considerar también que:

- La disponibilidad de memorias grandes, baratas y con tiempos de acceso menores de 60 ns en tecnología CMOS.
- Módulos SRAM (Memoria de acceso aleatorio estática) para memorias caché con tiempos de acceso menores a los 15 ns.
- Tecnologías de encapsulado que permiten realizar más de 120 pines. Esto ha hecho cambiar, en la segunda mitad de la década de los ochenta, esencialmente las condiciones técnicas para arquitecturas RISC:

Principios de diseño de las máquinas RISC

Resulta un tanto ingenuo querer abarcar completamente los principios de diseño de las máquinas RISC. Sin embargo, se intentará presentar de una manera general la filosofía básica de diseño de estas máquinas, teniendo en cuenta que dicha filosofía puede presentar variantes. Es muy importante conocer estos principios básicos, pues de éstos se desprenden algunas características importantes de los sistemas basados en microprocesadores RISC.

En el diseño de una máquina RISC, se siguen cinco pasos:

1. Analizar las aplicaciones para encontrar las operaciones *base*.
2. Diseñar un bus de datos que sea óptimo para las operaciones *base*.
3. Diseñar instrucciones que realicen las operaciones *base* utilizando el bus de datos.
4. Agregar nuevas instrucciones sólo si no hacen más lenta a la máquina.
5. Repetir este proceso para otros recursos.

El primer punto se refiere a que el diseñador deberá encontrar qué es lo que hacen en realidad los programas que se pretenden ejecutar. Ya sea que los programas a ejecutar sean del tipo algorítmicos tradicionales, o estén dirigidos a robótica o al diseño asistido por computador.

La parte medular de cualquier sistema es la que contiene los registros, la ALU y los buses que los conectan. Se debe optimizar este circuito para el lenguaje o aplicación en cuestión. El tiempo requerido (denominado tiempo el ciclo del bus de datos) para extraer los operandos de sus registros, mover los datos a través de la ALU y almacenar el resultado de nuevo en un registro, deberá hacerse lo más corto posible.

El siguiente punto a cubrir es diseñar instrucciones de máquina que hagan un buen uso del bus de datos. Por lo general se necesitan sólo unas cuantas instrucciones y modos de direccionamiento; sólo se deben colocar instrucciones adicionales si serán usadas con frecuencia y no reducen el desempeño de las más importantes.

Siempre que aparezca una nueva y atractiva característica, deberá analizarse y ver la forma en que se afecta al ciclo del bus. Si se incrementa el tiempo del ciclo, probablemente no vale la pena tenerla.

Por último, el proceso anterior debe repetirse para otros recursos dentro del sistema, tales como memoria *cache*, administración de memoria, coprocesadores de punto flotante, etcétera.

Una vez planteadas las características principales de la arquitectura RISC así como la filosofía de su diseño, podríamos extender el análisis y estudio de cada una de las características importantes de las arquitecturas RISC y las implicaciones que estas tienen.

Papel de los compiladores en un sistema RISC

El compilador juega un papel clave para la obtención de un sistema RISC equilibrado.

Todas las operaciones complejas se trasladan al microprocesador por medio de conexiones fijas en el circuito integrado para agilizar las instrucciones básicas más importantes. De esta manera, el compilador asume la función de un mediador inteligente entre el programa de aplicación y el microprocesador. Es decir, se hace un gran esfuerzo para mantener el hardware tan simple como sea posible, aún a costa de hacer el compilador considerablemente más complicado. Esta estrategia se encuentra en clara contraposición con las máquinas CISC, que tiene modos de direccionamiento muy complicados. En la práctica, la existencia de algunos modos de direccionamiento muy complicados en microprocesadores CISC, hacen que tanto el compilador como el microprograma sean bastante complejos. No obstante, las máquinas CISC no tiene características complicadas como carga, almacenamiento y salto que consumen mucho tiempo, las cuales aumentan efectivamente la complejidad del compilador.

Para suministrar datos al microprocesador de tal forma que siempre esté trabajando de forma eficiente, se aplican diferentes técnicas de optimización en distintos niveles jerárquicos del software.

Capacidad de procesamiento de los sistemas desde el punto de vista del usuario

Aparte de la base conceptual para el desarrollo de un sistema de computación de alta calidad, se requieren técnicas especiales para optimizar cada uno de los factores que determinan la capacidad de procesamiento, la cual sólo puede definirse con el programa de aplicación.

La información suministrada por un fabricante sobre la velocidad en MIPS (millones de instrucciones por segundo) que una arquitectura es capaz de realizar, carece de relevancia hasta que el usuario sepa cuántas instrucciones genera el respectivo compilador al traducir su programa de aplicación, y cuánto tiempo tarda la ejecución de estas instrucciones. Solo el análisis de diferentes pruebas y comparaciones de rendimiento (“benchmarks”) da una idea aproximada, que el usuario puede aplicar para delimitar las arquitecturas adecuadas.

Existen dos puntos de vista diferentes acerca de la capacidad de procesamiento del sistema:

- **Sistema Reprogramable**: un usuario que necesite desarrollar un sistema reprogramable no está interesado en obtener una alta capacidad de procesamiento.
- **Sistema incluido o dedicado**: en estos sistemas el principal objetivo es procesar de forma cíclica una serie de aplicaciones o funciones determinadas, y es de suma importancia la mayor cantidad posible de pruebas y comparaciones de rendimiento (“benchmarks”) diferentes.

Aplicaciones de los procesadores RISC

Las arquitecturas CISC utilizadas en los últimos 15 años han permitido desarrollar un gran número de productos de software. Ello representa una considerable inversión y asegura a estas familias de procesadores un mercado creciente.

Sin embargo, simultáneamente aumentan las aplicaciones en las cuales la capacidad de procesamiento que se pueda obtener del sistema es más importante que la compatibilidad con el hardware y el software anteriores, lo cual no sólo es válido en los subsistemas de alta capacidad en el campo de los sistemas llamados “embedded”, en los que siempre dominaron las soluciones especiales de alta capacidad de procesamiento, sino también para las estaciones de trabajo (“workstations”).

Esta clase de equipos se han ido introduciendo poco a poco en oficinas, en medicina y en los bancos, debido a los cada vez más voluminosos y complejos paquetes de software que con sus crecientes requerimientos de reproducción visual, sólo se encontraban en el campo técnico de la investigación y el desarrollo.

Conclusiones

Cada usuario debe decidirse a favor o en contra de determinada arquitectura de procesador en función de la aplicación concreta que desee realizar. Esto vale tanto para la decisión por una determinada arquitectura CISC o RISC, como para determinar si RISC puede emplearse de forma rentable para una aplicación concreta.

Nunca será decisiva únicamente la capacidad de procesamiento del microprocesador, y sí la capacidad real que puede alcanzar el sistema en su conjunto. Los costos, por su parte, también serán evaluados.

RISC ofrece soluciones atractivas donde se requiere una elevada capacidad de procesamiento y se presente una orientación hacia los lenguajes de alto nivel.

Sin embargo, en el campo industrial existe un gran número de aplicaciones que ni siquiera agotan las posibilidades de los controladores CISC de 8 bits actuales.

Si bien el campo de aplicaciones de las arquitecturas RISC de alta capacidad crece con fuerza, esto no equivale al fin de otras arquitecturas de procesadores y controladores acreditadas que también seguirán perfeccionándose, lo que sí resulta dudoso en la creación de familias CISC completamente nuevas. Adoptando técnicas típicas de los procesadores RISC en las nuevas versiones de procesadores CISC; se intenta encontrar nuevas rutas para el incremento de la capacidad de las familias CISC ya establecidas.

Entre tanto, los procesadores RISC han conquistado el sector de las estaciones de trabajo, dominado antes por los procesadores Motorola 68000 y es muy probable que acosen la arquitectura Intel en el sector superior de los PCs.

1.2.2 Arquitecturas del procesador: Von Neumann y Harvard

Antes de pasar a ver la estructura básica de un procesador, nos vemos obligados a revisar dos conceptos importantes: arquitectura Von Neumann y arquitectura Harvard. Dentro de este apartado introduciremos también el concepto de *máquina síncrona*.

Estructura Von Neumann

Al principio de los cincuenta, John von Neumann propuso el concepto de “computador con programa almacenado”, arquitectura que llegó a ser la base de la mayoría de procesadores que hoy en día podemos encontrar en el mercado.

En una máquina Von Neumann, el programa y los datos se almacenan en la misma memoria física, de tipo RAM o de tipo ROM, a las que accede secuencialmente la unidad central de procesos (CPU) a través de un bus de direcciones y otro de datos. La máquina consta de un contador de programa (PC) que apunta a la dirección de memoria de la instrucción actual. El PC es actualizado en cada instrucción, de modo que cuando no se producen *saltos*, las instrucciones que componen el programa son cargadas desde direcciones consecutivas de memoria de forma secuencial. Un *salto* consistirá básicamente en actualizar el PC con otra dirección de memoria cualquiera, dentro del espacio que ocupa el programa.

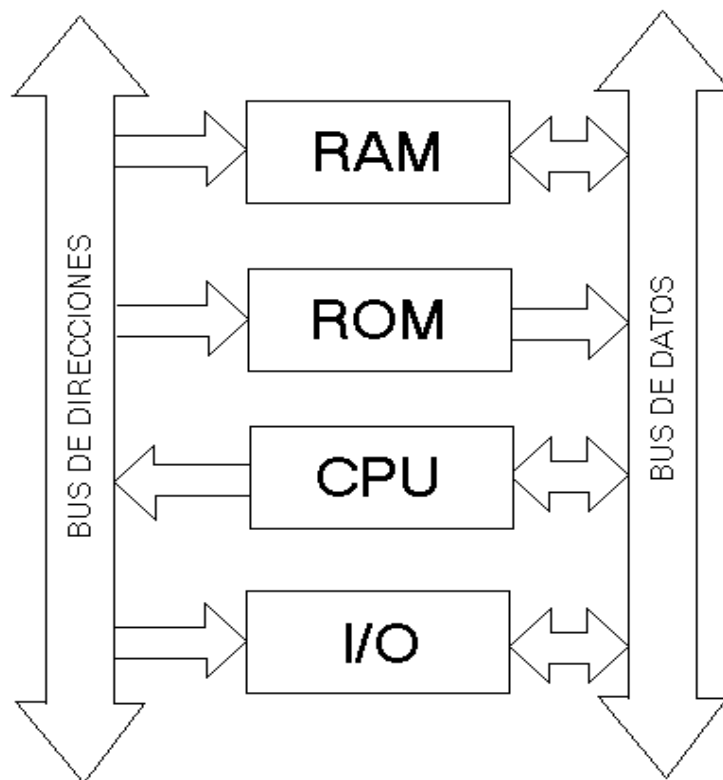


Figura 1-1: Arquitectura Von Neumann

La CPU lee las instrucciones de la memoria mediante el bus de datos y seguidamente ejecuta las instrucciones leídas previamente. El sistema microprocesador puede además leer y escribir datos en dispositivos externos mediante un buffer de

entrada-salida, de forma que el acceso a los periféricos externos es similar al acceso a la zona de memoria.

El sistema microprocesador, que sólo dispone de un bus de datos y otro de direcciones, accede a la memoria de programa para recoger la instrucción a ejecutar. Posteriormente decodifica la instrucción para determinar la secuencia de operaciones que ésta lleva asociada, y accede a la memoria de datos para leer los operandos asociados a la instrucción leída. Finalmente, la instrucción se ejecuta y comienza un nuevo ciclo.

Estructura Harvard

El bus de direcciones de acceso al programa es diferente del bus de direcciones de acceso a los datos. Las instrucciones son recibidas por la CPU a través de un bus de datos reservado para las instrucciones, separado del bus de datos propiamente dicho. La CPU lee las instrucciones de la memoria mediante el bus de datos y paralelamente ejecuta las instrucciones leídas previamente (esta estructura de acceso a datos y ejecución de instrucciones en paralelo es ideal para implementar lo que en terminología anglosajona se denomina *pipelining*).

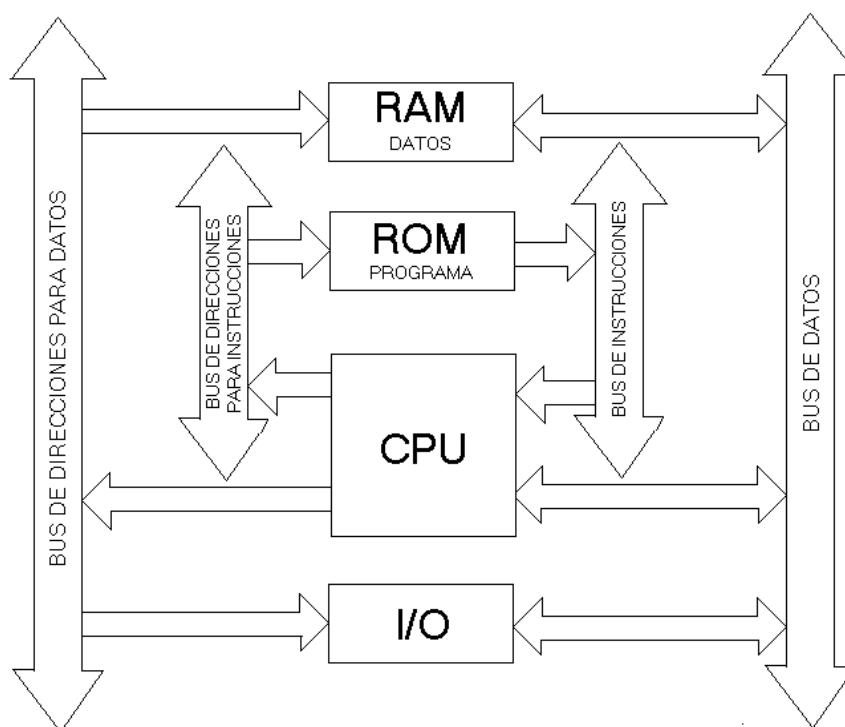
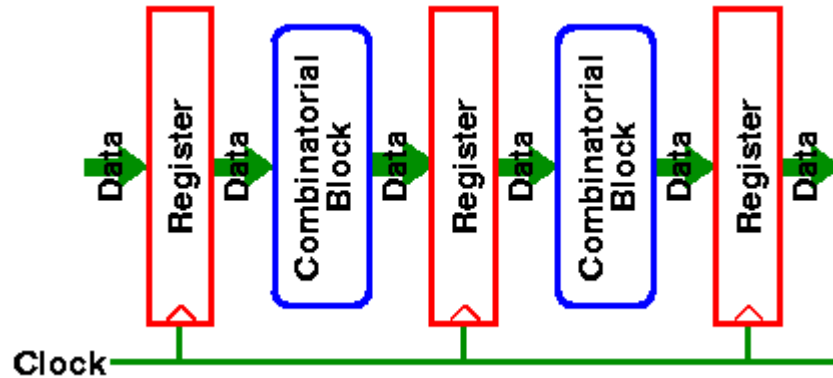


Figura 1-2: Arquitectura Harvard

Este tipo de estructuras es más complejo en hardware que las de tipo Von Neumann, pero permiten acelerar el tiempo efectivo de ejecución de la instrucción: la CPU prepara (fetch) instrucciones de programa mientras realiza la manipulación de datos de las instrucciones previamente recogidas. Esto es debido a que, con la estructura Harvard, el hardware destinado a procesar datos, y el destinado a ejecutar instrucciones, reside en diferentes partes de la CPU.

1.2.2.1 Máquinas Síncronas

La mayoría de las máquinas actuales son síncronas, esto es, son controladas por una señal de reloj.



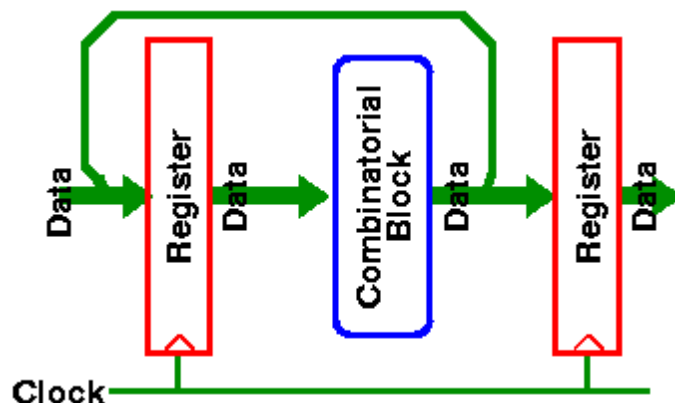
Bloques de lógica puramente combinacional y registros se alternan a lo largo de una cadena que conforma lo que se denomina estructura de datos. El dato avanza de un registro al siguiente con cada ciclo del reloj global del sistema: cuando se produce un flanco de reloj, la salida de un registro, tras ser procesada por el bloque combinacional, es capturada por el siguiente registro en lo que se denomina *pipeline*.

Los registros usan biestables *master-slave*, que permiten que la salida sea aislada de la entrada durante el proceso de carga de un nuevo dato en el registro.

En una máquina síncrona, el retardo de propagación es crítico. Sea $t_{pd}^{máx}$ el máximo retardo de propagación, y sea t_{cyc} el menor ciclo de reloj posible:

- si $t_{pd}^{máx} < t_{cyc}$ la salida de un registro es procesada por el bloque combinacional a tiempo para ser capturada por el siguiente registro cuando reciba el próximo flanco de reloj.
- en cambio, si $t_{pd}^{máx} > t_{cyc}$ el flanco de reloj llegará al registro antes que el dato se propague a través del bloque combinacional

En este tipo de máquinas es posible realizar bucles de realimentación, como los que se muestran en la figura: la salida una etapa cualquiera es *latcheada* por el mismo registro, mediante un lazo de realimentación. Este tipo de lógica se usa con frecuencia para determinar la próxima operación.



1.2.3 Arquitectura Básica de un Procesador

En este apartado vamos a tratar la estructura básica de un procesador simple. Examinaremos los distintos elementos que lo componen, dado que serán de gran ayuda a la hora de abordar el diseño de procesadores más complejos.

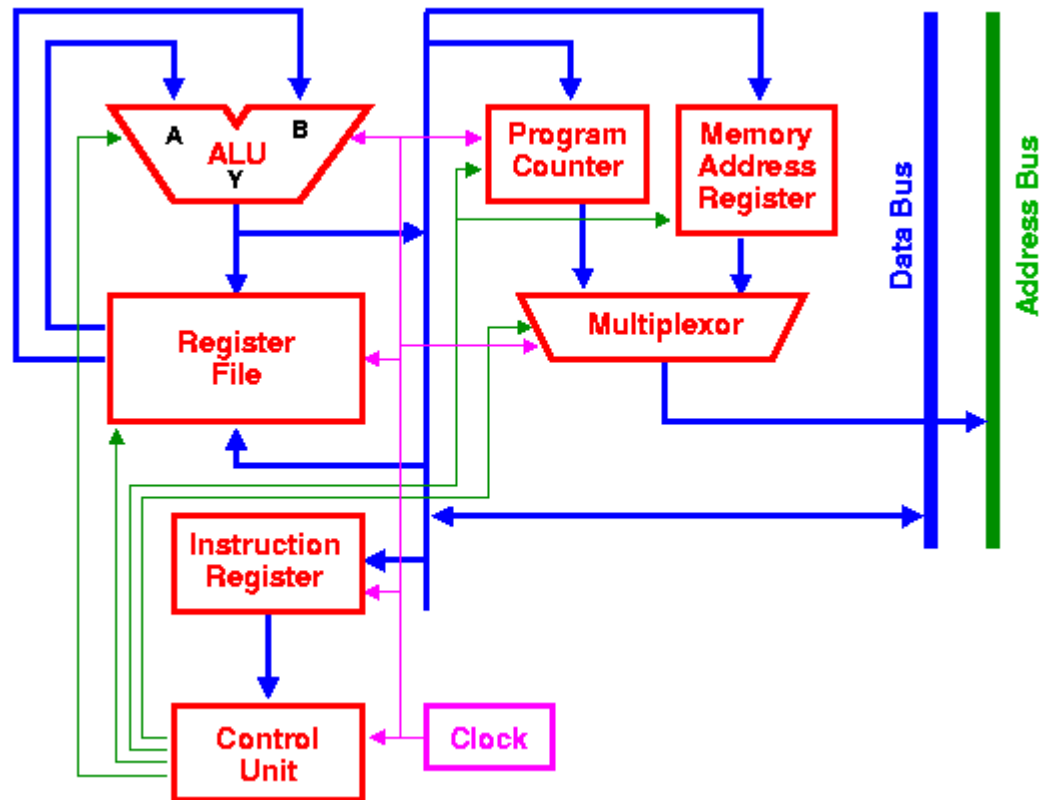


Figura 1-3: arquitectura básica de un procesador

En el diagrama anterior se muestra la estructura de un procesador bastante sencillo, como el que podemos encontrar en un pequeño microprocesador de 8 bits. Las diferentes unidades funcionales que lo componen son las siguientes:

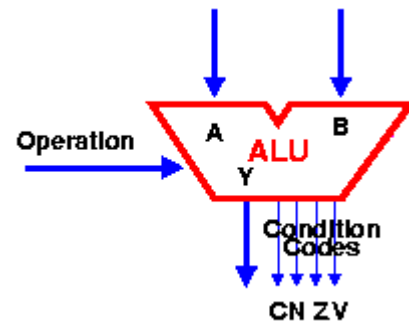
- ❑ **ALU:** unidad aritmético lógica. Este circuito toma dos operandos como entrada (A y B), y genera un resultado en la salida (Y). Entre las operaciones que se suelen incluir en una ALU, al menos encontraremos:
 - Aritméticas: suma, resta
 - Lógicas: and, or, not
 - Desplazamientos: desplazamiento de bits a derecha e izquierda
- ❑ **Banco de Registros:** (register file) es un conjunto de registros en los que se pueden almacenar resultados temporales. Las primeras máquinas solían tener un único registro (usualmente denominado acumulador). En cambio, los modernos procesadores RISC cuentan con al menos 32 registros.

- ❑ **Registro de Instrucción:** almacena la instrucción que se encuentra en ejecución
- ❑ **Unidad de Control:** decodifica la instrucción que se encuentra almacenada en el registro de instrucción, y activa las señales que controlan las operaciones de las demás unidades funcionales del procesador. Por ejemplo, ejemplo el código de operación (opcode) de la instrucción actual se usará para determinar las señales de control de la ALU, que a su vez determinarán la operación a realizar.
- ❑ **Reloj Global del Sistema (clk):** la gran mayoría de los procesadores son síncronos, esto es, usan una señal de reloj para determinar cuándo capturar la próxima palabra de datos y llevar a cabo todas las operaciones necesarias sobre ella. El reloj global debe llegar a todas las unidades funcionales que conforman el procesador.
- ❑ **Contador de Programa (PC):** mantiene la dirección de memoria de la próxima instrucción a ejecutar. Se actualiza con cada ciclo de instrucción a la dirección de la siguiente instrucción en el programa, que será la anterior más un cierto incremento constante, a no ser que se produzcan saltos (un salto no es más que actualizar el PC a un valor que no es el que se esperaría en la ejecución normal del programa, pero que apunta a una dirección dentro de la zona de memoria donde se ubica el mismo)
- ❑ **Registro de Dirección de Memoria:** (memory address register) almacena la dirección de memoria en la cual se almacenará el próximo dato, o de la cual se leerá el próximo dato.
- ❑ **Bus de Direcciones:** se usa para transferir direcciones tanto a la memoria principal, como a los periféricos mapeados en memoria. El procesador actúa como maestro del bus
- ❑ **Bus de Datos:** se encarga de llevar los datos desde y hacia el procesador, memoria y periféricos. El encargado de gestionar el bus será en cada caso el origen de datos, es decir, el procesador, la memoria, o el periférico en cuestión.
- ❑ **Bus Multiplexado:** muchos procesadores de altas prestaciones, que tienen bus de datos y bus de direcciones separados, por limitaciones en el número de pines, y por la complejidad misma de los buses, suelen multiplexar direcciones y datos en el mismo bus. Esto tiene un efecto muy negativo en el rendimiento del sistema.

1.2.4 Unidad Aritmético-Lógica (ALU)

La Unidad Aritmético-Lógica es el corazón del procesador, pues se encarga de realizar todos los cálculos.

En los procesadores más simples, la ALU es un gran bloque combinacional que toma como entradas dos operandos (A y B), y el código de operación (operation code), y a su salida muestra el resultado de la operación (Y), y además también activa las correspondientes banderas (flags), que se utilizarán para actualizar los códigos de condición del registro de control de estado (CPSR) del procesador.



Estos *flags* son cuatro, y su significado es el siguiente:

- bit C: (carry) es el bit de acarreo
- bit N: (negative) se activa si el resultado es negativo
- bit Z: (zero) se activa si el resultado es cero (todos los bits a '0')
- bit V: (overflow) se activa si se produce un overflow

Los operandos y el código de operación se aplican cuando se produce un flanco de reloj, y el circuito esperará por tanto, que se produzca un resultado antes del próximo flanco de reloj. Por tanto, el retardo de propagación a través de la ALU determina el mínimo periodo de reloj y establece una cota superior de la frecuencia del reloj (si aplicamos los resultados obtenidos en el apartado máquinas síncronas, $1/t_{pd}^{max} > f_{clk}$).

Operaciones

Cualquier ALU, por simple que sea, realiza como mínimo las siguientes operaciones:

- Aritméticas: suma, resta
- Lógicas: and, or, not
- Desplazamientos: derecha, izquierda, rotación

ALUs más modernas y complejas pueden soportar un amplio rango de operaciones con enteros (multiplicación, división), operaciones en punto flotante (suma, resta, multiplicación y división), e incluso algunas funciones matemáticas más complejas (raíz cuadrada, seno, coseno, logaritmo, ...).

De todos modos, en el mercado de los procesadores programables de propósito general, las operaciones aritméticas más comunes son la suma y la resta, pues multiplicaciones con enteros y otras operaciones más complejas pueden ser implementadas con software. Aunque ello suponga que requieran un tiempo de

ejecución considerable (una multiplicación de enteros de 32 bits necesita 32 sumas y desplazamientos), esta medida mejora considerablemente la velocidad de ejecución de las operaciones más sencillas (que son a su vez las más comunes), con lo cual se mejora el rendimiento del sistema.

Con el paso del tiempo, se ha conseguido reducir considerablemente el tamaño del transistor, con lo cual muchos chips incluyen unidad de punto flotante en la ALU (que ocupa un área bastante extensa en comparación con otras unidades funcionales que podamos encontrar en el layout del procesador)

¿Hardware o Software? La respuesta a esta pregunta es la primera decisión a tomar en el diseño de las operaciones que soporta una ALU, y para poder responderla habrá que tener en cuenta muchos factores, como puede ser la aplicación, la velocidad de procesamiento requerida, etcétera.

1.2.5 Anatomía de Procesadores Modernos

En esta sección vamos a introducir los principios básicos del funcionamiento de procesadores en paralelo. Estudiaremos tres conceptos: superscalar, pipelined, y VLIW. En realidad, algunas definiciones incluyen la arquitectura superpipelined y la VLIW como variantes de la arquitectura superscalar.

Posteriormente nos centraremos en la arquitectura pipelined, pues es la elegida en este proyecto para la implementación ARM8/9.

Superscalar

Es un uniprocador que puede ejecutar dos o más *operaciones escalares* en paralelo. Para ello, requiere de múltiples unidades funcionales, que pueden ser o no idénticas unas a otras.

En algunos procesadores *superscalar* el orden de ejecución de las instrucciones se determina estáticamente (al compilar), mientras que en otros en cambio se determinan dinámicamente durante la ejecución.

Dentro de este último tipo, tiene fundamental importancia lo que se llama Unidad de Selección de Instrucción a Cursar (Instruction Issue Unit), que escoge la próxima instrucción a cursar en función de la disponibilidad de las unidades funcionales requeridas, y de la consistencia de los operandos necesarios para la operación (si ya han sido actualizados).

Pipelined

Consiste en una secuencia de unidades funcionales (stages) que realizan una actividad dividiéndola en múltiples pasos (y cada unidad funcional se encarga de realizar una parte del proceso).

Cada unidad funcional captura sus entradas y produce un resultado que se almacena en su buffer de salida. El buffer de salida de una etapa es a su vez el buffer de entrada de la etapa siguiente. Esta filosofía de diseño permite que todas las etapas trabajen en paralelo, dando como resultado un flujo de salida mucho más elevado que si cada entrada pasara por toda la *pipeline* antes de que la próxima entrada sea pudiera ser capturada.

¿Qué coste tiene la obtención de las prestaciones anteriormente descritas? Una mayor latencia y complejidad debido a la necesidad de sincronizar de alguna manera todas las etapas, de modo que las diferentes entradas no interfieran entre ellas al ser procesadas (nos centraremos en este tema en los dos próximos apartados).

Para que una arquitectura *pipelined* pueda obtener su eficiencia máxima, es necesario que pueda ser llenada y vaciada al mismo tiempo mientras se lleva a cabo la ejecución de un programa.

VLIW (Very Long Instruction Word)

El juego de instrucciones implementado en este tipo de máquinas se realiza utilizando lo que se denomina microcódigo horizontal. Una instrucción codificada horizontalmente que contiene cuatro o más operaciones, puede ser considerada como muy larga (very long). Las instrucciones se dividen en *slot*, que se mapean directamente en la unidad funcional correspondiente, lo cual simplifica enormemente el papel de la Unidad de Selección de Instrucción a Cursar (Instruction Issue Unit).

Instruction Format

FP Add	FP Mult	Int ALU	Branch	Load/Store
--------	---------	---------	--------	------------

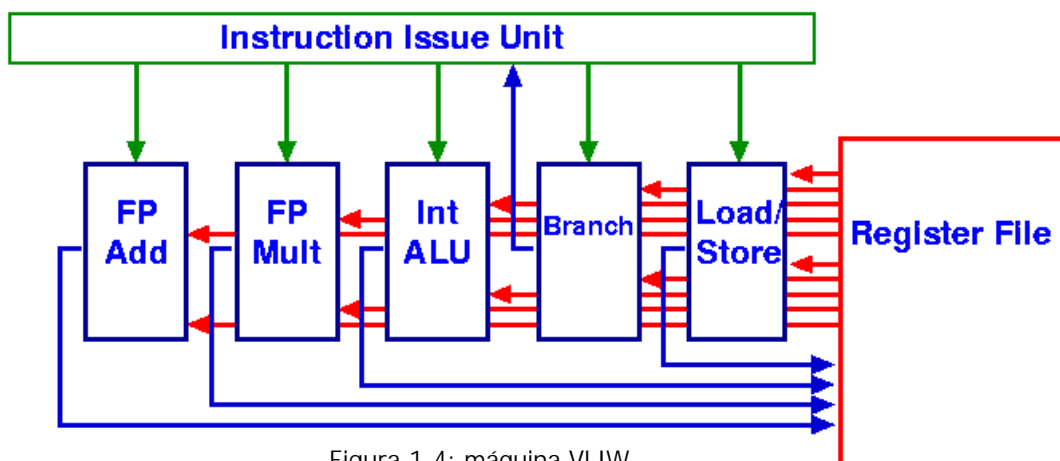


Figura 1-4: máquina VLIW

En cambio, generar código para máquinas VLIW es muy complicado (el papel de los compiladores es decisivo y además bastante complejo), pues es difícil decidir para cada instrucción qué misión asignaremos a cada slot. Para ello, se utiliza una técnica denominada, en la literatura anglosajona *trace scheduling*.

1.2.5.1 Procesador «*pipelined*»

Para ilustrar el principio básico de funcionamiento de esta arquitectura, consideremos el siguiente ejemplo: tenemos 100 estudiantes haciendo un examen, 5 preguntas por examen, y 5 personas corrigiendo el examen. Supongamos que se tarda en corregir una pregunta 0.5 horas

- Si se corrige individualmente cada examen, se tarda 2.5 horas por examen, lo que hace un total de 250 horas.
- En cambio, si utilizamos 5 personas, cada uno corrigiendo una pregunta, nos encontramos con que al transcurrir 50 horas, tenemos la primera pregunta de cada examen corregida, y tras 52.5 horas del inicio, todos los exámenes estarán corregidos (aproximadamente en 5 veces menos tiempo)

Este principio se ha utilizado para incrementar el flujo de instrucciones (throughput) generado, de modo que los procesadores de altas prestaciones modernos hacen uso de una técnica que se denomina *pipelinig*.

Un procesador *pipelined* no espera a que el resultado de una operación previa se haya alojado en el registro destino correspondiente, sino que va leyendo de memoria instrucción tras instrucción tan pronto una haya sido capturada por el registro de instrucción.

Un procesador *pipelined* cuenta con un *pipeline* estructurado en un cierto número de etapas (stages), que por ejemplo, en los primeros procesadores RISC oscilaba entre 4 ó 5. Así, las instrucciones que se van leyendo de memoria, pasan de una etapa a otra, de modo que en un momento dado, encontramos en la *pipeline* un número de instrucciones activas igual al número de etapas en que se divide la estructura.

En un procesador RISC típico, podemos encontrar una estructura *pipeline* dividida en cuatro etapas:

1. **IF (Instruction Fetch):** lee la instrucción de memoria y actualiza el PC
2. **DEC (Decode and Operand Fetch):** su misión consiste en decodificar la instrucción, seleccionar los operandos requeridos del banco de registros, y activar las señales de control necesarias para la ejecución de la instrucción en la siguiente etapa
3. **EX (Execute):** ejecuta la instrucción en la ALU
4. **MEM (Memory):** se realizan los accesos necesarios a memoria
5. **WR (Write and WriteBack):** se escribe el resultado en el registro destino

Con una pipeline de n etapas, después de $n-1$ ciclos de reloj, la *pipeline* está completamente llena y entonces, en cada ciclo de reloj se completa una instrucción. Esto implica que se aumente la velocidad de ejecución del procesador aproximadamente en un factor n (posteriormente estudiaremos el parámetro speedup que se encarga de cuantizar esta mejora).

Para que en un sistema se pueda implementar una estructura *pipeline*, éste debe satisfacer los siguientes requisitos:

- Un perfecto candidato para la implementación de un *pipeline* ejecuta repetidamente la misma función básica.
- Dicha función básica debe ser divisible en actividades (etapas) independientes, las cuales se solapan entre ellas lo menos posible.
- La complejidad de las etapas resultantes, deben ser similares.

Speedup

A continuación vamos a proceder al estudio de este parámetro, que mide la mejora del tiempo de ejecución de las instrucciones utilizando una estructura *pipelined*, con respecto al tiempo que tardaría la misma arquitectura, si no utilizara la técnica *pipelining*.

$$speedup = \frac{CPI_{no\ pipelined} \times t_{cyc}^{no\ pipelined}}{CPI_{pipelined} \times t_{cyc}^{pipelined}}$$

Si se utiliza la misma frecuencia de reloj en ambos casos, la expresión anterior puede reducirse a la ecuación:

$$speedup = \frac{CPI_{no\ pipelined}}{CPI_{pipelined}}$$

Vamos a suponer ahora un caso ideal, en el que tenemos una estructura *pipeline* que cuenta con d etapas, y queremos a ejecutar un total de n instrucciones. Vamos a suponer que no se producen *bubbles* (ya estudiaremos más adelante en qué consiste este fenómeno), ni tampoco saltos. Podemos definir entonces el parámetro *speedup* considerando el número de ciclos necesarios para ejecutar la totalidad de las n instrucciones si está presente la estructura *pipeline* ($n + d$) y si prescindimos de la misma ($n \cdot d$), con lo cual la expresión quedaría:

$$speedup = \frac{n \cdot d}{n + d}$$

Normalmente $n \gg d$, lo cual implica que se puede despreciar d en el denominador, y el parámetro *speedup* coincide con el número de etapas de la *pipeline* d .

$$speedup \gg d$$

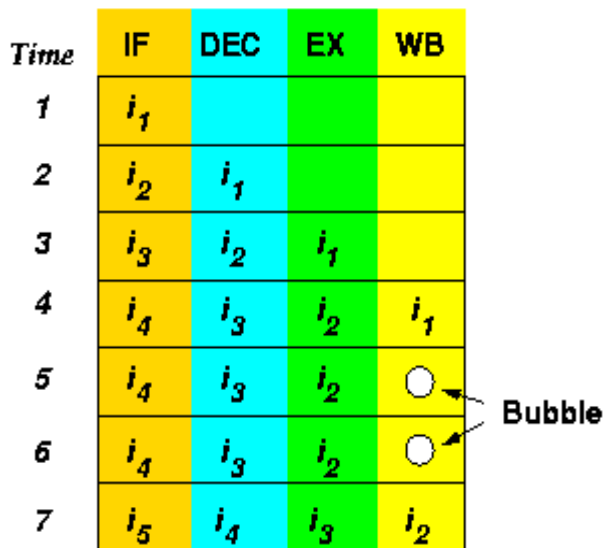
Para hacernos una idea, supongamos que queremos sumar 1000 pares de números, por ejemplo, sumando dos vectores de 1000 elementos. Si para cada suma se necesita 5 ciclos, una máquina sin *pipeline* tardaría 5000 ciclos. En cambio, con una *pipeline* de 5 etapas, la última suma se concluirá tras 1000+5 ciclos, así que el procesador *pipelined* es $5000/1005 = 4.97$ veces más rápido.

Tiempo de Latencia

Para el valor del PC correspondiente a una instrucción concreta, faltan aún N ciclos para que esta instrucción se complete. Se define la *latencia* como el tiempo entre que el procesador obtiene una instrucción, y el tiempo en el que de él emerge el resultado.

Bubbles

Prácticamente la mayoría de las máquinas RISC completan sus instrucciones aritméticas en un ciclo de reloj, pero existen instrucciones más complejas (ej. dividir) que necesitan de más de un ciclo para ejecutarse. Tales instrucciones permanecen en la misma etapa (por ej. EX) durante más de un ciclo, de modo que es necesario insertar lo que se llama un *bubble*. Esto no es más que una NOP (non operation) que la etapa en la que la instrucción va a ocupar más de un ciclo de reloj propaga a la siguiente etapa, pues ésta espera recibir una instrucción para realizar su parte del proceso de la misma. Este fenómeno degrada el parámetro *speedup* ideal obtenido anteriormente.



Es necesario generar *bubbles* cuando nos encontramos con:

- Operaciones de latencia elevada, como por ejemplo divisiones, operaciones con punto flotante y funciones matemáticas.
- Acceso a memorias que necesitan más de un ciclo para estabilizar los datos.
- Cuando se produce *interlocking* entre dos instrucciones, que acceden al mismo registro, una en lectura y la otra en escritura.
- Saltos

Saltos (“branches”)

Para realizar una ejecución eficiente de los saltos, es necesario realizar cambios bastante significativos en la arquitectura de base del computador. Un salto simple, como el retorno de una subrutina, requiere que un cierto número de instrucciones que le siguen en memoria, sean cargadas en la *pipeline* para luego ser vaciadas de la misma.

Ocurre que cuando una instrucción es un salto, éste no se realiza de forma efectiva hasta que no llega a la etapa EX. Esto quiere decir que en el momento de producirse el salto (es decir, en el momento en el que se actualiza el PC a la dirección de salto), existe una instrucción en DEC y otra en IF que no deberían estar allí, pues se ha producido un salto, y las instrucciones cargadas en la *pipeline* no son las correctas.

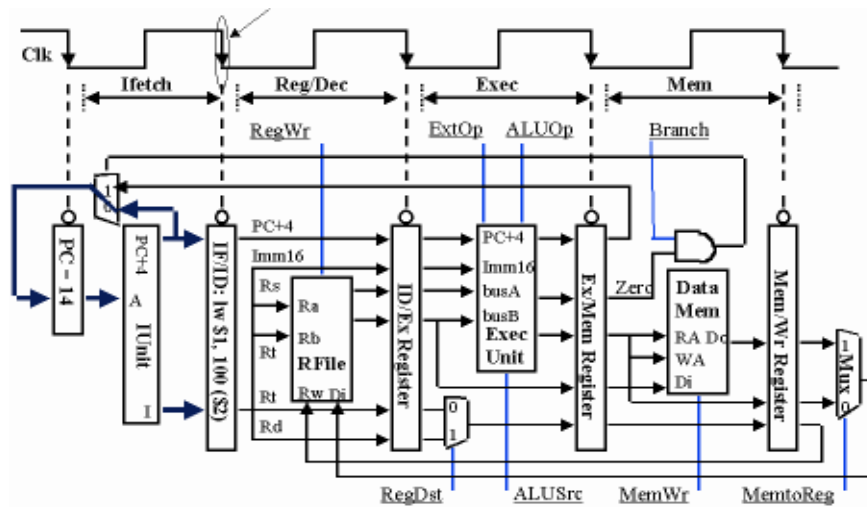
En la figura adjunta se ilustra este hecho. Nótese que los saltos disminuyen dramáticamente el rendimiento ideal del sistema, y por supuesto el *speedup* calculado anteriormente.

Este hecho muestra la importancia de utilizar *estrategias de predicción de saltos*.

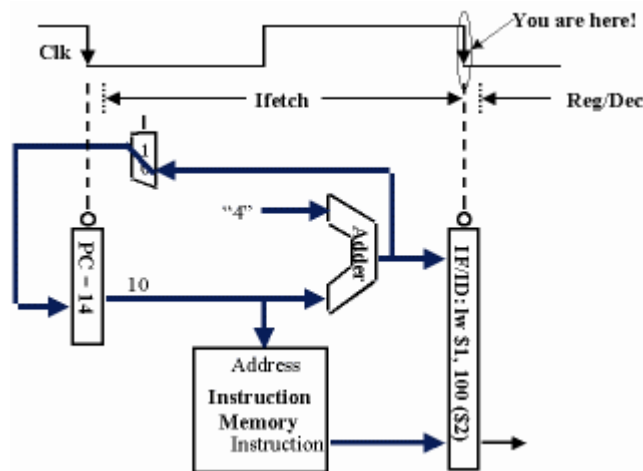
Time	IF	DEC	EX	WB
1	i_1			
2	i_2	i_1		
3	i_3	i_2	i_1	
4	i_4	i_3	i_2	i_1
5	○	○	○	i_2
6	i_a	○	○	○
7	i_b	i_a	○	○

A continuación se muestra el diagrama de bloques de una arquitectura *pipeline*, y cómo cambian e interactúan las señales de control al avanzar el reloj global del sistema.

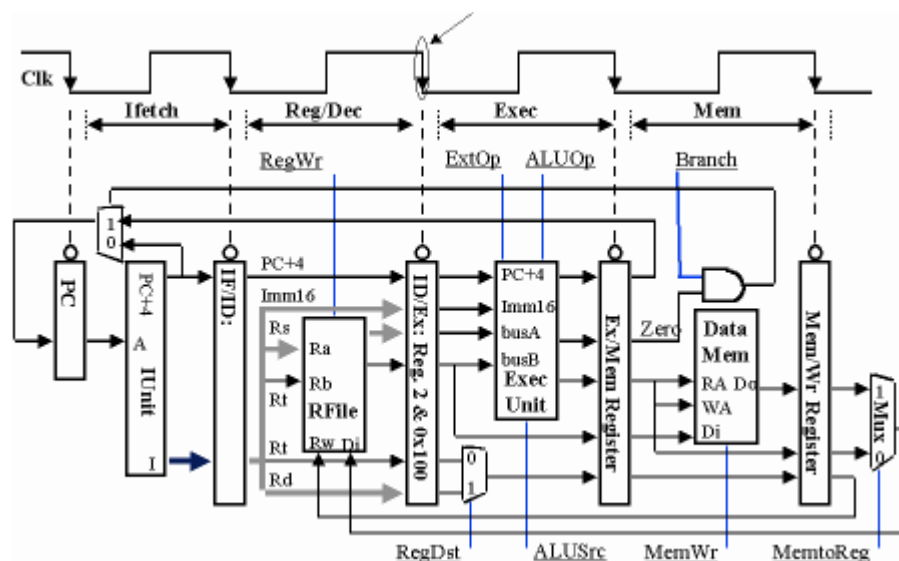
Primera Etapa: cargando (fetch) la instrucción



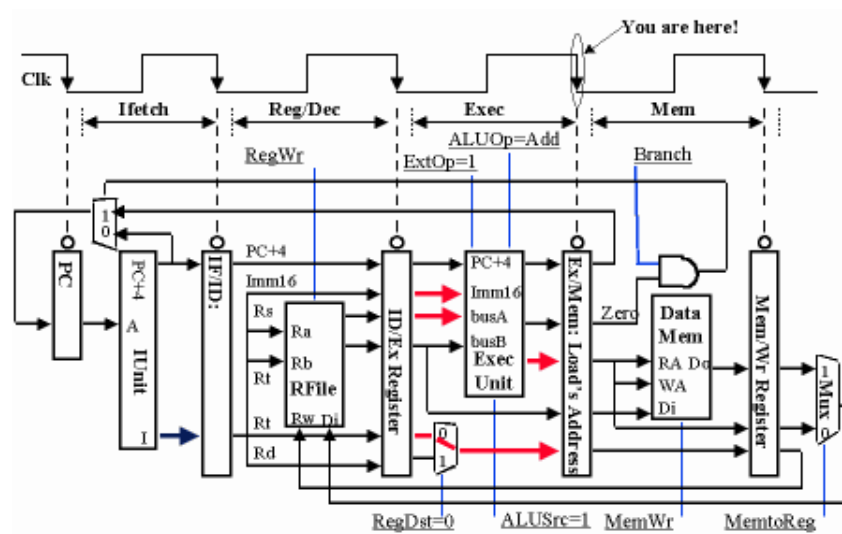
Unidad de Instrucción



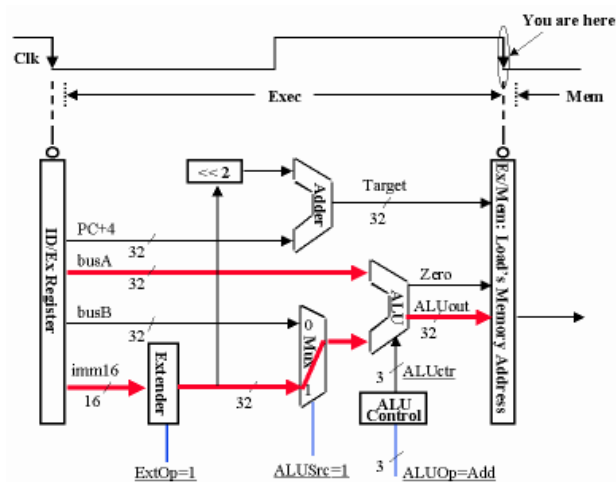
Decodificación de la Instrucción / Selección de Registros



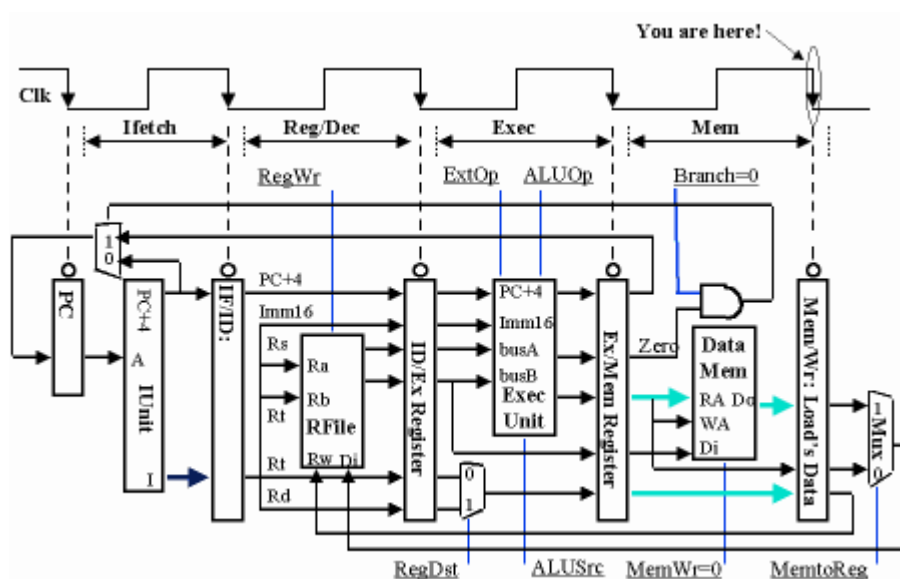
Cálculo de Direcciones



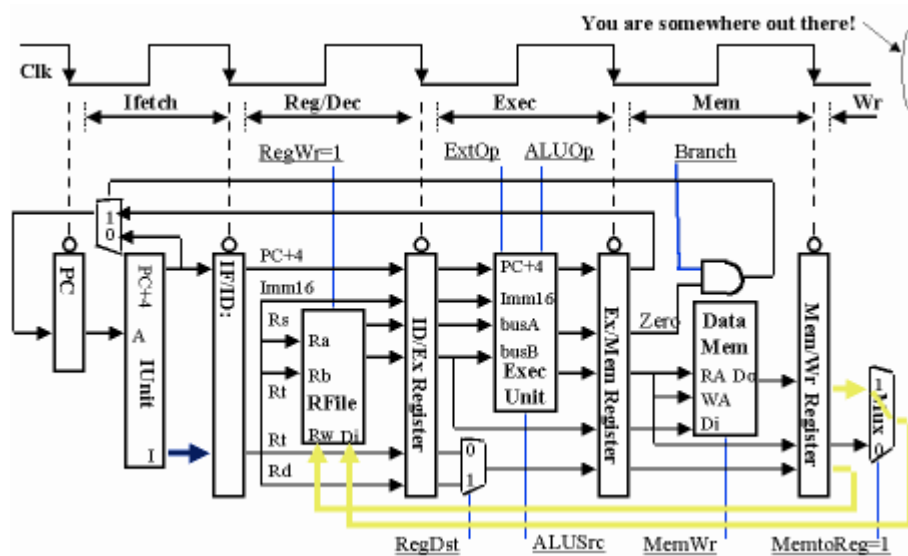
Unidad de Ejecución



Etapas de Acceso a Memoria



Escritura de Registros (WriteBack)

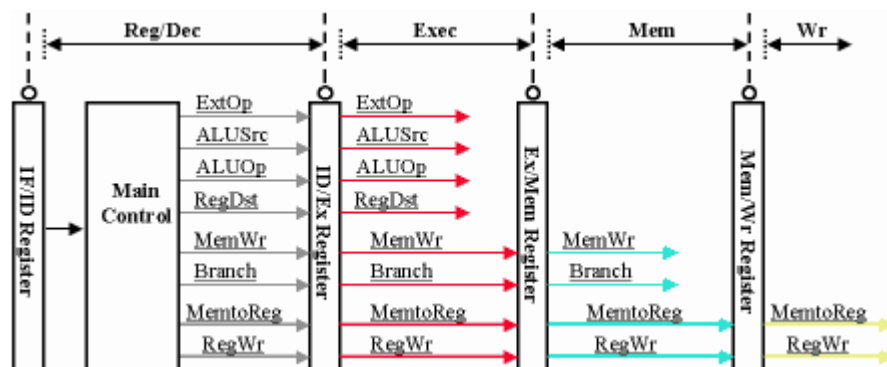


Señales de Control

Las señales de control en la etapa N dependen de la instrucción que se encuentra en esa misma etapa (para $N = EX, MEM, \text{ o } WR$).

Control de la estructura pipeline

- El control principal genera las señales de control durante la etapa DEC/Reg
- Las señales de control para EX se usan 1 ciclo más tarde
- Las señales de control para MEM se usan 2 ciclos después
- Las señales de control para WR se usan 3 ciclos más tarde



1.2.5.2 Riesgos (hazard) en procesadores “pipelined”

Hasta ahora hemos visto que idealmente, una nueva instrucción debería entrar en la *pipeline* en cada ciclo de reloj, de modo que cada etapa está ocupada en cada ciclo. Desafortunadamente, existen algunas secuencias de instrucciones que simplemente son imposibles. Estas secuencias se denominan *hazards* y su naturaleza se encuadra en uno de los siguientes tipos:

1. Riesgos Estructurales: ocurre cuando un componente hardware es requerido por más de una instrucción en la *pipeline*. En general, el problema se debe a que existe poco paralelismo en la ruta de datos, y la única solución consiste en añadir más componentes hardware, de modo que no sea necesario compartirlos entre etapas distintas. Estos riesgos estructurales pueden deberse a que:
 - Necesitamos que un recurso estuviera duplicado: el mismo componente es requerido por dos etapas en la *pipeline*.
 - La Unidad Funcional no es completamente *pipelined*: si ésto ocurre, su operación requerirá de más de un ciclo de reloj, lo que nos obligará a introducir *bubbles* en etapas posteriores.
2. Riesgos de Datos: se producen cuando una instrucción depende del resultado de una instrucción anterior, pero el resultado aún no está disponible porque dicha instrucción anterior aún se encuentra en la *pipeline* y no ha sido completada. La situación se resuelve introduciendo *bubbles* hasta que la instrucción anterior, de la que depende la actual, sea completada.
3. Riesgos de Control: se deben a que una instrucción ha cambiado el PC. Algunas instrucciones dentro de la *pipeline* deben ser invalidadas. Ya hemos tratado este tema en el apartado anterior, y la solución consistía en vaciar ciertas etapas de la *pipeline* e introducir *bubbles* en etapas posteriores.

Todos los *hazard* anteriores se producen con relativa frecuencia, y repercuten en una disminución efectiva del *speedup* calculado con anterioridad. Vamos a intentar cuantificar el efecto de los *hazard* sobre el parámetro *speedup*.

$$speedup = \frac{CPI_{no\ pipelined} \times t_{cyc}^{no\ pipelined}}{CPI_{pipelined} \times t_{cyc}^{pipelined}}$$

Asumiendo que el periodo de reloj es el mismo en ambos casos, y que el CPI ideal en una máquina *pipelined* es siempre 1, podemos escribir que

$$\begin{aligned} CPI_{pipelined} &= CPI_{ideal} + \text{Ciclos de Espera introducidos por instrucción} = \\ &= 1 + \text{Ciclos de Espera introducidos por instrucción (WS)} \end{aligned}$$

Basta con sustituir en la ecuación anterior, y obtenemos la siguiente expresión para el parámetro *speedup*:

$$speedup = \frac{CPI_{no\ pipelined}}{1 + N_{ws}}$$

donde N_{ws} es el número medio de ciclos de espera por instrucción. Si nos damos cuenta que el número de ciclos por instrucción de un procesador no *pipelined* es igual al número de etapas del mismo procesador con arquitectura *pipelined*, ésto es, d , tenemos que

$$speedup = \frac{d}{1 + N_{ws}}$$

Si no existiesen estados de espera, es decir, si el número de ciclos de espera por instrucción es 0, entonces el parámetro *speedup* es igual a d , que era el valor ideal calculado en el apartado anterior.

Para calcular N_{ws} , habría que tener en cuenta la probabilidad de que aparezca una instrucción en un programa, ponderando así el número de ciclos de espera que necesita dicha instrucción.

De este modo, el hecho de introducir una o dos instrucciones que requieran de un ciclo adicional de reloj, pero que usualmente no sean empleadas, no empeorará notablemente el *speedup*, pues al ponderar el número de ciclos de espera adicionales que requieren (1) con su probabilidad de aparición en un código (un valor muy pequeño, y siempre inferior a 1), tendremos que N_{ws} es bastante próximo a 0, y podríamos decir que el *speedup* sigue siendo igual a d .

1.3 Memorias

La evolución de las memorias ha sido de vital importancia para la proliferación y el desarrollo de las máquinas RISC, y por eso vamos a dedicar un apartado al estudio de las dos tecnologías más extendidas para almacenar bits en RAM (random access memory) de semiconductores: RAM estática (static RAM ó SRAM) y RAM dinámica (dynamic RAM ó DRAM)

1.3.1 SRAM y DRAM

Static RAM (SRAM)

La celda básica SRAM usa de 4 a 6 transistores para almacenar un único bit de datos. Esto implica una gran disminución del tiempo de acceso a expensas de no poder conseguir una baja densidad de bits. La memoria interna de un procesador (registros y *cache*) se suele fabricar utilizando tecnología SRAM.

Debido a que la industria se ha centrado en producir en masa DRAM e incrementar su densidad, la memoria SRAM ha sido considerada más cara que la DRAM (lógico, pues a menor volumen de producción mayor coste).

Se ha usado SRAM para memorias *cache* de nivel 2 (Level 2 ó L2), las cuales necesitan mucha rapidez y suponen una memoria relativamente pequeña (de 512 kB a 4 Mbyte de L2 *cache*).

Además, como no necesitan refresco, el consumo de la SRAM es mucho menor que el de las memorias DRAM, así que también podemos encontrar células SRAM en sistemas de bajo consumo limitados por la capacidad de la batería.

También la ausencia de refresco implica una mayor simpleza en la circuitería, de modo que en sistemas de tamaño reducido ésta simpleza en la circuitería puede repercutir en una compensación del coste debido a la utilización de SRAM.

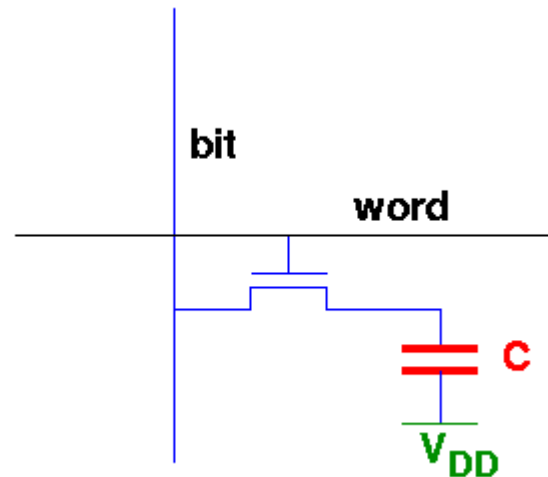
Dynamic RAM (DRAM)

Aunque más lenta que la anterior, la gran densidad de bits conseguida en este tipo de memoria implica que se puedan utilizar microprocesadores de más capacidad, con lo cual se incrementa la velocidad total del sistema.

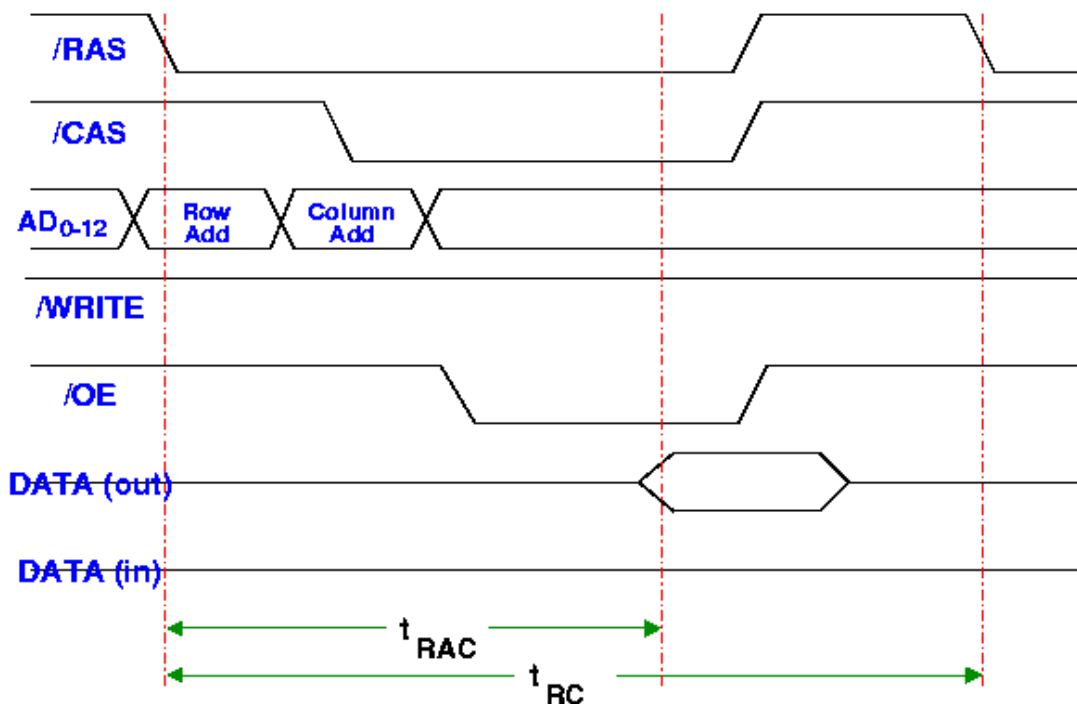
Una célula de memoria DRAM utiliza un transistor simple y un condensador para almacenar un bit de datos. Hoy en día se fabrican dispositivos de hasta 256 Mbits de capacidad en un único chip. A la vez, han proliferado las CPUs con 10 millones de transistores.

En la figura se muestra una celda típica DRAM con un transistor MOSFET y un condensador capaz de almacenar un bit.

Esta celda se distribuye regularmente a lo largo y ancho del dispositivo, de modo que una memoria DRAM es tan regular que se puede imaginar como una inmensa matriz de dos dimensiones compuesta por celdas capaces de almacenar un bit. En contraste, una CPU es un gran montón de lógica irregular.

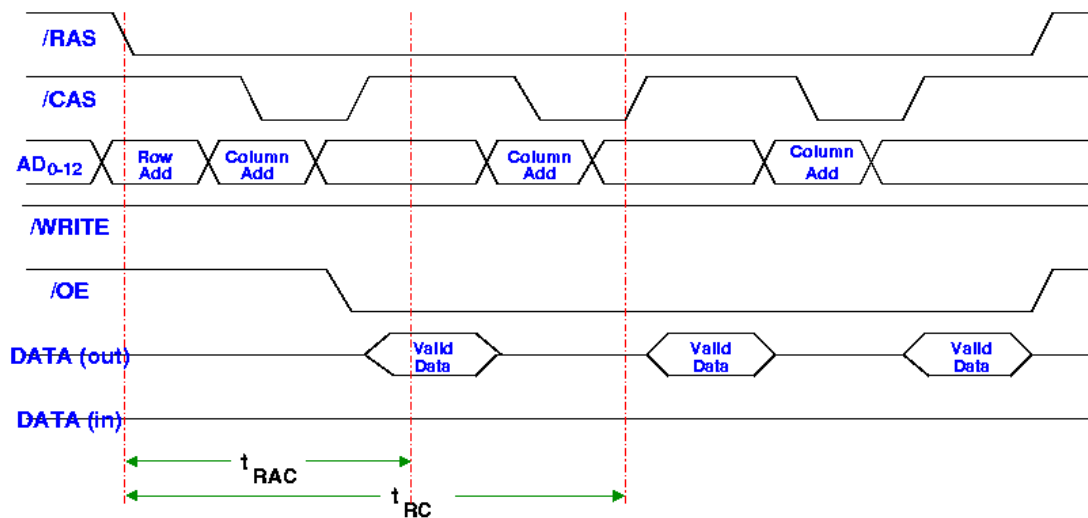


- ❑ Modo de Acceso: la mayoría de las memorias DRAM fabricadas requieren que la dirección aplicada sea dividida y aplicada en dos partes: una dirección de fila y una dirección de columna. Esto contribuye al deterioro del tiempo de acceso, pero permite que dispositivos con una gran capacidad (gran número de bits) sean fabricados con pocos pines



- ❑ Tiempo de Acceso: el rendimiento de las DRAM comerciales se mide en términos del *tiempo de acceso* (t_{RAC} en la figura). Es el tiempo transcurrido desde que la señal RAC se activa (es activa a nivel bajo), hasta que se encuentra disponible el dato en la salida.
- ❑ Ciclo de memoria: t_{RC} es el tiempo mínimo entre dos accesos sucesivos al mismo dispositivo, y es el factor que determina el flujo de datos total.

- ❑ Necesidad de refresco: existen pérdidas que hacen que el condensador poco a poco vaya perdiendo su carga, con lo cual se necesita una circuitería de refresco que periódicamente restaure el valor almacenado en cada celda. Cuando la memoria DRAM es refrescada, no puede ser accedida. Los tiempos de refresco rondan en torno a los milisegundos. Todo esto incrementa la complejidad de la circuitería de refresco y tiene como consecuencia el hecho de que en pequeños sistemas se utilicen memorias SRAM, donde el coste extra del chip SRAM se contrarresta con el ahorro de la complicada y cara circuitería de refresco.
- ❑ Paginado: la capacidad de un chip DRAM se ve incrementada gracias al modo de paginado que utiliza. Se pueden aplicar muchas direcciones de columnas para cada dirección de fila:



Esto compensa el hecho de que había que insertar la dirección en dos etapas, en beneficio del flujo total de datos, que se ve incrementado.

Esta estrategia es muy efectiva: se accede a cada dirección una a una dentro de la página, existiendo una gran probabilidad de que otra dirección en la misma página sea accedida también.

1.4 Diseño de Circuitos Integrados: ASIC y FPGA

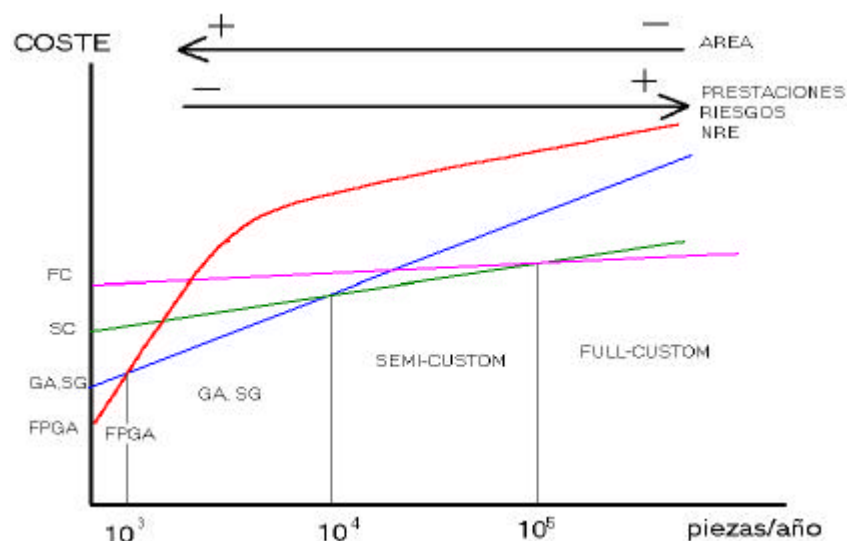
El aumento en los últimos años del uso de FPGAs en diseños microelectrónicos (junto al tremendo avance de la familia de procesadores ARM), ha motivado la realización del presente proyecto, y por ello queremos mostrar las diversas tendencias que se están siguiendo en la actualidad para la creación de circuitos integrados.

A la hora de plantearse el diseño de un circuito integrado digital, existen dos tendencias entre las cuales hay que elegir. No vamos a entrar en sus características, pero son:

- ASIC (Application Specific Integrated Circuit)
 - Full Custom
 - Semi-Custom
 - Standard Cell
 - Gate Array (GA)
 - Sea of Gates (SG)
- FPGA (Programmable Field Gate Array)

Para tomar una decisión, hay que llegar a un compromiso entre los siguientes factores:

- Funcionalidad a Integrar
- Coste Inicial (NRE): ingeniería no retornable, que incluye, por ejemplo, diseñamos un ASIC, la elección de la tecnología, las máscaras y el área a integrar. En cambio, en una FPGA es prácticamente despreciable
- Coste Total: ingeniería + NRE
- Volumen de Fabricación: aunque el coste inicial de nuestra elección sea alto, si vamos a producir un volumen elevado puede compensarse
- Prestaciones: puede que si nuestros requisitos son muy estrictos, no tengamos más remedio que optar por una tecnología Standard Cell, o incluso Full Custom (por ejemplo, si tenemos los retardos máximos de propagación acotados por una frecuencia de reloj deseada).
- Riesgo que estamos dispuestos a asumir



2

Introducción al ARM

El objetivo de este apartado es, en primer lugar, intentar justificar la motivación de este proyecto, y para ello nos apoyamos en la evolución histórica y tecnológica que ha sufrido esta familia de procesadores desde sus inicios.

En segundo lugar, se intenta en esta sección realizar un acercamiento al procesador ARM, resaltando sus características generales, y describiendo brevemente la arquitectura utilizada en el diseño del procesador comercial.

- 2.1 Breve reseña histórica
- 2.2 Evolución tecnológica
- 2.3 Versiones de la arquitectura del ARM
- 2.4 Características generales
- 2.5 Arquitectura del núcleo del ARM8
- 2.6 Arquitectura del núcleo del ARM9
- 2.7 Núcleo (CORE) del ARM8
- 2.8 Núcleo (CORE) del ARM9

2.1 Breve reseña histórica

A finales de los ochenta, una compañía llamada Acorn Computers, con sede en Cambridge (Inglaterra), ideó una UCP RISC que soportaba varios chips. Acorn había estado utilizando el procesador de 8 bits 6502 para su gama de ordenadores personales.

La compañía pretendía extender su arquitectura a 32 bits, pero sin que este aumento fuera en detrimento de la eficiencia obtenida en sus diseños anteriores. Su primera CPU fue bautizada con el nombre de ARM2 (que por entonces significaba Acorn RISC Machine 2), y resultó ofrecer una gran eficiencia a un coste muy bajo, además de un bajo consumo.

Apple Computer, una empresa con sede en USA, puso sus ojos en esta pequeña compañía, y al finales de 1990, un equipo de doce ingenieros dejaron la compañía británica Acorn y se pusieron bajo las órdenes de Robin Saxby para fundar una nueva compañía de microprocesadores con el nombre de Advanced RISC Machines Ltd. (ARM).

2.2 Evolución Tecnológica

¿Cuál es la diferencia entre la tecnología del ARM y otras existentes? Su mejor característica es la *eficiencia*. En el núcleo (CORE) del ARM cada transistor está ahí por una razón específica, y ofrece un beneficio notable con respecto a su coste.

Los diseñadores del ARM no buscaban sólo un compromiso entre área y velocidad, sino que buscaban el mejor compromiso entre varios requisitos.

- ❑ ARM1 (1985): pequeño, simple, de bajo coste y ofrecía bajo consumo
- ❑ ARM2 (1987): con la misma arquitectura que su predecesor, contaba con un multiplicador de enteros y una interfaz con coprocesador en el propio chip. Fue una CPU de bajo consumo casi por accidente, y resultó ser mucho más rápida que los procesadores Intel 286 y 386 de aquellos días, y técnicamente superior a los procesadores embedded que se encontraban en el mercado. Y todo ello con pocos transistores y una lógica interna muy simple
- ❑ ARM3 (1989): integra *cache* y utiliza una arquitectura más avanzada.
- ❑ ARM7 (1993): fue el último diseñado con arquitectura de Von Neumann

Después del ARM7, ARM experimentó con la arquitectura Harvard, que utiliza buses separados para datos e instrucciones. Esto implica el uso de *caches* separadas para datos e instrucciones, que son conectadas a la misma memoria.

El ARM8 (1996) fue un paso en el camino, pero fue el ARM9 el que se erigió como verdadero sucesor del ARM7.

Incluimos la siguiente tabla para mostrar el vertiginoso avance tecnológico que ha sufrido esta familia de procesadores a lo largo de la historia.

Evolución de las prestaciones de los Microprocesadores ARM						
Producto	Año de Introducción	Performance (MIPS)	Frecuencia (Mhz)	Voltaje (V)	Tecnología (µm)	Eficiencia [MIPS/W]
CORES básicos						
ARM7	1993	36	40	3,3	0,6	450
ARM7TDMI	1994	54	60	3,3	0,35	690
ARM8	1996	55	50	3,3	0,5	340
ARM9TDMI	1997	133	120	3,3	0,35	615
CORES de Microprocesadores completos						
ARM710	1994	43	48	5,0	0,6	80
ARM710	1996	27	24	3,3	0,6+	230
ARM710T	1998	45	50	3,3	0,35	200
ARM720T	1998	45	50	3,3	0,35	200
ARM740T	1998	36	40	3,3	0,35	230
ARM810	1996	55	50	3,3	0,5	170
ARM940T	1997	133	120	3,3	0,35	290
StrongARM110	1995	268	233	2,0	0,35	630

Fuente: Morgan Stanley Dean Writer, *Prospectus - ARM Holdings plc*, Issued April 14, 1998.

2.3 Versiones de la arquitectura del ARM

En este apartado se pretende presentar la Versión 4 de la arquitectura ARM, más conocida como ARMv4, que es la que utilizaremos en la implementación del ARM9.

Existen cuatro versiones de la arquitectura ARM:

- Versión 1: se utilizó en la implementación del ARM1, pero nunca fue comercializada.
- Versión 2: surgió como extensión de la Versión 1, tras añadirle instrucciones de multiplicación y sus variantes, y un soporte completo (interfaz e instrucciones) para coprocesador; además aparecen dos bancos adicionales de registros para el modo FIQ, y más tarde (Versión 2a) aparece una instrucción atómica de load-store denominada SWAP. La Versión 2 cuenta con tres modos de procesamiento privilegiados: FIQ, IRQ y Supervisor

Las Versiones 1, 2 y 2a son soportadas por un bus de direcciones de 26 bits, y combinan en el registro 15 un PC de 24 bits y 8 bits de estado del procesador.

- Versión 3: se extiende la arquitectura a 32 bits, definiendo en R15 un PC de 30 bits, y un CPSR separado del contador de programa. Además incorpora dos nuevos modos privilegiados de procesamiento, *undefined* y *abort*. Se define además cinco nuevos registros de estado (SPSRs), cada uno de los cuales se corresponde con un modo privilegiado del procesador. La Versión 3 tiene compatibilidad con las versiones anteriores, mediante emulación tanto software como hardware de las arquitecturas primitivas.
 - Versión 3G: igual que la versión 3, pero sin la compatibilidad con las versiones inferiores.
 - Versión 3M: incorpora multiplicaciones con y sin signo, y multiplicaciones de 64 bits.
- Versión 4: añade *halfword load/store*, simple, con signo, sin signo, y *byte load* con signo, y añade un nuevo modo privilegiado que utiliza los mismos registros que el modo *user*.
 - Versión 4T: incorpora un decodificador de instrucciones para un subconjunto de instrucciones de 16 bits derivado del juego de instrucciones original del ARM, y que se conoce con el nombre de THUMB. Esto no supone variación en la arquitectura del ARM, pues se realiza una descompresión para obtener instrucciones de ARM (32 bits) a partir de las instrucciones de 16 bits del subconjunto THUMB.

La última versión de la arquitectura ARM se conoce como ARMv4, y su derivada se suele denominar ARMv4T. Además de implementar dos ISAs distintos, existe otra diferencia sustancial entre ambas arquitecturas: en ARMv4T no existe el modo privilegiado *system*.

2.4 Características Generales

Tanto el ARM8 como el ARM9 forman parte de la familia de procesadores ARM (Advanced RISC Machines), compuesta por procesadores de propósito general con arquitectura de 32 bits, que ofrecen altas prestaciones a un bajo precio, y sobre todo con bajo consumo.

La arquitectura se basa en los principios de diseño de máquinas RISC, y el juego de instrucciones y la lógica de decodificación son por tanto mucho más simples que los de las máquinas CISC microprogramadas. Estos principios son:

- ❑ Utilizar un banco de registros extenso y uniforme
- ❑ Arquitectura *load-store* (load y store son las únicas operaciones que acceden a memoria. Las operaciones de proceso de datos sólo actúan sobre los registros internos)
- ❑ Modos de direccionamiento simples (el dato se almacena o se lee de la dirección de memoria especificada en registros y el campo de la instrucción)
- ❑ Longitud de instrucciones y de campos de las mismas uniforme.

Además, la arquitectura del ARM proporciona las siguientes ventajas:

- ❑ Control sobre la ALU y el SHIFTER en cada instrucción de proceso de datos, para maximizar el uso de los mismos.
- ❑ Autoincremento y autodecremento en los modos de direccionamiento, para optimizar los bucles.
- ❑ Ejecución condicional de todas las instrucciones para maximizar el flujo de ejecución del programa.

Banco de Registros del ARM

ARM cuenta con 31 registros de 32 bits de propósito general. Para un modo de funcionamiento concreto, sólo dieciséis son visibles, y el resto se utilizan únicamente para aumentar la velocidad de proceso de las excepciones. Todas las direcciones de registros especificadas en las instrucciones del ARM pueden referenciar a cualquiera de los dieciséis. El banco de registros es objeto de un análisis detallado en el capítulo siguiente (3. Modelo del programador)

- ❑ R15 es el contador de programa PC
- ❑ R14 es el registro de enlace (Link Register ó LR), y se utiliza para volver después de un salto con retorno.
- ❑ El resto de los registros no tienen un propósito específico, aunque R13 se ha venido utilizando normalmente como el puntero de pila (stack pointer ó SP).

Excepciones

El ARM soporta 5 tipos de excepciones, y existe un modo privilegiado para cada una de ellas. Estos cinco tipos son los siguientes:

- ❑ dos niveles de interrupción (rápido ó FIQ y normal ó IRQ)
- ❑ para implementar una protección de memoria, existe una señal de *abort*
- ❑ Atención a la excepción producida por la ejecución de instrucciones no definidas.
- ❑ Interrupciones software (SWIs) utilizadas para hacer llamadas al sistema operativo.

Cuando se produce una interrupción, algunos registros principales son substituidos por registros específicos del modo al que se entra tras la excepción.

CPSR y SPSR

El estado del procesador en cada instante de operación se encuentra almacenado en el CPSR (Current Program Status Register). Este registro mantiene:

- 4 banderas (flags) de condición: Negative, Zero, Carry, Overflow)
- 2 bits de desactivación de interrupciones (I, F)
- 5 bits que codifican el modo actual del procesador

Para los 5 tipos de excepciones existen 5 registros *SPSR_<modo>* en los que se almacena el estado del CPSR en el momento de producirse una interrupción, para poder restaurarlo al terminar la atención de la misma.

Unidad de Predicción de Saltos

El ARM8 incluye una unidad de predicción de saltos (Predicting Prefetch Unit) que reduce el CPI y el consumo total del sistema.

Interfaz con Coprocesador en chip

Un avance importante del ARM fue la integración de la interfaz con el coprocesador en el propio chip.

Juego de Instrucciones del ARM

El conjunto de instrucciones del ARM comprende un total de 11 instrucciones básicas, que se pueden dividir en cuatro clases de instrucciones:

- Saltos
- Proceso de Datos, transferencias entre CPSR, SPSR y registros
- *load, store* (operaciones de memoria)
- Coprocesador

Todas las instrucciones del ARM pueden ser ejecutadas condicionalmente, en función del estado de las cuatro banderas del CPSR (N, Z, C, V).

En el capítulo siguiente se estudiará en profundidad todas las características enunciadas en este apartado.

2.5 Arquitectura del núcleo (CORE) del ARM8

El avance más significativo en la arquitectura del ARM8 con respecto a su antecesor, el ARM7, ha sido la extensión de la *pipeline* de 4 a 5 etapas. Esto significa que la ejecución de una instrucción se extiende sobre más ciclos, reduciendo el esfuerzo realizado por cada etapa, lo que implica una simplificación de la lógica, un menor retraso, y por tanto, se soportaran mayores frecuencias de reloj.

El papel de los compiladores es importantísimo en este tipo de procesadores, pues los efectos de fenómenos tales como el interlocking, los saltos, etc, que disminuyen notablemente el rendimiento, pueden ser reducidos con una correcta reorganización de las instrucciones, que evite conflictos entre etapas.

Otra nueva característica se encuentra en la ALU: ahora el shifter y el sumador pueden operar en paralelo, en lugar de en serie. Ahora un desplazamiento y una suma se realizan en el mismo ciclo de reloj, y además, al estar en paralelo, el retardo combinacional es menor, repercutiendo este hecho en un nuevo aumento de la frecuencia de reloj.

El multiplicador del ARM8 es mucho mayor que el del ARM7.

Además, todo el código existente hasta el ARM8 puede correr en este procesador sin ningún problema, pues no se han añadido nuevas instrucciones (sólo se ha mejorado la arquitectura).

Aunque esta arquitectura no es completamente harvard, podríamos decir que se trata de una arquitectura mixta a medio camino entre la arquitectura Von Neumann y la arquitectura harvard; este hecho se puede apreciar en el apartado 2.7

2.6 Arquitectura del núcleo (CORE) del ARM9

La versión comercial que existe del CORE del ARM9 se conoce como ARM9TDMI, y se corresponde con la arquitectura ARMv4T expuesta en el apartado 2.3 del presente capítulo (*aunque nuestro diseño utilizará arquitectura ARMv4*).

Aunque es capaz de decodificar una ISA adicional al estándar de ARM (conocido como THUMB), este hecho no tiene porque implicar una modificación de la arquitectura hardware, pues existe un proceso de descompresión de las instrucciones THUMB, en el cual éstas son mapeadas en instrucciones ARM, de manera que éste ISA pasa inadvertido para el resto de la arquitectura del microprocesador.

Las diferencias más importante entre ambos COREs son:

- Arquitectura totalmente harvard en el ARM9
- Posibilidad de generación de estados de espera al acceder a RAM

2.7 Núcleo (CORE) del ARM8

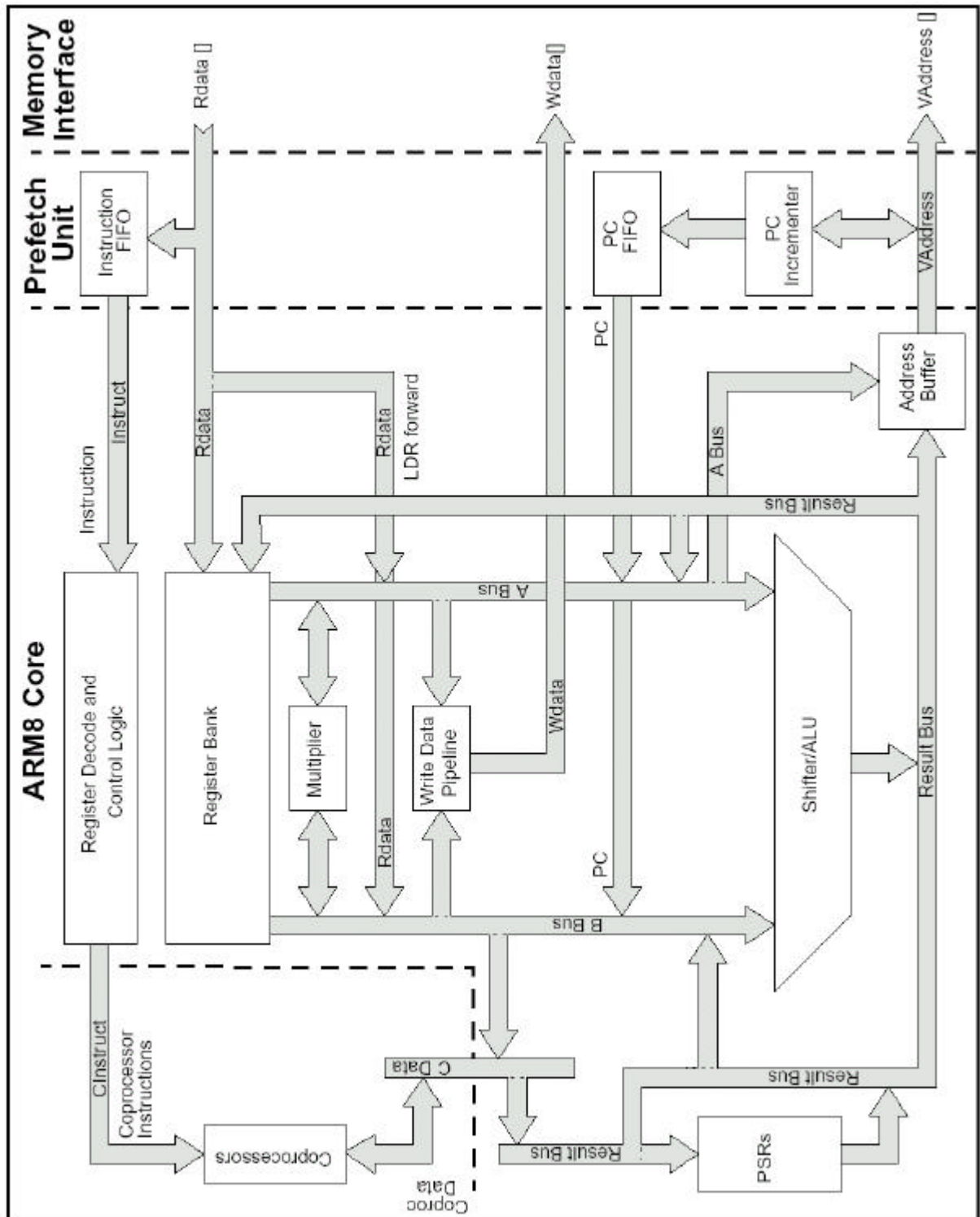


Figura 2-1: Arquitectura del CORE del ARM8

2.8 Núcleo (CORE) del ARM9

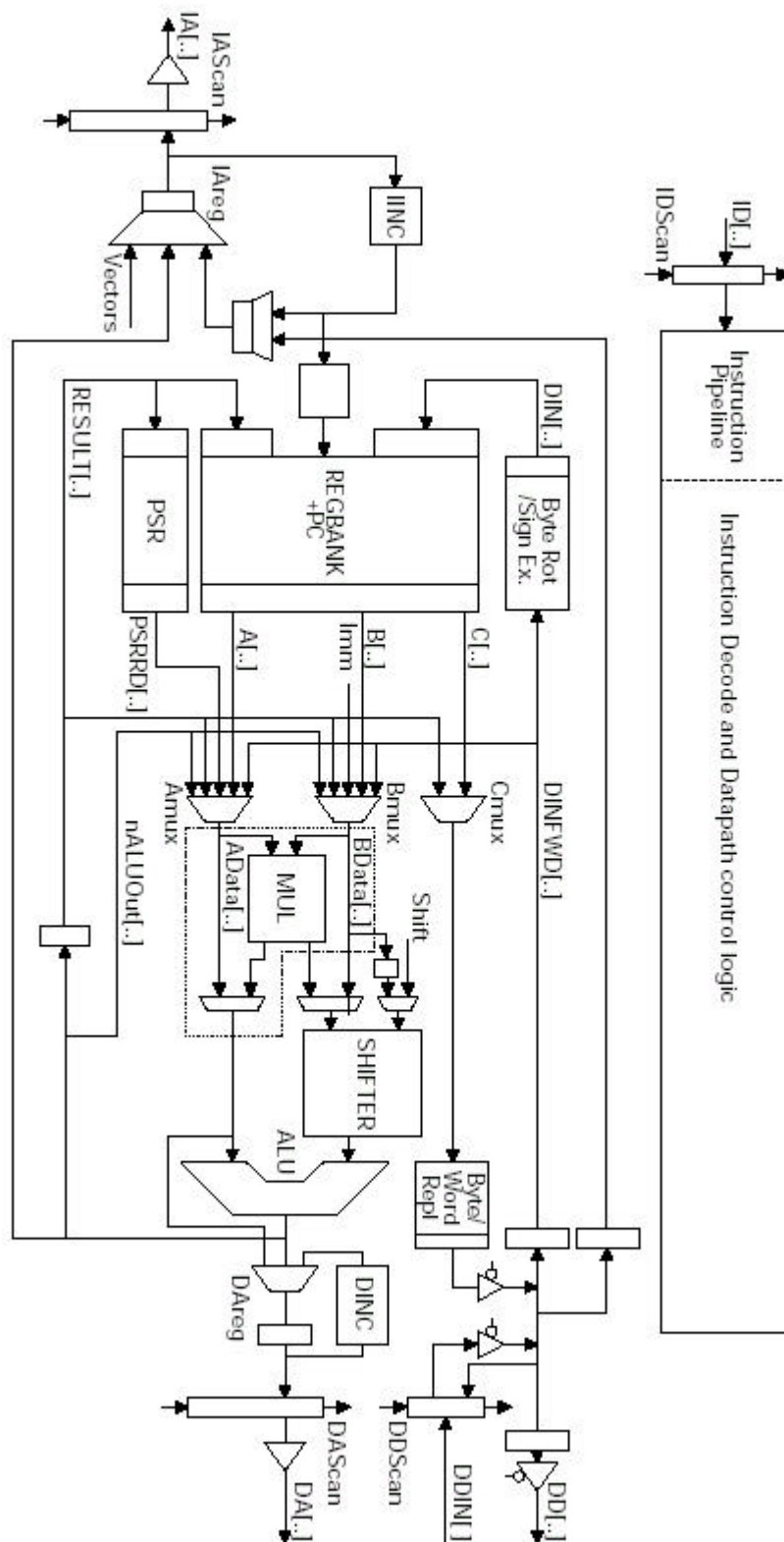


Figura 2-2: Diagrama de bloques del ARM9

3

Modelo del Programador

Este modelo se refiere al CORE implementado en este proyecto, y a la arquitectura ARMv4. Se ha incluido en el documento con una doble intención:

- Por un lado, servir de descripción detallada de las características generales de los procesadores ARM que brevemente se expusieron en el capítulo 2.
- Y por otro, cumplir la función propia de un “modelo del programador”, es decir, servir al futuro usuario del CORE generado para poder introducirlo en un sistema propio y ser capaz de configurarlo y programarlo correctamente (para ésta última tarea no basta con este modelo, sino que el lector tendrá que consultar también el capítulo 4 referente al juego de instrucciones del CORE del procesador ARM con arquitectura ARMv4 implementado en este proyecto)

3.1 Configuración Hardware

3.1.1 La señal BIGEND

3.1.2 Generación de estados de espera

3.2 Modos de Operación

3.3 Banco de Registros

3.3.1 Registro de Estado del Programa (PSR)

3.4 Excepciones

3.4.1 FIQ

3.4.2 IRQ

3.4.3 Fallos de Memoria (Memory ABORT)

3.4.4 Interrupción Software (SWI)

3.4.5 Instrucciones no reconocidas

3.4.6 Sumario de Vectores de Interrupción

3.4.7 Prioridades

3.5 Reset global

3.1 Configuración Hardware

3.1.1 La señal BIGEND

Típicamente existen dos tendencias de “numeración” de los bytes dentro de la palabra de memoria: Big-Endian y Little-Endian. Para el CORE del procesador ARM8 propiamente dicho sólo existe la segunda tendencia, e internamente trata las palabras de memoria como si fuesen little-endian, pero ha sido dotado de una interfaz que admite los dos formatos, e internamente realiza la conversión. Esta interfaz es lo que configura la señal BIGEND. Si el usuario, en su aplicación, desea realizar una interfaz propia, puede hacerlo, pero no debe perder de vista la existencia de esta señal.

Formato Little-Endian

Se configura asignando a la señal BIGEND un valor bajo (BIGEND=LOW). Así, el byte menos significativo de la palabra de memoria (word) se considerará como el byte 0, y al byte más significativo se le asignará el valor 3. O mejor dicho, como norma general, el byte menos significativo será el de numeración más baja, y el más significativo el de numeración más alta.

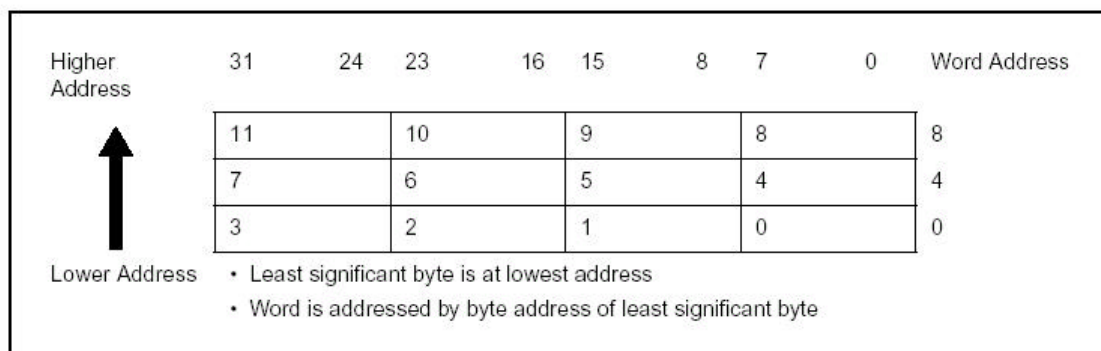


Figura 3-1: Formato Little-Endian

Formato Big-Endian

Si asignamos un valor alto a la señal BIGEN (BIGEN=HIGH), entonces el byte más significativo de la palabra de memoria será almacenado en el byte etiquetado con la numeración más baja, y viceversa. El byte 0 del sistema de memoria será conectado a las líneas 24-31 del bus de datos.

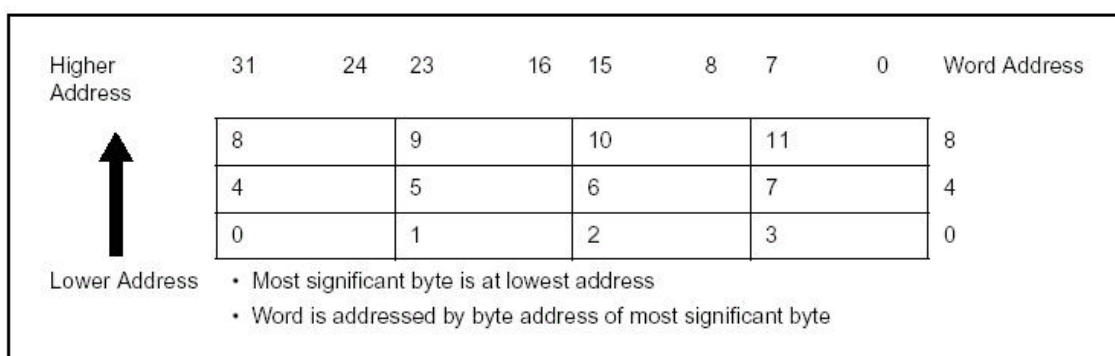


Figura 3-2: numeración Big-Endian

3.1.2 Generación de Estados de Espera

El CORE implementado posee dos bits (WS[1:0]) que se usan para pedir al procesador que genere ciclos o estados de espera a la hora de acceder a memoria externa. Esto es de gran utilidad cuando tenemos memorias o periféricos mapeados en memoria, que necesiten más de un ciclo de reloj para tener listo el dato.

En la tabla siguiente se adjuntan las posibles configuraciones:

ws[1:0]	Nº estados de espera	Ciclo de Acceso
“00”	0	El acceso dura 1 ciclo de reloj
“01”	1	El acceso dura 2 ciclos de reloj
“10”	2	El acceso dura 3 ciclos de reloj
“11”	3	El acceso dura 4 ciclos de reloj

Ejemplo de Uso

Imaginemos que tenemos una memoria DRAM que necesita para ser accedida al menos dos ciclos de reloj (pues ese es su ciclo de acceso: ver apartado 1.3).

Para poder utilizar esa memoria con nuestro CORE, basta con asignar a la señal externa ws[1:0] el valor “01”, y crear una interfaz entre la propia interfaz del CORE y la interfaz requerida por la memoria DRAM.

Una vez configurada la señal ws[1:0] con un valor distinto de “00”, cada vez que se produzca un acceso a memoria, éste durará al menos los ciclos de reloj descritos en la tabla anterior.

3.2 Modos de Operación

ARM soporta siete modos de operación diferentes. Para cambiar de modo, podemos utilizar las instrucciones software que se han concebido para ese fin, y que se describen en el siguiente capítulo.

También ARM8 cambia automáticamente de modo cuando se produce una excepción (para más información, consultar el apartado 3.4. “Excepciones”).

Modo	Descripción
User	ejecución normal del programa
FIQ	se usa para atender interrupciones de alta prioridad
IRQ	se usa para atender interrupciones de propósito general
Supervisor	modo protegido par al sistema operativo
System	modo privilegiado para el usuario
Abort	se usa cuando se produce un fallo de memoria
Undefined	se entra en este modo cuando se intenta ejecutar una instrucción no definida

3.3 Registros

El ARM cuenta con un total de 37 registros de 32 bits, 31 de los cuales son de propósito general, y los seis restantes son registros de estado.

No todos los registros pueden “ser vistos” al mismo tiempo: en un instante dado sólo 16 registros de propósito general pueden ser accedidos, y uno o dos registros de estado son visibles, en función del modo de operación. La situación anteriormente descrita, se puede representar de la siguiente forma:

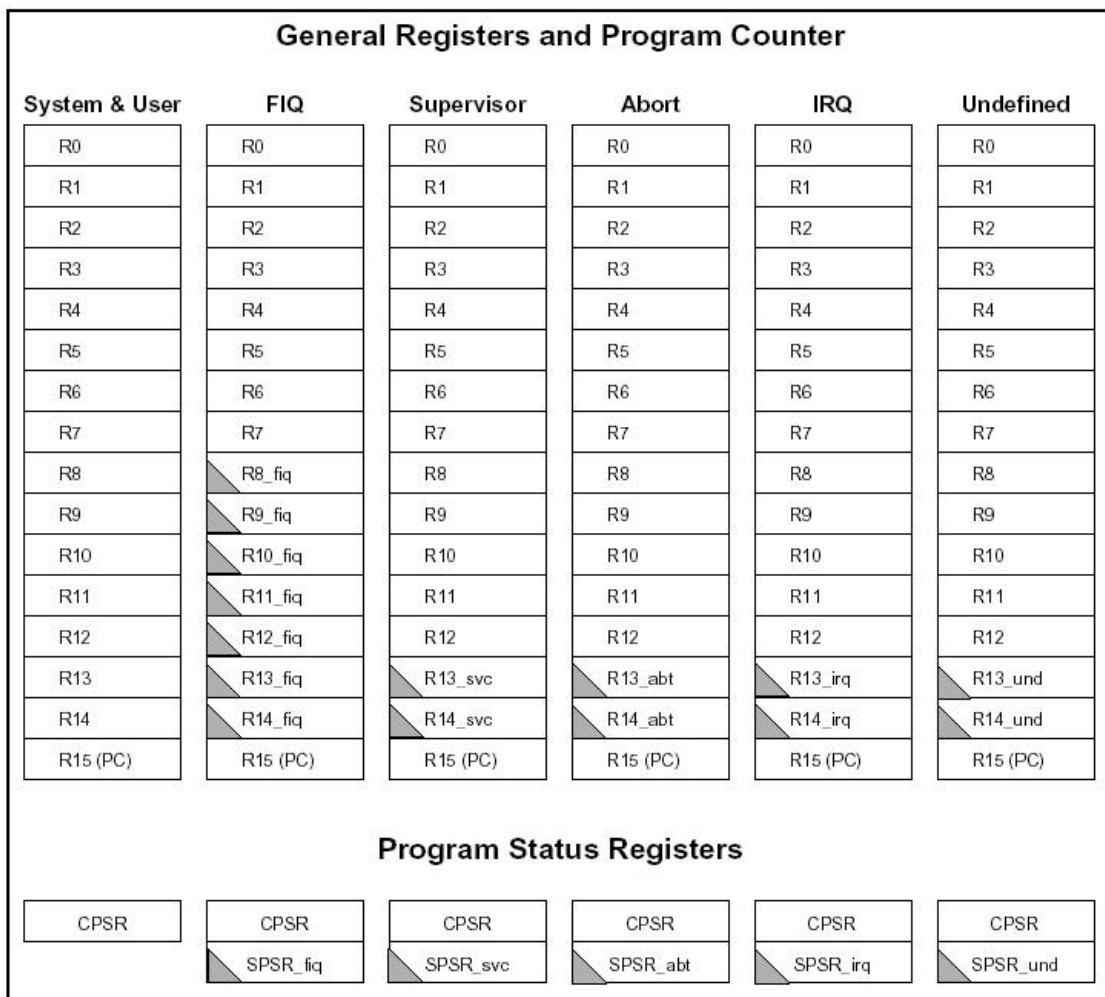


Figura 3-3: banco de registros

- En todos los modos hay 16 registros directamente accesibles: R0-R15
- Todos son de propósito general, excepto R15, que contiene el contador de programa (PC). Como las instrucciones son de 32 bits (4 bytes), los bits [1:0] del PC son ignorados, y deben estar siempre a '0'.
- R14 es utilizado como el registro de enlace para retorno de una subrutina (link register o LR). En el LR, al producirse un salto con retorno (branch with link ó BL), se almacena una copia del PC, para después poder volver. También al producirse ciertas excepciones, en R14_excepción se guarda la dirección de retorno.

- R13 se suele utilizar para guardar el puntero de pila (stack pointer ó SP). Por eso cada modo cuenta con un R13_modulo_actual: para así poder mantener el puntero de pila correspondiente a la subrutina de atención de la interrupción en cuestión, mientras que en R13 se mantiene el valor que existía antes de producirse el cambio de modo y así, al volver de una interrupción, todo se mantiene intacto.
- Existe otro registro accesible, el CPSR (Registro de Estado Actual del Programa, ó Current Program State Register). Además, existen cinco SPSRs (Saved Program Status Registers), en los cuales se ubica una copia del CPSR cuando se produce una excepción, y así, al volver de la misma, el CPSR es restaurado a su valor original (*no existen SPSR para el modo user ni para el modo system porque ninguna excepción entra en ese modo*).

3.3.1 Registro de Estado del Programa (PSR)

La siguiente figura muestra la estructura del PSR:

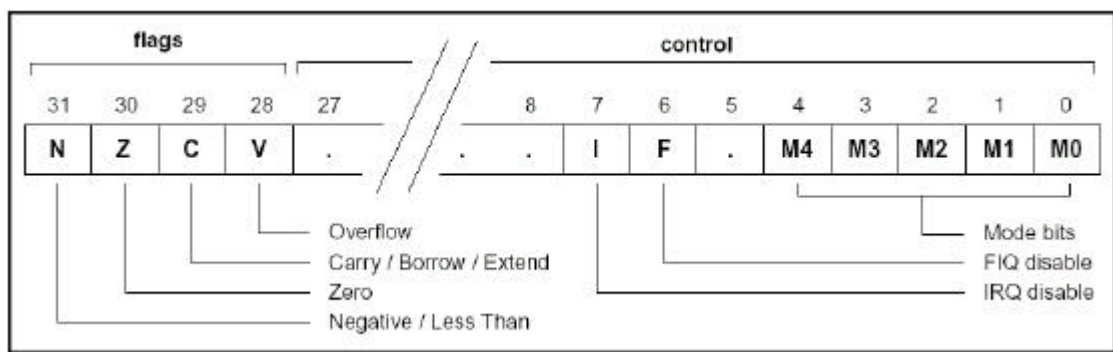


Figura 3-4: estructura del CPSR

Banderas (flags) de condición

Los bits N,Z,C,V son actualizados como resultado de una operación aritmética o lógica en la ALU, y deben ser comprobados por la etapa de ejecución antes de ejecutar una instrucción (ejecución condicional de las instrucciones, con acuerdo a estos bits).

- bit C: (carry) es el bit de acarreo de la operación
- bit N: (negative) se activa si el resultado es negativo
- bit Z: (zero) se activa si el resultado es cero (todos los bits a '0')
- bit V: (overflow) se activa si se produce un overflow

Bits de desactivación de interrupciones

Los bits I y F desactivan las interrupciones. Si alguno de ellos está activo, la respectiva interrupción, IRQ ó FIQ respectivamente, estará desactivada.

Bits de Modo

M[4:0] son los bits de modo, y su valor determina el modo en el que el procesador opera en un momento dado. La interpretación de estos bits se muestra en la tabla adjunta (no todas las combinaciones son posibles, y el programador sólo debe utilizar las que se describen a continuación).

M[4:0]	Modo	Registros Accesibles	
10000	User	PC,R14...R0	CPSR
10001	FIQ	PC,R14_fiq,R8_fiq,R7..R0	CPSR, SPSR_fiq
10010	IRQ	PC,R14_irq,R13_irq,R12...R0	CPSR, SPSR_irq
10011	Supervisor	PC,R14_svc,R13_svc,R12...R0	CPSR, SPSR_svc
10111	Abort	PC,R14_abt,R13_abt,R12...R0	CPSR, SPSR_abt
11011	Undefined	PC,R14_und,R13_und,R12...R0	CPSR, SPSR_und
11111	System	PC, R14...R0	CPSR

Bits de control

Los 8 bits menos significativos del PSR (incluidos I,F y M[4:0]) se conocen en conjunto como los *bits de control*. Estos bits son alterados cuando se produce una excepción. Si el procesador esta operando en modo privilegiado, solo pueden manipularse por software.

Bits reservados

El resto de los bits del PSR están *reservados*. Cuando se altere un flag o bit de control en el PSR, el programador debe asegurarse que no se cambia ningún bit de los que están reservados. Así podrá estar seguro que su programa podrá ejecutarse en futuros procesadores sin provocar ningún conflicto (pues puede que en futuras versiones del procesador esos bits tengan algún cometido concreto)

3.4 Excepciones

Una excepción tiene lugar cuando es necesario romper el flujo normal de ejecución del programa porque por ejemplo, el procesador debe ocuparse de atender una interrupción procedente de un periférico.

Cuando el ARM8/9 comienza a atender una interrupción, hace uso del banco de registros para salvar el estado actual:

- el PC antiguo y el contenido del CPSR son copiados en los registros R14 y SPSR apropiados, en función de la situación.
- los bits del PC y del CPSR son forzados a un valor que depende de la excepción.

En el caso de que el programador desee admitir interrupciones reentrantes, debe encargarse de salvar R14 y CPSR en una pila en memoria (cuando se transfiera el SPSR a memoria, ***es importante almacenar los 32 bits completos y no solo los bits de control o los flags***).

Nótese que al entrar en una excepción, las interrupciones son enmascaradas por el ARM, de modo que el programa tiene opción de salvar en la pila los registros deseados, y volver a habilitar las excepciones, para permitir así interrupciones reentrantes.

Cuando se produzcan simultáneamente varias interrupciones, una circuitería combinacional se encarga de determinar en orden en función de las prioridades expuestas en el apartado 3.4.7)

3.4.1 FIQ

Una excepción FIQ (Fast Interrupt reQuest) es generada externamente, activando la señal externa nFIQ (activa a nivel bajo). Este tipo de excepciones han sido ideadas para soportar interrupciones de alta prioridad, y cuenta con suficientes registros para cubrir todas las necesidades de almacenamiento temporal que en dicha aplicación pudieran surgir.

Para enmascararlas (desactivarlas), basta con activar el bit F del CPSR (cosa que no es posible desde el modo *user*). Cuando se detecta una interrupción FIQ, el ARM8/9:

- salva la dirección de la próxima instrucción en ejecutarse más 4 en R14_fiq
- almacena el CPSR en SPSR_fiq
- fuerza M[4:0]=”10001” (modo FIQ) y los bits I, F del CPSR se ponen a ‘1’
- actualiza el PC al valor del vector FIQ (ver apartado 3.4.6)

Para volver normalmente de la rutina de atención de la excepción basta con usar *SUBS PC, R14_fiq, #4*. Esto restaura el PC (de R14_fiq) y el CPSR (del SPSR_fiq).

3.4.2 IRQ

Una excepción IRQ (Interrupt ReQuest) es generada externamente, activando la señal externa nIRQ (activa a nivel bajo). Este tipo de excepciones han sido ideadas para soportar interrupciones de propósito general, y tienen una prioridad inferior a las FIQ, de modo que son automáticamente enmascaradas cuando se produce una interrupción FIQ.

Si se detecta una interrupción IRQ, el ARM8/9:

- salva la dirección de la próxima instrucción en ejecutarse más 4 en R14_irq
- almacena el CPSR en SPSR_irq
- fuerza M[4:0]=”10010” (modo IRQ) y el bit I del CPSR se pone a ‘1’.
- actualiza el PC al valor del vector IRQ (ver apartado 3.4.6)

Para volver normalmente de una excepción IRQ, basta con utilizar la instrucción *SUBS PC, R14_IRQ, #4*. Esto restaura el PC desde R14_irq y el CPSR desde el SPSR_irq.

3.4.3 Fallos de Memoria (Memory ABORT)

Puede ocurrir que una transacción entre el sistema de memoria de datos y el CORE no se complete con éxito, por causas muy distintas (por ejemplo, la dirección suministrada a la memoria principal no existe, o es incorrecta).

Si durante una lectura o escritura de memoria se produce un ABORT, entonces el registro destino y el registro base deben ser corregidos, y restaurados a su valor original (antes de que se produjese el ABORT), y además el ARM8/9 debe:

- salvar la dirección la instrucción que causó el ABORT más 8 en R14_abt
- almacenar el CPSR en SPSR_abt
- forzar M[4:0]=”10111” (modo ABORT) y el bit I del CPSR se pone a ‘1’.
- actualizar el PC al valor del vector ABORT (ver apartado 3.4.6)

Retorno de un ABORT: tras encontrar la causa del fallo de memoria, basta con utilizar *SUBS PC, R14_abt, #8*.

3.4.4 Interrupción Software (SWI)

Se utilizan para realizar llamadas al sistema operativo. Cuando se produce una interrupción software, el sistema entra en modo Supervisor, pero ejecutando código controlado por el sistema operativo. Cuando se produce una SWI, el ARM8/9:

- salva la dirección de la instrucción SWI más 4 en R14_svc

- almacena el CPSR en SPSR_svc
- fuerza M[4:0]=”10011” (modo Supervisor) y el bit I del CPSR se pone a ‘1’.
- actualiza el PC al valor del vector SWI (ver apartado 3.4.6)

Para volver de una SWI, basta con usar *MOVS PC, R14_svc*. Esto restaura el valor adecuado del PC (desde R14_svc), y el CPSR (desde SPSR_svc).

3.4.5 Instrucciones no reconocidas

En el juego de instrucciones del ARM existe un tipo de instrucción denominado “no definida”, que no se utiliza en ninguna versión del ARM, y que se conoce como *instrucción de espacio no definido*, ya que genera un espacio de instrucciones que no son reconocidas por el decodificador del procesador. Esta instrucción constituye uno de los mecanismos que originan la interrupción por instrucción no reconocida.

También se genera la excepción si se intenta ejecutar una instrucción de coprocesador, y éste no está físicamente (hardware) presente. Al no obtener respuesta, se genera una excepción por instrucción no reconocida, y mediante la rutina de atención a la excepción se puede implementar una emulación software del coprocesador.

Cuando el ARM8/9 detecta la excepción:

- salva la dirección de la instrucción no reconocida más 4 en R14_und
- almacena el CPSR en SPSR_und
- fuerza M[4:0]=”11011” (modo Undefined) y el bit I del CPSR se pone a ‘1’.
- actualiza el PC al valor del vector Undefined (ver apartado 3.4.6)

Para volver de esta interrupción, tras ser atendida, se utiliza *MOVS PC, R14_und*. Esto restaura el PC desde R14_und y restituye el CPSR desde el SPSR_und.

3.4.6 Sumario de Vectores de Excepción

Cuando se produce una excepción, y tras salvar el PC en R14_adequado, el contador de programa se actualiza al valor del vector de la excepción correspondiente.

Dirección	Excepción	Modo de Entrada
0x00000000	Reset	Supervisor
0x00000004	Instrucción no reconocida	Undefined
0x00000008	SWI	Supervisor
0x0000000C	(**)	----
0x00000010	ABORT	Abort
0x00000014	-- Reservado --	----
0x00000018	IRQ	IRQ
0x0000001C	FIQ	FIQ

(**) No disponible en esta implementación del CORE del ARM8/9

3.4.7 Prioridades

Cuando se producen múltiples interrupciones al mismo tiempo, existe un sistema puramente combinacional, que se encarga en determinar el orden de atención en función de la siguiente asignación de prioridades:

1. Reset (prioridad más alta)
2. ABORT
3. FIQ
4. IRQ
5. Instrucción no reconocida y SWI

No todas las interrupciones se pueden dar al mismo tiempo: por ejemplo, SWI y una instrucción no reconocida es evidente que son mutuamente exclusivas.

Parece contradictorio que si las excepciones FIQ se llaman “prioritarias”, se les asigne menos prioridad que al ABORT, pero esto tiene su lógica: si ocurre un fallo (ABORT) en memoria y al mismo tiempo una interrupción FIQ, y el bit F del CPSR está a ‘0’ (activo), el procesador ARM entrará primero en el proceso de atención al ABORT y cuando solucione el problema, entrará inmediatamente en la rutina de atención de excepciones FIQ (vector FIQ).

En cambio, si las interrupciones FIQ fuesen más prioritarias, se entraría en el ABORT, e inmediatamente en la rutina de atención de la interrupción FIQ, de modo que se ejecutaría dicha rutina sin que el fallo de memoria estuviese solucionado. Al volver normalmente de la rutina, se solucionaría el fallo, pero es imposible saber lo que ha ocurrido mientras tanto.

3.5 Reset

Un ‘reset’ puede ser insertado asincrónicamente, simplemente con activar la señal nRESET (activa a nivel bajo). En la implementación del CORE del ARM8/9 realizada, la señal nRESET ataca a todos los dispositivos presentes en el diseño capaces de almacenar de alguna forma el estado (es decir, a todos los biestables), de modo que al activarse dicha señal, se inicializan todos ellos a un valor conocido.

Cuando se produce un ‘reset’, el ARM8:

- se inicializan R14_svc y SPSR_svc con cualquier valor (0x00000000)
- se fuerza M[4:0]=10011 (modo supervisor) y los bits I, F del CPSR son inicializados a ‘1’ (interrupciones enmascaradas)
- se actualiza el PC al valor del vector de Reset (0x00000000)
- se reinician el resto de registros (los de propósito general a 0x00000000)

4

Juego de Instrucciones

En este capítulo se describe con cierto grado de detalle el conjunto de instrucciones que soporta el microprocesador ARM8/9 con arquitectura ARMv4, y sirve como complemento al apartado anterior para facilitar la utilización del CORE generado en este proyecto.

Además de proceder a la exposición de las instrucciones implementadas, se enumeran ciertas instrucciones de las que se ha prescindido en el presente diseño, así como los motivos que han conducido a su exclusión.

- 4.1 Sumario del juego de instrucciones del ARM
 - 4.1.1 Instrucciones implementadas en el CORE
 - 4.1.2 Instrucciones reservadas y restricciones de uso
- 4.2 Ejecución condicional: el campo de condición
- 4.3 Saltos (B, BL)
- 4.4 Instrucciones de proceso de datos
- 4.5 Instrucciones de transferencia con el PSR (MRS, MSR)
 - 4.5.1. Operandos en MSR
 - 4.5.2. Restricciones en los operandos
 - 4.5.3. Bits Reservados
- 4.6 Multiplicaciones (MUL, MLA)
- 4.7 Multiplicaciones de 64 bits (MULL, MLAL)
- 4.8 Transferencia simple de datos con memoria (LDR, STR)
- 4.9 Transferencia compleja de datos con memoria (halfwords)
 - 4.9.1 Direccionamiento autoindexado y offset
 - 4.9.2 Modos de funcionamiento
- 4.10 SWP (Single Data Swap)
- 4.11. Fallos de Memoria (Data Aborts)
- 4.12 SWI (Software Interrupt)
- 4.13 Instrucciones no reconocidas

4.1 Sumario del Juego de Instrucciones del ARM

El juego de instrucciones del ARM se puede resumir en el siguiente cuadro:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0									
Cond	0	0	I	Opcode				S	Rn				Rd				Operand 2																Data Processing / PSR Transfer							
Cond	0	0	0	0	0	0	0	A	S	Rd				Rn				Rs				1	0	0	1	Rm				Multiply										
Cond	0	0	0	0	1	U	A	S	RdHi				RdLo				Rn				1	0	0	1	Rm				Multiply Long											
Cond	0	0	0	1	0	B	0	0	Rn				Rd				0	0	0	0	1	0	0	1	Rm				Single Data Swap											
Cond	0	0	0	P	U	0	W	L	Rn				Rd				0	0	0	0	1	S	H	1	Rm				Halfword Data Transfer: register offset											
Cond	0	0	0	P	U	1	W	L	Rn				Rd				Offset				1	S	H	1	Offset				Halfword Data Transfer: immediate offset											
Cond	0	1	I	P	U	B	W	L	Rn				Rd				Offset																Single Data Transfer							
Cond	0	1	1																									1					Undefined							
Cond	1	0	0	P	U	S	W	L	Rn				Register List																					Block Data Transfer						
Cond	1	0	1	L	Offset																														Branch					
Cond	1	1	0	P	U	N	W	L	Rn				CRd				CP#				Offset																Coprocessor Data Transfer			
Cond	1	1	1	0	CP Opc				CRn				CRd				CP#				CP	0	CRm				Coprocessor Data Operation													
Cond	1	1	1	0	CP Opc				CRn				Rd				CP#				CP	1	CRm				Coprocessor Register Transfer													
Cond	1	1	1	1	Ignored by processor																														Software Interrupt					
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0									

Tabla 4-1: Juego de Instrucciones del ARMv4

4.1.1 Instrucciones implementadas en el CORE

En el diseño abarcado por este proyecto, se han implementado todas las instrucciones anteriores, con excepción de:

- Las instrucciones del Coprocesador: se ha prescindido de ellas, en primer lugar, porque no se dispone de las especificaciones del coprocesador. Además, resulta absurdo un coprocesador en una versión del ARM en la que se dispone de un multiplicador que realiza operaciones en 32 y 64 bits, tanto de enteros, como de operandos representados en complemento a2. De todos modos, el CORE implementado es capaz de decodificar estas instrucciones, y en los ficheros se dispone de una serie de “espacios comentados”, en los que la introducción de estas instrucciones podría realizarse con bastante sencillez. Además, en los próximos capítulos se detalla la configuración de los registros de control de la *pipeline* que se ha de adoptar si se desea la inclusión de esta familia de instrucciones

- Block Data Transfer: se ha prescindido de esta instrucción porque viola gravemente los principios de diseño de una estructura *pipeline*, puesto que su ejecución requiere de un número irregular y a priori indeterminado de ciclos de reloj (tantos como número de registros se desee transferir a memoria). De todos modos, el motivo principal no es ese, sino el hecho de que esta instrucción se puede implementar mediante software, utilizando las instrucciones de transferencia simple de memoria (LDR, STR)
- THUMB: aunque el único procesador comercial de la familia de ARM9 es el ARM9TDMI con arquitectura ARMv4T, se ha decidido que queda fuera de este proyecto todo lo relacionado con el juego de instrucciones THUMB de 16 bits, es decir, que vamos a diseñar un CORE con el mismo comportamiento hardware que el ARM9, pero que implementa una arquitectura ARMv4, similar a la que utilizaba su predecesor, el ARM8. Este es el motivo de que a lo largo de los capítulos anteriores se haya mencionado al microprocesador ARM8.

4.1.2 Instrucciones reservadas y restricciones de uso

En los microprocesadores ARM8/9, se producirá una excepción por instrucción no reconocida cuando el decodificador se encuentra con una secuencia de bits que no es capaz de reconocer.

De todos modos, existen algunos patrones de bits que aunque no están definidos, no deben causar excepción alguna, ya que están reservados para futuras versiones del juego de instrucciones.

Además, las instrucciones que en este capítulo se detallan no permiten todos los usos posibles, y algunas combinaciones de registros no son admitidas por el procesador. En el caso de ser utilizadas, el procesador podría mostrar un comportamiento inesperado, e incluso podría *alcanzar un estado interno inconsistente*. Se recomienda leer detenidamente todas las puntualizaciones realizadas al describir cada instrucción a lo largo del presente capítulo.

Las instrucciones de coprocesador también pueden causar una excepción por instrucción no reconocida, si el coprocesador (hardware) no está presente físicamente. Cuando se ejecuta una instrucción de este tipo y el coprocesador no responde, se produce una excepción; la rutina de atención a esta interrupción puede servir para emular un coprocesador mediante software.

En el último apartado de este capítulo se exponen los patrones de bits que provocarían una excepción por instrucción no reconocida.

4.2 Ejecución condicional: el campo de condición

Todas las instrucciones del ARM con ejecutadas condicionalmente. Esto quiere decir que su ejecución tiene o no lugar dependiendo del valor de los bits N, Z, C y V (flags) del registro CPSR. En la siguiente ilustración se muestra el significado de los bits del campo de condición:

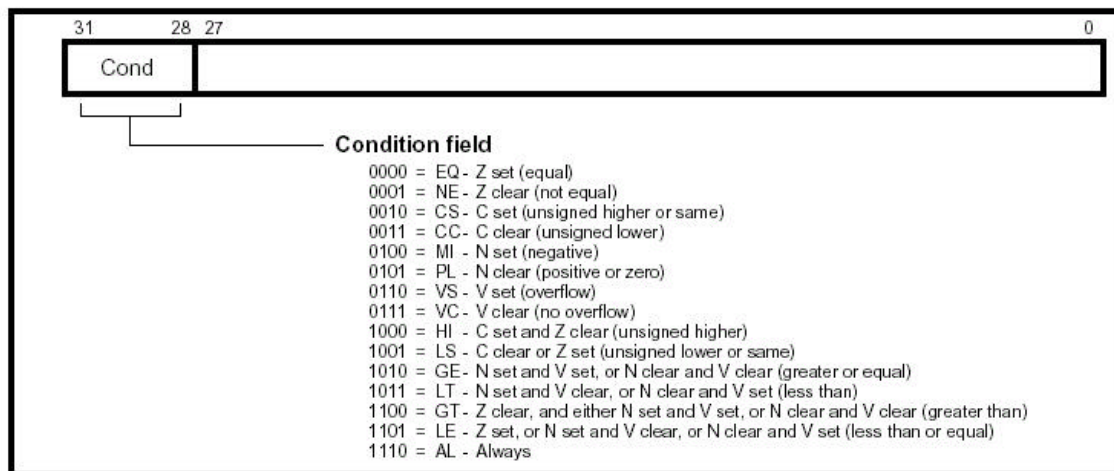


Figura 4-2: campo de condición

Si la condición especificada en una instrucción determinada es AL (always), su ejecución se producirá siempre, sea cual sea el contenido de los flags del CPSR.

El resto de códigos de condición tiene el significado especificado en la figura. Por ejemplo, el código “0000” (Igual) implica que la instrucción será ejecutada si y solo si el flag Z del CPSR está activo. Este es el caso de que en una instrucción de comparación (CMP) los dos operandos sean iguales (internamente el microprocesador realiza una resta, y si son iguales, el resultado será 0, con lo cual el bit Zero se activa). En caso contrario, si los operandos son distintos, la comparación obligará al CPSR a desactivar el bit Z (lo fuerza a cero), y la instrucción no será ejecutada.

Cuando se desee una no operación (NOP), debe utilizarse *MOV R0, R0*.

Nota: el código de condición “1111” está reservado, y no debe ser usado bajo ningún concepto, pues tiene significado interno para el CORE generado en este proyecto. Además, podría ser redefinido en futuras variantes de la arquitectura del ARM.

4.3 Saltos (B, BL)

Una instrucción de salto rompe el flujo normal de actualización del PC, actualizando éste a un valor que depende del *offset* especificado en un campo de la instrucción. Su ejecución sólo es posible si se valida la condición especificada en el campo correspondiente.

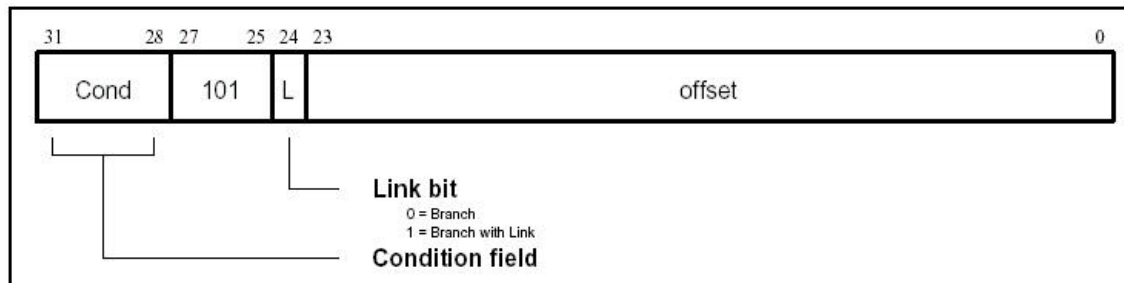


Figura 4-3: formato de la instrucción de salto

La instrucción de salto contiene un *offset* de 24 bits codificado en complemento a2. El valor especificado en este campo sufre el siguiente tratamiento:

- se le aplica un desplazamiento de dos bits a la izquierda
- se extiende en signo hasta completar 32 bits
- posteriormente es añadido al valor actual del PC

Por tanto, una instrucción puede especificar un salto de $\pm 32\text{MB}$. Si se desea especificar un salto que exceda $\pm 32\text{MB}$, el *offset* o la dirección absoluta debe ser cargada previamente en un registro. Si además se requiere que el salto sea con retorno, y el *offset* excede de $\pm 32\text{MB}$, el PC debe salvarse manualmente en R14 y el PC debe actualizarse bien con una operación ADD si se trata de un *offset*, o bien con una operación MOV si se trata de un valor absoluto.

Nota: Al especificar el *offset* es muy importante tener en cuenta el hecho de que el PC contiene un valor 8 bytes superior a la dirección de la instrucción actual, debido a las características de la arquitectura *pipeline*.

Uso del bit 'L' (the Link bit)

Los saltos con retorno (branch with link, ó BL) escriben el valor antiguo del PC en el registro de retorno (R14) del banco correspondiente al modo actual. En este proceso, se le resta 4 al valor almacenado, de modo que R14 contiene la dirección de la instrucción que sigue inmediatamente a la instrucción BL.

El CPSR no se salva al ejecutar esta instrucción.

Para volver desde la rutina invocada por el Salto con Retorno (BL), se debe utilizar *MOV PC, R14*, si el registro de enlace es todavía válido, o en su defecto *LDM Rn!, {..PC}* si la dirección de retorno se ha almacenado en una pila, y Rn es el puntero de pila (Stack Pointer).

4.4 Instrucciones de Proceso de Datos

Una instrucción de transferencia de datos solo es ejecutada si la condición especificada se verifica. El formato de esta instrucción es el siguiente:

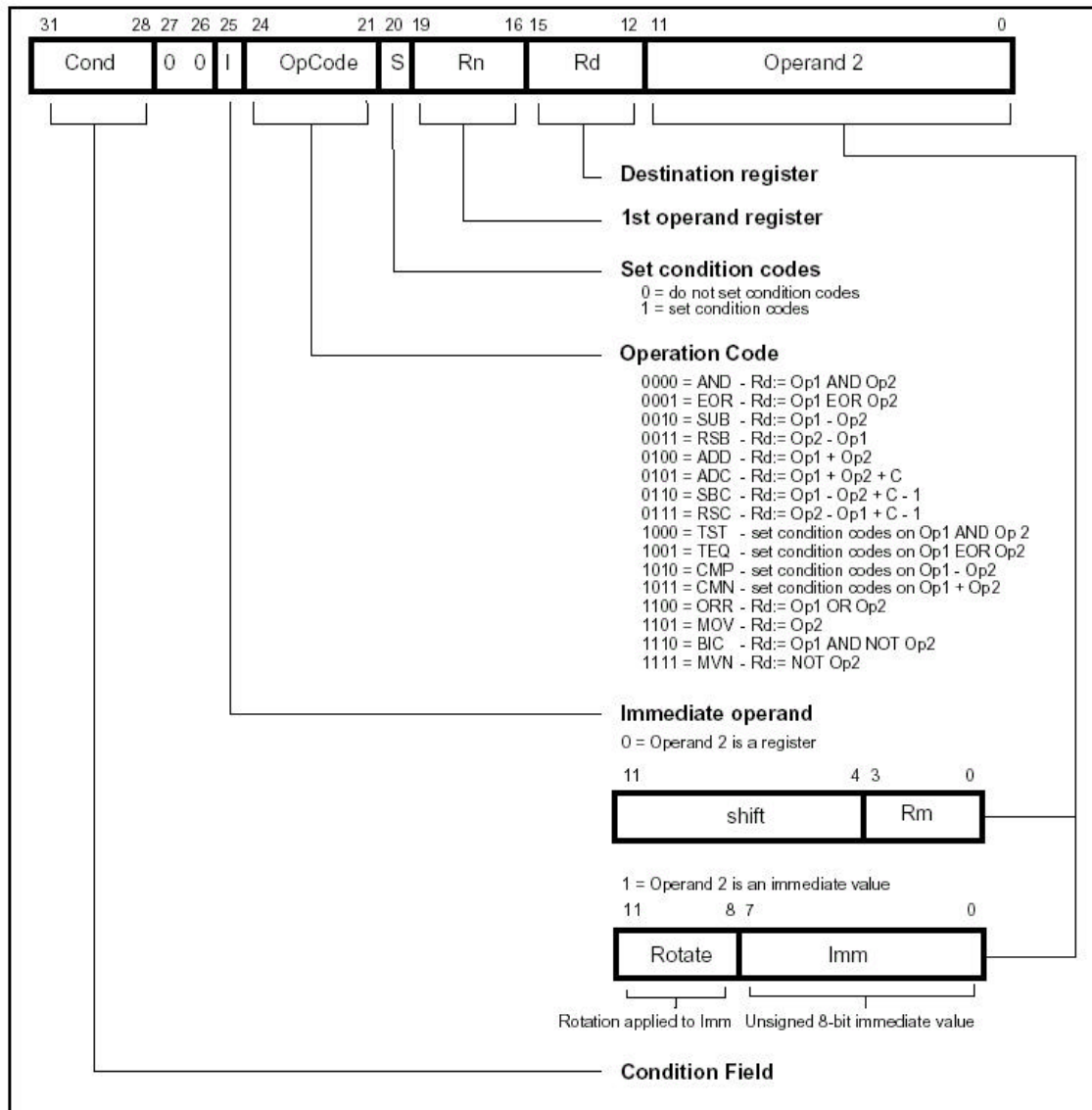


Figura 4-4: instrucciones de proceso de datos

Esta clase de instrucción genera un resultado, tomando para ello uno o dos operandos, de modo que:

- el primer operando es siempre un registro (Rn)
- el segundo operando puede ser un registro (Rm) desplazado (bit I=0), o un valor inmediato de 8-bit rotado o no (bit I=1)

Los flags del CPSR se mantienen o se actualizan con el resultado de la instrucción en función del valor del bit 'S'

Ciertas operaciones (TST, TEQ, CMP, CMN) no escriben resultado alguno en Rd. Simplemente se utilizan a modo de comprobación de condicione, y su misión consiste en actualizar los flags del CPSR en función del resultado. Por ello, este tipo de instrucciones siempre tiene el bit 'S' activo (lo contrario no tiene sentido).

A continuación se muestran las distintas instrucciones de proceso de datos, y sus efectos:

Assembler mnemonic	OpCode	Action	Note
AND	0000	operand1 AND operand2	
EOR	0001	operand1 EOR operand2	
SUB	0010	operand1 - operand2	
RSB	0011	operand2 - operand1	
ADD	0100	operand1 + operand2	
ADC	0101	operand1 + operand2 + carry	
SBC	0110	operand1 - operand2 + carry - 1	
RSC	0111	operand2 - operand1 + carry - 1	
TST	1000	as AND, but result is not written	Rd is ignored and should be 0x0000
TEQ	1001	as EOR, but result is not written	Rd is ignored and should be 0x0000
CMP	1010	as SUB, but result is not written	Rd is ignored and should be 0x0000
CMN	1011	as ADD, but result is not written	Rd is ignored and should be 0x0000
ORR	1100	operand1 OR operand2	
MOV	1101	operand2	Rn is ignored and should be 0x0000
BIC	1110	operand1 AND NOT operand2	Bit clear
MVN	1111	NOT operand2	Rn is ignored and should be 0x0000

Actualizando los flags del CPSR

Las instrucciones de proceso de datos se clasifican en lógicas y aritméticas:

Operaciones Lógicas

Las operaciones lógicas (AND, EOR, TST, TEQ, ORR, MOV, BIC, MVN) realizan una operación lógica indicada con la totalidad de los bits correspondientes a los operandos especificados, para producir un resultado. Si el bit S está activo (y Rd no es R15), se actuará sobre los flags del CPSR como sigue:

- N: es actualiza al valor del bit 31 del resultado
- Z: se activa si y solo si el resultado es todo cero
- C: se actualiza al valor de salida del carry del desplazador
- V: se mantiene su valor

Operaciones Aritméticas

Las operaciones aritméticas (SUB, RSB, ADD, ADC, SBC, RSC, CMP, CMN) tratan cada operando como si se tratase de un entero de 32 bits. Si el bit 'S' está activo y Rd es distinto de R15,

- N: se actualiza al valor del bit 31 del resultado. Esto indica si el resultado es negativo o no (si estamos considerando que trabajamos en complemento a2)
- Z: se pone a '1' si el resultado es todo cero
- C: toma el valor de salida del carry de la ALU
- V: se activa si ocurre un desbordamiento cuando estamos representando el valor en complemento a2. Esto quiere decir que si a un número positivo le sumamos otro número positivo, el resultado debe ser positivo; de lo contrario, hemos superado el umbral representable, aunque no exista desbordamiento. En terminología anglosajona, este hecho recibe el nombre de *signed overflow*

Desplazamientos (Shifts)

Cuando el segundo operando es un registro desplazado, el campo 'shift' de la instrucción controla la operación del desplazador (shifter).

- Indica el tipo de desplazamiento a realizar (lógico a derecha o izquierda, aritmético a derechas, o Rotación a derechas).
- Indica la cantidad de bit en la que el registro ha de ser desplazado, y puede ser especificada mediante un valor inmediato en el propio campo de la instrucción, o mediante el byte menos significativo de un registro de propósito general (distinto de R15).

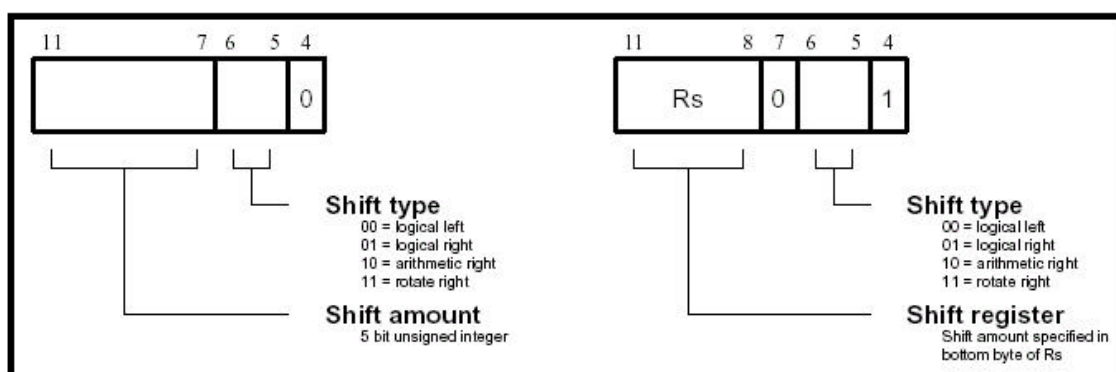


Figura 4-5: campos de control del shifter

Realización de desplazamientos

Desplazamientos especificados en la propia instrucción

Cuando la cantidad de bits a desplazar se especifica explícitamente en la instrucción, aparece contenida en un campo de 5 bits, que puede tomar cualquier valor entre 0 y 31.

- Un desplazamiento lógico a la izquierda (logical shift left LSL) toma el contenido de Rm, y mueve cada bit a una posición más significativa hasta completar la cantidad especificada. Los bits menos significativos que van quedando “vacíos” se rellenan con ceros. El último bit más significativo en “salir” del registro será el carry.
- Un desplazamiento lógico a la derecha (logical shift right LSR) sigue la misma mecánica, pero el sentido del desplazamiento es el contrario
- Un desplazamiento aritmético a derechas (Arithmetic Shift Right ASR) es similar al LSR, excepto que los bits más significativos son rellenos con el valor del bit 31 del registro Rm en lugar de con ceros (preservando así el signo si el valor almacenado en el registro está representado en complemento a2)
- Una rotación a la derecha (rotate right ROR) reutiliza los bits “desbordados” del registro para reintroducirllos por el extremo opuesto, como se indicaba en la figura anterior.

Algunos valores tienen un significado especial, que no coincide precisamente con el desplazamiento que aparentemente codifican, y que se detallan a continuación:

- LSL #0 opB = B Cout = Cin
- LSR #0 Se utiliza para codificar el caso LSR #32, de modo que el resultado del desplazamiento es 0x00000000 y Cout = B(31)
- ASR #0 Se utiliza para codificar el caso ASR #32, de modo que el resultado del desplazamiento será todo ‘0’ o todo ‘1’ en función del signo del operando B, es decir, Resultado <= (others => B(31)) y Cout = B(31)

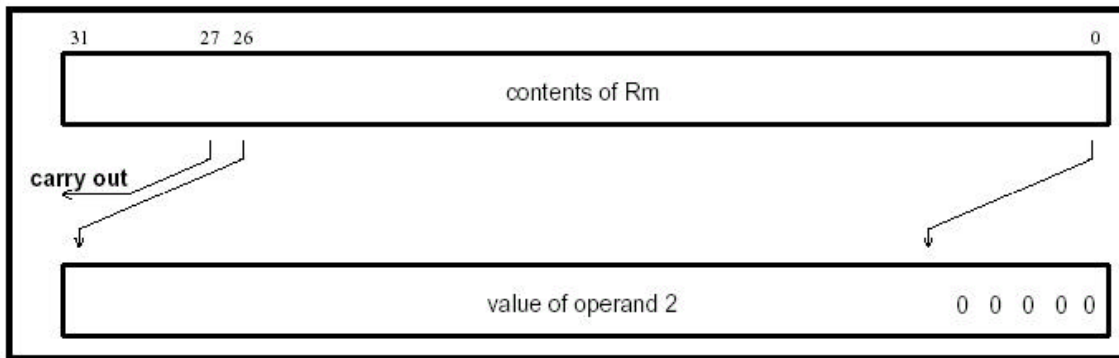


Figura 4-6: Desplazamiento Lógico a la Izquierda

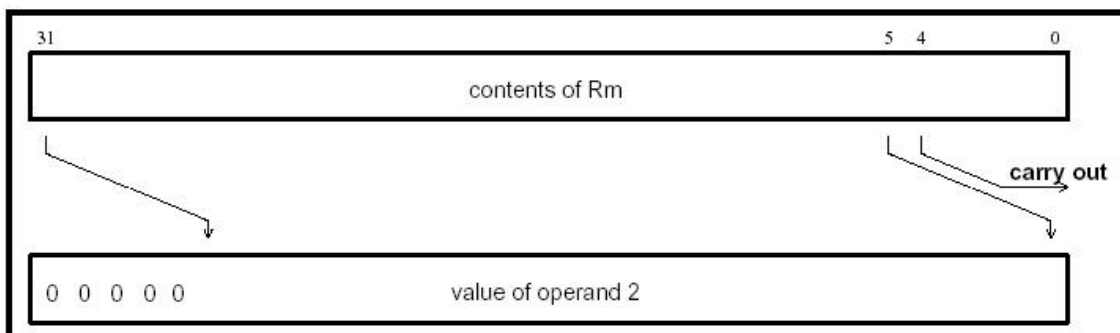


Figura 4-7: Desplazamiento Lógico a la Derecha

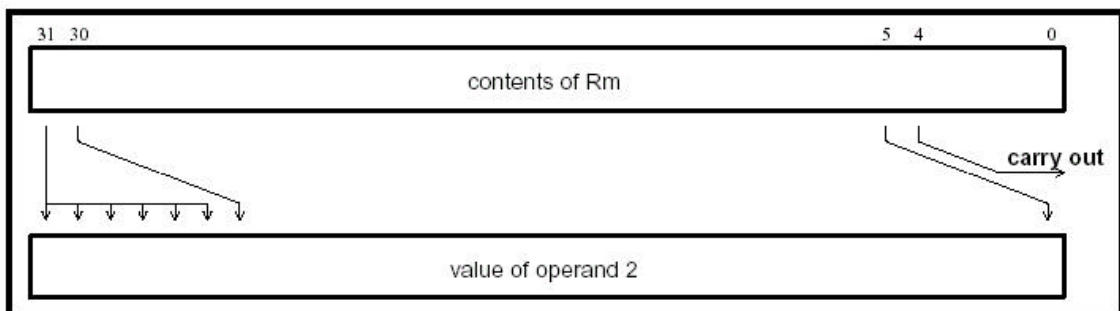


Figura 4-8: Desplazamiento Aritmético a la Derecha

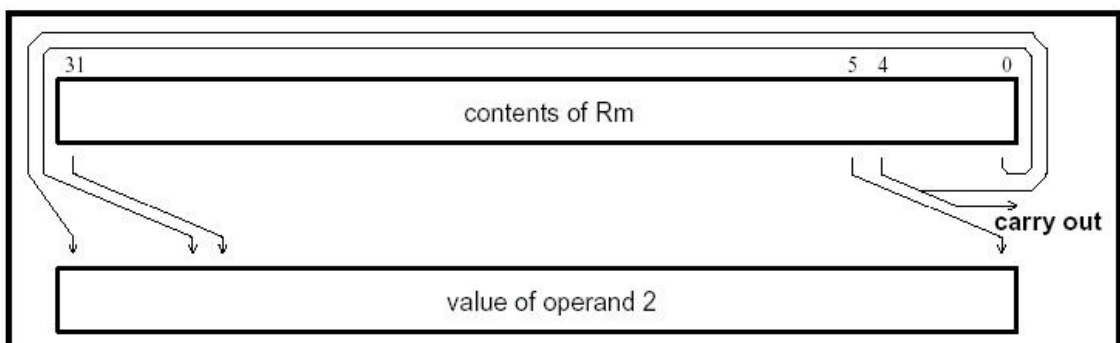


Figura 4-9: Rotación a la Derecha

Las figuras anteriores muestran el funcionamiento del desplazador del ARM8/9.

La cantidad de bits a desplazar es indicada por el byte menos significativo de Rs

El significado del desplazamiento estará en función del valor del byte menos significativo del registro Rs

Byte	Descripción
0	Se usa el valor inalterado de Rm como segundo operando, y el valor antiguo del CPSR se utiliza como salida del carry
1-31	Mismo significado que en el caso del valor inmediato
32	LSL: resultado 0, activa Z y carry=Rm[0] LSR: resultado 0, activa Z y carry=Rm[31] ASR: todos los bits y el carry son igual a Rm[31] ROR: el resultado es igual a Rm, y carry=Rm[31]
> 32	LSL: resultado 0, activa Z y carry='0' LSR: resultado 0, activa Z y carry='0' ASR: todos los bits y el carry son igual a Rm[31] ROR: #n, con $n > 32$, se hace $n-32$ tantas veces como sea necesario para que quede en el rango 1-32.

Valor Inmediato Rotado

El segundo operando también puede ser un valor inmediato, que se construye a partir del valor especificado en los 8 bits del propio campo de la instrucción dedicado a tal efecto, y realizándole las siguientes operaciones:

- se extiende a 32 bits rellenando con '0' (no se extiende en signo)
- se rota dos veces el valor especificado en el campo "Rotate"

Rotated Right Extended

Existe un caso excepcional, dado por ROR #0, que se usa para codificar una función especial del desplazador, que se denomina *rotated right extended (RRX)*. Su efecto sobre Rm es el siguiente:

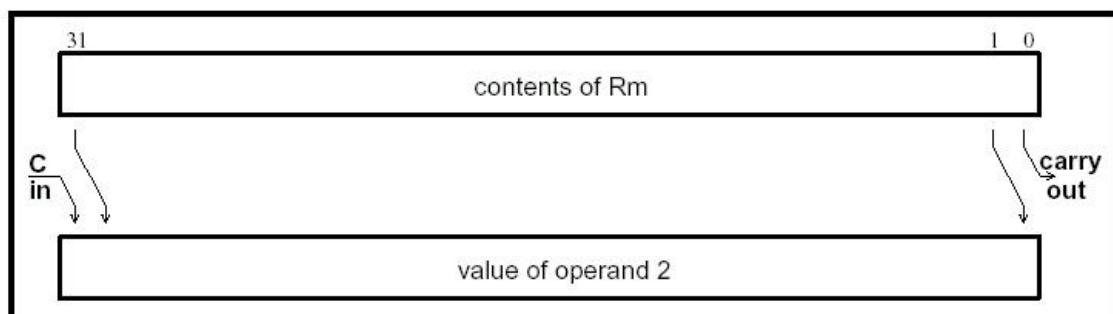


Figura 4-10: Rotated Right Extended (RRX)

Escribiendo R15

Cuando Rd es un registro distinto de R15, los flags del CPSR se actualizan directamente de los flags de la ALU si el bit S de la instrucción está activo.

En cambio, cuando Rd es igual a R15 y S=0, el resultado de la operación se almacena directamente en R15, y el CPSR queda inalterado.

Ahora bien, si Rd es R15 y S=1, el resultado de la operación es almacenado en R15, y el valor del SPSR correspondiente al modo actual es restaurado al CPSR. De esta forma, se recupera automáticamente el PC y el CPSR con una instrucción.

Nota: este uso de la instrucción no debe realizarse en modo *User* ni en modo *System*

Usando R15 como operando

Si R15 (PC) es utilizado como operando en una instrucción de proceso de datos, hay que tener en cuenta que el valor utilizado del PC será la dirección de la instrucción más 8 bytes.

No debe utilizarse R15 para control de desplazamientos, ni Rn o Rm pueden ser R15.

TEQ, TST, CMP, CMN, MOV y MVN

- Las primeras 4 instrucciones no escriben resultado alguno en registro, y solo sirven para actualizar el CPSR. Por ello deben llevar siempre el bit S activo.
- Cuando se usa MOV y MVN, Rn es ignorado, y debe ser siempre “0000”

4.5 Instrucciones de transferencia con el PSR (MRS, MSR)

MRS, MSR sólo son ejecutadas si la condición especificada en el campo correspondiente es cierta.

Realmente son un subgrupo de las instrucciones de proceso de datos, y son implementadas utilizando las instrucciones TEQ, TST, CMN y CMP, pero con el bit S desactivado.

MRS mueve el contenido del CPSR o del SPSR_modos_actual a un registro de propósito general. MSR, en cambio, hace la operación contraria, es decir, actualiza el CPSR ó el SPSR_modos bien al valor de un registro especificado, o bien a un valor inmediato contenido en la instrucción, pero afectando a:

- sólo los bits de flags
- sólo los bits de control
- ambos

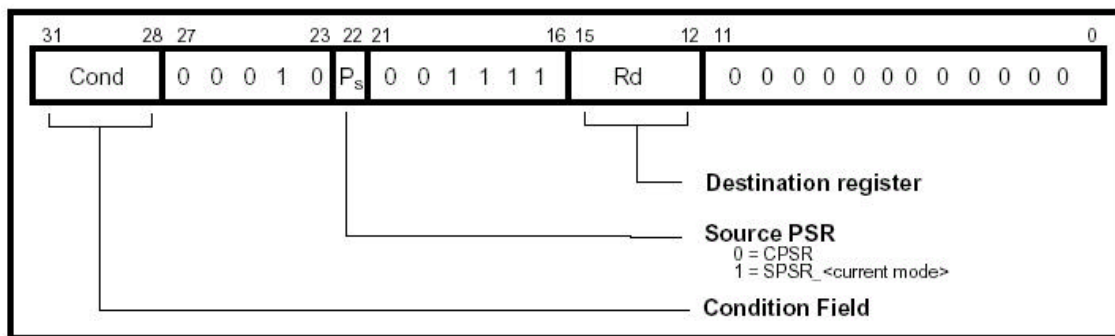


Figura 4-11: Formato de la instrucción MRS

4.5.1 Operandos en MSR

En la instrucción MSR se puede utilizar cualquier registro de propósito general que no sea R15.

Un operando inmediato se construye tomando los 8 bits del campo “Imm” (valor inmediato), extendiéndolo a 32 bits rellenando con ‘0’, y rotándolo dos veces el valor especificado en el campo “rotate”.

4.5.2 Restricciones en los operandos

En modo *user*, los bits de control del CPSR están protegidos, de modo que sólo los bits de condición pueden ser alterados. En otros modos (privilegiados), es posible modificar el contenido completo del CPSR.

El modo existente en el momento de la ejecución determina qué registro SPSR es accesible: por ejemplo, SPSR_fiq sólo puede ser accedido si el procesador se encuentra en modo FIQ.

R15 no puede ser especificado como registro fuente o registro destino.

Nota: no intentar acceder a un SPSR en modo *User* o en modo *System*, porque para el procesador no existe tal registro en los modos especificados.

4.5.3 Bits Reservados

Únicamente 11 bits del PSR están definidos en ARM8/9 (N, Z, C, V, I, F y M[4:0]). Los bits restantes están reservados para futuras versiones del procesador.

Para asegurar la máxima compatibilidad entre los programas del ARM8/9 y futuros procesadores, deberían adoptarse las siguientes medidas:

- los bits reservados deben ser preservados cuando se modifique el PSR
- los programas no deben proporcionar valores específicos para los bits reservados cuando alteren el estado del PSR, pues puede que en futuras versiones estos bits sean interpretables.

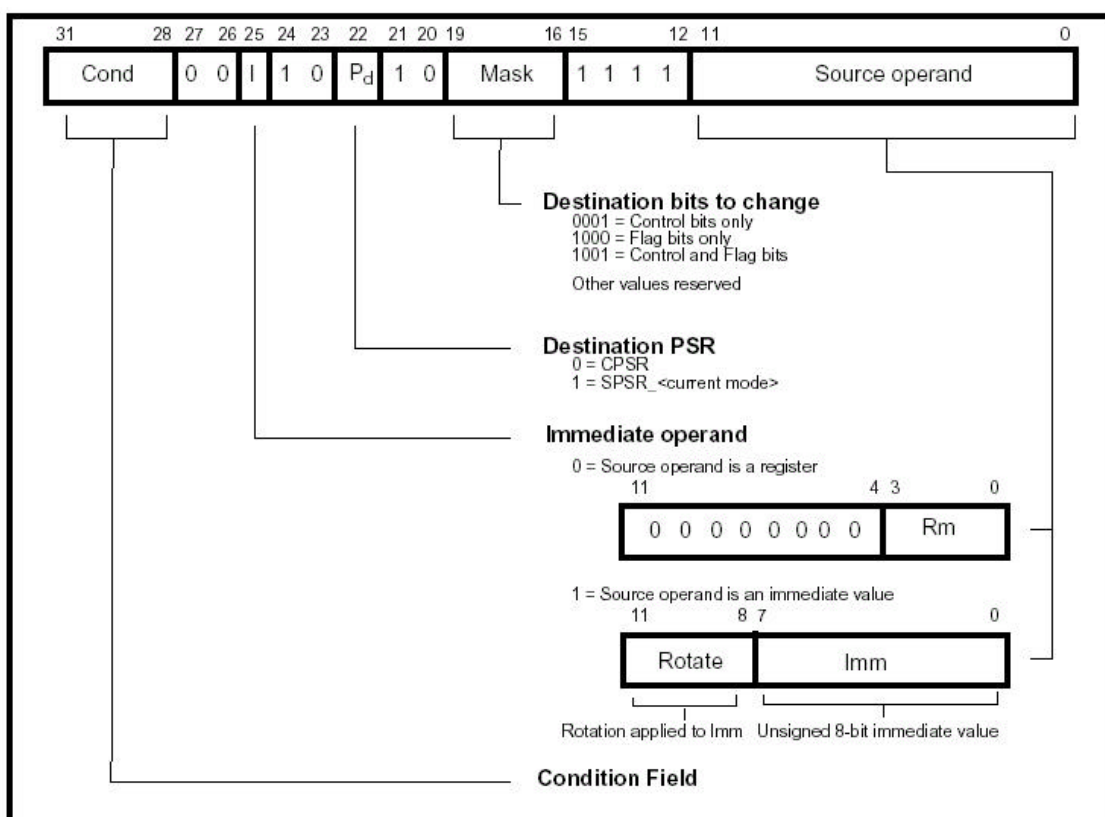


Figura 4-12: Formato de la instrucción MSR

4.6 Multiplicaciones (MUL, MLA)

Una multiplicación sólo es ejecutada si se verifica la condición especificada en el campo correspondiente.

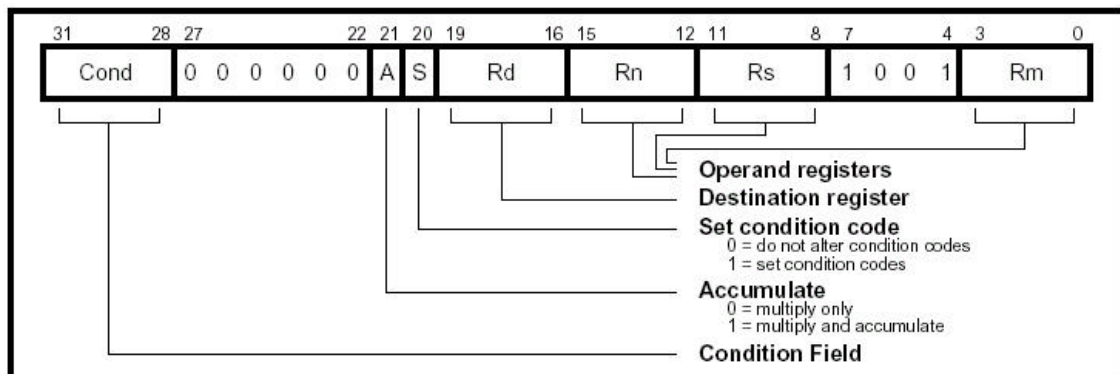


Figura 4-13: instrucciones de multiplicación

Las instrucciones de multiplicación (MUL) y multiplicación-acumulación (MLA) realizan una multiplicación de enteros, opcionalmente añadiéndole al resultado (suma) otro entero en el producto.

- **Multiplicación (MUL):** $Rd := Rm * Rs$. El operando Rn es ignorado, y debe ser "0000" por compatibilidad con futuras actualizaciones del juego de instrucciones.
- **Multiplicación y Acarreo (MLA):** $Rd := Rm * Rs + Rn$.

El resultado de una multiplicación con operandos de 32 bits codificados en complemento a2, difiere de una multiplicación de enteros sólo en bits superiores a los 32 bits representables en el resultado, luego para nuestro propósito podemos afirmar que los resultados de multiplicaciones con y sin signo son idénticos. Consideremos el siguiente ejemplo para ilustrar el hecho:

Operando A: 0xFFFFFFFF6 (-10 en Ca2, ó 4294967286 si se considera entero)

Operando B: 0x00000014 (20 en ambos casos)

Resultado (si los considero con signo): 0xFFFFFFFF38

Resultado (si los considero enteros): 0x13FFFFFF38

Restricciones de Operandos

- Rd debe ser distinto a Rm
- R15 no puede ser usado como Rd, Rm, Rn ó Rs

Efectos sobre los flags del CPSR

- N: igual al bit 31 del resultado
- Z: se activa si el resultado es todo '0'
- C: se actualiza a un valor carente de significado
- V: se queda inalterado.

4.7 Multiplicaciones de 64 bits (MULL, MLAL)

Una instrucción de multiplicación de 64 bits sólo es ejecutada si la condición especificada en el campo correspondiente, resulta ser cierta.

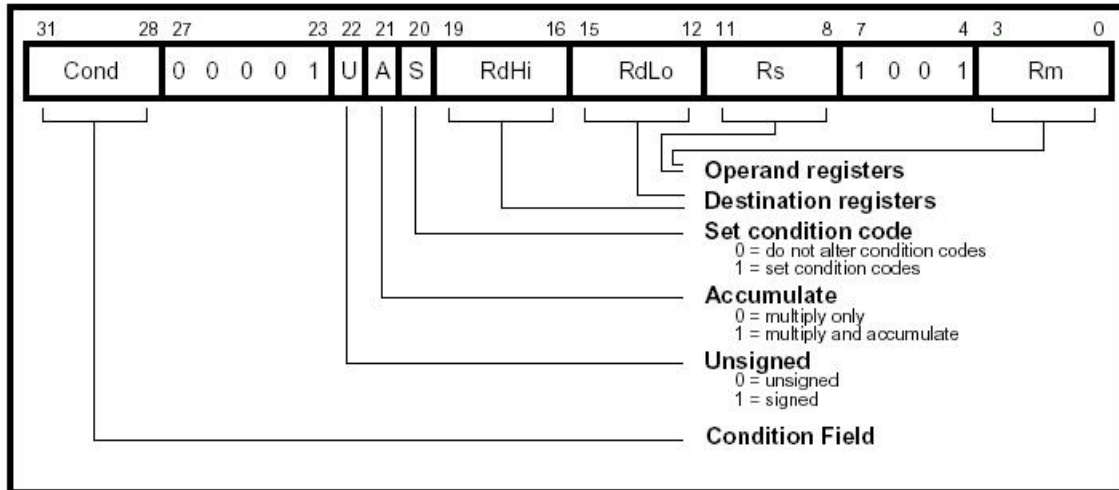


Figura 4-14: instrucciones de multiplicación de 64 bits

La multiplicación de 64 bits (multiply (Accumulate) Long realiza una multiplicación de enteros de 32 bits, dando como resultado un entero de 64 bits. Es posible elegir entre operaciones con y sin signo, y también existe la opción de acumular o no. Esto da lugar a 4 variantes de la instrucción:

- **Multiplicación (UMULL y SMULL):** $RdHi, RdLo := Rm * Rs$ y en cada caso serán interpretados como enteros o números codificados en Ca2. Los 32 bits menos significativos del resultado se escriben en RdLo, y los 32 bits más significativos se almacenan en RdHi.
- **Multiplicación con Acarreo (UMLAL y SMLAL):** igual que el anterior, pero ahora $RdHi, RdLo := Rm * Rs + RdHi, RdLo$

Restricciones en los operandos

- RdHi, RdLo, y Rm deben ser registros distintos
- R15 no puede ser usado como registro operando o destino

Efectos sobre los flags del CPSR

- N: igual al bit 31 del resultado
- Z: se activa si el resultado es todo '0'
- C: se actualiza a un valor carente de significado
- V: se queda inalterado.

4.8 Transferencia simple de datos con memoria (LDR, STR)

Una operación de transferencia simple de memoria sólo es ejecutada si se valida la condición especificada en el campo correspondiente.

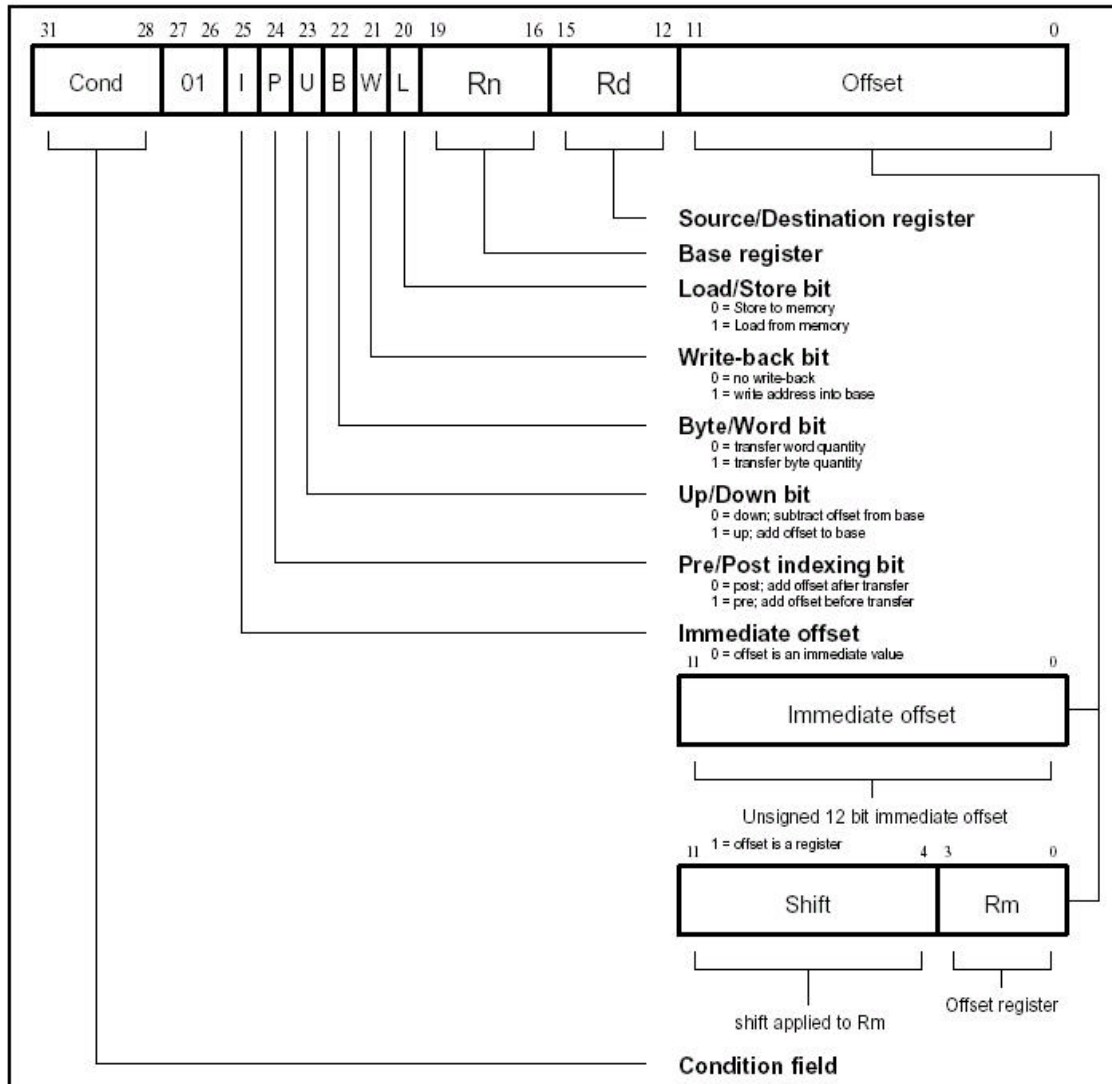


Figura 4-15: Formato de la instrucciones LDR/STR

Esta familia de instrucciones se utiliza para almacenar en memoria o leer de memoria bytes o palabras de datos.

La dirección utilizada para acceder a memoria es calculada sumando o restando un *offset* al valor almacenado en un registro base.

Offset y autoindexado.

El *offset* puede ser un valor inmediato especificado en la propia instrucción, o podría ser un segundo registro, probablemente desplazado de alguna forma.

El *offset* puede ser sumado ($U=1$) o restado ($U=0$) al registro base R_n . La nueva dirección obtenida tras la operación aritmética puede ser utilizada antes (preindexado, $P=1$) o después (postindexado, $P=0$) de la transferencia de memoria.

El bit W sirve para poder utilizar opcionalmente los modos de direccionamiento autoincremento y autodecremento. Estos modos modifican el registro base, escribiendo en él la nueva dirección obtenida tras aplicarle el *offset* ($W=1$), o mantienen su valor ($W=0$).

En el caso del modo de direccionamiento postindexado, el bit W es redundante y **obligatoriamente debe ser 0**, pues no tiene sentido aplicar después de la transferencia la dirección obtenida, si no es para almacenarla de vuelta en el registro base, para el siguiente acceso a memoria.

El caso $P=0$ y $W=1$ es un caso especial y se utiliza cuando el sistema de memoria está protegido por hardware, para poder realizar un acceso a memoria en modo *user* (salida *Privileg* = 0) aunque el programa esté ejecutándose en modo privilegiado (este es el caso por ejemplo, de la emulación software de una instrucción: el programa estará ejecutándose en modo *system*, aunque está emulando una instrucción que realmente debería ejecutarse en modo *user*, y por tanto los accesos a memoria no deberían ser privilegiados).

Registro desplazado como offset

Los 8 bits de control del desplazador están descritos en la sección 4.4. del presente capítulo. La única diferencia es que en este tipo de instrucción no está disponible la opción de especificar la cantidad de bit a desplazar mediante un registro.

Bytes y Palabras (words)

Esta familia de instrucciones puede transferir un byte ($B=1$) o una palabra completa ($B=0$) entre el registro correspondiente del ARM8/9 y la memoria.

Para el CORE, la configuración utilizada a la hora de realizar operaciones con memoria, es Little-Endian.

El CORE tiene una interfaz que permite realizar una conversión big-endian little-endian si el sistema de memoria externo así lo requiere, y está controlada por la señal externa BIGEND (consultar el apartado 3.1.1. La señal BIGEND).

Internamente, el CORE trabaja del siguiente modo:

- Byte Load (LDRB): espera el dato en los bits 7-0 del bus de datos si la dirección suministrada está alineada con la palabra de memoria, en los bits 15-8 si la dirección suministrada es la dirección de la palabra + 1 byte, y así sucesivamente. El dato obtenido se almacena en los 8 bits menos significativos del registro destino, y el resto se rellena con ceros.

- Byte Store (STRB): repite los 8 bits menos significativos del registro fuente cuatro veces a lo largo del bus de salida. El sistema de memoria externo debe encargarse de activar el byte apropiado.
- Word Load (LDR): una dirección de memoria que no esté alineada con una palabra (que no sea múltiplo de 4), provocará que aunque se seleccionará la palabra correcta, se almacenará en el registro interno rotada un número de bytes igual al offset (con offset me refiero al número de bytes que la dirección está desalineada con la palabra).

La siguiente figura ilustra este hecho:

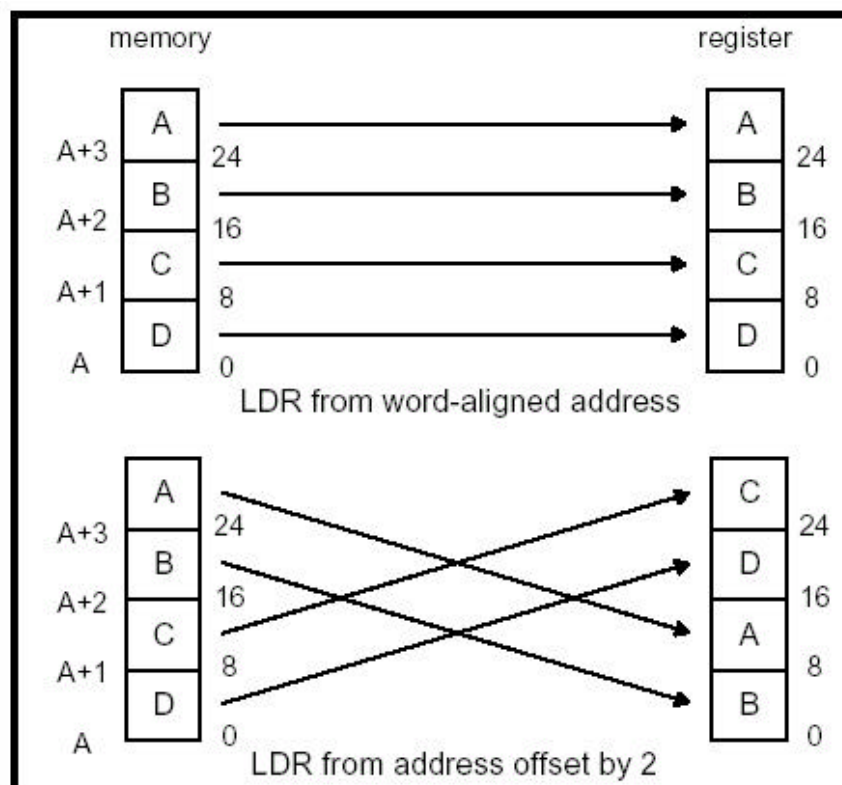


Figura 4-16: rotación al cargar en registro

La dirección A está alineada con la palabra. Ahora bien, si en lugar de aplicar la dirección A, utilizo la dirección A+1, la palabra se almacenará en el registro interno desplazada 1 byte, y si utilizo la dirección A+2, ocurrirá exactamente lo mismo, pero el desplazamiento será de 2 bytes. Esta es, precisamente la situación ilustrada en la figura. El caso análogo sucede si aplico A+3.

- Word Store (STR): normalmente se debe generar una dirección alineada con la palabra. Si no es así, no ocurre nada, y se trata el acceso a memoria como si lo fuese.

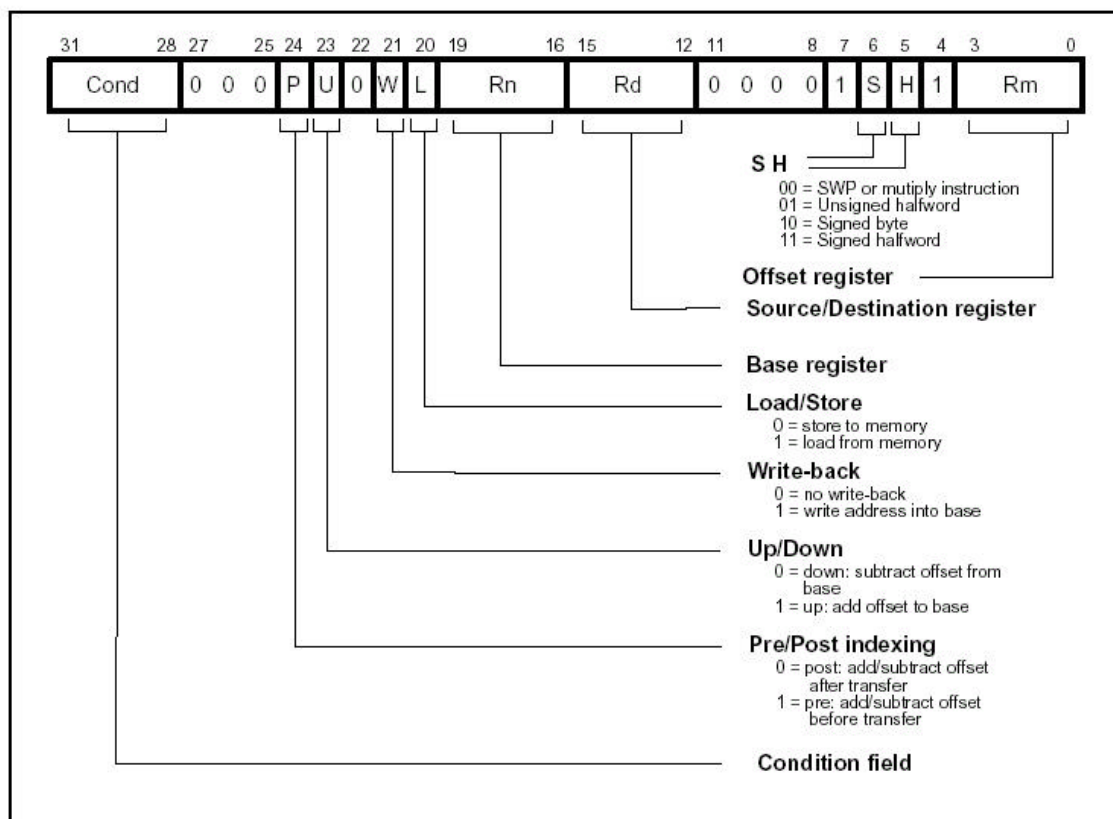
Utilización de R15

- No se debe especificar la opción *write-back* ($W=1$) si el registro base es R15. Además, cuando se use R15 como registro base, se debe tener presente que contiene una dirección 8 bytes por encima de la dirección de la instrucción actual.
- No especificar post-indexado cuando Rn es R15.
- No especificar R15 como registro de offset (R15)
- Si R15 es el registro fuente (Rd) de una instrucción STR, el valor almacenado será la instrucción almacenado más 8, y no más 12 como ocurría en versiones anteriores del ARM.

4.9 Transferencia compleja de datos con memoria (halfwords)

Estas instrucciones se utilizan para leer o almacenar en memoria lo que se denomina *halfwords*, o lo que es lo mismo, la mitad de una palabra completa. También son capaces de leer de memoria *bytes* o *halfwords* y extenderlos en signo al almacenarlos en un registro de propósito general. Las instrucciones sólo son ejecutadas si la condición especificada es cierta.

La dirección utilizada para el acceso a memoria se calcula sumando o restando un *offset* a un registro base (Rn). El resultado de esta operación puede opcionalmente ser escrito de vuelta en el registro base (Rn) si se requiere un modo de direccionamiento autoindexado.

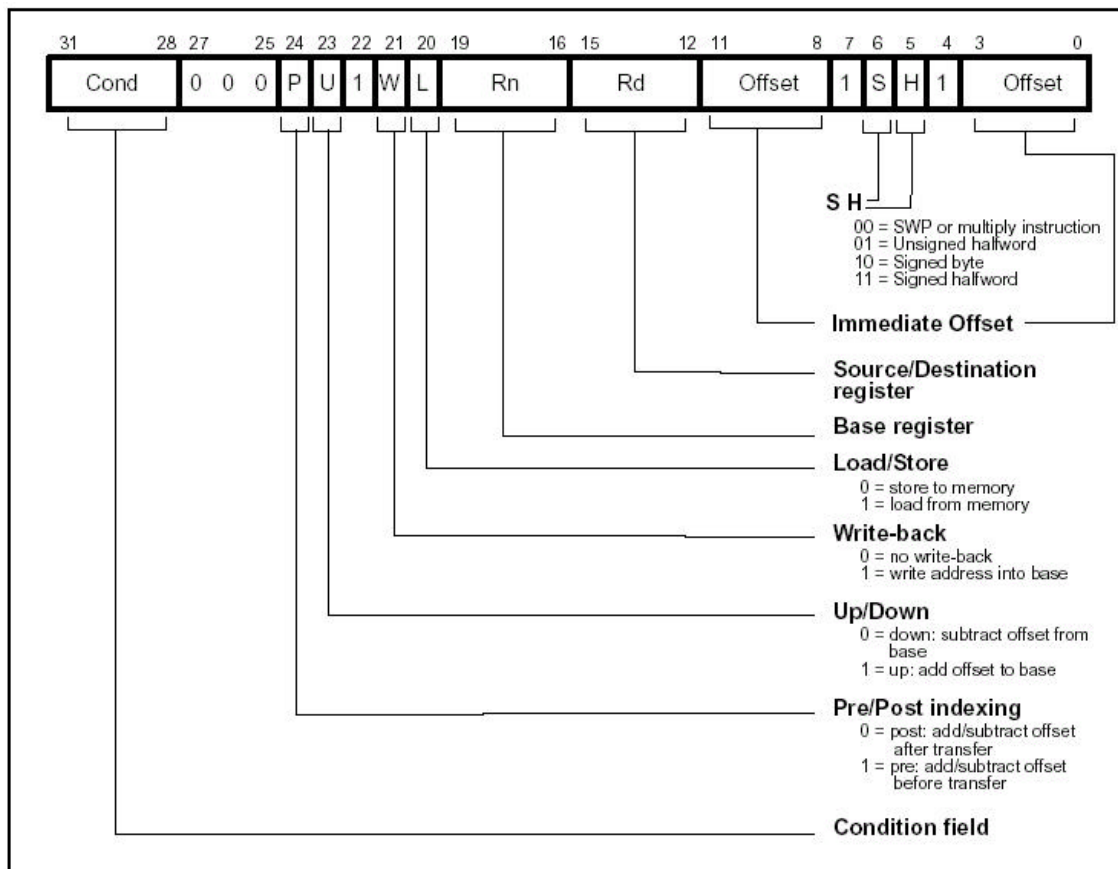


4-17: Transferencia de *halfwords* y datos con signo, usando registro de *offset*

4.9.1 Direccionamiento autoindexado y “offset”

El *offset* puede ser un entero de 8 bits especificado en la propia instrucción, o un segundo registro. El *offset* puede ser sumando (U=1) o restado (U=0) al registro base Rn. El resultado generado en esta operación puede ser aplicado antes (modo preindexado, P=1) o después (postindexado, P=0) de la transferencia a memoria. En éste último caso se utilizaría directamente el registro base para realizar la transferencia.

En el caso de que el *offset* sea un valor inmediato, los bits 11:8 (xxxx) de la instrucción, y los bits 3:0 (yyyy) se combinan para formar el *offset* (xxxxyyyy).



4-18: Transferencia de *halfwords* y datos con signo, usando *offset* inmediato

El bit W ofrece la posibilidad opcional de utilizar modos de direccionamiento con autoincremento y autodecremento. El valor modificado del registro base puede ser escrito de vuelta en el mismo tras el procesamiento (W=1), o también se puede conservar su valor original (W=0).

En el caso de modo de direccionamiento postindexado, el bit **W es redundante, y debe valer siempre cero**. Si se requiere que el registro base sea conservado, basta con establecer como *offset* el valor cero.

El caso P=0 y W=1 ya fue explicado en el apartado anterior, y se utiliza cuando el sistema de memoria está protegido por hardware, para poder realizar un acceso a memoria en modo *user* (salida *Privileg* = 0), aunque el programa esté ejecutándose en modo privilegiado.

El bit W no debe nunca ponerse a valor alto (W=1) cuando se esté utilizando el modo de direccionamiento post-indexado.

4.9.2 Modos de Funcionamiento

De los distintas combinaciones de los bits S, H, y L, se pueden extraer los siguientes modos de operación.

S	H	L	Nemónico	Descripción
0	0	--	-----	Corresponde a SWP ó multiplicación
0	1	0	STRH	Halfword store
0	1	1	LDRH	Halfword load
1	0	1	LDRSB	Signed byte load
1	1	1	LDRSH	Signed halfword load

- STRH: almacena en memoria los 16 bits meons significativos del registro fuente, los cuales repite dos veces a lo largo del bus de datos. El sistema de memoria externo debe activar los bytes apropiados para que la operación se lleve a cabo con éxito.
- LDRH: esta instrucción espera un dato en las líneas 15-0 del bus de datos si la dirección suministrada está alineada con la palabra de memoria (A[1]=0), o espera un dato en las líneas 31- 16 si A[1]=1). Siempre A[0]=0, de lo contrario, el comportamiento del sistema será impredecible. Una vez esté el dato correcto en el bus, es almacenado en los 16 bits menos significativos del registro destino, y el resto es rellenado con ceros.
- LDRSB: espera un dato en el byte menos significativo del bus de datos si la dirección está alineada, en el segundo byte menos significativo si la dirección es la de la palabra más un byte, y así sucesivamente. El byte seleccionado se almacena en los 8 bits menos significativos del registro destino, y el resto de bits son rellenados con el valor del bit más significativo del byte almacenado.
- LDRSH: el funcionamiento es idéntico al de LDRH, pero en lugar de rellenar con ceros el registro destino, se rellena con el valor del bit más significativo del *halfword* almacenado.

Uso de R15

No especificar R15 como:

- Registro de offset (Rm)
- Registro destino (Rd) de una instrucción tipo *load*
- Registro fuente (Rd) de una instrucción tipo *store*

Si se especifica como registro base, hay que tener en cuenta que el valor que se utilizará en el acceso a memoria será la dirección de la instrucción actual más 8 bytes.

Restricciones de uso del registro base

No especificar modo post-indexado cuando Rm y Rn son el mismo registro, pues sería imposible recuperar el estado después de un abort.

Rd y Rn deben ser distintos registros cuando una instrucción tipo *load* implique la escritura de vuelta del registro base, es decir, si L=1 y W=1, Rd debe ser distinto de Rn.

4.10 SWP (Single Data Swap)

Una instrucción SWP sólo es ejecutada si la condición especificada es cierta.

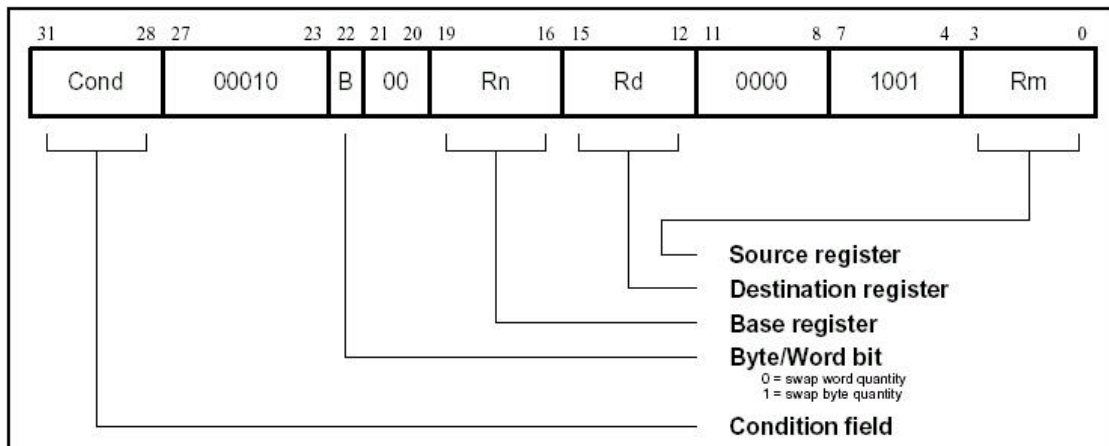


Figura 4-19: Formato de la instrucción SWP

La instrucción SWP se utiliza para intercambiar un byte o un word entre un registro y la memoria externa. Simplemente se trata de una lectura de memoria, seguida de una escritura de memoria, pero que se realizan de forma atómica. El procesador no puede ser interrumpido hasta que ambas operaciones se completen, y el gestor de memoria se encarga de que sean tratadas como “inseparables” (atómicas).

Este particular tipo de instrucción es bastante utilizado para la implementación software de semáforos.

La dirección del swap se determina a través del contenido del registro base Rn. El procesador primero lee el contenido de la dirección de acceso a memoria; entonces describe el contenido del registro fuente (Rm) en memoria utilizando la dirección anterior, y almacena el valor que existía en esa posición de memoria anteriormente en el registro destino (Rd).

Bytes y Palabras (words)

Mediante el bit B, se puede seleccionar la unidad de memoria a utilizar para el intercambio (*swap*). Si el bit B=1, entonces se trata de un byte; en caso contrario, será una palabra completa lo que se intercambiará con la memoria.

La instrucción SWP es implementada como un LDR seguido de un STR, y su acción ya ha sido descrita en apartados anteriores.

Uso de R15

R15 no debe ser utilizado como operando (Rd, Rn, ó Rm) en una instrucción SWP.

4.11 Fallos de memoria (Data Aborts)

En algunas situaciones, una transferencia desde o hacia memoria pueden provocar un error del sistema de memoria, lo que hará que éste genere una excepción *ABORT*. Ya en el capítulo 3 se analizó con detalle el comportamiento del CORE ante cualquier excepción.

En este apartado nos vamos a ocupar de las garantías que debe ofrecer y de hecho ofrece el hardware tras producirse un *Abort*, si no se puede completar la transferencia deseada.

En ese caso, ni el contenido de la dirección de memoria del acceso causante del *abort*, ni el registro destino de la instrucción correspondiente deben ser alterados bajo ningún concepto.

El ARM9 garantiza además que el contenido del registro base de la instrucción de memoria que generó el ABORT permanecerá intacto si el acceso a memoria no ha tenido éxito.

El software deberá contar con una rutina de atención de la excepción generada capaz de encontrar y reparar en la medida de lo posible el problema surgido.

Al finalizar su labor, devolverá el control al programa original, y la instrucción que causó el *fallo* será ejecutada de nuevo.

4.12 Interrupción Software (SWI)

Una instrucción SWI sólo es ejecutada si la condición especificada se cumple.

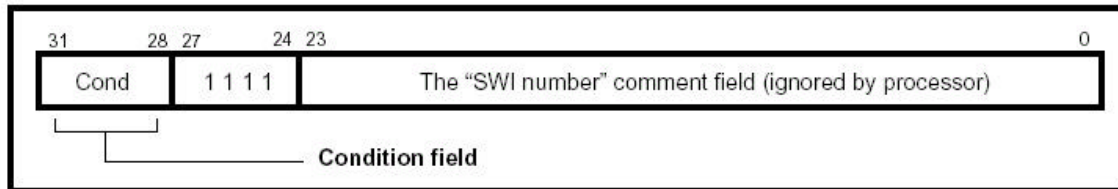


Figura 4-20: Formato de la instrucción SWI

La interrupción software se utiliza para entrar en modo supervisor, de manera que cuando se ejecuta una SWI, los bits de modo del CPSR son alterados, y el PC es forzado a actualizarse al valor del vector SWI, y el CPSR es salvado en el SPSR_svc.

Volver desde modo Supervisor

El PC fue salvado en R14_svc, y el CPSR en el SPSR_svc, al entrar en la interrupción SWI, de modo que basta con utilizar *MOVS PC, R14_svc* para devolver el control al programa original, y restaurar el valor anterior del CPSR.

El mecanismo de retorno es **no reentrante**, de modo que si se desea que sea posible utilizar SWI dentro de la rutina de atención, debe salvarse primero una copia de la dirección de retorno y del SPSR en una pila implementada en memoria.

El campo "comentario"

Este campo es ignorado por el microprocesador, y se utiliza para comunicar información al código supervisor. Por ejemplo, el supervisor puede mirar este campo y utilizarlo para indexar una matriz de punteros para rutinas que realizan distintas funciones de supervisión.

Este campo se conoce comúnmente con el nombre de " número SWI " (*SWI number*).

4.13 Instrucciones no reconocidas

Existen ciertos patrones de bits, que no se corresponde con ninguna de las instrucciones implementadas en el CORE. Estos patrones, al intentar ser ejecutados, causan lo que se denomina excepción por instrucción no reconocida.

Algunas de estas secuencias de bits pueden ser utilizadas para emular instrucciones no implementadas, a través de la rutina de servicio de la excepción provocada.

Las instrucciones no reconocidas se clasifican por clases de la siguiente manera:

- Clase A: instrucciones no definidas en implementaciones previas del ARM
- Clase B: MSR/MRS no asignadas como instrucciones
- Clase C: multiplicaciones no asignadas como instrucciones
- Clase D: SWP no asignada como instrucción
- Clase E: STRH/LDRH/LDRSH/LDRSB no asignadas como instrucción

Class	Instruction Bit Pattern								Notes
A	Cond	011x	xxxx	xxxx	xxxx	xxxx	xxx1	xxxx	
B	Cond	0001	0xx0	xxxx	xxxx	xxxx	yyy0	xxxx	yyy != 000
	Cond	0001	0xx0	xxxx	xxxx	xxxx	0xx1	xxxx	
	Cond	0011	0x00	xxxx	xxxx	xxxx	xxxx	xxxx	
C	Cond	0000	01xx	xxxx	xxxx	xxxx	1001	xxxx	
D	Cond	0001	yyyy	xxxx	xxxx	xxxx	1001	xxxx	yyyy != 0000 or 0100
E	Cond	0000	xx1x	xxxx	xxxx	xxxx	1yy1	xxxx	yy != 00
	Cond	000x	xxx0	xxxx	xxxx	xxxx	11x1	xxxx	

Estos patrones generan *espacios de instrucciones no definidas*, y se pueden utilizar para emular instrucciones que se implementen en futuras versiones de la arquitectura ARM, utilizando para ello la rutina de atención a la excepción.

Todos los bits que quedan en estos *espacios no definidos* (es decir, los bits que aparecen con x en la tabla anterior), pueden utilizarse como campos de esas instrucciones. En el caso de la instrucción *undefined*, su patrón de bits no ha sido utilizado por ninguna versión de la arquitectura del ARM.

El CORE generado en este proyecto implementa todos los *espacios no definidos* anteriores, para ser compatible con futuras versiones de arquitectura ARM.

5

Diseñando el CORE

En este apartado se detalla el alcance del proyecto, así como las decisiones adoptadas y la estrategia de diseño utilizada en el desarrollo de este proyecto. También se realiza una descripción por bloques de todos los componentes del CORE.

- 5.1 Consideraciones Iniciales
 - 5.1.1 Alcance del proyecto
 - 5.1.2 Decisiones sobre la arquitectura
 - 5.1.3 Algoritmo de diseño
- 5.2 Descomposición previa del diseño
- 5.3 Arquitectura del CORE
 - 5.3.1 Reducción de buses en la ruta de datos
- 5.4 Generación de “*bubbles*” y vaciado de la *pipeline*
- 5.5 ALU
 - 5.5.1 Códigos de Operación
- 5.6 Diseñando la “*pipeline*”
 - 5.6.1 Fase I
 - 5.6.2 Fase II
 - 5.6.3 Generación de estados de espera
- 5.7 Interfaz de memoria
 - 5.7.1 Creando una interfaz personalizada
 - 5.7.2 Accediendo a la memoria de programa
- 5.8 Introducción del control de excepciones
- 5.9 Vista del *pin-out* final del CORE
- 5.10 Layout de una mega-celda del microprocesador
 - 5.10.1 Optimización del diseño
- 5.11 Observaciones
 - 5.11.1 Unidad de predicción de saltos

5.1 Consideraciones Iniciales

En esta sección se pretende especificar de forma clara las especificaciones de este proyecto en lo que se refiere a

- alcance
- decisiones sobre la arquitectura
- algoritmo de diseño

La motivación de este trabajo está relacionada de forma clara con el capítulo segundo, en el que se muestra el tremendo desarrollo que ha experimentado la familia de microprocesadores ARM, desde que a mediados de los ochenta vio la luz el ARM1.

En el capítulo tercero se presentó el modelo del programador del ARM9, que sirve de base para que un futuro usuario pueda incluir nuestro CORE como parte integrante de un sistema de nivel jerárquico superior.

El juego de instrucciones se muestra en el capítulo cuarto, y es el único dato del que disponemos, junto con las características generales expuestas en el capítulo segundo, para iniciar el diseño de nuestro CORE.

Por tanto, las decisiones que se tomen a priori son de vital importancia, pues del buen o mal acierto que se tenga al principio dependerá el que el proyecto llegue o no a buen fin.

5.1.1 Alcance del proyecto

El objetivo de este proyecto es diseñar el CORE (núcleo) del microprocesador ARM9 **con arquitectura ARMv4**, de modo que pueda usarse como celda básica en una librería destinada al diseño de ASICs.

El CORE generado deberá ser capaz de decodificar y ejecutar todo el juego de instrucciones del ARM definidas para la arquitectura ARMv4 (excepto las que han sido descartadas por algún motivo), y también implementará todo el comportamiento hardware descrito en dicha arquitectura. Ésto incluye a:

- las interrupciones
- el acceso a memoria
- la configuración del banco de registros interno
- los modos de funcionamiento
- el formato de los registros de control de estado del programa PSR's
- el valor que debe utilizarse cuando el PC es operando de una operación

En pocas palabras, se requiere de un cierto trabajo de ingeniería inversa, para crear un CORE cuyo comportamiento sea una réplica exacta del auténtico ARM8/9.

El alcance del proyecto incluye:

- CORE del microprocesador ARM9 **con arquitectura ARMv4**
- interfaz de memoria del CORE, con formato propio, y posibilidad de configurar la numeración de memoria como BIG-ENDIAN o LITTLE-ENDIAN mediante hardware
- generación de estados de espera para acceso a memoria externa y periféricos mapeados en memoria, mediante configuración hardware
- gestión de interrupciones: se deben incluir los pines externos nFIQ, nIRQ, nRESET y nABORT
- realización de todas las pruebas y simulaciones pertinentes para verificar el correcto comportamiento del diseño, y reducción del espacio de errores a su mínima expresión
- realización de un prototipo de pruebas, que será implementado en la VIRTEX 800, para demostrar que verdaderamente el diseño es capaz de funcionar en un entorno real.

Queda fuera del alcance de este proyecto:

- todo lo referente a la arquitectura ARMv4T que no aparezca en la ARMv4
- la implementación de la interfaz con el coprocesador, pues no se disponen de datos del funcionamiento del mismo, y además, como ya se explicó en el capítulo 4, carece de sentido utilizar un coprocesador en un microprocesador que ya contiene multiplicador (la interfaz es más bien un vestigio de versiones anteriores)
- la implementación física de la instrucción “Block Data Transfer” por:
 - suponer una grave violación de los principios básicos de una estructura *pipeline*, al requerir un número de ciclos de reloj a priori incierto y variable para su ejecución
 - porque se puede implementar mediante software, utilizando simplemente la instrucción *LDR*, y además de forma mucho más eficiente.
- La realización de una unidad de predicción de saltos, aunque como consideramos que es algo de vital importancia en una máquina RISC con estructura *pipeline* como ésta (del correcto funcionamiento de la unidad de predicción de saltos se desprendería un incremento notable del parámetro *speedup*), dejamos sentadas las bases para realizarla en el último apartado.

5.1.2 Decisiones sobre la arquitectura

Cualquier labor de diseño microelectrónico, requiere asumir una serie de compromisos entre distintos parámetros que afectan al resultado final de forma contrapuesta.

En nuestro caso, y tras un detenido estudio de todos los factores en juego, podemos afirmar que existe un compromiso *velocidad-área, de modo que un incremento de velocidad solo puede lograrse a costa de aumentar el área del diseño.*

Tras considerar detenidamente esta cuestión, se tomó la decisión de que el parámetro que queremos optimizar es el *speedup*, que estudiamos en apartados anteriores, y que para nuestro diseño, se puede definir mediante la siguiente ecuación:

$$speedup = \frac{d}{1 + N_{ws}}$$

Dado que *d* (profundidad de la *pipeline*) es un valor constante, la única forma de mejorar el *speedup* será reducir el número de ciclos de espera adicionales (N_{ws}) al ciclo por instrucción deseado.

Una reducción de área, por ejemplo en el multiplicador, implicaría un aumento del número de ciclos adicionales de espera, con lo que el *speedup* empeoraría.

Después de considerar todos los factores expuestos, se adoptaron las siguientes decisiones de diseño, que se utilizarán como premisas fundamentales a la hora de tomar cualquier decisión:

- Se pretende diseñar un microprocesador rápido y eficiente, así que nuestra meta será una arquitectura capaz de ejecutar una instrucción por cada ciclo de reloj, a excepción de aquellas que requieren doble ancho de bus, que necesariamente necesitan de dos ciclos reloj.
- De este modo se sacrificará el área que sea necesaria para poder cumplir con lo anterior.

Lo anterior no supone un gran sacrificio, pues optimizar área en un diseño orientado a FPGAs de este tipo, no es una labor fácil (y no está claro si es posible).

Además hay que tener en cuenta que estamos utilizando una arquitectura que ocupa bastante más área que una arquitectura simple como la del PIC17, debido a su alta eficiencia y su velocidad. No tiene por tanto sentido ahorrar “un poco de área” (despreciable frente al área total del diseño) a costa de empeorar la velocidad que se había logrado al introducir dicha arquitectura.

¿Pipelined o no-pipelined?

La respuesta a esta pregunta está más que clara, teniendo en cuenta todo lo que se ha expuesto hasta este punto. Será **pipelined**, debido a que:

- es una arquitectura más actual
- es la arquitectura que utiliza el verdadero CORE del ARM9
- mejora el *speedup* en un factor aproximadamente igual al número de etapas de la estructura *pipeline*
- mejora por tanto el rendimiento global del sistema
- es una arquitectura más adaptada a la filosofía RISC, como ya hemos expuesto anteriormente, pues prácticamente la estructura *pipelined* surge de forma natural como la más adecuada para un procesador Advanced RISC

¿De cuántas etapas será la *pipeline*?

El número de etapas de la *pipeline* estuvo claro desde el primer momento por dos factores fundamentales:

- El primero, aunque menos relevante, fue el hecho de que el ARM8 fue el primer procesador de la familia ARM que utilizó una arquitectura *pipeline* de 5 etapas, en lugar de las tradicionales de 3 y 4 etapas que venía usando anteriormente.
- El segundo y el más importante, fue el cálculo de un valor óptimo para la implementación de una estructura *pipeline* en una FPGA, de modo que ésta sea la que consiga funcionar a mayor frecuencia de reloj

Cálculo del óptimo número de etapas que maximiza la frecuencia de reloj en la FPGA

Según un artículo de la revista X-FILES, "News From the leading provider of FPGA y desktop ASIC", el número óptimo de etapas que consigue la mayor frecuencia de reloj posible al implementar la estructura en una FPGA, se calcula como:

$$N_{opt} = \log_2 (\text{Ancho del Bus de Datos})$$

En nuestro caso, tenemos un bus de datos de 32 bits, luego el número de etapas óptimo para maximizar la frecuencia de operación del reloj, será de 5.

A continuación se adjunta una tabla que compara la implementación de una pequeña estructura pipeline con el número óptimo de etapas según el criterio anterior, y la de la misma estructura pero no-pipeline. Además se muestra el resultado de la implementación en dos FPGAs de diferentes fabricantes.

Tecnología	Tipo	Area	Frecuencia de reloj
Altera APEX	non pipelined	541 LCS	36.49 MHz
	Pipelined	587 LCS	78.12 MHz
Xilinx VIRTEX	non pipelined	156 Slices	42 MHz
	Pipelined	192 Slices	88 MHz

¿Von Neumann o Harvard?

La solución a esta cuestión también estuvo muy definida desde el primer momento, aunque es más difícil de justificar

Hasta ahora, los procesadores ARM, venían utilizando la arquitectura Von Neumann, aunque a partir del ARM7 se comenzó a ensayar con la arquitectura Harvard.

Nosotros buscábamos una arquitectura moderna y eficiente para la realización de nuestro CORE, así que preferimos optar por una **arquitectura interna** totalmente Harvard, ya que es la estructura que proporciona mayor eficiencia al ser usada conjuntamente con una estructura paralelo, como es la *pipelined*.

* * * * *

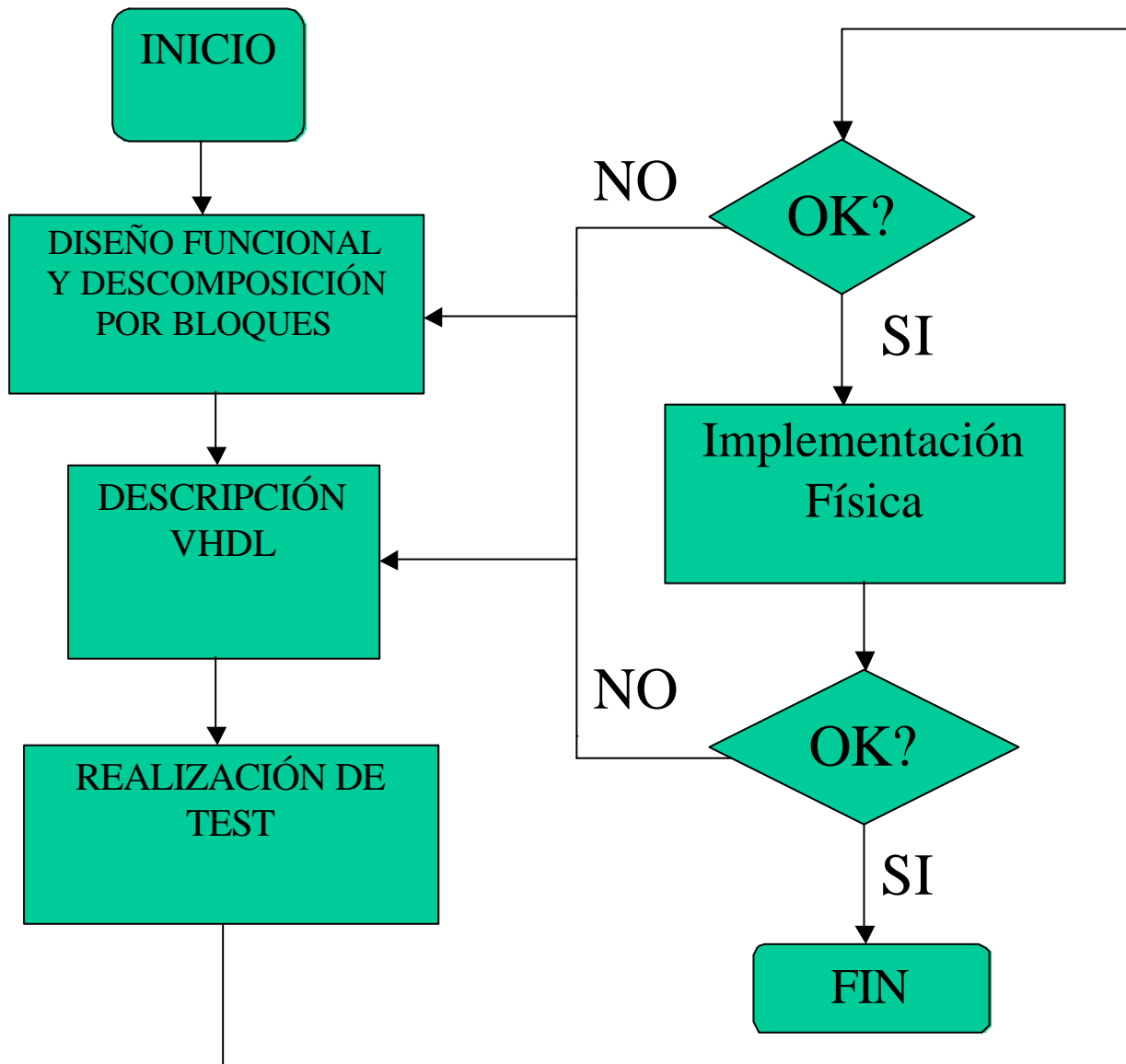
Todas las decisiones anteriores, pueden resumirse en la siguiente tabla:

<i>Todas las instrucciones deben ejecutarse en un ciclo de reloj, excepto la multiplicación de 64 bits (que requiere de dos ciclos para poder sacar el resultado, pues tanto el bus como los registros son de 32 bits), y la instrucción SWAP (porque no se puede leer y escribir una memoria en el mismo ciclo de reloj).</i>
--

Decisiones iniciales respecto a la arquitectura del CORE del ARM9		
Arquitectura	Estructura	Nº de etapas de la <i>pipeline</i>
Hardvard	<i>Pipelined</i>	5

5.1.3 Algoritmo de diseño

A continuación se incluye una especificación formal de lo que sería el algoritmo a seguir a la hora de abordar el diseño de cualquier circuito digital integrado implementado en una FPGA.



En el caso particular de nuestro diseño, el mayor coste en tiempo se distribuye en el bucle:

- realización de test
- ¿son los resultados satisfactorios?
- NO: modificar bien la descripción VHDL, o bien si el error se debe a un fallo de apreciación o a un error de tipo conceptual, volver a realizar la descripción funcional y la descomposición por bloques.

5.2 Descomposición previa del diseño

Antes de resolver otras cuestiones, es importante sentarse cierto tiempo a meditar qué requisitos impone el juego de instrucciones a implementar y, dentro de las diferentes opciones, que arquitectura permite introducir de la forma más sencilla posible todas las entidades funcionales para suplir las especificaciones anteriormente determinadas.

Un ejemplo ilustrativo de este hecho sería, por ejemplo, determinar de todo el conjunto de instrucciones, cuál sería la instrucción que más operandos requeriría en un momento determinado, y por tanto cuántos buses serían necesarios para poder cursarlos.

De este modo, no nos encontraríamos con la sorpresa de, estando bastante avanzado el diseño de la arquitectura, tener que modificar completamente toda la ruta de datos, porque resulta insuficiente para cierto tipo de instrucción.

Otro proceder interesante sería inspeccionar detenidamente todas las instrucciones, y determinar así en qué unidades básicas se pueden descomponer, de modo que una vez implementadas estas “funcionalidades”, introducir una instrucción sería tan sencillo como configurar los registros de control de las unidades funcionales de una forma determinada.

Nótese que la estructura *pipeline* lleva asociada de forma intrínseca una característica que simplifica bastante el diseño: *una vez construida la ruta de datos y los registros de configuración necesarios, añadir cualquier instrucción en particular, o cualquier operación en general, es tan sencilla como introducir una determinada configuración en función de la instrucción decodificada, o de las señales de control aplicadas.*

Con esto se quiere significar que la estructura *pipeline* puede ir creciendo conforme se van añadiendo “comportamientos”, y cada vez que una instrucción nueva se añade al diseño, se puede simular completamente el nuevo CORE. Esta característica simplifica enormemente la depuración de la arquitectura, en primer lugar, porque se puede simular completamente el diseño cada vez que se añade una nueva instrucción, y en segundo lugar, porque la inclusión de un nuevo “comportamiento” no tiene por qué afectar al resto del diseño; así, la depuración se centrará en el código añadido, pudiendo partir de la base de que el resto del diseño (previamente depurado) funciona correctamente, y el error se tiene que encontrar forzosamente en los últimos cambios realizados.

Una vez que todas estas indicaciones son consideradas, y tras un minucioso análisis del juego de instrucciones, se llega a la conclusión de que la descomposición funcional del diseño es la que sigue a continuación:

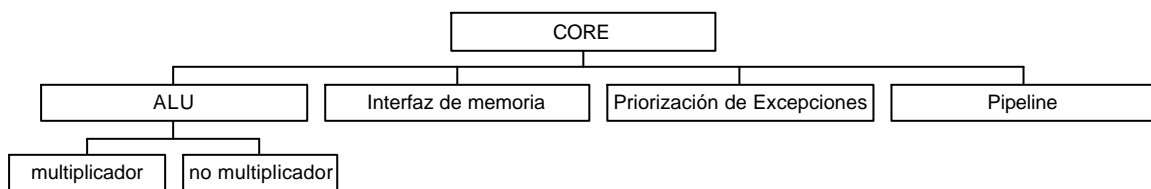
- En primer lugar, el CORE necesita de una ALU que sea capaz de realizar todas las operaciones aritméticas, lógicas y desplazamientos que son requeridas por todas las instrucciones del ARM8/9.
- Hay que prestar especial atención al multiplicador, pues todas las simplificaciones del mismo que pudieran realizarse repercutirán directamente en una reducción de área importante.
- Para implementar la ruta de datos, bastará solamente con 3 buses de 32 bits, denominados A, B y C. A esta conclusión se llegó tras realizar las siguientes consideraciones:
 - en principio existen instrucciones que requieren de 4 buses, como es el caso de la multiplicación de 64 bits, pero si se tiene en cuenta que el resultado, de 64 bits, requiere de dos ciclos de reloj para poder ser cursado por el bus de resultado (Result), que tiene un ancho de 32 bits, y si además consideramos las descomposiciones y simplificaciones que se detallan en el apartado siguiente, el único operando de 64 bits podría descomponerse en dos subconjuntos de 32 bits. Cada uno de ellos se puede aplicar por separado en cada uno de los 2 ciclos de reloj que requiere esta instrucción, aprovechándonos de la propiedad asociativa de la multiplicación y la suma.
 - el bus C, en algunas ocasiones, se utiliza para informar a la ALU de la cantidad de bit a desplazar el operando contenido en el bus B, por lo que existe una instrucción que requiere el bus A para el primer operando, el bus B para el segundo, el bus C para el desplazamiento aplicado al segundo operando, y además requiere un cuarto bus para cursar un operando que será almacenado en etapas posteriores en memoria. Esta situación podría resolverse utilizando un registro auxiliar para almacenar el número de bits a desplazar, y que sería conectado a la entrada C de la ALU al ejecutar esta instrucción, y el bus C quedaría libre para cursar el cuarto operando. Nótese que el nuevo registro requiere sólo de 5 bits, lo cual es un pequeño sacrificio para el gran ahorro logrado (el registro auxiliar permite no tener que utilizar un nuevo bus de 32 bits más su correspondiente decodificador para decidir qué registro de propósito general escribir en el bus)
- El siguiente paso sería la implementación de una máquina de estados que controlase la *pipeline*, y un conjunto de señales que fuesen capaces de “congelar” y/o de “vaciar” ciertas etapas de la *pipeline*, si el control así lo requiere.
- Ahora estaríamos en condiciones para implementar lo que hemos denominado FASE I, es decir, las instrucciones de tratamiento de datos, los saltos, las multiplicaciones, y las operaciones con los PSRs.

- Una vez depurada la fase anterior, podríamos emprender la FASE II: la implementación de todas las instrucciones relacionadas con el acceso a memoria, que tratarían todos los datos con formato little-endian.
- Lo siguiente sería realizar la interfaz de memoria del CORE, que se adaptaría a un formato propio, que posteriormente será detallado. Así el usuario del CORE podría de forma sencilla realizar una interfaz de conversión entre nuestra interfaz, y el sistema de memoria que desee emplear.
- El siguiente paso sería incluir una circuitería que permitiese acceder a memoria con más de un ciclo de reloj, es decir, que el CORE generase estados de espera. Esto no debería ser muy complicado, pues nos aprovecharíamos del control que “congela” y/o “vacía” ciertas etapas de la *pipeline*.
- Por último, bastaría con añadir el control de interrupciones para que el CORE implementado tuviera toda la funcionalidad del ARM8/9.

Nótese que si en la realización de la ruta de datos y del control global del diseño, se han tenido en cuenta todos los requisitos exigidos por todas las instrucciones, la inclusión de cada fase debería poder llevarse a cabo de forma simple (bastaría con multiplexar ciertas configuraciones de registros, en función de la instrucción decodificada, y de los “comportamientos” deseados por parte del CORE).

A posteriori se puede afirmar que la descomposición del diseño fue bastante acertada, y que la inclusión de una nueva fase pudo completarse sin ningún problema añadido a los que a priori fueron previstos.

Desde el punto de vista de la implementación en VHDL, la jerarquía deseada sería algo como lo que sigue a continuación:



5.3 Arquitectura del CORE

En el siguiente diagrama se muestra la estructura simplificada de la arquitectura *pipeline*.

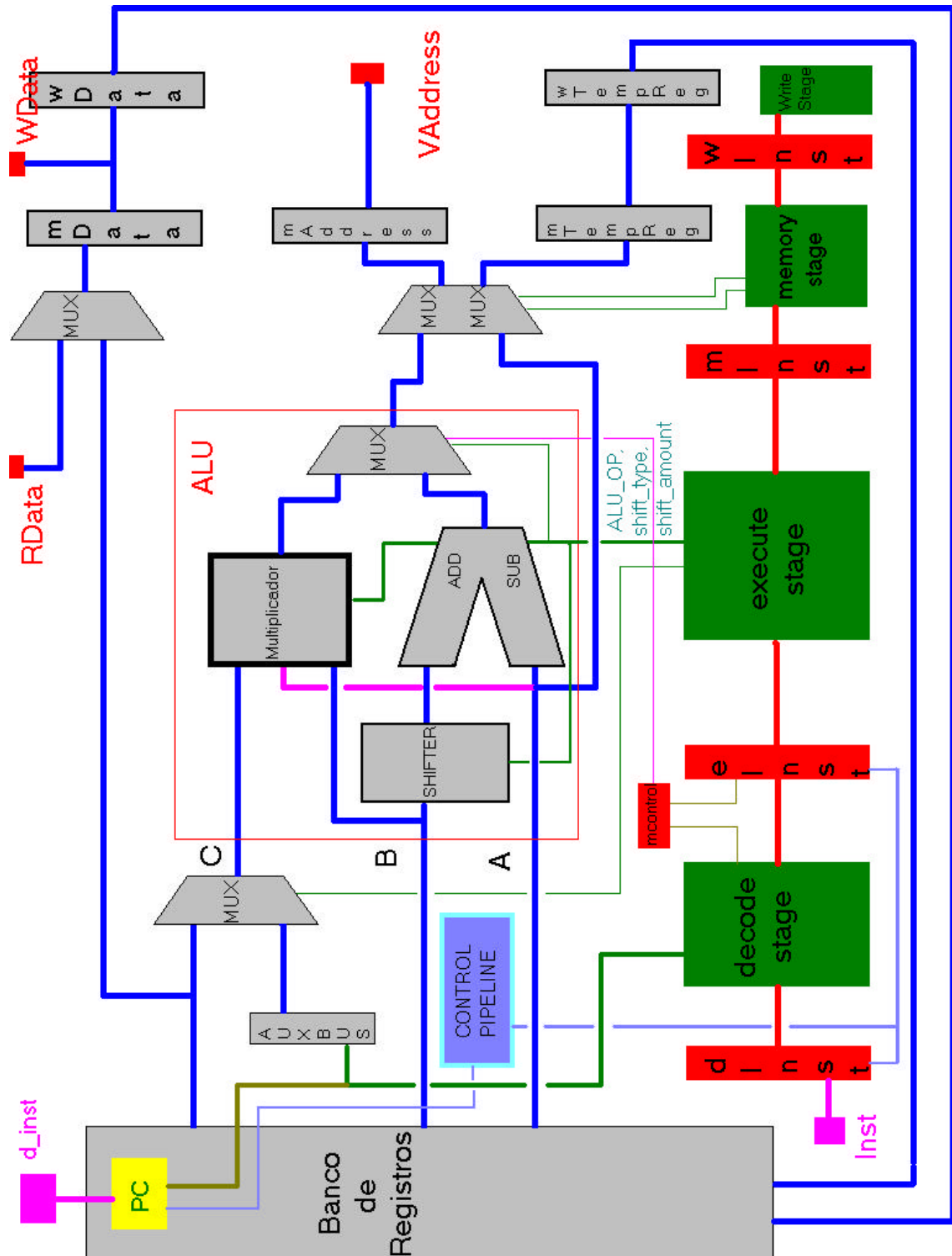


Figura 5-1: diagrama de bloques de la estructura *pipeline*

En el diagrama anterior se ha intentado simplemente reflejar las relaciones entre las distintas entidades funcionales que componen la arquitectura *pipeline*. Por tanto, se han omitido señales de control, reloj, reset asíncrono global, etc, así como el control de excepciones.

Lo único que se pretende con el diagrama anterior es mostrar la arquitectura de partida, con la que se empezó a codificar en VHDL la entidad *pipeline*.

Cada una de ellas será explicada detalladamente en las siguientes secciones de este capítulo, aunque vamos a comentarlas un poco por encima:

Banco de Registros

Está compuesto por 37 registros de 32 bits, entre los que se incluye el contador de programa (R15), el CPSR y los registros de propósito general.

Además, el bloque también incluye a los decodificadores de los tres buses de salida A, B y C, y de los dos buses de entrada (escritura del resultado, y escritura del nuevo registro base cuando en una instrucción de acceso a memoria, el bit W=1).

Control de la *pipeline*

Este bloque es el encargado de vaciar la pipeline y/o introducir bubbles (ciclos de espera) en aquellas etapas que lo necesiten en un momento dado, cuando por ejemplo se produzca “interlocking” de registros, o saltos, o alguna situación anómala.

Registros de instrucción

Contienen la configuración para el ciclo de reloj actual de la etapa correspondiente (por ejemplo, m_inst contiene la configuración actual de la etapa de memoria, ó en nomenclatura anglosajona, *memory-stage*).

La única excepción a esta regla es el registro d_inst, que contiene la instrucción que se acaba de “coger” (fetch) de memoria, está siendo decodificada. Una vez determinada la configuración de las etapas posteriores, ésta es propagada de un registro a otro, sin ser alterada (lo cual simplifica la lógica combinacional de las siguientes etapas, pues todo el peso recae sobre la etapa *decode*, que de todos modos tenía que decodificar la instrucción: una vez realizada esta operación, supone el mismo coste configurar diez bits de un registro, que veinte, pues el mayor consumo de área recaerá sobre el multiplexor que realiza la decodificación).

Registros de datos (m-data , w-data)

Almacenan temporalmente los operandos o resultados correspondientes a la etapa en concreto (memory, write) en el ciclo de reloj actual.

En el caso de las operaciones de proceso de datos, es el resultado de la operación lo que se propaga por estos registros hasta que se escribe de vuelta en el banco de registros. En cambio, en el caso de las operaciones de memoria, es el dato a escribir en memoria, o el dato leído de la misma lo que se almacena.

Registro de dirección de acceso a memoria (m-address)

Contiene la dirección con la que se va a acceder a memoria (si se trata de una operación de memoria), o no contiene nada.

Registros temporales de almacenamiento del registro base modificado (m/w-temp-reg)

Almacenan temporalmente el nuevo valor del registro base ($W=1$), para poder propagarlo a través de toda la ruta de datos, hasta que en la etapa quinta sea escrita de vuelta en el registro base.

En el caso de que la instrucción que procese la etapa de memoria y la etapa de escritura no sea una instrucción de memoria con el bit $W=1$, estos registros podrán ser utilizados como registros temporales de propósito general.

AUXBUS

El bus C se utiliza en todas las instrucciones que no son de acceso a memoria, para introducir a la ALU la cantidad de bit a desplazar, es decir, en la mayor parte de los casos el bus C se corresponde con la entrada C de la ALU.

Pero en las instrucciones de acceso a memoria, que prácticamente en la totalidad de los casos (excepto uno) no utilizan el desplazador, el bus C se ha aprovechado para propagar hasta la etapa de memoria, el contenido del registro a escribir en memoria, ahorrando así un bus con toda su lógica de decodificación.

No obstante, existe una instrucción de memoria, variante de *STR* que utiliza un registro rotado como offset para calcular la dirección a utilizar en el acceso a memoria. En ese caso, se requiere introducir un valor en la entrada C de la ALU, pero el bus C estará ocupado con el valor que deseamos escribir en memoria.

Como esta instrucción contiene un valor de tan sólo 5 bits para indicar el desplazamiento, basta con añadir un registro de 5 bits (AUXBUS), que será multiplexado junto con el bus C, para dar lugar a la señal que ataca la entrada C de la ALU.

ALU (Unidad Aritmético-Lógica)

Vamos a describir este bloque con poca rigurosidad, porque a continuación se le dedica un apartado entero. Básicamente, la ALU contiene:

- un desplazador de 32 bits
- la posibilidad de introducir un multiplicador de 32*32 bits con resultado de 64 bits, aunque en función de la aplicación podría bastar con un multiplicador de 32*16 bits y resultado en 48 bits. También es posible realizar una implementación sin multiplicador alguno (ahorro de área)
- un bloque que contiene el conjunto de operaciones aritméticas y lógicas más completo que cualquier ALU pueda contener.
- un multiplexor que en función del código de operación (opcode) y las señales de control aplicada, relaciona los distintos bloques entre sí, con las entradas, y además con la salida.

La ALU contiene tres entradas (A, B, y C), y una salida (Result).

- A: es el operando primero del multiplicador, y el operando primero del bloque de operaciones
- B: es el segundo operando del bloque de operaciones, que puede ser desplazado o no, y además también puede ser el segundo operando del multiplicador, pero en este caso no es desplazado.
- C: en el caso de las operaciones aritméticas y lógicas, se usa este BUS para indicar al desplazador la cuantía de bits a desplazar el operando B. En el caso de las multiplicaciones, como el operando B no se desplaza, se puede utilizar esta entrada como operando de acarreo (esta ALU puede hacer operaciones del tipo $Rd = Rm * Rs + Rn$).

Otras operaciones que puede se pueden realizar son:

- Dejar pasar el bus A, B ó C directamente a la salida (Result)
- Suma y Resta sin desplazamiento: de gran utilidad en las operaciones de memoria, en las que se requiere de operaciones de suma y resta, pero no se necesita de desplazamientos. En ese caso, podemos utilizar el bus C para propagar el dato a almacenar en memoria, pero la ALU requiere de un operando que contenga el número de bits a desplazar, aunque éste sea cero. Para no tener que echar mano del registro auxiliar, hemos creado estas dos operaciones que no hacen uso del desplazador (para la ALU son como las multiplicaciones, por lo que no utilizan el desplazador), y así nos ahorramos el tener que indicar un desplazamiento cero.

Etapas combinacionales de la *pipeline*

La estructura *pipeline* es una sucesión de parejas registro síncrono – bloque combinacional puro que se suceden, como se muestra en el esquema anterior. Su función es procesar el contenido del registro y actuar de forma determinada, dando como salidas los valores que el siguiente bloque toma como entradas.

La etapa *decode* es la única que trata con la instrucción original, la decodifica, se genera la próxima instrucción para la etapa siguiente, y seleccionan los operandos que hay que introducir en los buses para la próxima operación de la ALU. En esta etapa se detecta si se produce interlocking entre registros, y si es así, se introducen estados de no operación, también denominados *bubbles*.

En la etapa *execute* se reciben los operandos de la etapa anterior, y la configuración establecida por la decodificación de la instrucción. A partir de esta etapa la configuración se propaga inalterada de una etapa a otra, hasta completar las cinco. Así sólo se necesita decodificar la instrucción una vez, puesto que las configuraciones cargadas en los registros son cableadas de forma casi directa sobre los distintos componentes combinacionales, pues en la fase de diseño de las etapas se obró con cierta lógica, para que el mapeado del contenido de los registros sobre las entidades funcionales fuera lo más directo posible.

* * * * *

El diagrama completo de partida del CORE, es el siguiente

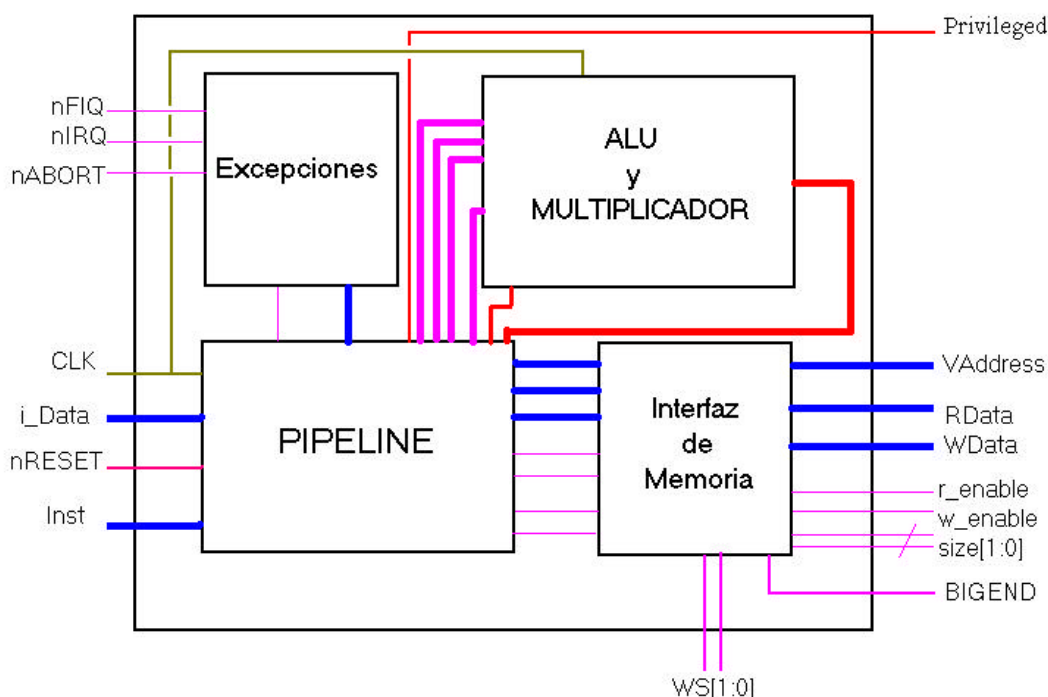


Figura 5-2: diagrama de bloques del CORE

5.3.1 Reducción de buses en la ruta de datos

En la ruta de datos del diagrama que se incluye al principio del capítulo 5.3, se puede observar la existencia de tres buses (A, B, C). A priori, se requerirían cuatro buses para poder cursar los datos necesarios correspondientes a las instrucciones que requieren del máximo número de operandos.

Ya en la descripción del registro AUXBUS incluida en el capítulo 5.3 se adelantaba un poco el razonamiento seguido para conseguir la simplificación de 4 a 3 buses de datos, y en el que el propio registro AUXBUS era una pieza clave.

El problema, fundamentalmente, tenía una doble naturaleza:

- Por un lado, el hecho de que el bus C era compartido por el multiplicador y el shifter, pues eran dos subsistemas mutuamente excluyentes (en las multiplicaciones no hay desplazamiento de operando en paralelo). Ésto aparentemente es un ahorro, pero cuando se usa la ALU para calcular direcciones, y no se requiere de desplazamiento, hay que forzar el bus C a 0x00000000 (es decir, no desplazar es equivalente a desplazar 0 bits). Por tanto, no puede utilizarse este bus para otros fines.
- Existe una instrucción de memoria, variante de STR que utiliza un registro rotado como *offset* para calcular la dirección a utilizar en el acceso a memoria. En ese caso, se requiere introducir un valor en la entrada C de la ALU, esta vez distinto de cero, y además necesitamos también el bus C para cursar el dato que deseamos escribir en memoria.

El problema planteado parecía de difícil solución, pero finalmente se encontró un método para resolverlo:

- En primer problema es bastante sencillo: para aquellas instrucciones, cuyas operaciones de ALU no requieran desplazamiento, pero que necesiten del bus C para cursar datos, se podrían crear dos nuevas operaciones de ALU (MADD, MSUB) con código de operación 1xxxx (ver apartado 5.5.1 – las operaciones cuyo código de operación empieza por ‘1’ deshabilitan el shifter), semejantes a ADD y SUB en su implementación, pero que no hacen uso del shifter, sino que utilizan los operandos originales sin desplazar. De este modo no es necesario indicarle a la ALU que el desplazamiento es de ‘0’ bits, pues de alguna manera estas operaciones “se saltan” el shifter.
- Como el desplazamiento a introducir en la entrada C de la ALU es un valor de tan sólo 5 bits, basta con añadir un registro de 5 bits (AUXBUS), que será multiplexado junto con el bus C, para dar lugar a la señal que ataca la entrada C de la ALU. En el caso de la instrucción que causaba el problema, el multiplexor conecta la entrada C de la ALU con el registro AUXBUS, en lugar de con el bus C (el decodificador de instrucciones debería haber introducido el desplazamiento entonces en el registro AUXBUS en lugar de en el bus C, en el que habría introducido el dato a cargar en memoria)

5.4 Generación de “*bubbles*” y vaciado de la *pipeline*

Prácticamente la mayoría de las máquinas RISC completan sus instrucciones aritméticas en un ciclo de reloj, pero existen instrucciones más complejas (como es el caso de la multiplicación de 64 bits en el ARM8/9) que necesitan de más de un ciclo para ejecutarse.

Tales instrucciones permanecen en la misma etapa (por ej. EX) durante más de un ciclo, de modo que es necesario insertar lo que se llama un *bubble*. Esto no es más que una NOP (non operation) que la etapa en la que la instrucción va a ocupar más de un ciclo de reloj propaga a la siguiente etapa, pues ésta espera recibir una instrucción para realizar su parte del proceso de la misma.

Time	IF	DEC	EX	WB
1	i_1			
2	i_2	i_1		
3	i_3	i_2	i_1	
4	i_4	i_3	i_2	i_1
5	i_4	i_3	i_2	○
6	i_4	i_3	i_2	○
7	i_5	i_4	i_3	i_2

Bubble

Es necesario generar *bubbles* cuando nos encontramos con:

- Operaciones de latencia elevada: en nuestro caso, la multiplicación de 64 bits, o la instrucción SWP..
- Acceso a memorias o a periféricos mapeados en memoria, que necesitan más de un ciclo para estabilizar los datos.
- Cuando se produce *interlocking* entre dos accesos (lectura y escritura) al mismo registro.
- Saltos

Saltos (“branches”)

Para realizar una ejecución eficiente de los saltos, es necesario realizar cambios bastante significativos en la arquitectura de base del computador. Un salto simple, como el retorno de una subrutina, requiere que un cierto número de instrucciones que le siguen en memoria, sean cargadas en la *pipeline* para luego ser vaciadas de la misma.

Ocurre que cuando una instrucción es un salto, éste no se realiza de forma efectiva hasta que no llega a la etapa EX. Esto quiere decir que en el momento de producirse el salto (es decir, en el momento en el que se actualiza el PC a la dirección de salto), existe una instrucción en DEC

Time	IF	DEC	EX	WB
1	i_1			
2	i_2	i_1		
3	i_3	i_2	i_1	
4	i_4	i_3	i_2	i_1
5	○	○	○	i_2
6	i_a	○	○	○
7	i_b	i_a	○	○

y otra en IF que no deberían estar allí, pues se ha producido un salto, y la instrucción siguiente debería ser la apuntada por la dirección de salto, y no la que secuencialmente sigue al salto.

Si no se toman las oportunas medidas, estas instrucciones finalizarán su ejecución, con lo cual, el flujo del programa no sería el deseado. La única solución es vaciar las etapas del *pipeline* implicadas (IF y DEC), e introducir *bubbles*.

Implementación en el CORE

Para poder detectar estas situaciones, cada etapa tiene que ofrecer al control global una señal especificando si va a modificar un registro (destino) o no, y si así fuese, indicar qué registro (mediante un bus de 4 bits).

De este modo, cuando una señal es decodificada, se comprueba al poner un operando en el bus correspondiente, si coincide con uno de los registros que se van a modificar por etapas posteriores:

- si coincide con el destino de una instrucción que está en *execute*, habrá que introducir 3 *bubbles*.
- si coincide con el destino de una instrucción que está en *memory*, habrá que introducir 2 *bubbles*
- si coincide con el destino de una operación que está en *write*, habrá que introducir un único *bubble*.

Simplemente hay que añadir al control global una máquina de estados con tres estados, que se encargue de procesar todas las señales anteriormente expuestas.

Las señales introducidas reciben el nombre de:

- *stop_execute*: el registro de instrucción de la etapa *execute* se actualiza al mismo valor que contiene, y al registro de la siguiente etapa, se le manda una instrucción especial de NOP. Así esta etapa queda congelada.
- *stop_decode*: exactamente el mismo funcionamiento que la anterior.
- *stop_fetch*: aquí basta con dejar de incrementar el contador de programa (PC) para congelar esta etapa, y mandar una instrucción a *decode* con código de condición “1111”, que para nuestro CORE significa nunca ejecutar (*never*)
- *reset_fetch*: el próximo valor del contador de programa es igual al actual.
- *reset_decode*: el próximo valor del registro de instrucción tiene un código de operación “1111”, que significa no ejecutar.
- *reset_execute*: el próximo valor del registro de instrucción contendrá una instrucción especial denominada NOP (no operación).

En el caso del *interlocking* entre registros (fenómeno que se produce cuando una instrucción necesita como operando un registro que se supone ha sido modificado por una etapa posterior, pero aún no se ha actualizado el contenido del registro, pues esto ocurre en la quinta etapa), sólo es necesario detener la etapa *fetch* y *decode*.

Igual ocurre con las instrucciones que tienen una latencia de dos ciclos de reloj. Estas se implementan de la siguiente forma:

- existe un biestable, mcontrol, que normalmente está a '0'. Cuando se decodifica una instrucción de alta latencia, se congela el PC, y la próxima instrucción a decodificar, es exactamente la misma, pero ahora el biestable mcontrol estará a '1'. Así se sabe que estamos en el segundo ciclo de una instrucción que requiere de dos ciclos de reloj.
- la configuración transmitida a la siguiente etapa será diferente según mcontrol esté a '0' o a '1', y como a partir de *execute*, el contenido del registro de instrucción se transmite íntegro, todas las demás etapas realizarán el comportamiento correcto, y el fenómeno acaecido será totalmente transparente para ellos.
- el contenido del biestable mcontrol, también es transmitido al ALU, pues así, cuando mcontrol='1', el multiplicador sabe que lo que tiene que escribir en el bus de resultado son los 32 bits más significativos de la multiplicación extendida de 64 bits.

La última situación de este tipo que se puede dar, son los saltos, bien por instrucciones de saltos, o bien porque se produzcan excepciones (el comportamiento del microprocesador es el mismo, sea cual sea la fuente del salto).

El salto se ejecuta de forma efectiva en la tercera etapa, y no es más que forzar el próximo valor del PC a la dirección de salto.

El problema es que ya hay dos instrucciones, una en decode y otra en fetch, que no son las que deberían seguir a la instrucción de salto, pues la dirección de la instrucción correcta se acaba de calcular ahora.

Aquí no basta con parar las etapas anteriores; de hecho ni siquiera es necesario. El procedimiento correcto sería vaciar físicamente esas dos etapas, y además, a partir de la etapa donde se ejecuta el salto, propagar NOP (esto último no es estrictamente necesario, pues basta con que las etapas "memory" y "write" traten el salto como una NOP).

Existe otro problema que requiere introducción de bubbles, y es cuando se desea un acceso a memoria que se prolongue durante más de 1 ciclo de reloj; pero esta situación será añadida al diseño después de la FASE II, cuando se hayan implementado las instrucciones de memoria, y sepamos a ciencia cierta el comportamiento del CORE ante las mismas.

Nótese que las señales reset/stop_execute no han sido necesarias aún.

El control de la *pipeline*, por ahora, implementaría el siguiente algoritmo escrito en pseudocódigo:

```
SEGUN ctrl_pipeline HACER
    CUANDO e0 ENTONCES
        stop_fetch<='0';
        stop_decode<='0';
        SI ws>0 ENTONCES
            stop_fetch<='1';
            stop_decode<='1';
        SI ws=3 ENTONCES ctrl_pipeline<=E2
        SI ws=2 ENTONCES ctrl_pipeline<=E1
        SI ws=1 ENTONCES ctrl_pipeline<=E0
        ELSE ctrl_pipeline<=E0;
        FIN SI
    CUANDO e1 ENTONCES
        stop_fetch<='1';
        stop_decode<='1';
        ctrl_pipeline<=E0;
    CUANDO e2 ENTONCES
        stop_fetch<='1';
        stop_decode<='1';
        ctrl_pipeline<=E1;
FIN SEGÚN
```

La señal WS es el número de *bubbles* o estados de no operación a introducir, y es función de la etapa en la que se haya producido el interlocking (nótese que el interlocking siempre es entre *decode*, y una de las siguientes etapas, como ya se ha explicado anteriormente), y es generada por los decodificadores de los buses, a la hora de seleccionar qué registro introducir en qué bus.

Por otro lado, las señales de 'reset', pueden ser aplicadas directamente desde la etapa *execute*, simplificando así la lógica interna del control general, aunque funcionalmente estas señales se englobarían dentro del propio control, y por eso aparecen así dibujadas en las ilustraciones.

NOTA: en esta descripción, se están ilustrando los mismos pasos que se siguieron en el diseño, y por el mismo orden. Con esto queremos decir que el control definitivo de la *pipeline* no es exactamente como aquí se muestra, sino que inicialmente era así. Posteriormente, y tras descubrir nuevas situaciones que posteriormente serán descritas, se fue modificando, hasta alcanzar el diseño definitivo.

A continuación se adjunta una sección del diagrama funcional original de la pipeline, pero que incluye todas las modificaciones propuestas anteriormente.

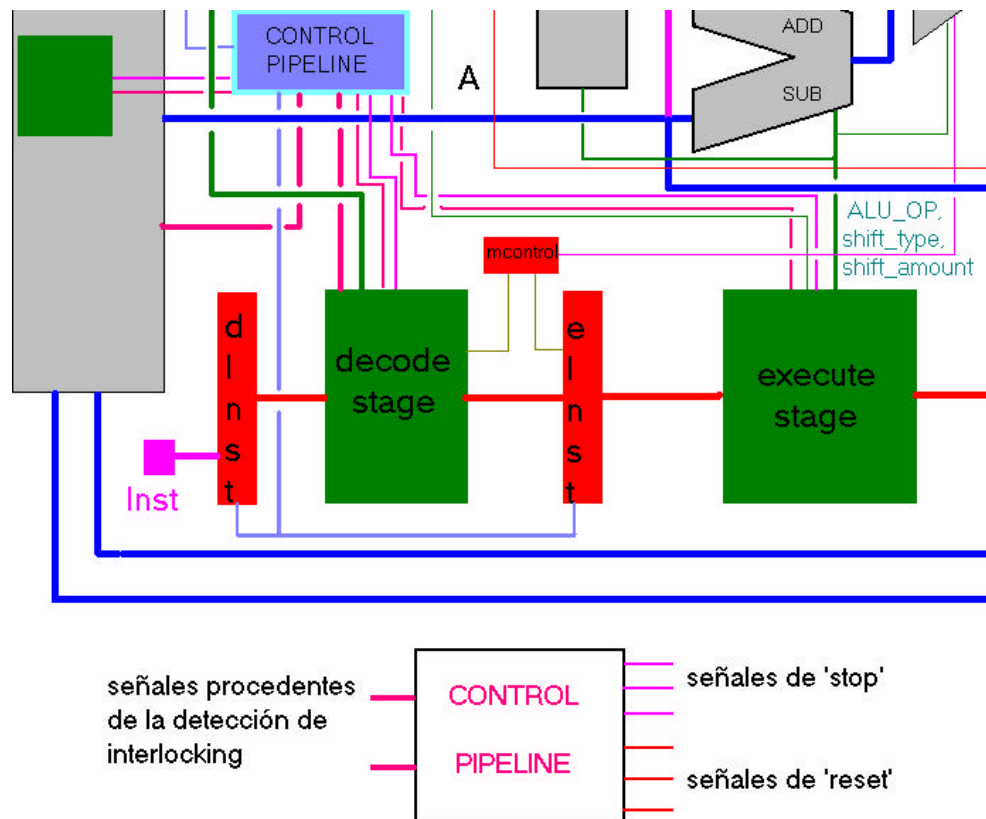


Figura 5-3: inclusión de las señales globales de control

Nótese que la etapa de *fetch*, que a priori era tan simple como un pequeño sumador que incrementase el PC de 4 en 4, se ha ido complicando, al tener que resolver situaciones de “congelación” de la cuenta, actualización del PC a una dirección de salto, o generación de un ‘reset’ en la etapa.

Este bloque puede realizar cuatro tipos de operaciones de rotación, en función del tipo de desplazamiento especificado en el campo “shift type” de la instrucción:

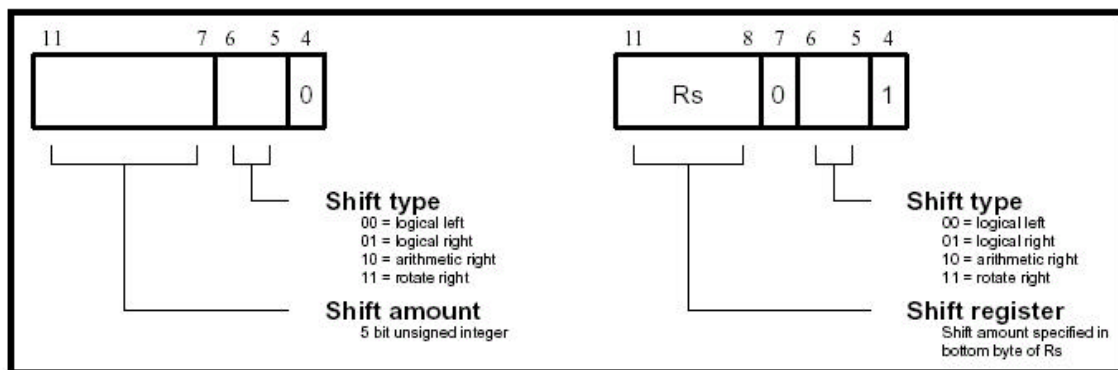


Figura 5-5: campos de control del shifter

- **Desplazamiento Lógico a la Izquierda:** la posición de cada bit es alterada, de modo que el bit x pasa a ocupar la posición $x+y$, siendo y la cantidad de bits de desplazar. Los ‘ y ’ bits menos significativos serán rellenos por ‘0’. Nótese que ‘ y ’ bits quedarán “fuera del registro”. El último bit en salir será el bit de acarreo del desplazador, que en el caso de que la operación fuese lógica, sería el bit Cout de la ALU.

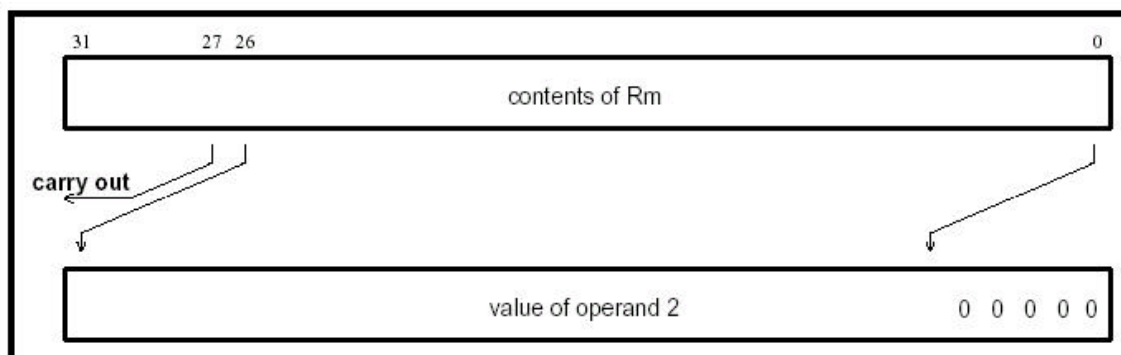


Figura 5-6: desplazamiento lógico a la izquierda (LSL)

- **Desplazamiento Lógico a la Derecha:** es igual al anterior, pero el bit x pasa a ser el $x-y$; los y bits más significativos se rellenan con ‘0’, y el último bit en salir por la derecha (que sería el bit $y-1$) será el acarreo del desplazador.

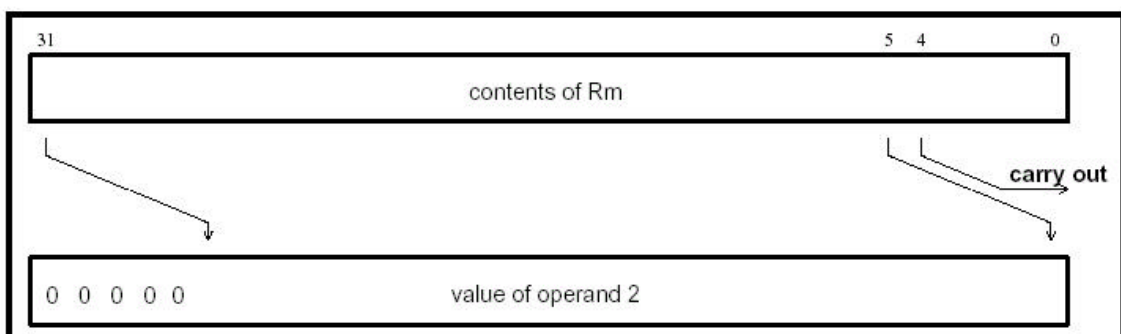
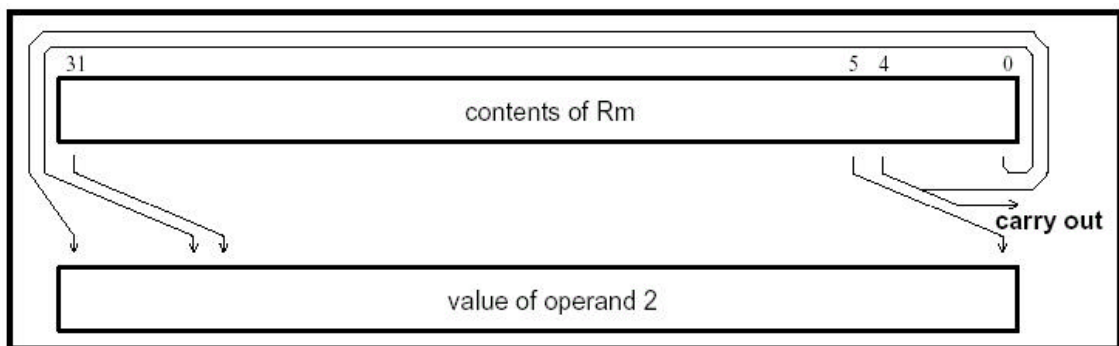


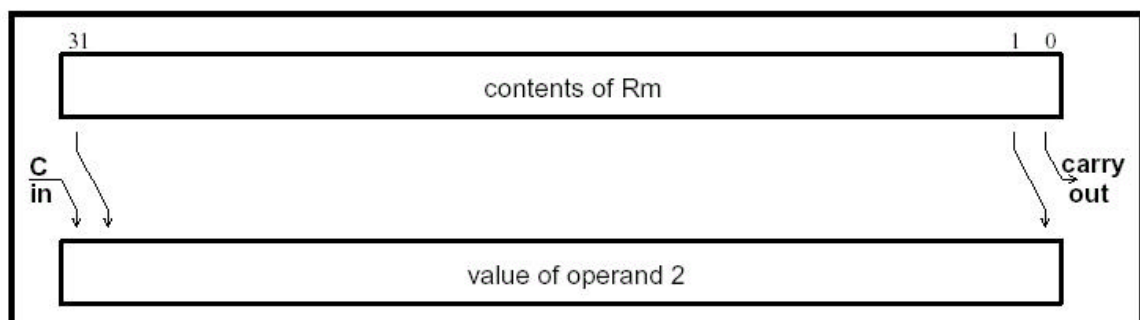
Figura 5-7: desplazamiento lógico a la derecha (LSR)

-
- The diagram shows two horizontal registers. The top register is labeled 'contents of Rm' and has bit positions 31, 30, ..., 5, 4, ..., 0 marked above it. Below this register, a series of downward arrows indicates a range of bits. The bottom register is labeled 'value of operand 2'. To the right of the bottom register, a bracket labeled 'carry out' points to the right, indicating the carry-out from the addition.

- Rotación a la derecha: es igual que el desplazamiento lógico a la derecha, pero los bits que se van perdiendo por la derecha son reutilizados para ir rellenando los que van quedando libres por la izquierda. Es decir, se realiza lo que literalmente se denomina una rotación.



Existe un caso excepcional, dado por ROR #0, que se usa para codificar una función especial del desplazador, que se denomina *rotated right extended (RRX)*. Su efecto sobre el operando B es el siguiente:



Existe una serie de valores especiales, que quedan fuera de lo que podría denominarse “funcionamiento normal”, y que acaba ser descrito. Además el comportamiento ante alguno de estos valores depende de si el número de bits a desplazar aparece explícitamente en la instrucción, o es especificado en el byte menos significativo de un registro. Por ello, el formato de la entrada C del shifter (9 bits) es el siguiente:

8	7 – 5	4 – 0
---	-------	-------

- Bit 8: se activa ('1') si el desplazamiento es especificado en el byte menos significativo de un registro
- Bit 7-5: se utilizan junto con los bits 4-0 para expresar el desplazamiento, cuando este viene especificado en el byte inferior de un registro
- Bit 4-0: se utilizan además cuando el desplazamiento es un valor inmediato de 4 bits, especificado en la propia instrucción.

Operando Inmediato (C(8)='0')

- LSL #0 opB = B Cout = Cin
- LSR #0 opB = 0x00000000 Cout = B(31)
- ASR #0 opB=(others=>B(31)) Cout = B(31)

Opeando byte menos significativo de un Registro (C(8)='1')

Byte	Descripción
0	Se usa el valor inalterado de Rm como segundo operando, y el valor antiguo del CPSR se utiliza como salida del carry
1-31	Mismo significado que en el caso del valor inmediato
32	LSL: resultado 0, activa Z y carry=Rm[0] LSR: resultado 0, activa Z y carry=Rm[31] ASR: todos los bits y el carry son igual a Rm[31] ROR: el resultado es igual a Rm, y carry=Rm[31]
> 32	LSL: resultado 0, activa Z y carry='0' LSR: resultado 0, activa Z y carry='0' ASR: todos los bits y el carry son igual a Rm[31] ROR: #n, con n > 32, se hace n-32 tantas veces como sea necesario para que quede en el rango 1-32.

Funcionamiento del bloque aritmético-lógico

Este bloque es capaz de realizar un total de 16 operaciones entre aritméticas y lógicas, utilizando como entradas el operando A y el operando B (éste último, una vez aplicado el desplazamiento), y dando como salida el resultado de la operación.

También es capaz de dejar pasar un operando tal cual, en el caso de que la operación de la ALU así lo requiera.

En la tabla que se adjunta a continuación se muestran el conjunto de operaciones que este bloque combinacional es capaz de realizar, de modo que el diseño en VHDL consistirá en un multiplexor que según el código de operación conecte los buses adecuados a la unidad funcional adecuada (sumador, restador, AND, etc).

Mnemónico	Acción del bloque aritmético-lógico	Notas
AND	(operando A) AND (operando B)	
EOR	(operando A) EOR (operando B)	
SUB	(operando A) - (operando B)	
RSB	(operando B) - (operando A)	
ADD	(operando A) + (operando B)	
ADC	(operando A) + (operando B) + Cin	
SBC	(operando A) - (operando B) + Cin - 1	
RSC	(operando B) - (operando A) + Cin - 1	
TST	como AND pero se ignora el resultado	Rd debe ser 0x0000
TEQ	como EOR pero se ignora el resultado	Rd debe ser 0x0000
CMP	como SUB pero se ignora el resultado	Rd debe ser 0x0000
CMN	como ADD pero se ignora el resultado	Rd debe ser 0x0000
ORR	(operando A) OR (operando B)	
MOV	(operando B)	Rn debe ser 0x0000
BIC	(operando A) AND NOT (operando B)	
MVN	NOT (operando B)	Rn debe ser 0x0000

Bloque de Control

Su función es calcular Cout y Vout en función del código de operación, de Vin, de Cin, y del resultado de la operación.

Además es el encargado de procesar la señal mult_long, de modo que el resultado de 17 bits temporalmente almacenado en el registro “acumulador” debe ser añadido al resultado cuando esta señal esté a ‘1’.

En el siguiente párrafo se va a explicar con detalle el bloque multiplicador, y se ilustrará la necesidad de este mecanismo, así como las ventajas que proporciona al diseño.

Multiplicador

En este apartado se van a exponer las especificaciones del multiplicador que se debe añadir al CORE generado para poder conseguir un correcto funcionamiento.

En principio existe un fichero VHDL con el multiplicador combinacional adecuado para la ALU, pero no se va a sintetizar en la FPGA porque ocupa un área enorme en comparación con el resto del CORE. Además, como el objetivo de este proyecto es que el diseño realizado sirva como una celda de librería para el diseño de ASICs, es preferible que el futuro usuario de la misma añada otra celda que contenga un multiplicador de las mismas características, pero que con toda seguridad será mucho más óptimo desde el punto de vista del área ocupada.

Para que ésto pueda ser así, en el fichero “core.vhd” existe una variable de nombre NOMULT y de tipo GENERIC, que permite una implementación con o sin multiplicador.

- Así, en nuestra implementación, NOMUL=1, con lo cual el multiplicador es sustituido por un bloque que simplemente da como salida el valor 0x00000000.
- En el caso de que se disponga de un multiplicador más adecuado, bastará con añadir el fichero en cuestión, y cambiar la asignación anterior por NOMUL=0.

Para realizar las simulaciones con VHDL-Simili se ha utilizado un fichero con un multiplicador que no es nada óptimo, pero que nos sirve para verificar el correcto funcionamiento de la circuitería que permite la realización de operaciones con operandos de 32 y 64 bits, al añadir el multiplicador.

En principio, tenemos tres operandos, dos longitudes posibles de resultado, y seis tipos de multiplicaciones:

- Operandos:
 - A: 32 bits
 - B: 32 bits
 - C: 32 bits / 64 bits
- Posibles resultados:
 - Resultado1: 32 bits
 - Resultado2: 64 bits
- Operaciones de multiplicación:
 - MUL: Resultado1 = $A * B$
 - MLA: Resultado1 = $A * B + C[31:0]$
 - UMULL: Resultado2 = $A * B$ (tratados como enteros)
 - UMLAL: Resultado2 = $A * B + C [63:0]$ (enteros)
 - SMULL: idem UMULL, pero se considera el signo
 - SMLAL: idel UMLAL, pero tratados con signo

Luego en principio, sería necesario un multiplicador con dos operandos de 32 bits y resultado en 64 bits. Además se requeriría de un sumador de 64 bits para realizar la suma en las operaciones UMLAL y SMLAL.

Otra cuestión es el tratamiento de operaciones con números codificados en Ca2, que también a priori requerirían de un multiplicador que tuviese en cuenta el signo (vamos a demostrar que esto último no es necesario)

¿Es indiferente considerar los operandos con o sin signo?

El tratamiento de operandos con y sin signo en las multiplicaciones es indiferente, pues los bits menos significativos del resultado es el mismo en ambos casos. Para ilustrar este hecho, consideremos el siguiente ejemplo:

Operando A: 0xFFFFFFFF6 (-10 en Ca2, ó 4294967286 si se considera entero)

Operando B: 0x00000014 (20 en ambos casos)

Resultado (si los considero con signo): 0xFFFFFFFF38

Resultado (si los considero enteros): 0x13FFFFFF38

Se podría demostrar que ocurre exactamente lo mismo si ambos operandos son negativos, y además el producto es conmutativo, así que si la longitud requerida para el resultado es igual o menor al tamaño del operando de mayor longitud, el resultado sería siempre correcto, sin tener que preocuparse de si estamos trabajando con enteros, o con valores con signo.

¿Es posible reducir el tamaño del multiplicador?

Esta posibilidad depende exclusivamente de la aplicación que se le vaya a dar al diseño. Así, si las operaciones de multiplicación no requieren de más de 24 bits de precisión por ejemplo, podría incluirse un multiplicador de 32*16 bits, y resultado en 48 bits, para realizar todas las multiplicaciones anteriormente expuestas, sin pérdida de precisión en el resultado.

En primer lugar, hay que hacer notar que esta simplificación no aumenta la latencia de las instrucciones (es decir, el número de ciclos de reloj necesarios en cada etapa para ejecutarse), pues las multiplicaciones que ahora necesitan de dos ciclos de reloj ya requerían de este tiempo antes de la simplificación, pues el bus del resultado es de 32 bits, y el resultado es de 64 bits.

¿Quién se encarga de que op2 y op3 contengan los bits adecuados de B y C?

Realmente es la *pipeline* la que realiza esta labor, ya sea de forma explícita o implícita.

En las operaciones de 64 bits, hemos dicho que el tercer operando que se suma a la multiplicación es de 64 bits. En éste caso, en el ciclo 1, la *pipeline* escribe en C los 32 bits menos significativos del operando de acarreo, y en el ciclo 2, los 32 bits más significativos, de modo que todo este planteamiento es correcto.

En el caso de op2, cuando la multiplicación es de 64 bits, es la ALU quien explícitamente asigna el valor correcto, pues la *pipeline* manda el operando B completo (32 bits).

Bloque de control de la ALU

Se encarga de calcular los valores para la actualización de los flags del CPSR, es decir, de asignar valores a Cout y Vout. De todos modos, la *pipeline* es la que tiene la última palabra a la hora de llevar acabo esta actualización, pues puede ser que mantenga intacto el CPSR por algún motivo (por ejemplo, que la instrucción lleve el bit S=0, que significa que no deben actualizarse dichos flags).

En el caso de Vout (que sólo tiene sentido para operaciones **aritméticas y con signo**), se sigue la siguiente regla:

- hay *overflow* cuando al sumar dos números positivos el resultado se codifique como negativo, o al sumar dos números negativos, el resultado se codifique como positivo, aunque en ningún caso se rebase el número de bits representables (32).
- también existe *overflow* si al restar un número positivo a uno negativo el resultado es positivo, o viceversa, aunque en ningún caso se rebase el número de bits representables (32).
- en el resto de las combinaciones posibles, si los operandos son representables en Ca2, el resultado también lo será, pues siempre será menor que el mayor de los operandos.

Especificaciones del multiplicador

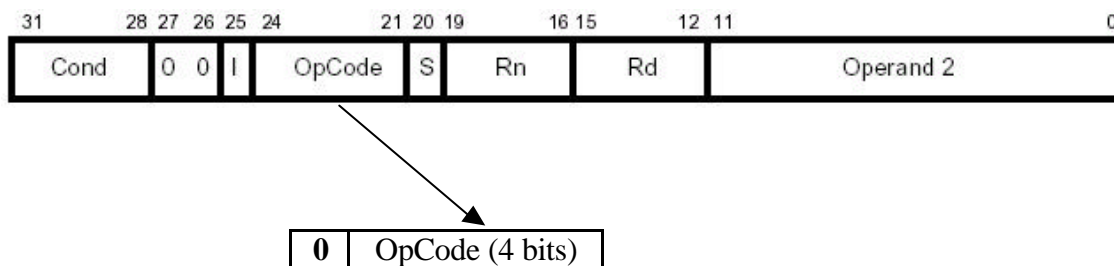
$$op1 (32 \text{ bits}) * op2 (32 \text{ bits}) = Y (64 \text{ bits})$$

Operación	Nº de ciclos de reloj necesarios
MUL	1
MLA	1
UMULL	2
UMLAL	2
SMULL	2
SMLAL	2

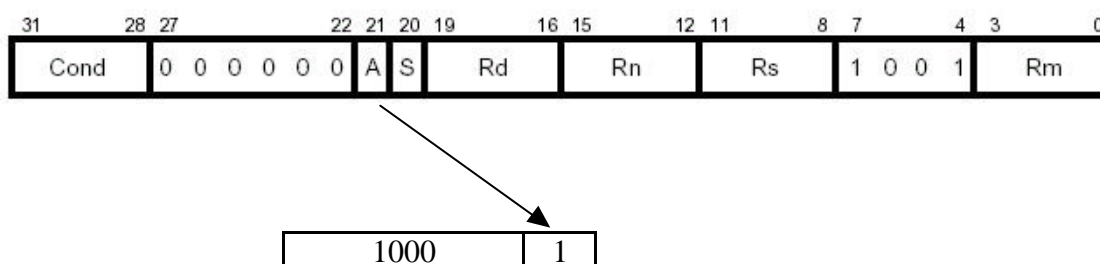
5.5.1 Códigos de Operación

El código de operación de la ALU es un vector de 5 bits. Para asignar el código a cada operación, se ha seguido un procedimiento que minimiza la lógica necesaria para generarlo.

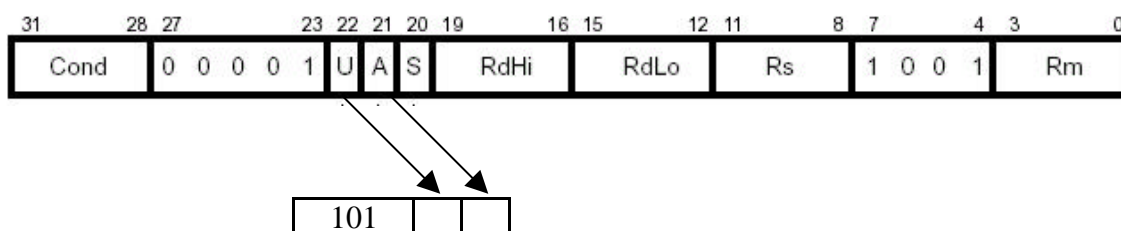
Los códigos que se refieren a operaciones del bloque aritmético-lógico y **desplazador**, se generan de la siguiente forma:



Los referidos a MUL, MLA, siguen el siguiente proceso:



Y en el caso de las multiplicaciones de 64 bits, el proceso de generación del código de operación para la ALU es:



Nótese que asignando los códigos de operación de la ALU de esta forma, se minimiza la lógica necesaria, pues la instrucción hay que decodificarla forzosamente, y una vez que ha sido decodificada, la generación del código de operación no es más que un mapeo directo de bits

Esta simplificación no es la única consecuencia de que los códigos se ordenen de esta forma. La repercusión en la ALU (en lo que a simplificación se refiere) es mucho mayor, pero antes de explicar los motivos, vamos a mostrar los códigos:

Código	Operación	Código	Operación
00000	ALU_AND	10000	ALU_MUL
00001	ALU_EOR	10001	ALU_MLA
00010	ALU_SUB		
00011	ALU_RSB		
00100	ALU_ADD	10100	ALU_UMULL
00101	ALU_ADC	10101	ALU_UMLAL
00110	ALU_SBC	10110	ALU_SMULL
00111	ALU_RSC	10111	ALU_SMLAL
01000	ALU_TST		
01001	ALU_TEQ	11000	ALU_MADD
01010	ALU_CMP	11011	ALU_MSUB
01011	ALU_CMN		
01100	ALU_ORR	11100	ALU_A
01101	ALU_MOV	11101	ALU_B
01110	ALU_BIC	11110	ALU_C
01111	ALU_MVN	11111	ALU_NOP

Nota: MADD y MSUB realizan la misma operación que ADD y SUB, pero utilizando el operando B original en lugar del desplazado.

El **primer bloque (bit más significativo del código a '0')**, corresponde con operaciones aritmético-lógicas, que además **requieren de desplazamiento** del operando B, con lo cual, basta con utilizar ese bit para saber si el operando B a utilizar es el original, o el desplazado. Cada subgrupo de operaciones que tienen alguna característica común, tiene los bits más significativos iguales, de modo que para implementar la operación que se deriva de esa característica común, basta con comprobar dichos bits.

El **segundo bloque (bit más significativo del código a '1')**, tiene una característica común (aunque luego cada subgrupo tenga otras funcionalidades comunes), y es que son operaciones que no requieren de shifter. Pero el significado de estos bits es más fuerte que la no necesidad de desplazamiento: *si este bit está activo, se cortocircuita la entrada y la salida del shifter, de modo que el operando B utilizado es el original, sin desplazar, a la fuerza.*

Así, si necesitamos el bus C para cursar algún dato a través de la *pipeline*, pero la operación de la ALU requiere de un desplazamiento nulo (0 bits), podemos utilizar una instrucción del segundo grupo, con lo cual nos ahorramos el tener que introducir un desplazamiento nulo en el bus C.

Precisamente MADD y MSUB, con código de operación 1xxxx, fueron introducidas para llevar a cabo la reducción de buses expuesta en el apartado 5.3.1.

5.6 Diseñando la *pipeline*

La estructura básica que se va a implementar ya ha sido expuesta en la sección 5.3 del presente capítulo. La arquitectura básica divide el proceso de las instrucciones en cinco etapas, que no son más que unidades funcionales puramente combinacionales, y existen ciertos registros donde se almacenan la configuración de dichos bloques, resultados intermedios, direcciones de acceso a memoria, ...

En su mínima expresión, la arquitectura *pipeline* podría verse como una sucesión de “bloques registro de configuración + bloque combinacional de procesado”

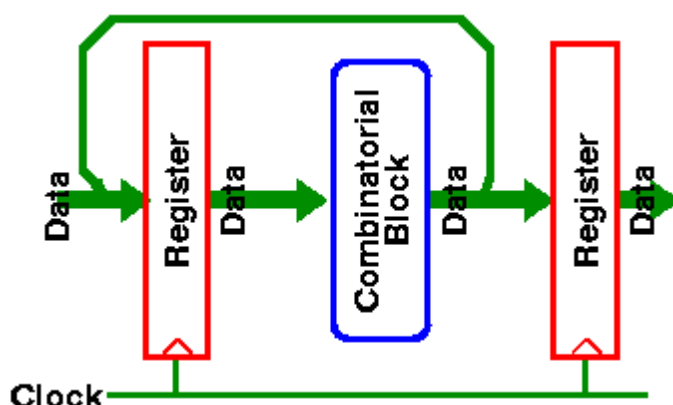


Figura 5-11: unidad básica de la estructura *pipeline*

Se puede observar en la figura anterior un lazo de realimentación: esta configuración permite congelar el proceso de la etapa, realimentando el registro de configuración con el mismo valor que contenía en el ciclo de reloj anterior. Básicamente congelar la *pipeline* resulta bastante simple, aunque hay que tener cuidado de que no se modifique ningún registro de propósito general o de estado, si el resultado de la modificación depende del número de veces que se ejecute la instrucción. Con ésto se quiere expresar que el que una instrucción quede congelada en una etapa, no es más que mantener durante varios ciclos de reloj la misma configuración, los mismos operandos, los mismos registros, etc, pero hay que tener cuidado de que no se esté escribiendo ningún resultado cuyo valor varíe con cada ciclo de reloj.

Por tanto, el primer paso en el diseño del CORE será implementar este tipo de estructura de proceso de datos, para después ir aumentando los bloques combinacionales mediante la adición de nuevas instrucciones a decodificar.

Nótese que en la mayoría de los casos, las instrucciones implementadas requerirán de la existencia de ciertos registros, que han sido explicados en el apartado 5.3, donde se mostraba un diagrama de la arquitectura de partida.

En segundo lugar, sería conveniente tener implementada desde un principio la máquina de estados y los mecanismos en las etapas que permiten parar o vaciar la estructura *pipeline*.

En resumidas cuentas, los dos primeros pasos a seguir en el diseño de la arquitectura del CORE son:

- Diseñar todas las posibles configuraciones de las etapas en función de las instrucciones que procesan, y de toda la ruta de datos, prestando un especial cuidado en que la generación de los códigos de configuración minimice la lógica necesaria para ello.
- Implementar la máquina de estados que se expuso en el apartado 5.4 para generar ciclos de espera, y todas las señales y mecanismos necesarios para que las distintas etapas obedezcan las órdenes de dicha máquina.

Formato de los registros de configuración de la *pipeline*

La pipeline contiene los siguientes registros que podríamos denominar “de configuración”:

- *d_inst*: propiamente dicho, es el registro de instrucción, pues contiene la instrucción a decodificar tal y como ha sido extraída de la memoria de programa. Por tanto, es el único que no se adapta al formato que vamos a desarrollar en esta sección; de hecho, es a partir de la instrucción que almacena de donde se genera la configuración que se propagará por el resto de los registros.
- *e_inst*: registro de configuración de la instrucción que está en la etapa *execute*.
- *m_inst*: contiene la configuración de la etapa de memoria
- *w_inst*: contiene la configuración de la etapa de escritura

Una vez que se decodifica la instrucción y se obtiene la configuración, ésta se propaga por los registros de configuración de las siguientes etapas, permaneciendo su contenido inalterado (a excepción de algún caso).

Sentar los criterios para generar las distintas configuraciones es fundamental, pues debe ser suficiente para gobernar el comportamiento de toda la lógica combinacional que se añada para implementar los “comportamientos” requeridos por cada instrucción. Estos criterios son los siguientes:

- es fundamental realizar un estudio de todas las instrucciones para poder extraer las operaciones básicas que se han de realizar en todas las etapas para su procesamiento.
- existirán entidades funcionales que operarán en la mayoría de las instrucciones. Los códigos que gobiernan su operación deberán aparecer en posiciones fijas en todos los formatos de configuración, porque así estos bits podrán ser cableados directamente con las entradas de control de dichas entidades, y no se necesitará ninguna lógica adicional.
- el número de configuraciones deberá ser el mínimo posible: al ser posible, una configuración por cada familia de instrucciones.

El formato de los registros de configuración e_inst, m_inst y w_inst será el siguiente:

31	28	27	24	23	0
condición	tipo_instruc.	configuración			

El código de condición será exactamente el mismo que contenga la instrucción a partir de la cual se genera la configuración.

El resto de campos se detallan a continuación. Hay que tener presente lo siguiente: los campos marcados en verde son de posición fija, y deben usarse

- bien para el fin establecido
- bien para dejarlos en blanco

pues dichos campos son cableados directamente con la lógica combinacional correspondiente.

Proceso de Datos / PSR

27	24	23	22	21	20	16	15	12	11	10	9	8	7	4	3	0
0000	s_t	S	alu_op	Rd	MRS	MSR	Ps	I	mask							

- s_t: indica a la ALU el tipo de desplazamiento
- S: es el bit 'S' de la instrucción
- alu_op: su función es indicar a la ALU el código de operación a realizar
- Rd: contiene el registro destino de la operación si ésta escribe el resultado en un registro, o bien se dejará en blanco (0x0000).
- MRS: si este bit está a '1', indica que la instrucción es MRS
- MSR: si este bit está a '1', indica que la instrucción es MSR
- Ps: su significado coincide con el bit Ps de la instrucción PSR correspondiente
- I: su significado coincide con el bit I de la instrucción PSR correspondiente
- mask: su significado coincide con el campo "mask" de la instrucción PSR

Instrucciones de Multiplicación

27	24	23	22	21	20	16	15	12	11	8	7	0
0001	L	mctrl	S	alu_op	RdLo	RdHi						

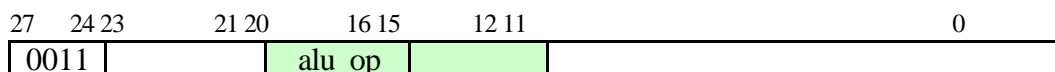
- L: si está activo, indica que la multiplicación es de 64 bits
- mctrl: solo tiene sentido en multiplicaciones de 64 bits, que requieren de dos ciclos de reloj para su ejecución. Si está a '0', indica que estamos en el ciclo primero, y si está a '1', indicará que se trata del ciclo segundo.

Salto

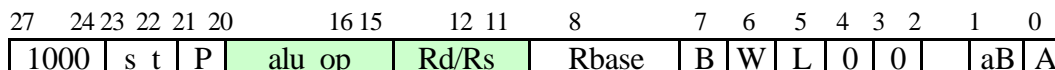
27	24	23	22	21	20	16	15	12	11	0
0010	L		alu_op							

- L: indica que se trata de un salto con retorno.

Undefined



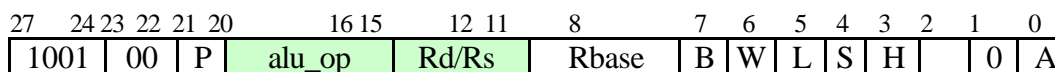
Transferencia Simple de Datos (LDR_STR)



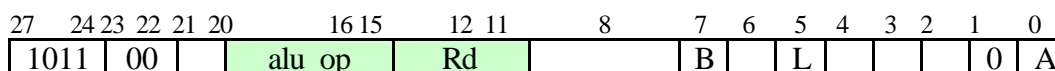
En esta configuración, todos los bits tienen el mismo significado que en la instrucción correspondiente, a excepción de:

- aB: recordemos que existe una instrucción de tipo STR, que necesita de un registro auxiliar para indicar al ALU la cantidad de bits a desplazar el operando B. Si este bits está activo, indica al multiplexor que muestra al ALU el operando correcto, que debe utilizar el registro auxiliar (AUXBUS) en lugar del bus C.
- A: se activa en la etapa 4^a (memoria) si se produce un fallo de memoria (abort), y sirve para indicar a la etapa 5^a que no debe escribir nada en ningún registro, pues se ha producido un error.

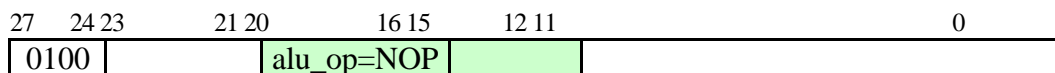
Transferencia Compleja de Datos (Halfword and Signed Data Transfer)



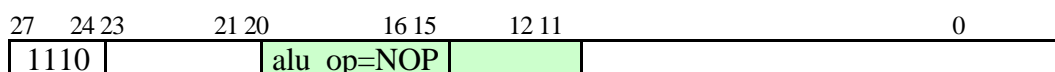
SWP



SWI



Instrucción NO RECONOCIDA

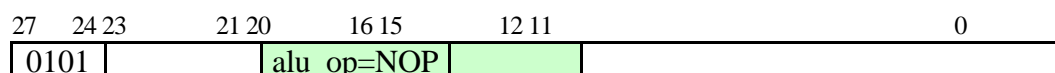


Instrucciones del COPROCESADOR

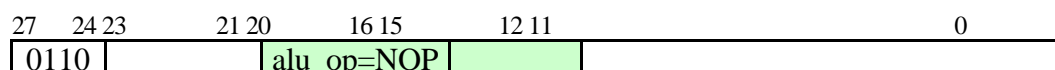
Estas instrucciones no han sido implementadas en este proyecto, pero el decodificador es capaz de reconocerlas, y generar las siguientes configuraciones. De este modo, de deja la puerta abierta a la introducción de las mismas de forma sencilla, pues para implementarlas bastaría con:

- Idear el formato para completar las configuraciones que se adjuntan, respetando los campos de posición fija (marcados de color verde).
- En la etapa *decode*, introducir en los buses los operandos necesarios, y generar las configuraciones de acuerdo a lo establecido en el paso anterior.
- En el resto de etapas, realizar las operaciones necesarias, según el contenido del registro de configuración.

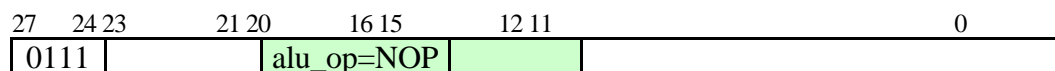
CPD:



LDC_STC:



MRC_MCR:



Observaciones:

Nótese que el formato de estas configuraciones permite simplificar la lógica en las etapas combinacionales de la *pipeline*, debido a que:

- existen campos de posiciones fijas, que se cablean directamente bien con la ALU, o bien con el decodificador de escritura de registros, de modo que no requieren de ninguna lógica adicional
- los bits que se refieren a la configuración de un mismo elemento, se encuentran en la misma posición, se trate de una familia de instrucciones u otra.
- a consecuencia de todo lo anterior, el diseño de las etapas combinacionales se reduce a implementar ciertas funciones lógicas, que serán habilitadas o inhabilitadas en función del valor de uno o varios bits de estos registros.

Generación de Estados de Espera

Las etapas *fetch*, *decode*, y *execute*, dispondrán de una señal de **stop_<etapa>**, y de una señal de **reset_<etapa>**.

Dichas etapas deberán “congelar” su funcionamiento, o “vaciar” su registro de configuración y sustituirlo por una no operación (código de operación NOP - “1111”), en función de la señal que se active.

Para la generación de estados de espera, sólo se requieren las señales ‘stop’. De hecho, las señales de ‘reset’ sólo son necesarias cuando se produce un salto o una excepción, y serán generadas directamente por la etapa *execute* cuando así lo detecten. Con esto queremos significar que están fuera del alcance del circuito que aquí describimos.

La máquina que a continuación vamos a especificar, se encargará de generar las señales de ‘stop’ únicamente. A esta altura del diseño, no podemos todavía tener conocimiento de cómo se implementarán los ciclos de espera para el acceso a memoria, pero sí podemos afirmar que la máquina que ahora ideemos, deberá ser modificada más adelante para activar las señales de ‘stop’ adecuadas si la situación así lo requiere.

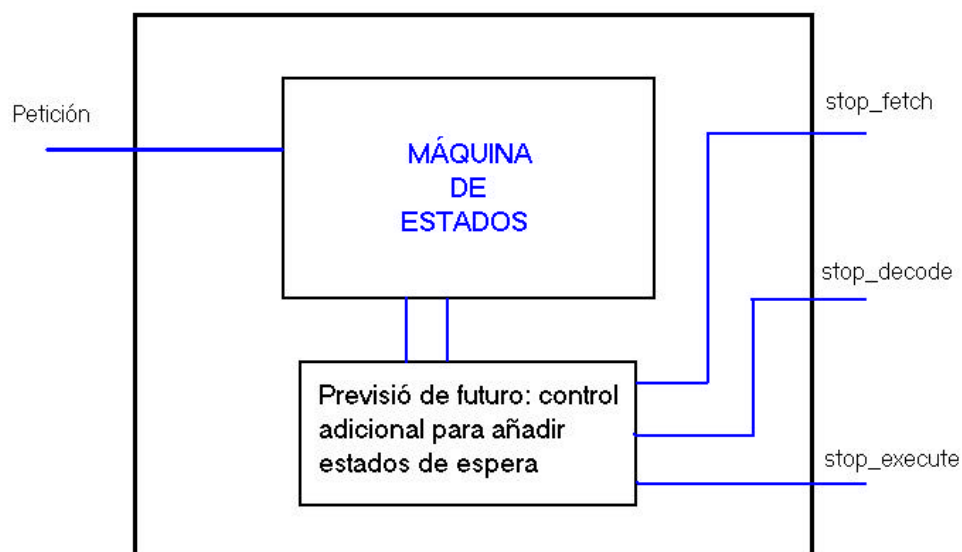


Figura 5-12: maquina global de control

Para resolver situaciones de conflicto entre registros, que son las previstas hasta el momento, sólo se requiere de las siguientes señales:

- **Petición:** el decodificador de registros a escribir en los buses A, B y C inspeccionará el registro destino (Rd) contenido en los registros de configuración de las etapas *execute*, *memory*, *write*. Si uno de los operandos requeridos se encuentra aún en una de estas etapas, se realizará una petición de ciclos de espera (1, 2, 3); en caso contrario, *petición* = 0.

- Sólo es necesario “congelar” las etapas *fetch* y *decode*, pues el operando requerido podría estar incluso en *execute*, y hay que dejar que siga su curso para que en un futuro se encuentre disponible.

La señal *stop_execute* ha sido incluida como previsión de futuro, pues sabemos que cuando se necesite que un acceso a memoria dure más de un ciclo de reloj, habrá que “congelar” también esta etapa (pues la etapa de memoria estará ocupada durante más de un ciclo con el mismo acceso).

También se ha añadido un bloque adicional para la generación de las señales, que en principio no es más que un cable. En un futuro, será un OR entre la señal generada por la etapa de memoria cuando requiera de más de un ciclo de reloj para el acceso, y de las señales generadas por la máquina de estos de este bloque.

5.6.1 Fase I

Hasta este momento, únicamente disponemos de una ruta de datos que incluye un banco de 37 registros y algunos registros de almacenamiento temporal (diagrama de la sección 5.3 del presente capítulo), de un ALU (sección 5.5), y de una serie de registros de configuración.

Una vez que tenemos implementada esta estructura, resta por diseñar las etapas combinacionales que realizarán el procesamiento de la información, es decir, queda por abarcar la realización física de las instrucciones en sí.

En esta fase nos vamos a dedicar únicamente a la implementación de las instrucciones de proceso de datos, de operaciones con el PSR, de multiplicación, de salto, y a la instrucción *undefined*.

Cuando esta fase concluya, su funcionamiento podrá ser verificado (todas las pruebas que se realizaron son explicadas en el siguiente capítulo), pues en ese momento la estructura será 100% funcional (aunque claro está, sólo podrá ejecutar las instrucciones implementadas hasta este momento).

Etapla *FETCH*

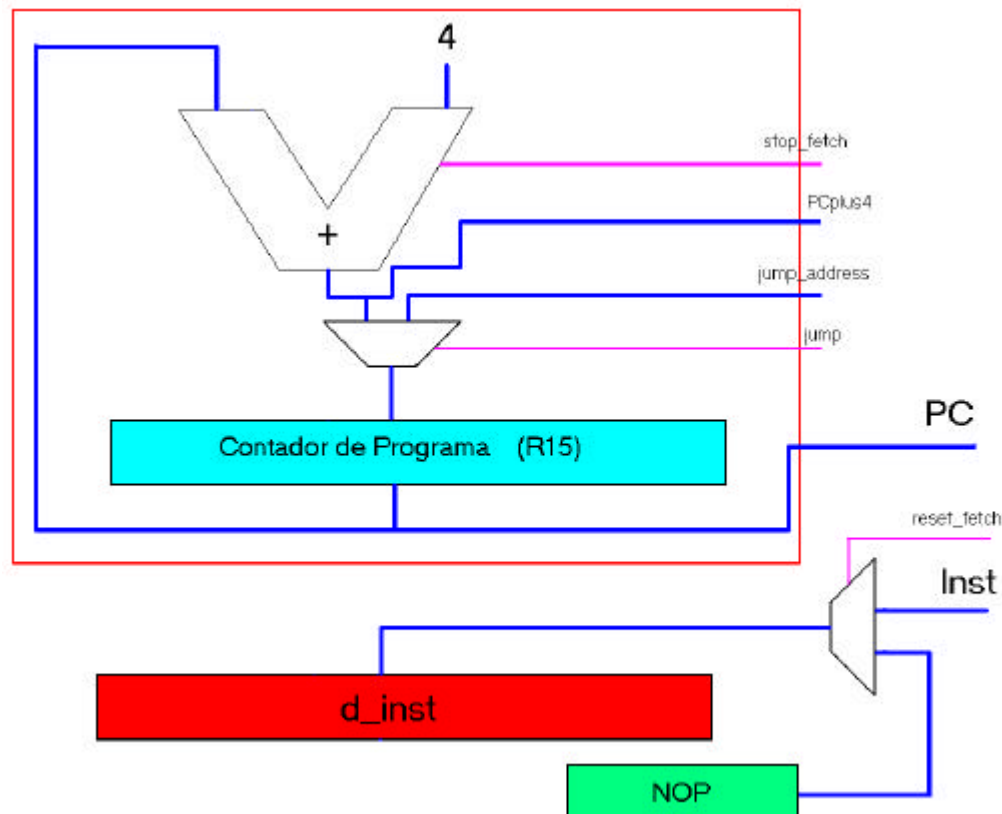


Figura 5-13: etapa *FETCH* tras la FASE I

Inicialmente, tras la FASE I de diseño, tenía únicamente dos funciones:

- Actualización del PC: el modo normal de funcionamiento consiste en incrementar el contador de programa de 4 en 4. Esto quiere decir que se mantienen intactos los dos bits menos significativos (que valdrán siempre '0'), pues las instrucciones son de 32 bits. Cuando se produce un salto, el nuevo valor del PC vendrá dado por la dirección absoluta de salto (que será calculada y suministrada por etapas posteriores).
- Carga de nueva instrucción: con la dirección contenida en el PC se accede a la memoria de programa, y se carga en el registro `d_inst` la instrucción correspondiente, para posteriormente ser decodificada por la siguiente etapa en la *pipeline*.

Nótese que existen dos señales especiales en el diagrama anterior:

- `Stop_fetch`: en lugar de actualizar el contador de programa con `PC+4`, lo hace con el valor original (PC), con lo cual la etapa queda congelada.
- `Reset_fetch`: cuando está activa, no se escribe en `d_inst` la instrucción leída de la memoria de programa, sino que se sustituye por una NOP (instrucción de no operación).

Gracias a estas señales, se pueden implementar los mecanismos de:

- introducción de ciclos de espera adicionales
- introducción de *bubbles*, tras modificar el valor del PC y sustituirlo por una dirección que no es `PC+4` (Por ej; tras un salto, o una excepción)

Las señales anteriores son generadas, respectivamente, por la máquina global de control en el caso de *stop_fetch*, y por la etapa que alteró el valor del PC (que siempre es la etapa *execute*), en el caso de la señal *reset_fetch*.

La funcionalidad de esta etapa será levemente alterada a lo largo de la fase II, pues al incluir el control de excepciones y la etapa de acceso a memoria, se necesitarán más funciones adicionales que las implementadas hasta el momento.

Etapla *DECODE*

Esta etapa se puede ver básicamente, por un lado, como un gigantesco multiplexor, que multiplexa una serie de configuraciones de registro de control para las etapas posteriores en función de la decodificación de la instrucción, y por otro lado, como una serie de multiplexores que escriben en cada bus (A, B, C) el operando procedente del registro correspondiente, o del propio campo de la instrucción.

El diseño de esta etapa es crítico, pues de la optimalidad del mismo dependerá la simplicidad de las etapas sucesivas.

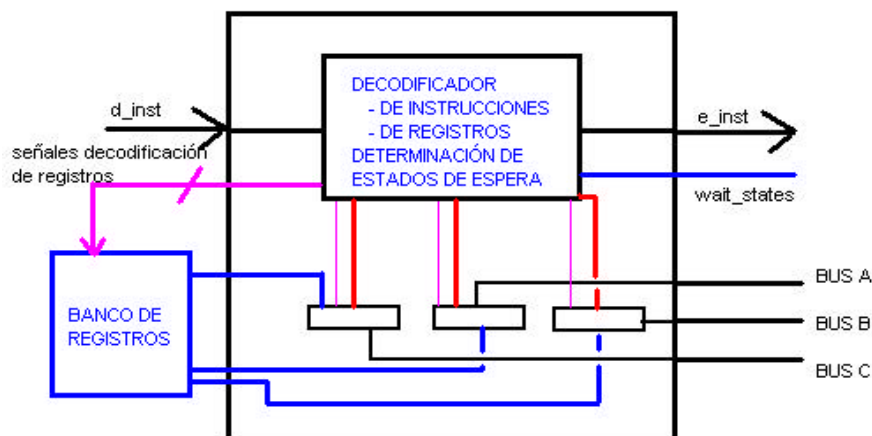


Figura 5-14: etapa DECODE tras la FASE I

En el diagrama anterior se ilustran los bloques funcionales que componen el bloque combinacional de la etapa, y también algunos registros, como los que gobiernan el contenido de los tres buses de la ruta de datos (A, B, y C). Esta etapa se encarga de:

- Generar la configuración adecuada para la etapa *execute*, que será almacenada en el registro *e_inst*, como resultado del proceso de decodificación de la instrucción contenida en el registro *d_inst*.
- Cargar en los buses A, B y C los operandos adecuados en función de la instrucción. Estos operandos proceden de
 - Un registro de propósito general, de los que componen el banco de registros
 - O bien son valores inmediatos especificados en cierto campo de la propia instrucción.
- En el proceso de generación de las señales de decodificación del banco de registros, se comprueban los registros destino de las instrucciones que están en etapas sucesivas (*execute*, *memory*, *write*). Si el registro a cargar en un bus, coincide con alguno de los registros destino de dichas etapas, se escribirá en valor 1, 2 ó 3 (en función de la etapa que sea) en el bus *wait_states*, mediante el cual se informa a la máquina global de estados de que es necesario generar 1, 2 ó 3 ciclos de espera, para así asegurarnos que el registro destino ha sido actualizado, y ya tenemos un operando válido para escribir en el bus correspondiente. En la descripción de la máquina global de control, este bus se denominó “*petición*”.

En el diagrama de la página anterior, aparecen tres registros de 32 bits, que mantienen el valor escrito en los tres buses, de manera que mientras no se actualicen estos registros, los buses mantendrán su valor.

Para realizar esta función se plantearon dos soluciones en su momento:

1. La anteriormente expuesta, que consistía en escribir el valor deseado en un registro, al que se conectaba directamente el bus. Controlando el registro se lograba el dominio del bus, y además permitía mantener un valor en el bus durante varios ciclos de reloj.
2. Utilizar tres registros de 4 bits únicamente, que contenían no el valor del registro, sino el código del registro a cargar en el bus, de manera que manteniendo este código durante varios ciclos de reloj, el decodificador de registros mantenía los buses conectados a los registros correspondientes, de manera que lograba el mismo efecto que en el punto anterior. El único inconveniente era mantener en el bus un valor inmediato especificado directamente en el campo de la propia instrucción; esto, a priori, se lograba manteniendo la instrucción durante varios ciclos, pero un detallado análisis desvela que se requieren de ciertas precauciones que complicarían bastante la lógica de decodificación.

En nuestro caso, el diseño consume muchísimos SLICES de la FPGA, pero no requiere de más de un tercio de flip-flops, de modo que no tenemos límite de bits a utilizar en los registros, pero sí lo tenemos con la lógica adicional, sobre todo si se puede evitar usarla, por lo que se optó por el primer método.

Este método tiene una consecuencia, que no es ni positiva ni negativa, pero que hay que tenerla muy en cuenta a la hora de realizar operaciones con el PC: cuando el operando es R15 (PC), el valor que se toma al operar en *execute*, es el que tenía cuando la instrucción estaba en *decode* debido al método que hemos usado (capturamos el valor en un registro, pero en la etapa de *decode*). Si hubiésemos utilizado la alternativa segunda, como lo que se escribe en *decode* es el código del registro a usar como operando en el siguiente ciclo, el valor del PC sería el que éste tendría en *execute*, con lo que habría una diferencia de 4 según el método que se utilice.

Este efecto es importante a la hora de almacenar direcciones de retorno, pues existirá una discrepancia con la dirección que almacenaría el ARM8/9 original y hay que asegurarse de que se añade o se resta 4 (según el método que se utilice), para que los programas funcionen de forma correcta.

Para ello se utiliza la señal PCplus4 generada en *fetch*. A la hora de escribir en uno de los registros anteriormente mencionados el valor del PC, se utiliza el valor PCplus4, que contiene el valor del PC actual +4. De este modo, cuando en el ciclo siguiente se lea el bus, el valor corresponderá con el que el ARM8/9 original habría escrito en el bus en este ciclo, luego el problema estaría solucionado. Como ya se dijo anteriormente, no es un efecto ni positivo ni negativo, pero hay que tenerlo presente.

Etapla EXECUTE

Si el diseño de la etapa anterior era vital para la simplificación de etapas posteriores, el diseño de la presente etapa es el más crítico a la hora de lograr que nuestro diseño opere de forma correcta.

Las funciones asignadas a esta etapa son las siguientes:

- Configurar la ALU de forma correcta en función del contenido del registro de configuración *e_inst*.
- También en función de dicho registro de configuración, y una vez obtenido el resultado de la operación de la ALU, activar/desactivar los flags adecuados en el registro de control de estado del programa (CPSR).
- Cargar el dato adecuado en el registro *m_data* (normalmente será el dato a escribir en registro en la etapa *write*).
- Generar direcciones absolutas de saltos (bien porque se está ejecutando una instrucción de salto, o bien porque se produce una excepción, y en función del tipo que sea, se carga el vector de excepción correspondiente en el bus *jump_address*), que son escritas en el bus *jump_address*, y se informa a la etapa *fetch* de que tiene que usar esta dirección para actualizar el PC a través de activación de la señal *jump*.
- En el bus de 4 bits *e_destiny* escribe el registro destino de la instrucción. En caso de que la instrucción no necesite escribir ningún resultado, tendrá un valor incierto, pero la señal *e_nro* estará activada, especificando así este hecho.

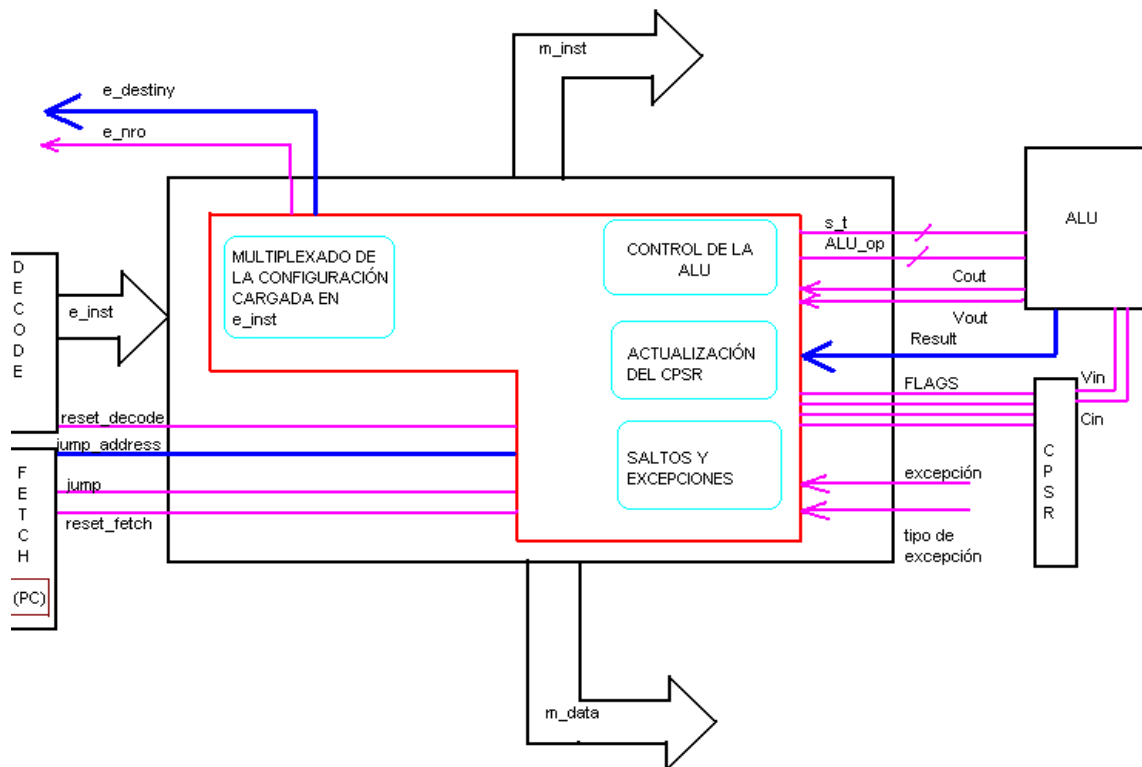


Figura 5-15: etapa EXECUTE tras la FASE I

Ejecución Condicional

La función más importante de esta etapa es verificar si se cumple la condición especificada en el campo correspondiente de la instrucción.

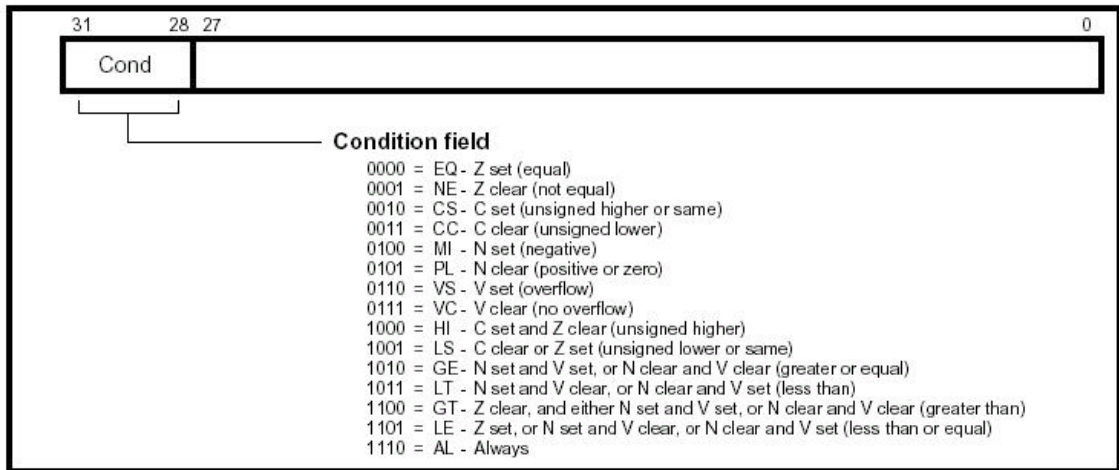


Figura 5-16: campo de condición

Realmente, lo que se hace es actuar si no se cumple la condición, evitando que se escriba ningún resultado en ningún registro.

En principio, parece tan sencillo como no propagar ningún dato a la siguiente etapa, y cambiar el campo de tipo de configuración por una cNOP (configuración de no operación) en el caso de que no se cumpla la condición especificada en la instrucción.

Esto sería así de sencillo en etapas donde no se modifique ningún registro, sino que simplemente se realiza un procesamiento de un dato, y luego se propaga. De éste modo, bastaría con no propagar el resultado. Pero en esta etapa en concreto, hay que tener en cuenta dos hechos:

- Es la etapa donde se modifica el CPSR
- La activación de la señal *jump* se traduce en una precarga asíncrona, y por tanto inmediata, del PC, de modo que en el siguiente ciclo se actualizaría al valor cargado en el bus *jump_address*.

Por eso, cuando no se cumple la condición de ejecución de la instrucción, no basta con lo anteriormente explicado, sino que hay que asegurarse de que no se modifique el CPSR, ni tampoco se altere el valor del contador de programa (PC) mediante las señales *jump* y *jump_address*.

Lo que finalmente se hace en el diseño es utilizar una señal *execute*, de manera que si ésta vale 0, nos aseguraremos que no se active la señal *jump*, y de que el contenido del CPSR quede inalterado.

Etapla *MEMORY*

En esta fase no se dota de ninguna funcionalidad a la etapa *memory*, pues las instrucciones a implementar en esta fase no requieren de acceso a memoria.

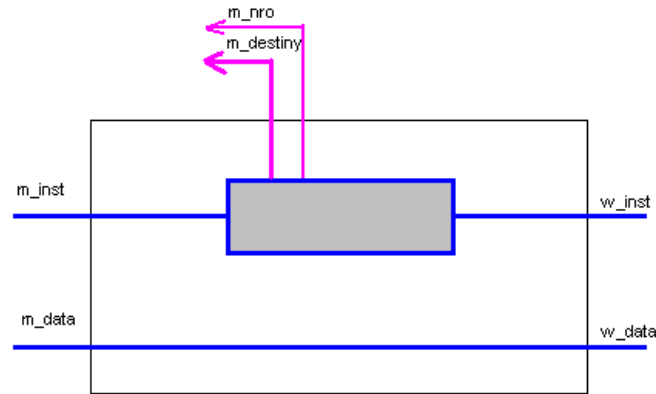


Figura 5-17: etapa *memory* tras la FASE I

En el diagrama se observa que sí se realiza una pequeña función, y es cargar en el bus *m_destiny* el código del registro destino de la instrucción que está en esta etapa. En el caso de que la instrucción actual no requiera de registro destino, se activará la señal *m_nro*.

Esto hay que hacerlo porque aunque no se quiera dotar por el momento de ninguna funcionalidad a esta etapa, es necesario que el decodificador de registros de la etapa *decode* pueda detectar si se produce *interlocking* con la instrucción que se encuentra en *memory*. De lo contrario, el funcionamiento de toda la *pipeline* sería erróneo.

Nota: la FASE II de la realización del diseño en VHDL se dedica prácticamente íntegra a la implementación de las instrucciones de memoria, con lo que esta etapa será desarrollada prácticamente en su totalidad en la siguiente sección del documento.

Etapla *WRITE*

Esta es la última de las etapas que compone la estructura *pipeline*. Cuando una instrucción llega a esta etapa, ya ha completado completamente su ejecución (y si se trata de un acceso a memoria, ya habrá concluido), y sólo resta escribir los resultados en el registro destino (en el caso de que haya que escribir algún resultado).

Muchas instrucciones no tendrán que escribir ningún resultado, por lo que la mayoría de las instrucciones no requieren de ninguna función (o desde otro punto de vista, la única función que requieren es una NOP). Es importante resaltar que **estamos hablando de las instrucciones implementadas en la FASE I** (en la FASE II las instrucciones de memoria tienen que escribir de vuelta el registro base, con lo cual casi todas las instrucciones implementadas hacen uso de esta etapa).

De todos modos, aunque la mayoría de instrucciones implementadas en esta fase no requieran de ninguna operación, las instrucciones que tienen que escribir un resultado son las que aparecen con más frecuencia, por lo que se trata de una fase vital en el diseño de la *pipeline*.

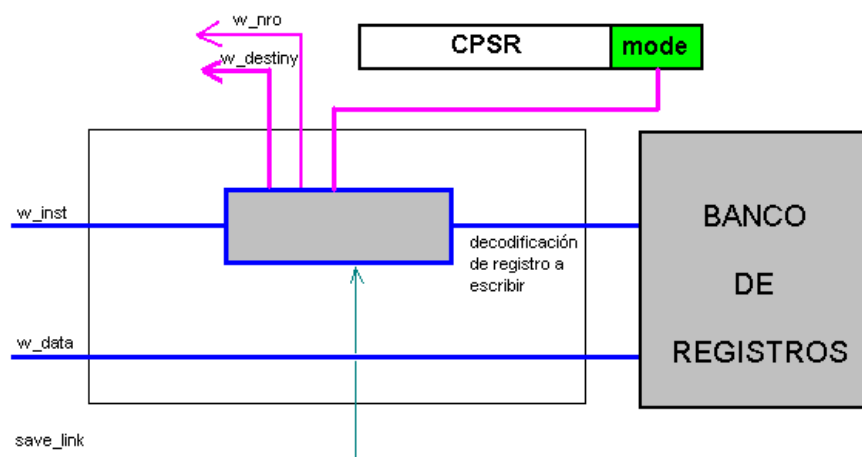


Figura 5-18: etapa *write* tras la FASE I

Las funcionalidades implementadas en esta etapa, son las siguientes:

- Escribir en el bus *w_destiny* el código del registro destino de la instrucción, para facilitar la detección de interlocking por la etapa *decode*. En el caso de que la instrucción no requiera de operando destino, este bus tendrá un valor indeterminado; para indicar este hecho, se activa la señal *w_nro*.
- La señal *w_nro* también sirve para indicarle al propio proceso *write* que tiene que escribir un registro. Al activarse, también se activa el decodificador de escritura de registros, de manera que escribe el dato contenido en *w_data* en el registro indicado por *w_destiny*.
- Cuando la señal *save_link* se activa, debe de salvar el contenido del contador de programa en el registro R14_<modo_actual>.

5.6.2 Fase II

Tras la finalización de la FASE I, se llevaron a cabo una serie de simulaciones y test detalladas en el capítulo sexto, y se verificó el correcto funcionamiento de la misma.

Además, tras la observación de los resultados se propusieron una serie de mejoras en el diseño, de modo que en la presente fase hay que dotar a las cinco etapas de funcionalidades adicionales para implementar las modificaciones propuestas y además poder añadir la circuitería de acceso a memoria, y de decodificación de las instrucciones de acceso a memoria.

Etapas *FETCH*

Se observó que en ciertas operaciones que utilizaban el contador de programa de alguna manera, el valor del contador de programa cargado estaba incrementado en cuatro unidades con respecto al que se debería usar.

Este hecho es algo completamente lógico, pues la pipeline cuenta con 5 etapas, y el diseñador tiene en algunas ocasiones libertad para escoger en qué etapa se lee el valor del contador de programa (al menos entre dos etapas). Pero como en cada ciclo de reloj el PC se incrementa en 4 unidades, el valor variará si lo capturo en una etapa u otra (es decir, en un ciclo de reloj, o en el siguiente).

En otras operaciones el valor era correcto, así que decidimos que en lugar de modificar la arquitectura, sería conveniente idear un método para obtener PC-4, pero sin utilizar un restador.

La solución es bien sencilla: basta con utilizar un registro de desplazamiento adicional, que almacena el valor anterior del PC cuando éste se incrementa en 4 unidades. Las mismas señales que habilitan/inhabilitan el PC se pueden utilizar para controlar el funcionamiento de este registro, de modo que siempre contendrá el valor PC-4.

Existen situaciones en las que este registro contendrá un valor erróneo; imaginemos por ejemplo, que el PC contiene el valor 8 y por tanto el registro contendrá el valor 4. Si ahora se produce un salto, en lugar de incrementarse el PC, se sustituiría por la dirección de salto (por ejemplo, 20), y el nuevo registro se actualizaría al valor antiguo (8).

En la situación descrita anteriormente, el nuevo registro no contendría PC-4, pero no importa porque en la situación anterior, el valor correcto desde el punto de vista práctico, es precisamente el que contiene el registro (luego el registro de todos modos era necesario), y un ciclo después el valor ya sí será PC-4.

Es importante sustituir un restador por un registro de desplazamiento en nuestro diseño, pues disponemos de exceso de flip-flops, pero tenemos limitación de SLICES.

La arquitectura de la nueva etapa se muestra en la siguiente ilustración:

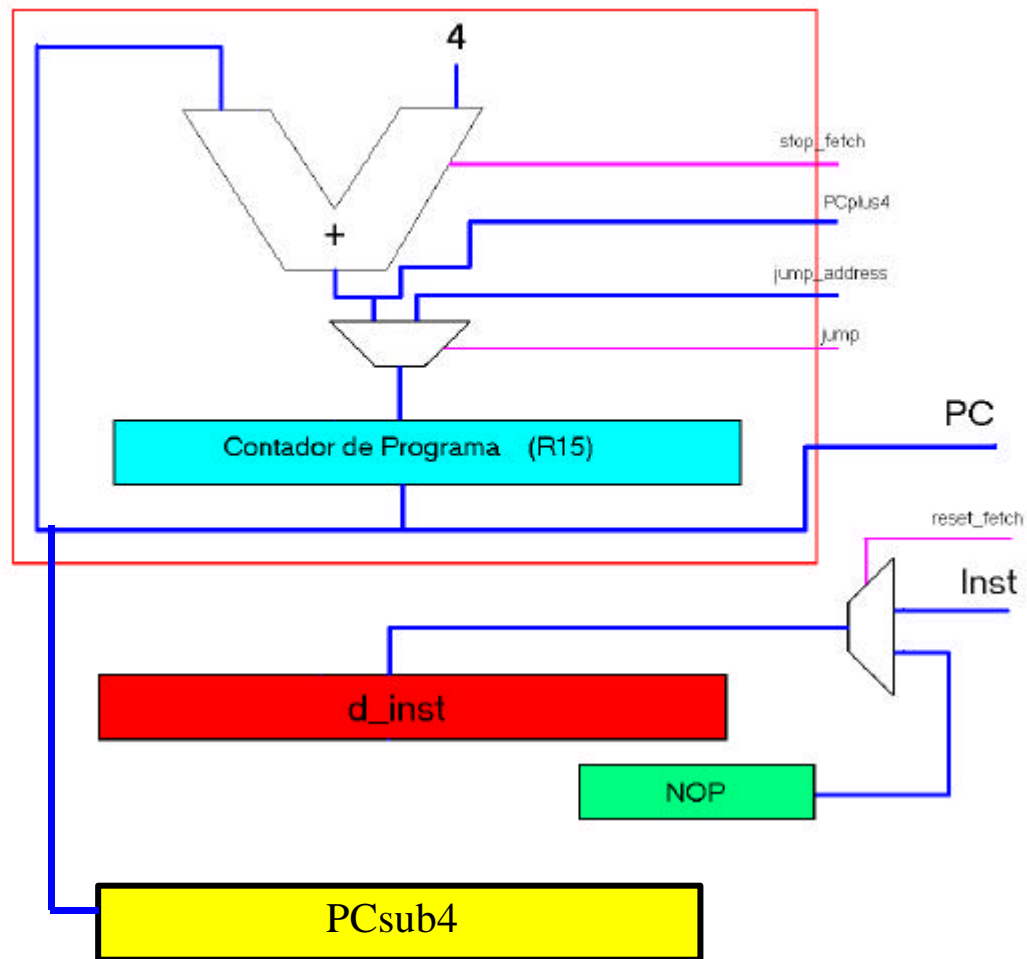


Figura 5-19: etapa FETCH tras la FASE II

Etapa *DECODE*

En esta etapa no hay que realizar ningún cambio significativo. Únicamente hay que añadir los patrones de bits para identificar las nuevas instrucciones al decodificador, y añadir las nuevas configuraciones de registro que hay que adoptar en función del resultado del proceso de decodificación antes mencionado.

Por supuesto, es necesario incluir en los buses adecuados los operandos necesarios para poder llevar a cabo la ejecución de las nuevas instrucciones.

También se realizaron algunas modificaciones a nivel de código VHDL, para lograr como resultado de las mismas una implementación con la misma funcionalidad, pero más óptima en porcentaje de ocupación de la FPGA. Estas modificaciones lograron reducir el porcentaje de ocupación prácticamente en un 50%, luego se puede considerar como un avance bastante significativo en el diseño (misma funcionalidad, mismas restricciones temporales, pero mitad de área ocupada).

Etapa EXECUTE

Aunque el diagrama de bloques no se vea alterado, los bloques que componen esta etapa son los que han sufrido las mayores variaciones, pues se requiere de las siguientes nuevas funcionalidades:

- En esta etapa, se deben implementar todos los modos de direccionamiento explicados en el capítulo cuarto.
- En algunas situaciones, el resultado de la operación con el offset y el registro base es la dirección de acceso a memoria, y además es la dirección que se almacena de vuelta en el registro base.
- Pero en otros casos, la dirección del registro base es la que se utiliza para acceder a memoria, y el resultado obtenido en la ALU al sumar/restar el offset es la que debe almacenarse de vuelta. En este caso, en principio se necesitarían cuatro buses en la ruta de datos, pero con las modificaciones expuestas en la sección 5.3.1 resulta suficiente con la ruta de datos ideada inicialmente, y que se muestra en el apartado 5.3.

Si recordamos en diagrama de partida mostrado en el apartado 5.3, observamos una serie de registros temporales, y de direcciones, que fueron concebidos precisamente para las instrucciones de acceso a memoria.

En los registros temporales se almacena la dirección calculada por la ALU, y que en algunas circunstancias será escrita de vuelta en el registro base, de modo que al igual que ocurría con los operandos normales, el registro base es susceptible de que se produzca el fenómeno de *interlocking*, con lo cual habrá que añadir una circuitería análoga a la que se añadió en su día, pero esta vez para evitar que ocurra este fenómeno con la lectura/escritura del registro base.

Para subsanar el problema, basta con añadir las señales:

- `e_wbReg`: bus de 4 bits que contiene el código del registro base en el caso de que la instrucción requiera escribir las modificaciones de vuelta en el mismo (implementando así distintos modos de direccionamiento).
- `e_wbr`: en el caso de que no se necesite alterar el registro base, el bus anterior tendrá un valor incierto; para notificar este hecho, estará **inactiva** la señal `e_wbr`.

Y modificar el decodificador que relaciona registros del banco de registros con los buses A, B y C, de modo que además de las comprobaciones que ya realizaba, deberá verificar si el registro que carga en un bus no se corresponde con ningún `<etapa>_wbReg` que tenga activa la señal `<etapa>_wbr`.

Etapa *MEMORY*

Realmente el objetivo de la FASE II del diseño es la implementación de esta etapa, pues es ella la que realmente se encarga del acceso a memoria. La etapa *memory* responde al esquema de bloques que se adjunta a continuación:

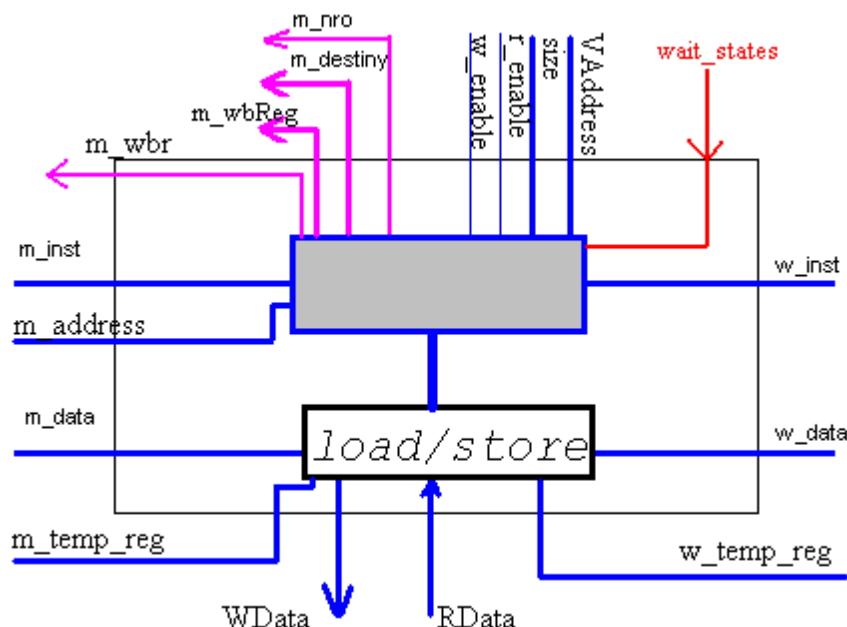


Figura 5-20: etapa *memory* tras la FASE II

Implementadas en los bloques que componen el diagrama anterior, deberán ir las siguientes funcionalidades:

- Load/store: es una unidad funcional que se encarga de cargar un dato de memoria, o salvar un dato en memoria. El método de acceso, dirección a utilizar, tratamiento del dato (rotaciones, desplazamientos, ...), el tamaño de la palabra en el acceso, etc vendrán dados como parámetros por el decodificador de la instrucción, el cual los incluye en la configuración de esta etapa.
- Activación de las señales *m_wbr* y *m_wbReg* en el caso de que sea necesario escribir de vuelta la dirección calculada por la ALU en el registro base, evitando así el fenómeno de *interlocking* entre registros, pues el decodificador de los buses comprueba estas señales antes de cargar cierto registro en un bus.
- Para el resto de instrucciones, basta con las funcionalidades descritas en la FASE I.

Existe también una etapa de control, compuesta por un contador de dos bits, que se encarga de aumentar la latencia de la instrucción de memoria en el caso de que la memoria necesite ciclos de espera para ser accedida. En el siguiente apartado se muestra un diagrama del contador, junto con las modificaciones necesarias en la máquina global de control, y el procedimiento de generación de estados de espera.

Etapa *WRITE*

En esta última fase, la única funcionalidad adicional que hay que incluir, es la escritura en el banco de registros del registro base, en caso de que sea necesario. Para ello, debemos añadir:

- Un decodificador adicional para poder salvar el contenido del registro *w_temp_reg* en uno de los 15 registros de propósito general (excepto R15)
- La activación de las señales *w_wbr* y *w_wbReg* en el caso de que sea necesario escribir de vuelta la dirección calculada por la ALU en el registro base, evitando así el fenómeno de *interlocking* entre registros, pues el decodificador de los buses comprueba estas señales antes de cargar cierto registro en un bus.
- Para el resto de instrucciones, basta con las funcionalidades descritas en la FASE I.

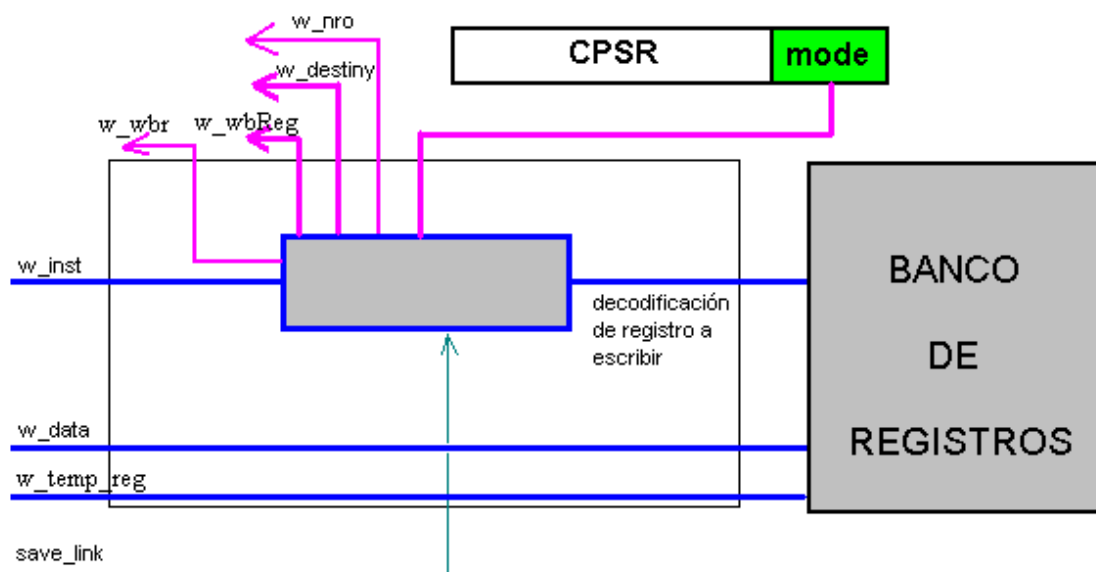


Figura 5-21: etapa *WRITE* tras la FASE II

5.6.3 Generación de Estados de Espera

Ya se han adelantado prácticamente todos los mecanismos para poder realizar el vaciado de la *pipeline*, la “congelación” independiente de las etapas, la generación de *bubbles*, ... pero aún no hemos explicado como es el proceso de generación de estados de espera, pues requiere de la acción conjunta y coordinada de todos los mecanismos anteriormente expuestos.

Antes de abordar este proceso, aún queda por describir tres entidades:

- por un lado, el contador que se añadió a la etapa *memory* en la FASE II del diseño de la *pipeline*
- por otro lado, un biestable denominado *mcontrol* cuya función es vital en la implementación de instrucciones que requieren de dos ciclos de reloj.
- por último, las modificaciones realizadas en la máquina global de control, como consecuencia de la inclusión de los dos mecanismos anteriormente expuestos.

Biestable ‘*mcontrol*’

Quizá este biestable se trate del mecanismo más simple, y a la vez más complejo de todo el diseño de la arquitectura, pues de la sencillez del mecanismo deriva del grado de flexibilidad del resto del diseño.

Existen dos instrucciones que requieren de dos ciclos de reloj para poder ejecutarse; se trata de SWP y de la multiplicación de 64 bits. Para implementar estas dos instrucciones, lo que se hace es generar dos configuraciones en la etapa *decode*, cada una de las cuales contiene las funcionalidades a llevar a cabo en cada ciclo de la ejecución de la instrucción.

Dicho desdoblamiento necesitará tener en cuenta varios factores:

- En primer lugar, deberá congelar las etapas anteriores durante un ciclo extra, pues de una instrucción genera dos configuraciones distintas (realmente lo que se hace es dividir una instrucción en dos, introduciendo un ciclo de reloj adicional)
- Deberá tenerse en cuenta el hecho de que se haya producido *interlocking* justo con esta instrucción, con lo cual, la máquina global de control ya estará “congelando” las etapas “*fetch*” y “*decode*”
- Además, podría pasar que la instrucción que está en la etapa *memory* esté realizando un acceso con más de un ciclo de espera, con lo cual estarán congeladas las etapas “*fetch*”, “*decode*” y “*execute*”
- Será de vital importancia el tener en cuenta el estado de *mcontrol* en el control de etapas, cuando se produzcan saltos o interrupciones IRQ y FIQ.

Pues para poder realizar una circuitería que tenga en cuenta todos estos factores, basta con un único biestable, *mcontrol*.

En la etapa *decode*, cuando se detecta una instrucción de larga latencia, se mira el valor de *mcontrol*, y

- Si *mcontrol*='0'
 - si las señales *reset* y *stop* están desactivadas, se procede a la ejecución de la instrucción. En el próximo ciclo, *mcontrol*=1
 - En caso contrario, se espera a que se desactiven
- Si *mcontrol*='1'
 - estamos en el segundo ciclo de la ejecución de una instrucción de larga latencia.

Además, el valor de este biestable se incluye en las configuraciones generadas por las instrucciones SWP y multiplicación de 64 bits, de la siguiente forma:

Instrucciones de Multiplicación

27	24	23	22	21	20	16	15	12	11	8	7	0
0001	L	mctrl	S	alu_op	RdLo	RdHi						

- mctrl: solo tiene sentido en multiplicaciones de 64 bits, que requieren de dos ciclos de reloj para su ejecución. Si está a '0', indica que estamos en el ciclo primero, y si está a '1', indicará que se trata del ciclo segundo (es decir, que hay que sacar los 32 bits más significativos del resultado, en lugar de los 32 menos significativos; y además, que hay que cargar en el bus C los 32 bits más significativos del operando de acarreo, y no los 32 menos significativos)

SWP

27	24	23	22	21	20	16		15	12		11	8		7	6	5	4	3	2	1	0
1011		00				alu_op		Rd						B		L				0	A

- L = **not** (mcontrol); de este modo, si está activo, significa que se trata de una instrucción *load* (que sería el primer ciclo de un swap entre registro y memoria), y si está inactivo, se trata de una instrucción *store* (que precisamente sería el segundo ciclo de un swap entre registro y memoria).

A partir de la etapa *execute* (incluido ella) no hay que preocuparse más por este tipo de instrucciones, pues basta con que cada etapa siga adaptando su funcionamiento a las configuraciones que recibe, y que ya serán correctas.

Esta última afirmación no es del todo correcta, pues la etapa 'write' debe comprobar el valor de 'mcontrol' para decidir qué valor almacenar en R14_abt cuando se produce una interrupción, pues si no se toman precauciones, el desdoblarse una instrucción podría almacenarse un valor erróneo provocado por el hecho de que una misma instrucción está ocupando dos etapas (al calcular la dirección de retorno, se mira el PC actual, y suponiendo que existe una instrucción por etapa, se calcula la dirección de la instrucción que causó el ABORT).

Contador para la generación de estados de espera

El funcionamiento a grandes rasgos del sistema para la generación de estados de espera al acceder a memoria es bien sencillo:

- Existe una señal externa, denominada *wait_state*, que en realidad son dos bits. Por defecto, deberá configurarse como “00” (**es muy importante darle algún valor, o de lo contrario el CORE no funcionaría**).

ws[1:0]	Nº estados de espera	Ciclo de Acceso
“00”	0	El acceso dura 1 ciclo de reloj
“01”	1	El acceso dura 2 ciclos de reloj
“10”	2	El acceso dura 3 ciclos de reloj
“11”	3	El acceso dura 4 ciclos de reloj

- Si la señal anterior está configurada a un valor distinto de “00”, entonces cada vez que llega una configuración de acceso a memoria a la etapa *memory*, un contador comienza a contar, desde 0 hasta el valor codificado en la señal anterior (1 ó 2), **y a la vez se activa una señal denominada *mem_inst***. Esta señal estará activa hasta que el contador alcance el valor programado por la señal externa *wait_state*.
- La señal *mem_inst* se conecta directamente a la máquina global de estados (aunque en el siguiente párrafo se explica con detalle, recordemos que en la figura 5-12 quedó un bloque funcional vacío que fue incorporado como previsión de futuro para la generación de estados de espera adicionales).

Modificaciones en la máquina global de control originaria

Precisamente, en la figura 5-12 podemos ver que existía un bloque que se dejó “en el aire”, y que se denominó *previsión de futuro para la generación de estados adicionales de espera*.

Llegados a este punto del diseño, sería conveniente modificar ahora la máquina añadiéndole a ese bloque la funcionalidad necesaria para poder atender todas las señales generadas por los mecanismos anteriormente expuestos, y generar patrones de comportamiento en función de las distintas situaciones (combinaciones de señales activas) que se puedan presentar, de modo que no se genere ni un ciclo más ni tampoco un ciclo menos de los que sean necesarios para resolver la situación en concreto.

Pueden darse situaciones que requieran de congelar la estructura en dos etapas de la *pipeline* a la vez, y puede ocurrir que cada una de ellas requiera de un número de ciclos de reloj adicionales distinto.

Supongamos que tenemos configurada la señal externa `size="10"`, de modo que el acceso a memoria debe durar 3 ciclos de reloj, es decir, se requieren dos ciclos adicionales al que duraría la ejecución de la instrucción en la etapa *memory*.

Imaginemos que al mismo tiempo existe en *decode* una instrucción que requiere un operando que es el registro destino de la instrucción que se encuentra en la etapa *write*. Estaremos entonces ante un fenómeno de *interlocking*, que en este caso concreto requiere de un ciclo adicional de reloj para solucionar el problema.

Los mecanismos que resuelven cada una de las situaciones anteriores son inicialmente distintos, pues los ciclos adicionales para la resolución de *interlocking* los genera la propia máquina global, mientras que inicialmente, la cuenta de los ciclos extras para acceso a memoria la lleva el contador introducido en la propia etapa *memory*.

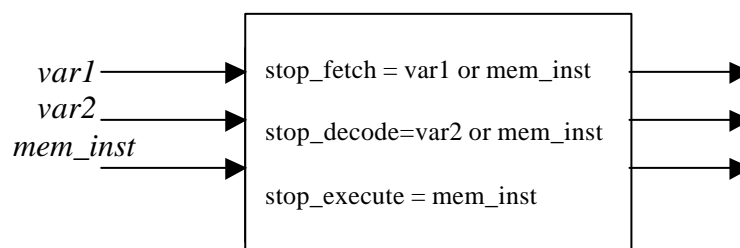
Sería conveniente que el control de la “congelación” de la *pipeline* resida exclusivamente en la máquina de control global, y que el resto de los mecanismos le solicitasen la generación de estados de espera, de forma que sea una única entidad funcional la que lleve la cuenta.

De este modo, en la situación anteriormente descrita, debería generar únicamente dos ciclos de reloj, pues son suficientes para resolver ambas situaciones (uno sería insuficiente para el acceso a memoria, y tres serían excesivos).

El problema planteado es más complejo que la solución en sí, que una vez resulta ser bastante sencilla:

- Las señales *stop_fetch* y *stop_decode* que generaba la máquina de la figura 5-12, se llamarán ahora *var1* y *var2*, y no atacarán directamente la salida, sino que se conectarán al nuevo bloque.
- Además, la señal *mem_inst*, procedente del contador de generación de estados de espera también será entrada del nuevo bloque de la máquina de control.
- Las señales *stop_fetch*, *stop_decode* y ahora también *stop_execute* (para accesos de más de un ciclo de reloj también será necesario detener la etapa *execute*, pues la siguiente etapa estará ocupada durante más de un ciclo de reloj con la misma instrucción) deberán generarse como combinación de todas las anteriores, y teniendo en cuenta todos los casos.

El nuevo bloque sería algo tan sencillo como dos puertas OR:



Así esta nueva unidad funcional coordina el funcionamiento de los dos mecanismos existentes para la generación de estados de espera, de modo que cada uno de ellos funciona de forma independiente, y las señales *stop_etapa* estará activa si alguno de los mecanismos está solicitando estados de espera (o ambos al mismo tiempo)

Es estos momentos estamos ya en condiciones de hacer un resumen sobre el funcionamiento global de nuestro CORE a la hora de resolver situaciones que no se corresponden con el flujo de funcionamiento normal. Estas situaciones son:

- Interlocking: una instrucción en *decode* necesita cargar en un bus un operando contenido en un registro que a su vez es el registro destino de una instrucción que está en *execute*, *memory* o *write*, y que por tanto aún no ha sido actualizado. Si se utiliza en valor actual del registro, estaremos cometiendo un grave error.
- Salto: las instrucciones de salto no se ejecutan de forma efectiva hasta la etapa *execute*, de modo que en el instante en el que el PC se actualiza al valor calculado por la ALU a partir de la instrucción de salto, existen dos instrucciones, una en *fetch* y otra en *decode* que son las que secuencialmente seguían a la instrucción del salto en la memoria de programa, pero que no son las que deberían estar si seguimos el flujo del programa, pues se ha realizado un salto. Si esas instrucciones alcanzan la etapa *execute* (y por supuesto si alcanzan una etapa posterior), realizarán modificaciones en ciertos registros de control que serán irreversibles, y por tanto, a partir de ese momento nada funcionará correctamente.
- A la hora de diseñar este CORE, se decidió poder utilizarlo con memorias que requieran de más de un ciclo de reloj. Pero como el acceso a memoria se produce en *memory*, conseguir que dure más de un ciclo es no trivial, pues tenemos detrás tres etapas que de alguna manera hay que “congelar”, y con ciertas garantías.
- Existen instrucciones que requieren de dos ciclos de latencia

Toda la circuitería que se ha descrito a lo largo de este apartado, y de algunos anteriores tiene como objetivo poder resolver los problemas anteriores. Para ello:

- En el caso del fenómeno de *interlocking*, las etapas *execute*, *memory* y *write* comunican a la etapa *decode* si están cursando una instrucción que deba escribir un registro, y en caso afirmativo, también le informan del registro en cuestión. Si el decodificador de los buses detecta coincidencia entre el registro a cargar y un registro destino, hace una petición de ciclos de espera a la máquina global de control
- En el caso de los saltos, **es la propia etapa *execute*** la que activa las señales *reset_fetch* y *reset_decode* que eliminan las instrucciones erróneamente cargadas en las respectivas etapas, y las sustituyen por *bubbles*.
- Para conseguir ciclos adicionales en el acceso a memoria, el mecanismo ya sido expuesto a lo largo de este apartado
- Las instrucciones de latencia elevada se resuelven con el biestable *mcontrol* y su lógica adicional, explicados al principio de este apartado

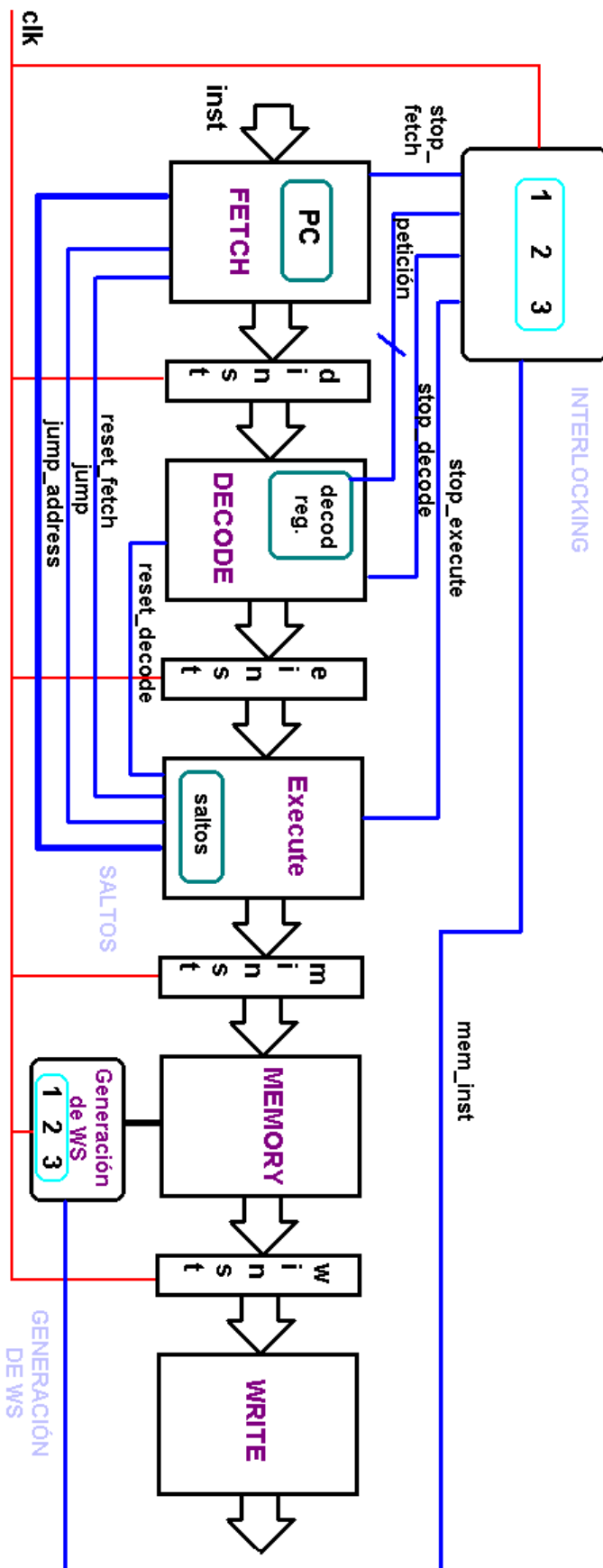


Figura 5-22: Circuitería de “vaciado” y “congelación” de la *pipeline*

5.7 Interfaz de memoria

El objetivo de este proyecto es realizar la implementación en VHDL del CORE del procesador ARM9 utilizando la arquitectura ARMv4. Por tanto, con el término interfaz de memoria nos referimos a la “interfaz del CORE”

En principio, parece algo absurdo pensar que un CORE tenga interfaz, pues cuando se utilice como parte de un diseño de jerarquía superior, será necesario crear una interfaz de memoria acorde al sistema de memoria que se vaya a utilizar.

Pero si hubiésemos omitido la creación de esta interfaz, la labor de diseño de la interfaz definitiva sería un trabajo altamente complejo, y en cambio, gracias al presente capítulo se convertirá en una tarea sencilla, pues vamos a incluir en él:

- Una descripción detallada de la interfaz de memoria del CORE
- Un ejemplo de creación de interfaz definitiva, para un sistema de memoria concreto.



Figura 5-23: Interfaz de Memoria del CORE del ARM9

Las señales a la izquierda de la figura 5-23 son señales internas del CORE, y carecen por tanto de interés para el futuro diseñador de la interfaz personalizada. En cambio, la comprensión de las señales a la derecha de la figura es esencial para la utilización del CORE.

- ❑ **BIGEND:** la interfaz que incluye el CORE **es capaz de realizar la conversión big-endian => little endian**, de manera que cuando el sistema de memoria utilice numeración big-endian, basta con poner esta entrada a “1”, y no hay que preocuparse más por el formato.

- ❑ VAddress: bus de direcciones de 32 bits para acceso a memoria de datos, o periféricos mapeados en memoria.
- ❑ WData: bus de 32 bits que contiene el dato a escribir en la dirección de memoria dada por VAddress.
- ❑ RData: bus de 32 bits de entrada que contiene el dato leído de la dirección de memoria especificada por VAddress.
- ❑ r_enable: se activa cuando el acceso a memoria es de lectura
- ❑ w_enable: es activa cuando el acceso a memoria es de escritura
- ❑ size: bus de 2 bits que codifica el tamaño del dato a leer/escribir

size[1:0]	Tamaño
"00"	No hay acceso a memoria
"01"	Byte
"10"	Halfword
"11"	Word

- ❑ wait_states: codifica el número de ciclos extra, con los que queremos que se realicen los accesos a memoria, y **es muy importante asignarle un valor**, aunque sea "00", pues de lo contrario el funcionamiento de la *pipeline* sería erróneo.

ws[1:0]	Nº estados de espera	Ciclo de Acceso
"00"	0	El acceso dura 1 ciclo de reloj
"01"	1	El acceso dura 2 ciclos de reloj
"10"	2	El acceso dura 3 ciclos de reloj
"11"	3	El acceso dura 4 ciclos de reloj

Otras funciones que realiza la interfaz de memoria

Además de la conversión big-endian => little-endian, la interfaz realiza una alineación de la dirección de memoria, de modo que:

- ❑ En Lectura: la dirección siempre está alineada con un Word, o lo que es lo mismo, siempre es múltiplo de cuatro (los dos bits menos significativos de VAddress se fuerzan a "00")
- ❑ En Escritura: dependá de *size* el que VAddress esté alineada con un *Word*, con un *Halfword* (únicamente el bit menos significativo de VAddress se fuerza a '0'), o que se mantenga intacta en el caso de un *Byte*.

5.7.1 Creando una interfaz personalizada

Para abordar la realización de una interfaz personalizada, basta con analizar las características del sistema de memoria que se quiere implantar, y compararlas con las especificaciones de la interfaz de memoria incluida en el CORE, y que se detallaban en la sección anterior.

Así, tras un detallado análisis, nos encontraremos en situación de poder generar lo que podríamos denominar *unidad de adaptación entre interfaces de memoria*.

Ejemplo de Uso

Imaginemos que disponemos de cuatro bancos de memoria DRAM, cada uno de los cuales requiere para ser accedido al menos dos ciclos de reloj (ver apartado 1.3). Además, resulta que cada banco tiene 8 bits (1 byte) de ancho, y la numeración de la memoria tiene formato little-endian.

Para poder utilizar esa memoria con nuestro CORE, debemos crear un bloque que realice las siguientes acciones:

- ❑ asignar a la señal externa ws[1:0] el valor "01"
- ❑ generar 4 señales de selección de chip (chip select) denominadas CS0, CS1, CS2, CS3, a partir de las señales r_enable, w_enable y size[1:0], de manera que cuando se acceda en lectura, siempre se activaran los cuatro bancos de memoria para leer en paralelo un Word completo (1 byte de cada banco). En cambio, si el acceso es en escritura, se puede activar 1 byte cualquiera, en función de los dos bits menos significativos de VAddress y de size[1:0], tal y como se ilustra en la siguiente tabla:

VAddress	Lectura (r_enable=1)	Escritura (w_enable=1)		
		byte	halfword	word
00	CS0, CS1, CS2, CS3	CS0	CS0, CS1	CS0, CS1, CS2, CS3
10	CS0, CS1, CS2, CS3	CS2	CS0, CS1	CS0, CS1, CS2, CS3
01	CS0, CS1, CS2, CS3	CS1	CS2, CS3	CS0, CS1, CS2, CS3
11	CS0, CS1, CS2, CS3	CS3	CS2, CS3	CS0, CS1, CS2, CS3

- ❑ conectar la señal BI GEND a tierra
- ❑ cablear todos los buses con todos los bancos de memoria, y conectar cada señal CS0....CS3 con su banco de memoria correspondiente, tal como se muestra en la figura 5-24

Nota: el código VHDL de este ejemplo está en un fichero llamado **memperso.vhd**, y se incluye en el CD-ROM, en el directorio correspondiente a los códigos VHDL

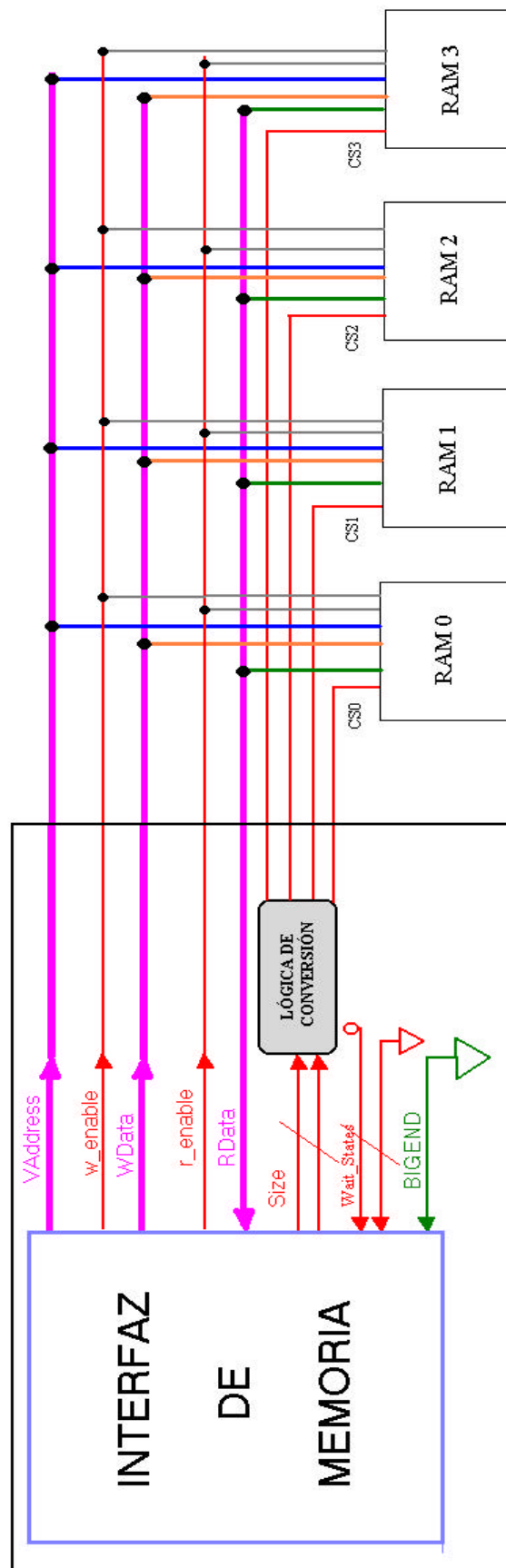


Figura 5-24: Interfaz Personalizada para el CORE del ARM9

5.7.2 Accediendo a la memoria de programa

Nótese que la arquitectura de nuestro CORE es completamente Harvard, por lo que a priori parece imposible acceder a la memoria de programa para leer una instrucción que no sea la que se ha de cargar en el CORE por el bus de instrucciones.

Efectivamente, podemos acceder a la memoria de programa mediante los buses de instrucciones, pero nosotros queremos además poder acceder a ésta mediante los buses de datos. El motivo de ello es la existencia de ciertas operaciones software que requieren acceso al programa mediante las instrucciones LDR, LDRH, ... Por ejemplo,

- ❑ cuando se produce un SWI, el software sabe qué instrucción ha sido la que ha producido la excepción mediante la lectura de R14 (recordemos que ahí se guardaba el valor del PC cuando se produce el trap). Sería interesante para la rutina de atención a la interrupción el poder acceder a la memoria de programa con una instrucción LDR, para poder leer así el *SWI number*
- ❑ más vital sería el acceso cuando se trata de emulación software de una instrucción no definida. Cuando se produce el trap, sería conveniente acceder a memoria de programa para leer el contenido de los campos que cuentan con los operandos, los parámetros de la instrucción a emular, El mecanismo debería ser igual que en el caso anterior, es decir, mediante R14 podemos calcular la dirección de acceso, y nos gustaría poder acceder a memoria con un LDR, como si de una memoria de datos se tratase
- ❑ el caso de las instrucciones de coprocesador cuando éste no se encuentra físicamente en el sistema, es análogo al anterior.

Para resolver la situación expuesta, basta con que el sistema de memoria del diseño en el que se incluya el CORE entregue la instrucción al bus de datos.

Normalmente, independientemente de que el CORE tenga arquitectura Harvard, datos e instrucciones utilizarán el mismo espacio de memoria, de modo que es el sistema de memoria creado por el usuario de nuestro diseño el que deberá encargarse de cargar los valores leídos en memoria en el bus correspondiente.

Al igual que existe un rango de memoria de datos asignado para la RAM y otro para los periféricos mapeados en memoria, también se puede asignar un rango de direcciones a la memoria de programa. Cuando se intente realizar un acceso a memoria a través del bus de datos, pero utilizando una dirección de las asignadas a la memoria de programa, el sistema de memoria accederá a ésta e introducirá el dato leído (en este caso se tratará de una instrucción) en el bus de lectura de datos. Así la memoria de instrucciones será accesible mediante los buses VAddress y Rdata (sería conveniente que WData no se utilizase, ya que por seguridad el código no debería poder alterar el contenido del programa).

5.8 Introducción del control de excepciones

Una vez llegados a este punto del diseño, ya contamos con un CORE completamente funcional, que es capaz de ejecutar todas las instrucciones de la arquitectura ARMv4, y que se puede comunicar con el exterior mediante una interfaz de memoria (que permite la inclusión de un sistema de memoria RAM, y el uso de periféricos mapeados en memoria).

Pero para que los dispositivos externos funcionen correctamente, necesitamos añadir al diseño la posibilidad de poder interrumpir el funcionamiento normal del procesador mediante algunas señales externas cuando se produzcan ciertos eventos. Para ello, basta con introducir un control de excepciones que implemente todas las características expuestas en la sección 3.4 del capítulo tercero.

Además es necesario que el CORE cuente con tres pines externos, que serán activos a nivel bajo, y que se denomina FIQ, IRQ, ABORT. Su significado también ha sido expuesto a lo largo de los capítulos tercero y cuarto, y no vamos a detenernos en ello.

El diseño del control de excepciones debe tener en cuenta que cuando se produzcan múltiples interrupciones simultáneas, deberá existir un sistema puramente combinatorial, que se encargará de determinar el orden de atención en función de la siguiente asignación de prioridades:

1. Reset (prioridad más alta)
2. ABORT
3. FIQ
4. IRQ
5. Undefined, SWI

No todas las interrupciones se pueden dar al mismo tiempo: por ejemplo, SWI y una instrucción no reconocida es evidente que son mutuamente exclusivas.

Cuando se produce una excepción, y tras salvar el PC en *R14_<adecuado>*, el contador de programa se actualiza al valor del vector de la excepción correspondiente. Para ello se utilizan los mecanismos *save_link*, *jump* y *jump_address*, con los que ya cuenta la estructura *pipeline*, lo cual, simplifica enormemente el control de excepciones.

Dirección	Excepción	Modo de Entrada
0x00000000	Reset	Supervisor
0x00000004	Instrucción no reconocida	Undefined
0x00000008	SWI	Supervisor
0x0000000C	(**)	----
0x00000010	ABORT	Abort
0x00000014	-- Reservado --	----
0x00000018	IRQ	IRQ
0x0000001C	FIQ	FIQ

(**) No disponible en esta implementación del CORE del ARM8/9

Rediseñando la máquina global de control

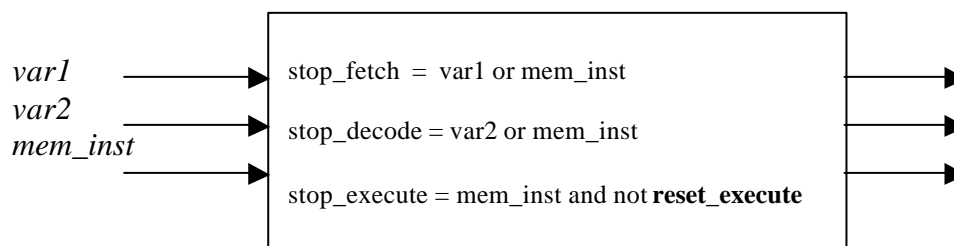
Tras añadir el control de excepciones, descubrimos que una interrupción FIQ, IRQ o ABORT en ciertas situaciones originaría la pérdida de ciertas instrucciones, o la ejecución errónea de otras.

- Instrucciones de larga latencia: tal y como está concebida la máquina global de control, una interrupción FIQ o IRQ generada cuando una instrucción de larga latencia está desdoblada, ocasionaría la pérdida del segundo ciclo de ejecución de dicha instrucción, debido a que la señal `reset_decode` vaciaría el contenido de la etapa *decode* cuando ésta contiene parte de la instrucción.
- Acceso a memoria de duración superior a un ciclo de reloj: cuando la etapa *execute* está deshabilitada por medio de la señal `stop_execute`, debe desactivarse la señal `stop_decode`, pues de lo contrario se estaría introduciendo *bubbles* en la etapa *execute*, perdiéndose así la instrucción que estaba en espera de que se completase el acceso a memoria.
- ABORT: cuando se produce un ABORT, hay que asegurar que la instrucción que está en *execute* no modifique ningún registro de estado, pues el fallo de memoria es el único tipo de excepción que requiere del vaciado de la etapa *execute*. Además, si `stop_execute` está activo en el instante que se produce el ABORT; hay que desactivarlo, o no podrá vaciarse la etapa *execute*.

Todas las anteriores situaciones, unidas además al *interlocking* y los saltos, requieren de una modificación en el diseño de la máquina global de control y en las etapas, de modo que el comportamiento se adapte al descrito en la siguiente tabla:

Situación	stop_fetch	stop_decode	stop_execute	reset_fetch reset_decode	reset_execute
Interlocking	1	1	0	0	0
Interlocking + ciclos de espera	1	1	1	0	0
Interlocking con mcontrol=1	1	1 (**)	0	0	0
ABORT	0	0	0	1	1
Salto con mcontrol='0'	0	0	0	1	0
Salto con mcontrol='1'	0	0	0	1,0	0
Salto con stop_execute='1'	1	1	1	1	0
ABORT con ciclos de espera	1	1	0	1	1

(**) La etapa *decode* debe asegurar que si `mcontrol='1'`, no se generan *bubbles* ni se vacía *decode*.



Pero la solución no es tan sencilla: si analizamos el comportamiento de la instrucción SWP, nos damos cuenta que es una instrucción que genera dos configuraciones que ocupan simultáneamente etapas distintas. Esto puede provocar diversas situaciones críticas, tales como:

- Si ocupa simultáneamente *decode* y *execute*, y se produce una interrupción FIQ o IRQ, habrá que asegurarse de que no se vacía *decode*, pues una configuración (correspondiente al primer ciclo de latencia) ya se ha ejecutado. Además, el valor del (PC) a almacenar en R14_fiq ó R14_irq es distinto al que normalmente se almacenaría, pues el contador de programa contiene un valor 4 bytes menor que el que se esperaría en condiciones normales. Todo esto ya ha sido descrito en la tabla anterior, y además también le ocurre a la instrucción de multiplicación de 64 bits.
- Pero además, SWP puede ocupar simultáneamente *memory* y *write*. Imaginemos que el primer ciclo del *swap* se realizó correctamente (correspondería a un *load*). Si en el segundo ciclo (*write*) se produce un ABORT, no debería alterarse el valor de ningún registro.
- Más allá aún, si estamos accediendo a memoria con varios ciclos de espera, el fallo de memoria podría producirse en cualquier ciclo, con lo que la etapa de *write* no debe escribir ningún registro hasta el último ciclo de acceso a memoria.

Todas las situaciones anteriores implican unas variaciones importantes en el control global y en las propias etapas, pero es la última situación descrita la que requiere de una reformulación de la etapa *write* prácticamente completa.

Si bien, el diseño es tan robusto que no se requiere modificar las etapas, cuando hablamos de cambios profundos nos referimos a una alteración del comportamiento (que se traduce en una variación de la configuración de la etapa), no del diseño hardware que sigue siendo válido.

La modificación introducida en la etapa *write* consiste en evitar que ésta escriba ningún registro hasta que se esté llevando a cabo el último ciclo de espera en un acceso a memoria (únicamente en el caso que se requieran ciclos adicionales de espera, claro está).

De este modo, cuando una instrucción SWP ocupe *memory* y *write* podremos garantizar que ningún registro será alterado a no ser que se lleve a cabo el *swap*.

Lo que estamos garantizando con este robusto mecanismo hardware, es la *atomicidad* de la instrucción SWP, siendo así fieles a lo expuesto en la arquitectura ARMv4.

5.9 Vista del *pin-out* final del CORE

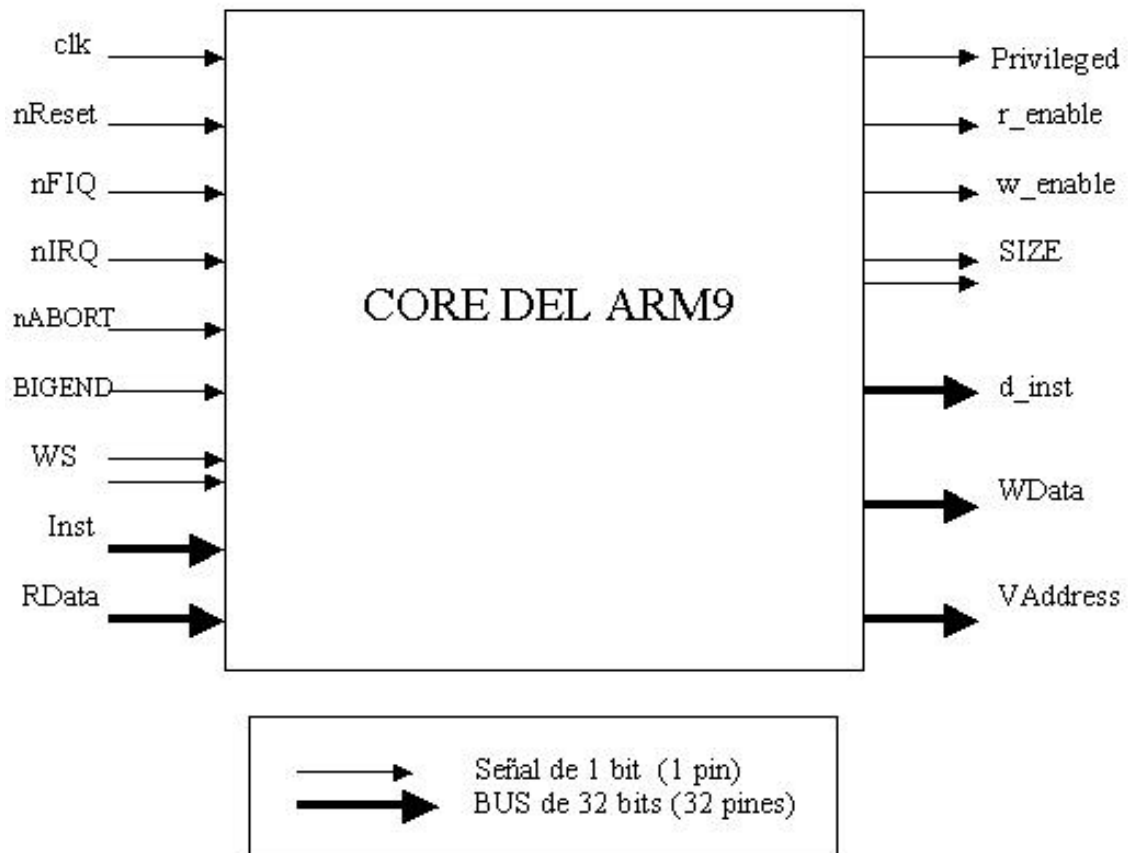


Figura 5-25: vista del *pin-out* del CORE del ARM9

SEÑAL	PINES	E/S	DESCRIPCIÓN
clk	clk	E	Reloj Global del Sistema
nReset	nReset	E	Reset Global (activo a nivel bajo)
nFIQ	nFIQ	E	Petición de Interrupción Rápida (idem)
nIRQ	nIRQ	E	Petición de Interrupción (activa a nivel bajo)
nABORT	nABORT	E	Fallo de Memoria (activa a nivel bajo)
BIGEND	BIGEND	E	0: little-endian - 1: big-endian
WS	ws0, ws1	E	Petición para la generación de estados de espera
INST	inst0....inst31	E	BUS de Instrucciones
Rdata	Rdata0....RData31	E	BUS de lectura de datos de memoria
r_enable	r_enable	S	Acceso a memoria en lectura
w_enable	w_enable	S	Acceso a memoria en escritura
size	size0, size1	S	Tamaño del dato de memoria
d_inst	d_inst0...d_inst31	S	BUS de direcciones de instrucciones
Wdata	Wdata0...Wdata31	S	BUS de escritura de datos en memoria
Vaddress	Vaddress0..Vaddress31	S	BUS de direcciones de memoria de datos
Privileged	Privileged	S	Acceso a memoria en modo (1:privilegiado; 0:user)

5.10 Layout de una mega-celda del microprocesador

El objetivo de este proyecto era la generación del CORE del microprocesador ARM8/9, para la posterior obtención de una celda de librería para diseño de ASICs.

El diseño ha sido sintetizado a puertas standard-cell utilizando *synopsys*, y el layout producido para las celdas de AMS de 0.35u se ha realizado utilizando *silicon ensemble*.

El resultado obtenido ocupa un área total de 2.4x2.4mm, y puede alcanzar una frecuencia máxima de 33 MHz (optimizado por velocidad). Esta frecuencia es pobre debido a que existe un retraso máximo crítico, mucho más alto que cualquiera de los demás, y que sería posible reducir considerablemente si se realizase un estudio detallado del origen del mismo; pero esto queda fuera del alcance de este proyecto.

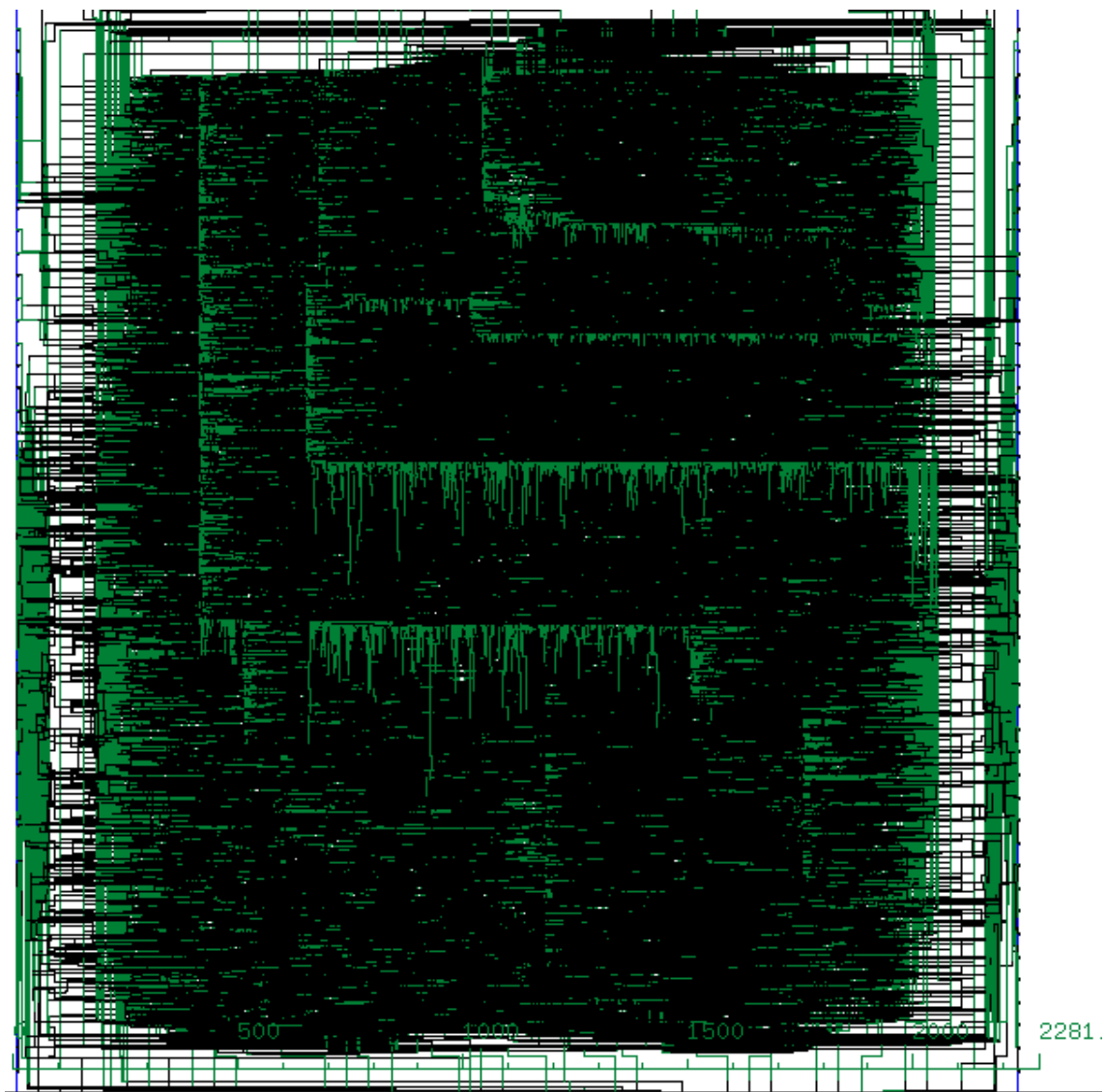


Figura 5-26: layout de la mega-celda

5.10.1 Optimización del diseño

La frecuencia máxima de funcionamiento del *layout* anterior está limitada por el máximo retardo combinacional.

Tras un detallado análisis del diseño resultante, se descubrió que el máximo retardo lo proporciona el shifter en serie con el sumador completo. Además es un retardo muchísimo mayor que el segundo mayor retardo, así que una reducción de un 50% de ese retardo, por ejemplo, ocasionaría un aumento de la frecuencia máxima de funcionamiento del doble (200%).

Una solución propuesta sería introducir el shifter en la etapa *decode*, y la ALU permanecería en *execute*, de modo que el operando B escrito en el bus B estaría ya desplazado.

El único problema sería el *carry* de entrada. El *shifter* utiliza el valor actual del *carry* almacenado en el registro CPSR para realizar ciertos desplazamientos. El caso es que si el desplazamiento se realiza en *decode*, el CPSR no tendrá aún el bit C actualizado al valor correcto.

El *shifter* utiliza el bit C del CPSR para utilizarlo como bit de signo, o bit menos significativo del operando desplazado, en ciertos casos excepcionales.

De este modo, el problema se podría solucionar simplemente realizando la siguiente división de funcionalidades entre *shifter* en *decode* y ALU en *execute*:

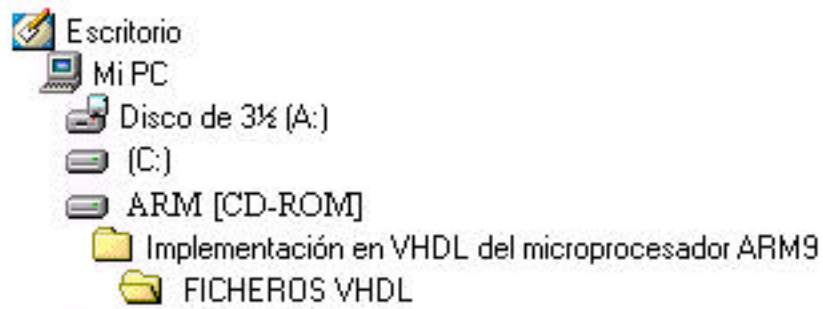
- El *shifter* realiza el desplazamiento en *decode*, y si necesitase incluir el bit C en un desplazamiento, utilizará cualquier valor (es indiferente, pues en la siguiente etapa se sustituirá por el valor correcto)
- Además generará el *carry* de salida del *shifter*
- La ALU, aunque no realizará desplazamientos, sí que utilizará las señales de control del *shifter* para tener conocimiento del tipo de desplazamiento que realizó, y así sustituir el valor introducido como bit C del CPSR por el valor correcto (ahora ya actualizado), de ser necesario
- Además la ALU implementará el resto de funcionalidades que incluía anteriormente.

La etapa *decode* tendrá que sufrir pequeñas variaciones para poder cargar en los buses del nuevo *shifter* los valores correctos, y para que el decodificador de registros utilice el valor desplazado como operando, en lugar del contenido del registro o del valor inmediato proporcionado por la instrucción.

Versión 2

Aunque queda fuera del alcance del proyecto, finalmente se optó por realizar un nuevo diseño que respondiese a las especificaciones anteriores, con motivo de obtener una mejora en la frecuencia máxima de reloj de la macrocelda.

Los ficheros VHDL referentes a esta nueva implementación se encuentran ubicados en el subdirectorio “**VERSION 2**”, en la misma ruta que el resto de ficheros VHDL.



Con la nueva versión se alcanza un incremento de aproximadamente el 10% en la frecuencia máxima de funcionamiento con respecto a la versión original.

5.11 Observaciones

5.11.1 Unidad de Predicción de Saltos

La unidad de predicción de saltos no entra dentro del alcance del presente proyecto, y además es algo que es externo a un CORE. En la figura 2-1 podemos observar la relación funcional entre la unidad de predicción de saltos y el CORE. Nótese que la predicción de saltos sólo tiene sentido en sistemas que utilicen lo que se denomina *prefetch buffer*, que no es más que una FIFO de instrucciones.

La justificación del presente capítulo es dejar sentadas las bases para poder en un futuro incorporar al diseño una unidad de predicción de saltos, que iría en beneficio del parámetro *speedup*

Aproximadamente una de cada diez instrucciones es un salto, y cada salto supone 3 ciclos de tiempo real, pues hay que vaciar la pipeline e introducir algunos *bubbles* para establecer el correcto flujo del programa.

Si cuando se ejecuta un salto, alguien ha introducido ya las instrucciones correctas en fetch y en decode, nos ahorraríamos todo este proceso, con la consiguiente mejora del *speedup*.

Para ello, hay que detectar un salto antes que se introduzca en el CORE. Es importante tener en cuenta los siguientes principios:

- ❑ se puede basar la detección en los patrones de bits particulares de las instrucciones de salto
- ❑ las instrucciones del ARM son condicionalmente ejecutadas, así que pueden existir incorrectas predicciones, lo que implicaría que instrucciones erróneas serían introducidas en el CORE. Sería conveniente poder cancelar las instrucciones incorrectamente introducidas en el CORE externamente, para poder implementar mecanismos de corrección (bastaría con hacer accesibles al exterior las señales *reset_fetch* y *reset_decode*).
- ❑ Nunca predecir un salto con registro de enlace (BL), si su ejecución es condicional, porque un error en la predicción de este tipo de saltos, sería incorregible.
- ❑ Luego **NO HE DE PREDECIR**
 - Si Instruction[27:24]!="1011" and Instruction [31:28]!="1110 (BL)
 - Si la señal externa de predicción está desactivada
 - Si Instruction[31:28]!="1111 (código de condición inválido)
 - Si Instruction[27:25]!="101" (no es una instrucción de salto)
- ❑ Y **HE DE PREDECIR**
 - Si Instruction[31:28]!="1110" (la condición se cumple siempre)
 - Si Instruction[24]='0' and Instruction[23]='1' (salto adelante)
 - Si Instruction[24]='0' and Instruction[23]='0' (salto hacia atrás)
- ❑ Estos dos últimos casos habría que analizarlos detenidamente.
- ❑ Habría que añadir a la implementación del CORE la instrucción IMB

6

Simulaciones

Esta es quizá la sección más importante de este proyecto como tal, pues los tests realizados al CORE generado han consumido la mayor parte del tiempo dedicado a la realización del mismo, y han sido de vital importancia en la depuración del diseño.

- 6.1 Introducción a la fase de simulaciones
- 6.2 Simulando la ALU
- 6.3 Usando VHDL Simili
 - 6.3.1 Generando las entradas: aplicación “Txt2vhdl”
 - 6.3.2 Ficheros de salida
- 6.4 Emulando el multiplicador
- 6.5 Simulando la *pipeline*
 - 6.5.1 Verificando la FASE I
 - 6.5.2 Verificando la FASE II
- 6.6 Comprobando el control de interrupciones
- 6.7 Ejecutando un programa completo
- 6.8 Simulación del Prototipo de pruebas
- 6.9 Ubicación y descripción de los archivos de test

6.1 Introducción a la fase de simulaciones

El mayor problema que presenta simular un diseño como el presente, es su complejidad. Complejidad residente, entre otros factores, en la dificultad para presentar datos que el diseñador pueda interpretar de forma más o menos sencilla, y que además le puedan facilitar la depuración del diseño.

El simulador que el paquete Xilinx Foundation incluye está orientado a cronogramas, luego no resulta adecuado para evaluar nuestro diseño.

El único servicio que aportó al proyecto, fue la simulación de la primera versión de la ALU, pues no requería de ningún tiempo determinado para realizar un proceso, y el simulador no resultaba demasiado engorroso de manejar a la hora de configurar los correspondientes tests.

En cuanto la primera versión simplificada de la *pipeline* estuvo terminada, el simulador de Xilinx resultó bastante inapropiado, y se optó por utilizar **VHDL Simili**, que presentaba una gran flexibilidad a la hora de introducir instrucciones al CORE, y una gran capacidad de personalización a la hora de mostrar las salidas (el usuario configura de forma manual y en su totalidad el formato de los ficheros de salida y de entrada, y simplemente añadiendo líneas de pseudo-código VHDL que a la hora de sintetizar e implementar el diseño, son ignoradas por completo)

Al final de este capítulo, en el apartado 6.9, se muestra una traza detallada de la ejecución de una secuencia de código que se adapta completamente a la estructura de un programa (incluye vectores de interrupción en las posiciones predeterminadas, etc).

En el resto de apartados se describe brevemente los objetivos y resultados de las simulaciones realizadas, intentando responder en cada uno de ellos a las siguientes preguntas:

- ¿Qué comportamiento se desea evaluar?
- ¿Qué fallos se pretende detectar?
- ¿Dónde se encuentran los ficheros referentes a esta simulación?

Nota: en este capítulo se muestran algunas simulaciones básicas, que se comentan con más o menos grado de detalle, pero que no componen en absoluto la totalidad de los test realizados, sino que son más bien un breve resumen. El resto de los test se enumeran en el último apartado, donde se muestra la ubicación de los mismos en el CD-ROM que acompaña este documento. Como se verá a continuación, estos ficheros son bastante complejos de entender por estar en código máquina, y sólo tienen sentido para las personas que han realizado las simulaciones. De todos modos, uno de los objetivos de este apartado es intentar ilustrar la filosofía de dichas simulaciones, e intentar demostrar que el espacio de errores de nuestro diseño se ha reducido a su máxima expresión.

6.2 Simulando la ALU

El simulador de Xilinx tiene una interfaz gráfica bastante intuitiva, pero carece de la potencia necesaria para realizar pruebas en un diseño tan complejo como el nuestro, debido a su diseño:

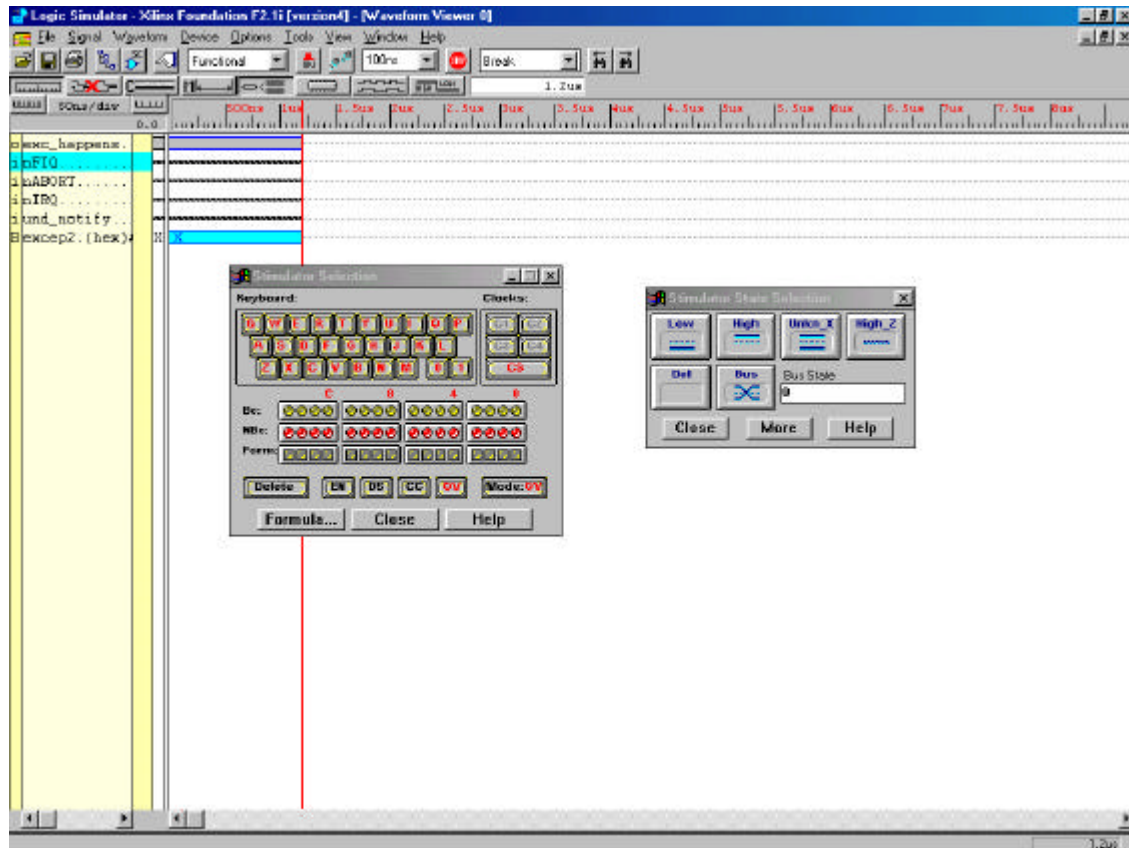


Figura 6-1: entorno de simulación de Xilinx Foundation

El hecho de que muestre gráficamente el contenido de las distintas líneas y el modo de operación para obtener valores resultan inconvenientes a la hora de simular un diseño como por ejemplo el de un microprocesador.

Pero existe un componente, la ALU, que tiene un formato de entradas/salidas y un modo de operación que se presta a ser simulado con esta herramienta. Las comprobaciones que se llevaron a cabo con la ALU fueron las siguientes:

- se comprobó el correcto funcionamiento de los desplazadores, haciendo especial hincapié en los casos excepcionales y límites.
- se realizaron sumas y restas con al menos un valor de cada subconjunto de operandos que soporta la ALU (por ejemplo, +534, +1, 0, -1, -234 y todas sus combinaciones como operando A y operando B serían buenos valores para realizar un test)
- se comprobaron tanto el carry como el *signed-overflow*
- se realizaron minuciosas comprobaciones de los multiplicadores, sobre todo de multiplicaciones con acarreo y con signo.

6.3 Usando VHDL Simili

Esta herramienta permite realizar una “compilación” del código VHDL, que a su vez es susceptible de una posterior ejecución.

¿Cómo trabajar con esta herramienta? Su funcionamiento es bastante sencillo:

- Existen una serie de sentencias en VHDL que permiten escribir el valor de ciertos buses en ficheros, pero que a la hora de sintetizar e implementar el código, son totalmente ignorados. Además existe una librería asociada a este tipo de expresiones.
- Basta con declarar una serie de vectores de test, que no son más que variables globales al diseño, y asignarles a éstas el valor de las señales que deseamos monitorizar
- Posteriormente se crea una entidad sin puertos de E/S, y en cuya arquitectura utilizamos estos comandos de escritura en disco para escribir ficheros a nuestro gusto, mostrando los vectores de test deseados, y con el formato que se nos ocurra:

Ejemplo
<pre>LIBRARY IEEE; USE IEEE.std_logic_1164.all; USE IEEE.std_logic_arith.all; USE STD.TEXTIO.ALL; USE IEEE.std_logic_textio.all; USE WORK.my_package.all;</pre>
<pre>ENTITY testbench IS END ENTITY;</pre>
<pre>ARCHITECTURE comp of testbench IS FILE fout: TEXT IS OUT "fout.dat"; -- declaración de señales, componentes, etc)</pre>
<pre>test_driver: PROCESS (clk) VARIABLE linea: LINE; VARIABLE str0:STRING(1 TO 18):= " esto es una prueba " BEGIN IF (clk='0' and clk'event) THEN write (linea,str0); write (linea, DEBUG0); WRITELINE (fout,linea); write (linea,str1); end if; end process; end COMP;</pre>

Este fichero será el de mayor nivel jerárquico, y tendrá como componentes al top del diseño a simular, y el *fichero.vhd* de entradas. Ambos se interconectarán por señales o variables. En el ejemplo, existe una variable global DEBUG0 declarada en el fichero *my_package.vhd* a la que en alguna parte del diseño se le ha asignado una señal a monitorizar. Supongamos que su valor, al simular es “10010111”. En el fichero de salida fout.dat aparecerá algo como:

esto es una prueba: 10010111

Tras esta primera introducción, vamos a pasar a explicar el procedimiento seguido a la hora de realizar simulaciones en este proyecto (si se desean más detalles de cómo realizar estas simulaciones, consultar en el último capítulo el código correspondiente al fichero “*test.vhd*”).

En nuestro caso, el fichero de entrada que se escribe en primera instancia es un fichero de texto que contiene --en hexadecimal-- todas las instrucciones que se desean ejecutar, una a continuación de la otra.

A este fichero se le aplica la herramienta “Txt2vhd”, que genera el código VHDL de una ROM que contiene las instrucciones anteriores.

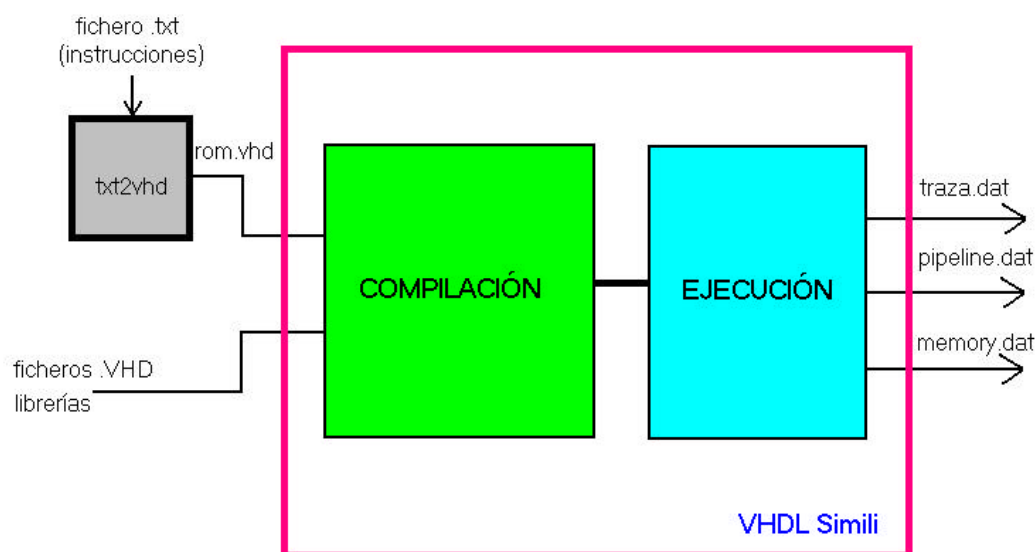


Figura 6-2: diagrama de bloques del proceso de simulación

NOTA: al final de este apartado se incluye el listado del código fuente en C de la aplicación *txt2vhd.exe*, creada específicamente para este proyecto.

Una vez que tenemos la ROM, hay que proceder a la compilación de todos los ficheros fuente, utilizando la herramienta VHDL-Simili, y a su posterior ejecución. Para ello, basta ejecutar los siguientes ficheros de proceso por lotes.

programa.bat		ejecuta.bat	
@echo off	VHDLP MEM_IF.VHD	@echo off	
cls	VHDLP CORE.VHD	cls	
VHDLP INSTRU~1.VHD	VHDLP ROM.VHD	VHDL TESTBENCH	
VHDLP NOMUL.VHD	VHDLP RAM.VHD	@echo on	
VHDLP ARM8_ALU.VHD	VHDLP -87 TEST.VHD		
VHDLP PIPELINE.VHD	@echo on		
VHDLP EXCEPTION.VHD			

Como resultado de todas estas operaciones, VHDL-Simili escribe en disco tres ficheros:

- traza.dat: contiene una traza de los valores de los 37 registros del ARM8/9 para cada ciclo de reloj.
- pipeline.dat: contiene una traza de los valores de los registros de control y configuración, y de los buses que componen la arquitectura *pipeline*, para cada ciclo de reloj
- memory.dat: contiene un registro de todos los accesos a memoria, y muestra el valor del (PC) para cada acceso.

El formato de estos archivos es el siguiente:

“traza.dat”							
----- NUEVO CICLO -----							
R0	00000000	R1	00000000	R2	00000000	R15 (PC)	00000000
R3	00000000	R4	00000000	R5	00000000		
R6	00000000	R7	00000000	R8	00000000	CPSR	000000D3
R9	00000000	R10	00000000	R11	00000000		
R12	00000000	R13	00000000	R14	00000000	SPSR_FIQ	00000000
R8FIQ	00000000	R9FIQ	00000000	R10FIQ	00000000	SPSR_IRQ	00000000
R11FIQ	00000000	R12FIQ	00000000	R13FIQ	00000000	SPSR_UND	00000000
R14FIQ	00000000	R13IRQ	00000000	R14IRQ	00000000	SPSR_ABT	00000000
R13SVC	00000000	R14SVC	00000000	R13ABT	00000000	SPSR_SVC	00000000
R14ABT	00000000	R13UND	00000000	R14UND	00000000		
busA	00000000	busB	00000000	busC	00000000	Result	00000000
----- NUEVO CICLO -----							

“pipeline.dat”							
----- NUEVO CICLO -----							
Inst	E3B00000	d_inst	F0000000	e_inst	0F000000	(PC)	00000000
m_inst	0F000000	w_inst	0F000000	m_data	00000000	w_data	00000000
m_wbReg	00000000	w_wbReg	00000000	emw-WBR	000	m_addr	00000000
eA	1	eB	1	eC	0		
busA	0	busB	0	busC	0		
op A	00000000	op B	00000000	op C	00000000	AUXBUS	00000
Result	00000000	(e/m/w)	111	execute	0		
JUMP	0	JUMP_ADDRESS	00000000	WS	0	M_CTRL	0
exc_happens	0	exc_report	0	EXCEPT:	000		
----- NUEVO CICLO -----							

“memory.dat”							
----- NUEVO CICLO -----							
VAddress	00000004	SIZE	11	(PC)	00000038		
WData	00000000	w_enable	0				
RData	23456789	r_enable	1				
----- NUEVO CICLO -----							
VAddress	00000004	SIZE	01	(PC)	00000040		
WData	00000000	w_enable	0				
RData	23456789	r_enable	1				
----- NUEVO CICLO -----							

En las ilustraciones anteriores se muestran lo que se podría definir como las celdas básicas de cada uno de los ficheros. Estas celdas se repiten para cada ciclo de reloj durante todo el desarrollo de las simulaciones, mostrando los distintos parámetros a monitorizar.

Nota: a continuación se detalla el código fuente de la aplicación “*txt2vhdl.exe*” que permite de forma sencilla generar ROMs escritas en VHDL con las instrucciones que se deseen ejecutar.

“*txt2vhdl.c*”

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
FILE *fptr;
FILE *fpo;
int main (int argc, char *argv[])
{
    int i = 0;
    int val;
    if (argc < 3) {
        fprintf(stderr, "Uso: txt2vhdl fichero_entada salida_vhdl\n");
        return 2;
    }
    fptr = fopen(argv[1], "r");
    fpo = fopen(argv[2], "w");
    fprintf (fpo, "\nLIBRARY IEEE;");
    fprintf (fpo, "\nUSE IEEE.STD_LOGIC_1164.ALL;");
    fprintf (fpo, "\nUSE IEEE.STD_LOGIC_ARITH.ALL;");
    fprintf (fpo, "\nUSE IEEE.STD_LOGIC_UNSIGNED.ALL;");
    fprintf (fpo, "\n\nENTITY arm_rom IS");
    fprintf (fpo, "\n    PORT    (");
    fprintf (fpo, "\n    Addr   : IN  STD_LOGIC_VECTOR(31 DOWNT0 0); ");
    fprintf (fpo, "\n    Data   : INOUT STD_LOGIC_VECTOR(31 downto 0)); ");
    fprintf (fpo, "\n\nEND arm_rom;");
    fprintf (fpo, "\n\n\nARCHITECTURE second OF arm_rom IS");
    fprintf (fpo, "\nBEGIN\n");
    fprintf (fpo, "\ns: process(Addr)\n");
    fprintf (fpo, "    variable vadd: integer;\n");
    fprintf (fpo, "\nBEGIN");
    fprintf (fpo, "\n    vadd := conv_integer(Addr(29 downto 0));\n");
    fprintf (fpo, "\n    CASE vadd IS\n");
    for (i = 0; fscanf(fptr, "%x", &val) > 0; i+=4) {
        fprintf (fpo, "when %d =>\n", i);
        fprintf (fpo, "    Data <= X\"%08X\";\n", val);
    }
    fprintf (fpo, "when others =>\n");
    fprintf (fpo, "-- saltar a 0\n");
    fprintf (fpo, "    Data <= X\"E3B0F000\";\n");
    fprintf (fpo, "\n    end case;\n");
    fprintf (fpo, "\nend process;\n");
    fprintf (fpo, "\nEND second;\n");
    fprintf (fpo, "\n\n\nCONFIGURATION cfg_arm_rom OF arm_rom IS");
    fprintf (fpo, "\nFOR second");
    fprintf (fpo, "\nEND FOR;\n");
    fprintf (fpo, "\nEND cfg_arm_rom;\n");
    fclose (fptr);
    fclose (fpo);
    return 0;
}
```

6.4 Emulando el multiplicador

El CORE generado no incluye el multiplicador, aunque está habilitado para ello. Se ha optado por esta alternativa debido a que:

- los multiplicadores combinatoriales, que operan en un ciclo de reloj, y que VHDL incluirían en el diseño, ocuparían un área enorme.
- existen células básicas que contienen multiplicadores combinatoriales muy optimizados y depurados, y que se pueden añadir al diseño

Por este motivo existen dos ficheros VHDL, que generan la misma entidad en función de una variable de tipo *generic*. Si el valor especificado para la variable `NOMUL=1`, entonces la entidad que se añade al diseño es la que no contiene multiplicador.

En caso contrario, se añade una entidad que contiene un multiplicador combinatorial de los que añade VHDL, y que nos sirve para poder simular la ALU, pero que carece de interés a la hora de realizar la síntesis, ya que el área ocupada por el mismo sería enorme.

Por todo ello, a la entidad que contiene el multiplicador, se le denomina “emulador de multiplicador” (porque aunque realmente es un multiplicador, su única función sería emular la operación, no implementarla).

El código de esta entidad es el siguiente:

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use work.ARM8.all;

entity mul_a is
    port (A: IN std_logic_vector(31 downto 0);
          B: IN std_logic_vector(31 downto 0);
          Y: OUT std_logic_vector(63 downto 0)
        );
end mul_a;

architecture comp of mul_a is
begin
    Y <= unsigned(A) * unsigned(B);
end comp;
```

Ya se ha expuesto en el apartado 5.5 del Capítulo Quinto las distintas simplificaciones y descomposiciones de operandos que se pueden abordar para conseguir una reducción de área en el multiplicador, en función de la aplicación para la que el CORE esté destinado (existirán situaciones en las que un multiplicador de 32*16 bits y el resultado en 48 bits, podría resultar suficientemente preciso).

El contenido de la ROM utilizada no lo vamos a mostrar, porque carece de interés, pero sí se van a indicar las operaciones que realiza cada instrucción:

- Se parte de que en R0, R1 y R2 existen los valores de la tabla que se adjunta a continuación.
- $MUL\ R3=R1*R0$
- $MUL\ R4=R2*R0$
- $MLA\ R5=R1*R2 + R0$
- $MLA\ R6=R0*R2+R1$

R0	R1	R2
00000005	FF000000	000000AC

----- NUEVO CICLO -----							
R0	00000000	R1	00000000	R2	00000000	R15(PC)	00000000
R3	00000000	R4	00000000	R5	00000000		
R6	00000000	R7	00000000	R8	00000000	CPSR	000000D3
R9	00000000	R10	00000000	R11	00000000		
R12	00000000	R13	00000000	R14	00000000	SPSR_FIQ	00000000
R8FIQ	00000000	R9FIQ	00000000	R10FIQ	00000000	SPSR_IRQ	00000000
R11FIQ	00000000	R12FIQ	00000000	R13FIQ	00000000	SPSR_UND	00000000
R14FIQ	00000000	R13IRQ	00000000	R14IRQ	00000000	SPSR_ABT	00000000
R13SVC	00000000	R14SVC	00000000	R13ABT	00000000	SPSR_SVC	00000000
R14ABT	00000000	R13UND	00000000	R14UND	00000000		
busA	00000000	busB	00000000	busC	00000000	Result	00000000
----- NUEVO CICLO -----							
R0	00000000	R1	00000000	R2	00000000	R15(PC)	00000004
R3	00000000	R4	00000000	R5	00000000		
R6	00000000	R7	00000000	R8	00000000	CPSR	000000D3
R9	00000000	R10	00000000	R11	00000000		
R12	00000000	R13	00000000	R14	00000000	SPSR_FIQ	00000000
R8FIQ	00000000	R9FIQ	00000000	R10FIQ	00000000	SPSR_IRQ	00000000
R11FIQ	00000000	R12FIQ	00000000	R13FIQ	00000000	SPSR_UND	00000000
R14FIQ	00000000	R13IRQ	00000000	R14IRQ	00000000	SPSR_ABT	00000000
R13SVC	00000000	R14SVC	00000000	R13ABT	00000000	SPSR_SVC	00000000
R14ABT	00000000	R13UND	00000000	R14UND	00000000		
busA	00000000	busB	00000000	busC	00000000	Result	00000000
----- NUEVO CICLO -----							
R0	00000000	R1	00000000	R2	00000000	R15(PC)	00000008
R3	00000000	R4	00000000	R5	00000000		
R6	00000000	R7	00000000	R8	00000000	CPSR	000000D3
R9	00000000	R10	00000000	R11	00000000		
R12	00000000	R13	00000000	R14	00000000	SPSR_FIQ	00000000
R8FIQ	00000000	R9FIQ	00000000	R10FIQ	00000000	SPSR_IRQ	00000000
R11FIQ	00000000	R12FIQ	00000000	R13FIQ	00000000	SPSR_UND	00000000
R14FIQ	00000000	R13IRQ	00000000	R14IRQ	00000000	SPSR_ABT	00000000
R13SVC	00000000	R14SVC	00000000	R13ABT	00000000	SPSR_SVC	00000000
R14ABT	00000000	R13UND	00000000	R14UND	00000000		
busA	00000000	busB	00000005	busC	00000000	Result	00000005
----- NUEVO CICLO -----							
R0	00000005	R1	00000000	R2	00000000	R15(PC)	0000000C
R3	00000000	R4	00000000	R5	00000000		
R6	00000000	R7	00000000	R8	00000000	CPSR	000000D3
R9	00000000	R10	00000000	R11	00000000		
R12	00000000	R13	00000000	R14	00000000	SPSR_FIQ	00000000
R8FIQ	00000000	R9FIQ	00000000	R10FIQ	00000000	SPSR_IRQ	00000000
R11FIQ	00000000	R12FIQ	00000000	R13FIQ	00000000	SPSR_UND	00000000
R14FIQ	00000000	R13IRQ	00000000	R14IRQ	00000000	SPSR_ABT	00000000
R13SVC	00000000	R14SVC	00000000	R13ABT	00000000	SPSR_SVC	00000000
R14ABT	00000000	R13UND	00000000	R14UND	00000000		
busA	00000005	busB	000000FF	busC	00000008	Result	FF000000

```

----- NUEVO CICLO -----
R0      00000005  R1      00000000  R2      00000000  R15(PC)  00000010
R3      00000000  R4      00000000  R5      00000000
R6      00000000  R7      00000000  R8      00000000  CPSR      A00000D3
R9      00000000  R10     00000000  R11     00000000
R12     00000000  R13     00000000  R14     00000000  SPSR_FIQ  00000000
R8FIQ   00000000  R9FIQ   00000000  R10FIQ  00000000  SPSR_IRQ  00000000
R11FIQ  00000000  R12FIQ  00000000  R13FIQ  00000000  SPSR_UND  00000000
R14FIQ  00000000  R13IRQ  00000000  R14IRQ  00000000  SPSR_ABT  00000000
R13SVC  00000000  R14SVC  00000000  R13ABT  00000000  SPSR_SVC  00000000
R14ABT  00000000  R13UND  00000000  R14UND  00000000
busA     00000005  busB     000000AC  busC     00000000  Result    000000AC
----- NUEVO CICLO -----
R0      00000005  R1      00000000  R2      00000000  R15(PC)  00000010
R3      00000000  R4      00000000  R5      00000000
R6      00000000  R7      00000000  R8      00000000  CPSR      200000D3
R9      00000000  R10     00000000  R11     00000000
R12     00000000  R13     00000000  R14     00000000  SPSR_FIQ  00000000
R8FIQ   00000000  R9FIQ   00000000  R10FIQ  00000000  SPSR_IRQ  00000000
R11FIQ  00000000  R12FIQ  00000000  R13FIQ  00000000  SPSR_UND  00000000
R14FIQ  00000000  R13IRQ  00000000  R14IRQ  00000000  SPSR_ABT  00000000
R13SVC  00000000  R14SVC  00000000  R13ABT  00000000  SPSR_SVC  00000000
R14ABT  00000000  R13UND  00000000  R14UND  00000000
busA     00000005  busB     00000000  busC     00000000  Result    00000000
----- NUEVO CICLO -----
R0      00000005  R1      FF000000  R2      00000000  R15(PC)  00000010
R3      00000000  R4      00000000  R5      00000000
R6      00000000  R7      00000000  R8      00000000  CPSR      200000D3
R9      00000000  R10     00000000  R11     00000000
R12     00000000  R13     00000000  R14     00000000  SPSR_FIQ  00000000
R8FIQ   00000000  R9FIQ   00000000  R10FIQ  00000000  SPSR_IRQ  00000000
R11FIQ  00000000  R12FIQ  00000000  R13FIQ  00000000  SPSR_UND  00000000
R14FIQ  00000000  R13IRQ  00000000  R14IRQ  00000000  SPSR_ABT  00000000
R13SVC  00000000  R14SVC  00000000  R13ABT  00000000  SPSR_SVC  00000000
R14ABT  00000000  R13UND  00000000  R14UND  00000000
busA     00000005  busB     00000000  busC     00000000  Result    00000000
----- NUEVO CICLO -----
R0      00000005  R1      FF000000  R2      000000AC  R15(PC)  00000014
R3      00000000  R4      00000000  R5      00000000
R6      00000000  R7      00000000  R8      00000000  CPSR      200000D3
R9      00000000  R10     00000000  R11     00000000
R12     00000000  R13     00000000  R14     00000000  SPSR_FIQ  00000000
R8FIQ   00000000  R9FIQ   00000000  R10FIQ  00000000  SPSR_IRQ  00000000
R11FIQ  00000000  R12FIQ  00000000  R13FIQ  00000000  SPSR_UND  00000000
R14FIQ  00000000  R13IRQ  00000000  R14IRQ  00000000  SPSR_ABT  00000000
R13SVC  00000000  R14SVC  00000000  R13ABT  00000000  SPSR_SVC  00000000
R14ABT  00000000  R13UND  00000000  R14UND  00000000
busA     00000005  busB     FF000000  busC     00000000  Result    FB000000
----- NUEVO CICLO -----
R0      00000005  R1      FF000000  R2      000000AC  R15(PC)  00000018
R3      00000000  R4      00000000  R5      00000000
R6      00000000  R7      00000000  R8      00000000  CPSR      800000D3
R9      00000000  R10     00000000  R11     00000000
R12     00000000  R13     00000000  R14     00000000  SPSR_FIQ  00000000
R8FIQ   00000000  R9FIQ   00000000  R10FIQ  00000000  SPSR_IRQ  00000000
R11FIQ  00000000  R12FIQ  00000000  R13FIQ  00000000  SPSR_UND  00000000
R14FIQ  00000000  R13IRQ  00000000  R14IRQ  00000000  SPSR_ABT  00000000
R13SVC  00000000  R14SVC  00000000  R13ABT  00000000  SPSR_SVC  00000000
R14ABT  00000000  R13UND  00000000  R14UND  00000000
busA     00000005  busB     000000AC  busC     00000000  Result    0000035C
----- NUEVO CICLO -----
R0      00000005  R1      FF000000  R2      000000AC  R15(PC)  0000001C
R3      00000000  R4      00000000  R5      00000000
R6      00000000  R7      00000000  R8      00000000  CPSR      000000D3
R9      00000000  R10     00000000  R11     00000000
R12     00000000  R13     00000000  R14     00000000  SPSR_FIQ  00000000
R8FIQ   00000000  R9FIQ   00000000  R10FIQ  00000000  SPSR_IRQ  00000000
R11FIQ  00000000  R12FIQ  00000000  R13FIQ  00000000  SPSR_UND  00000000
R14FIQ  00000000  R13IRQ  00000000  R14IRQ  00000000  SPSR_ABT  00000000
R13SVC  00000000  R14SVC  00000000  R13ABT  00000000  SPSR_SVC  00000000
R14ABT  00000000  R13UND  00000000  R14UND  00000000
busA     000000AC  busB     FF000000  busC     00000005  Result    54000005

```

```

----- NUEVO CICLO -----
R0      00000005  R1      FF000000  R2      000000AC  R15(PC) 00000020
R3      FB000000  R4      00000000  R5      00000000
R6      00000000  R7      00000000  R8      00000000  CPSR      000000D3
R9      00000000  R10     00000000  R11     00000000
R12     00000000  R13     00000000  R14     00000000  SPSR_FIQ 00000000
R8FIQ   00000000  R9FIQ   00000000  R10FIQ  00000000  SPSR_IRQ 00000000
R11FIQ  00000000  R12FIQ  00000000  R13FIQ  00000000  SPSR_UND 00000000
R14FIQ  00000000  R13IRQ  00000000  R14IRQ  00000000  SPSR_ABT 00000000
R13SVC  00000000  R14SVC  00000000  R13ABT  00000000  SPSR_SVC 00000000
R14ABT  00000000  R13UND  00000000  R14UND  00000000
busA     000000AC  busB     00000005  busC     FF000000  Result    FF00035C
-----
----- NUEVO CICLO -----
R0      00000005  R1      FF000000  R2      000000AC  R15(PC) 00000024
R3      FB000000  R4      0000035C  R5      00000000
R6      00000000  R7      00000000  R8      00000000  CPSR      800000D3
R9      00000000  R10     00000000  R11     00000000
R12     00000000  R13     00000000  R14     00000000  SPSR_FIQ 00000000
R8FIQ   00000000  R9FIQ   00000000  R10FIQ  00000000  SPSR_IRQ 00000000
R11FIQ  00000000  R12FIQ  00000000  R13FIQ  00000000  SPSR_UND 00000000
R14FIQ  00000000  R13IRQ  00000000  R14IRQ  00000000  SPSR_ABT 00000000
R13SVC  00000000  R14SVC  00000000  R13ABT  00000000  SPSR_SVC 00000000
R14ABT  00000000  R13UND  00000000  R14UND  00000000
busA     00000005  busB     00000000  busC     00000000  Result    00000000
-----
----- NUEVO CICLO -----
R0      00000005  R1      FF000000  R2      000000AC  R15(PC) 00000000
R3      FB000000  R4      0000035C  R5      54000005
R6      00000000  R7      00000000  R8      00000000  CPSR      00000000
R9      00000000  R10     00000000  R11     00000000
R12     00000000  R13     00000000  R14     00000000  SPSR_FIQ 00000000
R8FIQ   00000000  R9FIQ   00000000  R10FIQ  00000000  SPSR_IRQ 00000000
R11FIQ  00000000  R12FIQ  00000000  R13FIQ  00000000  SPSR_UND 00000000
R14FIQ  00000000  R13IRQ  00000000  R14IRQ  00000000  SPSR_ABT 00000000
R13SVC  00000000  R14SVC  00000000  R13ABT  00000000  SPSR_SVC 00000000
R14ABT  00000000  R13UND  00000000  R14UND  00000000
busA     00000005  busB     00000000  busC     00000000  Result    00000000
-----
----- NUEVO CICLO -----
R0      00000005  R1      FF000000  R2      000000AC  R15(PC) 00000004
R3      FB000000  R4      0000035C  R5      54000005
R6      FF00035C  R7      00000000  R8      00000000  CPSR      00000000
R9      00000000  R10     00000000  R11     00000000
R12     00000000  R13     00000000  R14     00000000  SPSR_FIQ 00000000
R8FIQ   00000000  R9FIQ   00000000  R10FIQ  00000000  SPSR_IRQ 00000000
R11FIQ  00000000  R12FIQ  00000000  R13FIQ  00000000  SPSR_UND 00000000
R14FIQ  00000000  R13IRQ  00000000  R14IRQ  00000000  SPSR_ABT 00000000
R13SVC  00000000  R14SVC  00000000  R13ABT  00000000  SPSR_SVC 00000000
R14ABT  00000000  R13UND  00000000  R14UND  00000000
busA     00000005  busB     00000000  busC     00000000  Result    00000000

```

En este último ciclo de reloj, todas las operaciones ya se han concluido, y además aparecen todos los resultados:

R0	R1	R2
00000005	FF000000	000000AC
R3	R4	R5
FB000000	0000035C	54000005
R6		
FF00035C		

Efectivamente se puede comprobar que los valores almacenados en R3, R4, R5, y R6 con el resultado de las operaciones descritas al principio de esta sección.

También, si observamos el valor del CPSR en cada ciclo de reloj, vemos como se van actualizando los flags según el resultado de la operación dos ciclos más atrás (nótese que cuando se almacena un valor en un registro --etapa 5ª --, éste fue calculado dos ciclos de reloj más atrás --etapa 3ª --, luego hay que entender qué está mostrando el CPSR en cada instante.

Para ver que ésto es así, basta con comprobar que el valor del CPSR se corresponde al resultado almacenado en un registro dentro de dos ciclos de reloj. Por ejemplo, el valor del CPSR para (PC) = 18 (en negrita en el listado anterior) se corresponde con el registro que se almacena para un (PC)=20, ésto es R3. Como R3 almacena un valor negativo, CPSR(31)=Z='1' y por tanto, en hexadecimal, el primer byte del CPSR vale 8 (NZCV="1000"=0x8).

Se puede comprobar que el resto de las operaciones también actualizan correctamente el registro de control de estado del programa.

Varias simulaciones como ésta fueron realizadas también con la multiplicación extendida de 64 bits, con y sin acarreo.

Tras realizar el diseño, la preocupación más importante que teníamos era si al ejecutarse una instrucción de dos ciclos de reloj de latencia, como la multiplicación de 64 bits, se iba a perder la siguiente instrucción.

De hecho, en las simulaciones previas a las definitivas (que se incluyen en el CD-ROM), ocurría que si bien la multiplicación de 64 bits funcionaba correctamente, la instrucción inmediatamente posterior se perdía. Esto era debido a que no se deshabilitaba correctamente la etapa *decode*.

En estas últimas simulaciones, la multiplicación de 64 bits funcionó correctamente. Además, se probaron toda serie de situaciones límites, como por ejemplo la utilización de operandos en la multiplicación, que estaban siendo utilizados por la instrucción anterior, de manera que se debían introducir 3 bubbles para que el operando fuese accesible, y ahora ejecutar la instrucción con 2 ciclos de reloj de latencia.

Todas las simulaciones fueron a priori satisfactorias, aunque posteriormente se detectó un problema cuando se producía una interrupción FIQ o IRQ durante el primer ciclo de latencia de la multiplicación de 64 bits, que será descrito en el apartado referente a la simulación de excepciones.

Los ficheros de las ROMs utilizadas para evaluar todos los casos posibles de multiplicaciones de 32 y 64 bits, con y sin signo, y con y sin acarreo, se muestran en la siguiente tabla:

FICHERO	DESCRIPCIÓN
multiply.txt	Contiene el programa anteriormente analizado
multlong.txt	Ejemplos críticos de multiplicaciones de 64 bits

6.5 Simulando la *pipeline*

Este tipo de estructuras *paralelo* es complicado de monitorizar, porque si se examinan los valores de los distintos registros de control y configuración en un instante dado (para un determinado valor del PC, por ejemplo), cada parámetro corresponde a una operación que puede haber sido realizada en el presente ciclo, o pudo ser realizada por una etapa anterior en un ciclo de reloj anterior.

Por eso hay que tener muy presente el funcionamiento y la estructura de la arquitectura *pipeline* para poder entender el fichero “*pipeline.dat*”, que constituye una de las salidas del simulador.

Igual ocurre con el resto de ficheros: por ejemplo, puede inducir a la no comprensión de una simulación el hecho de que para un valor del PC se muestren accesos de memoria correspondientes a una instrucción 4 ciclos más atrás. En cambio es un hecho lógico, pues cuando una instrucción está en *fetch*, la que entró cuatro ciclos antes en la *pipeline* será la que en estos momentos estará accediendo a memoria.

Se han realizado multitud de simulaciones que no expondremos en el documento, pero que se incluyen en el CD-ROM. En el apartado 6.8 se muestra la ruta y los contenidos de estas simulaciones, entre otras cosas.

Finalmente, cuando se tomó la determinación de que el funcionamiento del CORE era el correcto, se realizaron los siguientes *test*, que se pueden considerar como definitivos, y que pretenden comprobar que además de la ejecución normal de un programa, los casos excepcionales también funcionan correctamente.

En cada uno de los apartados que a continuación abordaremos, se debe intentar responder a las siguientes preguntas:

- ¿Qué comportamiento se desea evaluar?
- ¿Qué fallos se pretende detectar?
- ¿Dónde se encuentran los ficheros referentes a esta simulación?

En la siguiente tabla se adjuntan los nombres de dichos ficheros.

FICHERO	DESCRIPCIÓN
saltos.txt	Contiene saltos de todo tipo
data_psr.txt	Instrucciones de proceso de datos típicas
shifter.txt	Ejemplos de casos críticos del shifter
ldr_str.txt	Instrucciones de acceso a memoria de datos
halfword.txt	Idem al anterior, pero con la instrucción <i>halfword</i>
swp.txt	Instrucción LDR con registro de offset, y SWP
exception.txt	Incluye rutinas de atención a ciertas excepciones

6.5.1 Verificando la FASE I

Instrucciones de Salto (fichero *saltos.txt*)

El programa intenta evaluar el correcto funcionamiento de todas las máquinas de control involucradas en los saltos, pues estos constituyen una de las acciones más problemáticas en una estructura *pipeline*.

Con esta simulación se pretende descubrir problemas de cálculo de direcciones absolutas a través de los offset relativos especificados en las instrucciones, y sobre todo problemas de retorno en las instrucciones de salto con registro de enlace (BL).

El programa para verificar los saltos realiza las siguientes acciones:

- ☐ salta a una determinada posición, guardando en R14 la dirección de vuelta
- ☐ realiza unas cuantas operaciones absurdas de transferencia entre registros
- ☐ MOV PC, R14, que deberá volver a la siguiente instrucción al salto con retorno (BL)
- ☐ realiza otras cuantas operaciones sin sentido con registros para que transcurran unos cuantos ciclos
- ☐ se realiza un salto a 0, pero no absoluto, sino relativo, para comprobar que el hardware aplica bien el algoritmo de cálculo de direcciones de salto.

Instrucciones de Proceso de Datos (fichero *data_psr.txt*)

En esta sección queremos comprobar si la *pipeline* introduce los operandos y los parámetros en la ALU de forma correcta, y verificar en la medida de lo posible la ruta de datos, y las actualizaciones de los flags del CPSR.

- ☐ R0 = 0
- ☐ R1 = FF
- ☐ R2 = C
- ☐ R3 = R1 and R2 (Resultado = C)
- ☐ SI Z=1, R8=8 ; No debe ejecutarse, pues el resultado anterior no fue 0 !
- ☐ R3 = R0 and R2 (Resultado = 0)
- ☐ SI Z=1, R9=9 ; Ahora sí debe ejecutarse!
- ☐ R3 = R1 and R2 (Resultado = C)
- ☐ SI not(Z)=1, R8=8 ; También se debe ejecutar!
- ☐ R4 = FF000000 (como debe hacer un desplazamiento, hay un carry)
- ☐ SI N=1, R10=A ; Debe ejecutarse!
- ☐ R5 = FF
- ☐ SI N=1, R11=B ; No debe ejecutarse!
- ☐ R12 = C ; Debe ejecutarse!

Además de verificar ciertas instrucciones de datos, y de observar que funciona la ejecución condicional de instrucciones, este programa verifica cosas mucho más profundas de las que inicialmente se puedan deducir al observarlo. Y es que si funciona,

habremos probado que una instrucción cuya condición de ejecución no se cumple, no altera el contenido del CPSR, ni de ningún registro auxiliar.

No hemos incluido instrucciones de suma o resta porque aparecen en los sucesivos programas que se han realizado, y porque no es necesario, puesto que

- hemos verificado anteriormente el correcto funcionamiento de la ALU
- hemos verificado que el código de operación de la ALU se crea correctamente a partir del campo correspondiente de la instrucción

Verificando el Shifter (fichero *shifter.txt*)

Llegados a este punto, se consigue depurar el comportamiento del *shifter*, sobre todo en lo que se refiere a ciertos casos excepcionales, en los que el *shift_amount* es proporcionado por el byte menos significativo de un registro de propósito general.

Tras realizar las oportunas modificaciones en el diseño, se hace necesario una comprobación minuciosa de todos los casos clave en el funcionamiento del *shifter*.

No vamos a describir el programa porque sería muy complejo, no en sí mismo el describirlo, sino el tratar de explicar los resultados que deben darse y el por qué, puesto que el funcionamiento del desplazador es algo bastante complejo.

Remitimos al lector a las secciones 4.4 y 5.5, en las que se puede observar todas las operaciones que debe realizar el shifter en función de todos los parámetros relacionados con él. Así podrá observar la complejidad derivada de todas las posibles combinaciones de modos de operación, uso de registro o de operandos inmediatos, ... que es lo que motiva el que sea difícil explicar estas simulaciones.

Nosotros aseguramos que los resultados fueron del todo satisfactorios, y tanto los ficheros fuente, como las trazas de la simulación se incluyen en el CD-ROM.

Verificando las instrucciones MSR y MRS (*msr_mrs.txt*)

El objetivo fundamental de esta ROM es comprobar que la transferencia entre registros y PSRs, y viceversa, se realiza de forma correcta, y más aún, **verificar que el CPSR no puede ser modificado ni en modo *user* ni en modo *system***. Durante las simulaciones se encontró un error grave de seguridad: cuando nos encontramos en modo *user* o en modo *system*, los SPSRs no son visibles. Si intentamos escribir el CPSR, la operación no podría realizarse, pero en cambio, si intentamos escribir un SPSR, como no es visible, el CORE modificaba entonces el CPSR.

Este comportamiento constituye un fallo de protección física que tuvo que ser subsanado. El origen del problema estaba en que, al no ser visible el SPSR ni en modo *user* ni en modo *system*, el comportamiento del hardware era imprevisible, y resultó ser que escribía el resultado en el CPSR. La solución consistió en la implantación de una protección hardware explícita del CPSR para esos dos modos de funcionamiento.

6.5.2 Verificando la FASE II

En esta sección es necesario emular una RAM para poder simular los accesos a memoria en lectura y en escritura.

Realmente, emular un acceso **en escritura** es bastante sencillo, pues simplemente basta con monitorizar los buses de dirección y de escritura. El entorno de simulación creado ya escribía estos buses en el fichero *memory.dat*, así que basta con examinar dicho fichero para comprobar los accesos en escritura.

Simular accesos **en lectura** es algo más complejo, pues necesitamos poder leer algo; no basta con examinar los buses, pues el CORE del ARM realiza una serie de tratamientos al dato cargado de memoria (desplazamientos de bytes, ...) en función de lo desalineada que esté la dirección del acceso con un WORD.

Para emular la RAM, se crea un fichero denominado *ram.vhd*, que inicialmente fue generado a partir de un fichero de texto mediante la aplicación Txt2vhdl.exe. Posteriormente, se le realizaron una serie de modificaciones, como por ejemplo, se introduce la señal nABORT, para que genere una interrupción cuando se intente acceder a una dirección no válida. Aunque generar esta interrupción en un entorno real, no corresponde a la RAM, sino al decodificador general de todo el sistema de memoria (una dirección no genera ABORT cuando no corresponde a la RAM, sino cuando no es válida; es decir, cuando no correspondie a ningún dispositivo mapeado en memoria), nos va a servir para poder comprobar el comportamiento del ARM cuando se produce un acceso que podríamos denominar como *erróneo*.

En el apartado 8.1 se detallan todos los ficheros que se han involucrado de alguna manera con estas simulaciones.

Instrucciones LDR/STR (fichero *ldr_str.txt*)

Con estas simulaciones se pretende demostrar el correcto funcionamiento de las instrucciones más sencillas de acceso a memoria, pues internamente la arquitectura *load/store* utilizada en el CORE reutiliza las mismas unidades funcionales que implementan estas instrucciones, para poder llevar a cabo instrucciones más complejas.

Prácticamente podríamos asegurar que si el funcionamiento de LDR y STR es correcto, el resto de instrucciones de memoria también operaran satisfactoriamente; no obstante, nosotros vamos a comprobar cada una de ellas, con objeto de reducir así el espacio de fallos a su mínima expresión.

No vamos a dar detalles sobre el programa, simplemente mencionar que existen infinidad de combinaciones entre los distintos parámetros que se pueden proporcionar:

- W: se puede escribir o no de vuelta el registro base

- PRE/POST: se puede aplicar directamente la dirección del registro base, o también la calculada tras operar con el offset (preindexado, o postindexado)
- ADD/SUB: el offset se puede restar o sumar al registro base
- El offset puede ser un registro o un valor inmediato
- El offset puede sufrir desplazamientos
- El acceso puede ser de un byte cualquiera de los cuatro que compone una palabra de memoria, o bien del WORD completo.
- load/store: podemos acceder en lectura o en escritura
- Cuando se realiza una carga en registro de un WORD, y la dirección no está alineada, se realiza una rotación de los bytes que componen la palabra en tantas posiciones como esté desalineada la dirección.

En la sección correspondiente del Capítulo Cuarto se explican todas estas características, e incluso posiblemente alguna más.

Las simulaciones llevadas a cabo mediante este programa de pruebas, deben manifestar si el funcionamiento de la arquitectura *load-store* que nuestra *pipeline* utiliza es correcto o no.

Las entidades *load* y *store* son el motor del resto de instrucciones de memoria, así que si su funcionamiento es correcto, es muy probable que toda la FASE II funcione correctamente.

Instrucciones Halfword (fichero *halfword.txt*)

Estas instrucciones se basan en las anteriores, pero vamos a continuar con las simulaciones para así estar suficientemente seguros de haber examinado todas las posibles combinaciones de parámetros.

Además, aunque las unidades funcionales sean las mismas, hay que comprobar que el decodificador de instrucciones para correctamente los parámetros desde la instrucción *halfword* a la configuración de unidades funcionales que genera.

Instrucciones SWP (fichero *swp.txt*)

Esta instrucción es crítica, pues su latencia es de dos ciclos de reloj. El objeto de simularla, más que comprobar el acceso a memoria, es verificar el comportamiento de la *pipeline* cuando se produce interlocking, y esta instrucción está en la cuarta etapa, etc, es decir, combinaciones de los distintos casos críticos que se pueden dar en la *pipeline*, pero ahora simultáneamente.

Nota: todas las simulaciones anteriores, se hicieron primeramente con *ws="00"*, es decir, sin incluir estados adicionales de espera. Pero posteriormente, se repitieron con un estado de espera adicional, y también con dos. Aquí fue donde se determinó que aunque el caso "11" también corresponde a dos estados adicionales de espera, no existen garantías de su correcto funcionamiento, por lo que no debe utilizarse.

6.6 Comprobando el control de interrupciones

La primera particularidad que podemos encontrar en los programas realizados para la verificación de las interrupciones, es el hecho de que aparecen en las primeras líneas una serie de instrucciones de salto, una a continuación de otra.

Esto es debido a que estos programas se adaptan a la estructura real de un programa, en la cual las primeras posiciones se corresponden con los vectores de interrupción, y por ello en esas posiciones se debe encontrar una instrucción de salto a la dirección de memoria de programa en la cual se encuentra la rutina de atención a la correspondiente excepción.

Dirección	Excepción	Modo de Entrada
0x00000000	Reset	Supervisor
0x00000004	Instrucción no reconocida	Undefined
0x00000008	SWI	Supervisor
0x0000000C	(**)	----
0x00000010	ABORT	Abort
0x00000014	-- Reservado --	----
0x00000018	IRQ	IRQ
0x0000001C	FIQ	FIQ

(**) No disponible en esta implementación del CORE del ARM8/9

Las interrupciones FIQ y IRQ se probaron en una de las simulaciones previas a las definitivas. De todos modos, al ser externas al procesador, no dependen de la ROM del programa, sino de que se genere un evento externo; para emular el evento, lo que hicimos fue en el fichero *test.vhd*, tras un cierto número de ciclos en los que el programa se va ejecutando normalmente, y tras asegurarnos de que el programa ha habilitado las interrupciones, provocábamos un cambio en el pin IRQ (por ejemplo), con lo cual, el programa debía saltar a la posición donde se encontraba la rutina de excepción.

El programa que se encuentra en el fichero *exception.txt* realiza una serie de operaciones bien distintas:

- en las primeras posiciones, se encuentran los saltos a las direcciones donde comienzan las rutinas de atención correspondientes
- en primer lugar se intenta realizar un acceso a la dirección de memoria apuntada por el registro R2, que no existe, con lo cual genera ABORT
- la rutina de atención de la excepción corrige el valor de R2 por un valor que está dentro del rango de la RAM, así que al devolver de nuevo el control, deberá completar el acceso a memoria correctamente
- tras realizar algunas operaciones absurdas, se ejecuta una llamada al sistema, SWI. La rutina realiza dos o tres operaciones absurdas con registros, para poder comprobar en la simulación que realmente ha saltado a la rutina de atención, y devuelve el control al programa principal

Directorio EXCEPTION2

Contiene una serie de programas pensados para verificar todas las combinaciones posibles de situaciones críticas, como *interlocking*, sats, instrucciones de larga latencia, accesos a memoria de más de un ciclo de reloj y excepciones.

La finalidad de estas simulaciones es depurar el comportamiento del CORE, no sólo con la finalidad de obtener una respuesta correcta, sino además con afán de optimización, ésto es, queremos que nuestro diseño sea capaz de resolver todas estas situaciones, pero además sin invertir un ciclo de reloj más de los necesarios.

Para realizar estas simulaciones, debemos modificar ciertas líneas de código que aparecen comentadas, y utilizar unas u otras en función de la situación que se desee evaluar.

Las líneas del fichero *test.vhd* que emulan las excepciones son las siguientes:

```
-- nFIQ <= '0' after 1600 ns, '1' after 1705 ns;  
-- nIRQ <= '0' after 1600 ns, '1' after 1705 ns;  
-- nFIQ <= not DEBUG_MCONTROL;  
-- nIRQ <= not DEBUG_MCONTROL;
```

Para la implementación de los estados de espera, tenemos:

```
ws<="00";  
--ws<="01";  
--ws<="10";
```

Nótese que en el caso de los estados de espera, **nunca pueden estar todas las líneas comentadas a la vez, o el CORE alcanzaría un estado de bloqueo.**

Los ficheros utilizados para verificar estas situaciones son los siguientes:

- fiq2.txt: (hay que eliminar el comentario de la tercera línea nFIQ) se produce una excepción FIQ cuando una instrucción de larga latencia (multiplicación de 64 bits) está ocupando simultáneamente *decode* y *execute*. Esto implica que si no se han tomado las oportunas precauciones, se almacenará un valor de retorno de la excepción incorrecto, pues el contador de programa muestra un valor 4 bytes menor que el que presentaría en una situación normal; además, debemos verificar que únicamente en este caso, no se vacía la etapa *decode*, pues parte de la instrucción (el primer ciclo) ya estaba ejecutándose.
- fiq3.txt: (eliminar el comentario de la primera línea nFIQ) comportamiento normal ante un FIQ
- fiq4.txt: idem al anterior, pero se está produciendo un acceso a memoria de duración 3 ciclos de reloj cuando llega el FIQ. No debe perderse ninguna instrucción en el proceso.

- *fiq5.txt*: similar al primero (eliminar por tanto, el tercer nFIQ comentado), pero el acceso a memoria dura varios ciclos de reloj, y va seguido de una instrucción de larga latencia (multiplicación de 64 bits) cuando se produce el FIQ.
- *abort.txt*: (todos comentados) combina instrucciones de larga latencia en *decode* y *execute* cuando se produce el ABORT, y además se ha simulado con WS=0, 1 2 (el usuario puede modificar este valor a su gusto, simplemente cambiando la línea WS sin comentar).

Los ficheros IRQx.txt son análogos a los anteriores, pero las líneas que hay que utilizar en el fichero *test.vhd* (quitar comentarios siguiendo las mismas reglas que en el caso de las nFIQ), son las que empiezan por nIRQ.

Subdirectorio SWP\SWP2

Este subdirectorio almacena el resultado de una serie de simulaciones destinadas a verificar la atomicidad de la instrucción SWP, y la robustez del mecanismo de implementación de la misma frente a fallos de memoria.

Para lograr obtener estos resultados, fue necesario romper un poco el método habitual de simulación, para obtener situaciones límites, como por ejemplo fallos de memoria en el segundo ciclo (escritura) del *swap*, pero en el segundo o tercer ciclo de reloj del acceso (suponiendo acceso a memoria de varios ciclos de reloj de duración).

El proceso para alcanzar los ficheros de salida es bastante tedioso, así que lo obviaremos.

Basta simplemente con observar los resultados almacenados en los directorios *ciclo_load* y *ciclo_store*, para observar como efectivamente el mecanismo que garantiza la *atomicidad* de la instrucción SWP es totalmente robusto frente a cualquier situación que se pueda plantear.

- *ciclo_load*: el fallo de memoria se produce en el primer ciclo (lectura) del *swap*. El sistema debe garantizar que no se altera el contenido de ningún registro, pues se ha producido un ABORT. Estas simulaciones se realizaron con accesos de varios ciclos de duración.
- *ciclo_store*: el fallo de memoria se produce en el segundo ciclo (escritura). Esta es la situación más crítica, pues como el acceso dura varios ciclos de reloj, el ABORT no tiene por qué generarse en el primer ciclo del acceso, de modo que la etapa *write* no debe permitir que el primer ciclo de la instrucción (que estará en dicha etapa) altere ningún registro hasta que no existan garantías plenas de que el acceso a memoria ha concluido con éxito.

6.7 Ejecutando un programa completo

El programa que se analizará en este apartado no es muy diferente al anteriormente expuesto, pues tiene la misma estructura. Simplemente contiene una selección de las instrucciones más comúnmente utilizadas en los programas.

La ROM con el correspondiente programa, ha sido generada a partir del fichero **completo.txt** que se incluye en el CD-ROM.

Los ficheros de salida de la simulación son 3: traza.dat, pipeline.dat y memory.dat. No vamos a incluir los tres listados completos, pues necesitaríamos decenas de páginas solamente para los listados.

Para cada ciclo de reloj, en cada fichero se genera un cuadro con las lecturas de los registros, señales y buses más importantes. Lo que haremos será ir comentando los ciclos más relevantes de cada fichero, de modo que incluiremos el ciclo más relevante para cada valor del PC, generando así una traza que será una mezcla entre los tres ficheros.

Es decir, vamos a coger el primer ciclo de un fichero, el segundo de otro, e iremos comentando los resultados.

Para la correcta comprensión de las simulaciones, vamos a recordar una serie de conceptos, relacionados con las abreviaturas utilizadas, aunque lo ideal sería un estudio previo detallado del Capítulo Cuarto y Quinto antes de abordar la lectura de la traza. Ello facilitaría enormemente su comprensión

Recordatorio:

¿Cómo alinea la interfaz de memoria las direcciones?

La siguiente tabla ilustra cómo la interfaz del CORE altera los dos últimos bits de la dirección de acceso a memoria calculada, para forzar la alineación con el word, el halfword o el byte, según el caso (nótese que en lectura, siempre se generan direcciones WORD-alineadas, pues siempre se lee la palabra completa de 32 bits, y luego se selecciona el/los byte/s a cargar en el registro).

Bits 1 y 0 de la Dirección Calculada	Vaddress [1:0]			
	r_enable = 1	w_enable=1		
		byte	halfword	word
00	00	00	00	00
01	00	01	00	00
10	00	10	10	00
11	00	11	10	00

Abreviaturas utilizadas en la descripción de las instrucciones de memoria

Todas las instrucciones de acceso a memoria utilizadas en el programa de la siguiente simulación son del tipo LDR (Ej: `ldr, pre, up, byte, W=0 R0 R4 #0`).

Para calcular la dirección absoluta de acceso a memoria, se parte del valor contenido en el registro base, y bien se le suma (UP), o bien se le resta (DOWN) un *offset*, que puede ser un valor inmediato, o bien un registro de propósito general.

El modo de direccionamiento puede ser preindexado (PRE) ó postindexado (POST), éste es, que la dirección aplicada en el acceso de memoria puede ser la dirección calculada tras aplicar el *offset* al registro base, o bien se accede directamente a memoria con la dirección contenida en el registro base.

El bit W=1 implica que la dirección calculada tras sumar o restar el *offset* al registro base, hay que almacenarla de vuelta en el propio registro base. Cuando estamos utilizando direccionamiento postindexado (POST), este bit es redundante.

Ejecutando el programa

La primera instrucción es un salto a la dirección 0x00000020, pero no se realiza de forma efectiva hasta la etapa *execute*, es decir, hasta dos ciclos de reloj después, lo cual implica que el salto se realiza cuando (PC)=8. Obsérvese en el ciclo (PC)=8 las señales en negrita, que la etapa *execute* activa para que se genere el salto

----- NUEVO CICLO -----							
Inst	EA000006	d_inst	F0000000	e_inst	0F000000	(PC)	00000000
m_inst	0F000000	w_inst	0F000000	m_data	00000000	w_data	00000000
m_wbReg	00000000	w_wbReg	00000000	emw-WBR	000	m_addr	00000000
eA	1	eB	1	eC	0		
busA	0	busB	0	busC	0		
op A	00000000	op B	00000000	op C	00000000	AUXBUS	00000
Result	00000000	(e/m/w)	111	execute	0		
JUMP	0	JUMP_ADDRESS	00000000	WS	0	M_CTRL	0
exc_happens	0	exc_report	0	EXCEPT:	000		
----- NUEVO CICLO -----							
Inst	E3A0F000	d_inst	EA000006	e_inst	F0000000	(PC)	00000004
m_inst	0F000000	w_inst	0F000000	m_data	00000000	w_data	00000000
m_wbReg	00000000	w_wbReg	00000000	emw-WBR	000	m_addr	00000000
eA	0	eB	0	eC	0		
busA	0	busB	0	busC	0		
op A	00000000	op B	00000000	op C	00000000	AUXBUS	00000
Result	00000000	(e/m/w)	111	execute	0		
JUMP	0	JUMP_ADDRESS	00000000	WS	0	M_CTRL	0
exc_happens	0	exc_report	0	EXCEPT:	000		
----- NUEVO CICLO -----							
Inst	E3A0F060	d_inst	E3A0F000	e_inst	E2180000	(PC)	00000008
m_inst	FF000000	w_inst	0F000000	m_data	00000000	w_data	00000000
m_wbReg	00000000	w_wbReg	00000000	emw-WBR	000	m_addr	00000000
eA	0	eB	0	eC	0		
busA	0	busB	0	busC	0		
op A	00000008	op B	00000018	op C	00000000	AUXBUS	00000
Result	00000020	(e/m/w)	111	execute	1		
JUMP	1	JUMP_ADDRESS	00000020	WS	0	M_CTRL	0
exc_happens	0	exc_report	0	EXCEPT:	000		

Las próximas instrucciones son:

- (PC) = 20 MOVE R0, #0
- (PC) = 24 MOVE R1, #4
- (PC) = 28 MOVE R2, HEX 28
- (PC) = 2C MOVE R3, HEX FF

El resultado de la primera de ellas no estará en la etapa write hasta (PC)= 30, lo cual implica que hasta (PC)=34 no podremos ver el valor escrito en R0. Un ciclo después estará escrito R1, otro ciclo después R2, y así sucesivamente. Hasta (PC) = 30, vamos a seguir adjuntando ciclos extraídos del fichero *pipeline.dat*.

El significado de cada una de las etiquetas es exactamente el mismo que se le dio a los registros y buses en la descripción efectuada en el Capítulo 5. La figura 5.1 puede resultar bastante ilustrativa para la comprensión de esta traza, aunque será difícil comprender la simulación si no se ha asimilado toda la información contenida en los Capítulos 3, 4 y 5.

```

----- NUEVO CICLO -----
Inst      E3B00000    d_inst  F3A0F060    e_inst  EF0DF000    (PC)      00000020
m_inst    E2180000    w_inst  FF000000    m_data  00000000    w_data  00000000
m_wbReg   00000000    w_wbReg 00000000    emw-WBR 000        m_addr  00000000
eA        0          eB        0          eC        0
busA      0          busB      0          busC      0
op A      00000008    op B      00000000    op C      00000000    AUXBUS   00000
Result    00000000    (e/m/w)    111    execute  1
JUMP      0          JUMP_ADDRESS 00000000    WS        0          M_CTRL   0
exc_happens 0          exc_report 0          EXCEPT: 000

----- NUEVO CICLO -----
Inst      E3B01004    d_inst  E3B00000    e_inst  F00DF000    (PC)      00000024
m_inst    EF0DF000    w_inst  E2180000    m_data  00000000    w_data  00000000
m_wbReg   00000000    w_wbReg 00000000    emw-WBR 000        m_addr  00000000
eA        0          eB        0          eC        0
busA      0          busB      0          busC      0
op A      00000008    op B      00000060    op C      00000000    AUXBUS   00000
Result    00000060    (e/m/w)    111    execute  0
JUMP      0          JUMP_ADDRESS 00000000    WS        0          M_CTRL   0
exc_happens 0          exc_report 0          EXCEPT: 000

----- NUEVO CICLO -----
Inst      E3B02028    d_inst  E3B01004    e_inst  E02D0000    (PC)      00000028
m_inst    FF0DF000    w_inst  EF0DF000    m_data  00000000    w_data  00000000
m_wbReg   00000000    w_wbReg 00000000    emw-WBR 000        m_addr  00000000
eA        0          eB        0          eC        0
busA      0          busB      0          busC      0
op A      00000008    op B      00000000    op C      00000000    AUXBUS   00000
Result    00000000    (e/m/w)    011    execute  1
JUMP      0          JUMP_ADDRESS 00000000    WS        0          M_CTRL   0
exc_happens 0          exc_report 0          EXCEPT: 000

----- NUEVO CICLO -----
Inst      E3B030FF    d_inst  E3B02028    e_inst  E02D1000    (PC)      0000002C
m_inst    E02D0000    w_inst  FF0DF000    m_data  00000000    w_data  00000000
m_wbReg   00000000    w_wbReg 00000000    emw-WBR 000        m_addr  00000000
eA        0          eB        0          eC        0
busA      0          busB      0          busC      0
op A      00000008    op B      00000004    op C      00000000    AUXBUS   00000
Result    00000004    (e/m/w)    001    execute  1
JUMP      0          JUMP_ADDRESS 00000000    WS        0          M_CTRL   0
exc_happens 0          exc_report 0          EXCEPT: 000
----- NUEVO CICLO -----

```

```

----- NUEVO CICLO -----
Inst      E5B03004      d_inst  E3B030FF      e_inst  E02D2000      (PC)      00000030
m_inst    E02D1000      w_inst  E02D0000      m_data  00000004      w_data  00000000
m_wbReg   00000000      w_wbReg 00000000      emw-WBR 000      m_addr  00000000
eA        0            eB        0            eC        0
busA      0            busB      0            busC      0
op A      00000008      op B      00000028      op C      00000000      AUXBUS   00000
Result    00000028      (e/m/w)      000      execute  1
JUMP      0            JUMP_ADDRESS 00000000      WS        0            M_CTRL   0
exc_happens 0      exc_report 0      EXCEPT: 000
----- NUEVO CICLO -----

```

A continuación vamos a añadir los siguientes ciclos del fichero *traza.dat*. En ellos veremos el contenido de los registros, que se irán actualizando a los valores indicados por las instrucciones anteriores.

```

----- NUEVO CICLO -----
R0      00000000      R1      00000000      R2      00000000      R15(PC) 00000034
R3      00000000      R4      00000000      R5      00000000
R6      00000000      R7      00000000      R8      00000000      CPSR     000000D3
R9      00000000      R10     00000000      R11     00000000
R12     00000000      R13     00000000      R14     00000000      SPSR_FIQ 00000000
R8FIQ   00000000      R9FIQ   00000000      R10FIQ  00000000      SPSR_IRQ 00000000
R11FIQ  00000000      R12FIQ  00000000      R13FIQ  00000000      SPSR_UND 00000000
R14FIQ  00000000      R13IRQ  00000000      R14IRQ  00000000      SPSR_ABT 00000000
R13SVC  00000000      R14SVC  00000000      R13ABT  00000000      SPSR_SVC 00000000
R14ABT  00000000      R13UND  00000000      R14UND  00000000
busA     00000008      busB     000000FF      busC     00000000      Result   000000FF
----- NUEVO CICLO -----
R0      00000000      R1      00000004      R2      00000000      R15(PC) 00000038
R3      00000000      R4      00000000      R5      00000000
R6      00000000      R7      00000000      R8      00000000      CPSR     000000D3
R9      00000000      R10     00000000      R11     00000000
R12     00000000      R13     00000000      R14     00000000      SPSR_FIQ 00000000
R8FIQ   00000000      R9FIQ   00000000      R10FIQ  00000000      SPSR_IRQ 00000000
R11FIQ  00000000      R12FIQ  00000000      R13FIQ  00000000      SPSR_UND 00000000
R14FIQ  00000000      R13IRQ  00000000      R14IRQ  00000000      SPSR_ABT 00000000
R13SVC  00000000      R14SVC  00000000      R13ABT  00000000      SPSR_SVC 00000000
R14ABT  00000000      R13UND  00000000      R14UND  00000000
busA     00000000      busB     00000004      busC     00000000      Result   00000004
----- NUEVO CICLO -----
R0      00000000      R1      00000004      R2      00000028      R15(PC) 00000038
R3      00000000      R4      00000000      R5      00000000
R6      00000000      R7      00000000      R8      00000000      CPSR     000000D3
R9      00000000      R10     00000000      R11     00000000
R12     00000000      R13     00000000      R14     00000000      SPSR_FIQ 00000000
R8FIQ   00000000      R9FIQ   00000000      R10FIQ  00000000      SPSR_IRQ 00000000
R11FIQ  00000000      R12FIQ  00000000      R13FIQ  00000000      SPSR_UND 00000000
R14FIQ  00000000      R13IRQ  00000000      R14IRQ  00000000      SPSR_ABT 00000000
R13SVC  00000000      R14SVC  00000000      R13ABT  00000000      SPSR_SVC 00000000
R14ABT  00000000      R13UND  00000000      R14UND  00000000
busA     00000000      busB     00000003      busC     00000000      Result   00000003
----- NUEVO CICLO -----
R0      00000000      R1      00000004      R2      00000028      R15(PC) 00000038
R3      000000FF      R4      00000000      R5      00000000
R6      00000000      R7      00000000      R8      00000000      CPSR     000000D3
R9      00000000      R10     00000000      R11     00000000
R12     00000000      R13     00000000      R14     00000000      SPSR_FIQ 00000000
R8FIQ   00000000      R9FIQ   00000000      R10FIQ  00000000      SPSR_IRQ 00000000
R11FIQ  00000000      R12FIQ  00000000      R13FIQ  00000000      SPSR_UND 00000000
R14FIQ  00000000      R13IRQ  00000000      R14IRQ  00000000      SPSR_ABT 00000000
R13SVC  00000000      R14SVC  00000000      R13ABT  00000000      SPSR_SVC 00000000
R14ABT  00000000      R13UND  00000000      R14UND  00000000
busA     00000000      busB     00000003      busC     00000000      Result   00000003
----- NUEVO CICLO -----

```

Llama la atención el hecho de que en los últimos tres ciclos de reloj, el valor del PC no se ha alterado. Esto se debe a que se ha producido un fenómeno de *interlocking*.

La instrucción que en (PC)=30 ha entrado en la *pipeline*, es un *ldr*, que utiliza R0 como registro base, y tras aplicarle el offset, escribe de vuelta en R0 el resultado (W=1).

La instrucción que entra en (PC)=34 es también una instrucción *ldr*, y utiliza también R0 como registro base, así que cuando ésta llega a *decode* (en PC=38), requiere como operando el registro R0 que aún no está actualizado. Se ha producido entonces un fenómeno de *interlocking*, pues la instrucción en *decode* debe esperar a que una instrucción que se encuentra actualmente en *execute*, complete la fase de *write*. Ese es el motivo por el que la *pipeline* ha quedado congelada en (PC)=38.

Vamos a incluir algunos ciclos de reloj del fichero *pipeline.dat*, algunos de los cuales se solapan con los ciclos que acabamos de introducir del fichero *traza.dat*, pero que son interesantes de analizar para observar cómo resuelve la *pipeline* el fenómeno de *interlocking*.

En (PC)=30 se cargó en la *pipeline* la instrucción E5B03004, que es un LDR con W=1, modo preindexado, UP=1 y que utiliza como registro base a R0 y como offset, el valor absoluto #4. En el ciclo siguiente entra otra instrucción en la *pipeline* que utiliza también R0 como offset:

En negrita, hemos marcado los siguientes eventos:

- La instrucción anteriormente mencionada llega a *decode*. La instrucción E5B03004 es el primer *ldr*, y la instrucción E5D04003 es el está marcado en negrita cuando llega a *decode*.
- La etiqueta (e/m/w) se refiere a las señales *e_nro*, *m_nro* y *w_nro*. Cuando estas señales están a '1' significa que en la etapa respectiva (*execute*, *memory*, *write*) no hay operando destino. En nuestro caso, están todas a '0', luego el decodificador de los buses A, B y C comprueba el operando destino de las tres etapas, y resulta que hay coincidencia con el registro destino de la etapa *execute*. Por tanto, realiza una petición a la *máquina global de control* de 3 ciclos de espera. WS, por tanto, toma el valor 3.
- WS lleva la cuenta de los ciclos de espera restantes, así que en cada ciclo se va disminuyendo hasta llevar a 0.
- También se ha resaltado en negrita el valor del registro **e_inst=E53840A0** en ciertos ciclos. El segundo byte de la instrucción es "F", lo cual significa, independientemente del resto de los bytes del registro, que se trata de un *bubble*, es decir, de una NOP. Se ha resaltado en negrita también la propagación de este *bubble* mientras la *pipeline* está congelada.

```

----- NUEVO CICLO -----
Inst    E5D04003    d_inst    E5B03004    e_inst    E02D3000    (PC)    00000034
m_inst  E02D2000    w_inst    E02D1000    m_data    00000028    w_data    00000004
m_wbReg 00000000    w_wbReg    00000000    emw-WBR    000        m_addr    00000000
eA      1          eB        0          eC        0
busA    0          busB      0          busC      0
op A    00000008    op B      000000FF    op C      00000000    AUXBUS    00000
Result  000000FF    (e/m/w)    000          execute  1
JUMP    0          JUMP_ADDRESS 00000000    WS        0          M_CTRL    0
exc_happens 0    exc_report 0    EXCEPT: 000
----- NUEVO CICLO -----

```

```

----- NUEVO CICLO -----
Inst      E5D04002    d_inst  E5D04003    e_inst  E8383060    (PC)    00000038
m_inst    E02D3000    w_inst  E02D2000    m_data  000000FF    w_data  00000028
m_wbReg   00000000    w_wbReg 00000000    emw-WBR 100      m_addr  00000000
eA        1          eB        0          eC        0
busA      0          busB      0          busC      0
op A      00000000    op B      00000004    op C      00000000    AUXBUS   00000
Result    00000004    (e/m/w)      000      execute  1
JUMP      0          JUMP_ADDRESS 00000000    WS        3          M_CTRL   0
exc_happens 0      exc_report 0      EXCEPT: 000
----- NUEVO CICLO -----
Inst      E5D04002    d_inst  E5D04003    e_inst  EF3840A0    (PC)    00000038
m_inst    E8383060    w_inst  E02D3000    m_data  000000FF    w_data  000000FF
m_wbReg   00000004    w_wbReg 00000000    emw-WBR 010      m_addr  00000004
eA        1          eB        0          eC        0
busA      0          busB      0          busC      0
op A      00000000    op B      00000003    op C      00000000    AUXBUS   00000
Result    00000003    (e/m/w)      100      execute  1
JUMP      0          JUMP_ADDRESS 00000000    WS        2          M_CTRL   0
exc_happens 0      exc_report 0      EXCEPT: 000
----- NUEVO CICLO -----
Inst      E5D04002    d_inst  E5D04003    e_inst  EF3840A0    (PC)    00000038
m_inst    EF3840A0    w_inst  E8383060    m_data  000000FF    w_data  23456789
m_wbReg   00000004    w_wbReg 00000004    emw-WBR 001      m_addr  00000004
eA        1          eB        0          eC        0
busA      0          busB      0          busC      0
op A      00000000    op B      00000003    op C      00000000    AUXBUS   00000
Result    00000003    (e/m/w)      110      execute  1
JUMP      0          JUMP_ADDRESS 00000000    WS        1          M_CTRL   0
exc_happens 0      exc_report 0      EXCEPT: 000
----- NUEVO CICLO -----
Inst      E5D04002    d_inst  E5D04003    e_inst  EF3840A0    (PC)    00000038
m_inst    EF3840A0    w_inst  EF3840A0    m_data  000000FF    w_data  000000FF
m_wbReg   00000004    w_wbReg 00000004    emw-WBR 000      m_addr  00000004
eA        1          eB        0          eC        0
busA      0          busB      0          busC      0
op A      00000000    op B      00000003    op C      00000000    AUXBUS   00000
Result    00000003    (e/m/w)      111      execute  1
JUMP      0          JUMP_ADDRESS 00000000    WS        0          M_CTRL   0
exc_happens 0      exc_report 0      EXCEPT: 000
----- NUEVO CICLO -----
Inst      E5D04001    d_inst  E5D04002    e_inst  E83840A0    (PC)    0000003C
m_inst    EF3840A0    w_inst  EF3840A0    m_data  000000FF    w_data  000000FF
m_wbReg   00000004    w_wbReg 00000004    emw-WBR 000      m_addr  00000004
eA        1          eB        0          eC        0
busA      0          busB      0          busC      0
op A      00000004    op B      00000003    op C      00000000    AUXBUS   00000
Result    00000007    (e/m/w)      011      execute  1
JUMP      0          JUMP_ADDRESS 00000000    WS        0          M_CTRL   0
exc_happens 0      exc_report 0      EXCEPT: 000
----- NUEVO CICLO -----

```

Las instrucciones que se ejecutan a continuación son todas de memoria:

- (PC) = 30 ldr, pre, up, word, W=1 R0 R3 #4
- (PC) = 34 ldr, pre, up, byte, W=0 R0 R4 #3 (es la que provocó el interlocking con la anterior)
- (PC) = 38 ldr, pre, up, byte, W=0 R0 R4 #2
- (PC) = 3C ldr, pre, up, byte, W=0 R0 R4 #1
- (PC) = 40 ldr, pre, up, byte, W=0 R0 R4 #0

La primera de ellas es la que causo el interlocking anteriormente descrito con la segunda, y como utiliza modo preindexado (pre), y calcula la dirección absoluta sumando el valor de offset al registro base (up), y dado que R0=0, la dirección absoluta del acceso será 0x00000004. El dato almacenado en esa dirección de la RAM es 0x23456789 (basta con consultar el fichero ram.vhd para verificar que ésto es así).

Las cuatro últimas instrucciones, tienen como offset un valor inmediato. El registro base es R0 que inicialmente vale '0', y como W=1, no se escribe de vuelta el resultado de calcular la dirección absoluta en el registro base. La dirección absoluta del acceso será respectivamente 3, 2, 1 y 0, por lo que debe almacenarse en R4 (registro destino en todas ellas) en cada ciclo de reloj el byte referido por la desalineación de la palabra (una dirección está alineada con una palabra de memoria cuando es divisible por 4. Cuando se trata de bytes, la terminación 0 se refiere al menos significativo y la terminación 3 al más significativo, siempre que la numeración de bytes escogida sea *little-endian*)

Nótese que **debido al fenómeno de interlocking entre el primer ldr y el segundo, se introdujeron 3 bubbles así que a continuación observaremos después de almacenarse en R3 el valor 0x23456789, tres ciclos de reloj en los que no hay actividad alguna en los registros.** Ésto se debe a que los tres *bubbles* están pasando por la etapa *write* para salir definitivamente de la *pipeline*.

----- NUEVO CICLO -----							
R0	00000004	R1	00000004	R2	00000028	R15(PC)	00000038
R3	23456789	R4	00000000	R5	00000000		
R6	00000000	R7	00000000	R8	00000000	CPSR	000000D3
R9	00000000	R10	00000000	R11	00000000		
R12	00000000	R13	00000000	R14	00000000	SPSR_FIQ	00000000
R8FIQ	00000000	R9FIQ	00000000	R10FIQ	00000000	SPSR_IRQ	00000000
R11FIQ	00000000	R12FIQ	00000000	R13FIQ	00000000	SPSR_UND	00000000
R14FIQ	00000000	R13IRQ	00000000	R14IRQ	00000000	SPSR_ABT	00000000
R13SVC	00000000	R14SVC	00000000	R13ABT	00000000	SPSR_SVC	00000000
R14ABT	00000000	R13UND	00000000	R14UND	00000000		
busA	00000000	busB	00000003	busC	00000000	Result	00000003
----- NUEVO CICLO -----							
R0	00000004	R1	00000004	R2	00000028	R15(PC)	0000003C
R3	23456789	R4	00000000	R5	00000000		
R6	00000000	R7	00000000	R8	00000000	CPSR	000000D3
R9	00000000	R10	00000000	R11	00000000		
R12	00000000	R13	00000000	R14	00000000	SPSR_FIQ	00000000
R8FIQ	00000000	R9FIQ	00000000	R10FIQ	00000000	SPSR_IRQ	00000000
R11FIQ	00000000	R12FIQ	00000000	R13FIQ	00000000	SPSR_UND	00000000
R14FIQ	00000000	R13IRQ	00000000	R14IRQ	00000000	SPSR_ABT	00000000
R13SVC	00000000	R14SVC	00000000	R13ABT	00000000	SPSR_SVC	00000000
R14ABT	00000000	R13UND	00000000	R14UND	00000000		
busA	00000004	busB	00000003	busC	00000000	Result	00000007
----- NUEVO CICLO -----							
R0	00000004	R1	00000004	R2	00000028	R15(PC)	00000040
R3	23456789	R4	00000000	R5	00000000		
R6	00000000	R7	00000000	R8	00000000	CPSR	000000D3
R9	00000000	R10	00000000	R11	00000000		
R12	00000000	R13	00000000	R14	00000000	SPSR_FIQ	00000000
R8FIQ	00000000	R9FIQ	00000000	R10FIQ	00000000	SPSR_IRQ	00000000
R11FIQ	00000000	R12FIQ	00000000	R13FIQ	00000000	SPSR_UND	00000000
R14FIQ	00000000	R13IRQ	00000000	R14IRQ	00000000	SPSR_ABT	00000000
R13SVC	00000000	R14SVC	00000000	R13ABT	00000000	SPSR_SVC	00000000
R14ABT	00000000	R13UND	00000000	R14UND	00000000		
busA	00000004	busB	00000002	busC	00000000	Result	00000006
----- NUEVO CICLO -----							


```

----- NUEVO CICLO -(es un bubble)-----
R0      00000004    R1      00000004    R2      00000028    R15(PC)  00000044
R3      23456789    R4      00000000    R5      00000000
R6      00000000    R7      00000000    R8      00000000    CPSR      000000D3
R9      00000000    R10     00000000    R11     00000000
R12     00000000    R13     00000000    R14     00000000    SPSR_FIQ 00000000
R8FIQ   00000000    R9FIQ   00000000    R10FIQ  00000000    SPSR_IRQ 00000000
R11FIQ  00000000    R12FIQ  00000000    R13FIQ  00000000    SPSR_UND 00000000
R14FIQ  00000000    R13IRQ  00000000    R14IRQ  00000000    SPSR_ABT 00000000
R13SVC  00000000    R14SVC  00000000    R13ABT  00000000    SPSR_SVC 00000000
R14ABT  00000000    R13UND  00000000    R14UND  00000000
busA     00000004    busB     00000001    busC     00000000    Result    00000005
----- NUEVO CICLO -----
    Cuando el último bubble sale de la pipeline, comienza a almacenarse en
    R4 el byte seleccionado por el desalineamiento de la dirección de acceso a
    memoria. Como el registro destino de las cuatro instrucciones es R4, en cada
    ciclo se irá "sobrescribiendo" el valor cargado por la instrucción anterior.
----- NUEVO CICLO -----
R0      00000004    R1      00000004    R2      00000028    R15(PC)  00000048
R3      23456789    R4      00000023    R5      00000000
R6      00000000    R7      00000000    R8      00000000    CPSR      000000D3
R9      00000000    R10     00000000    R11     00000000
R12     00000000    R13     00000000    R14     00000000    SPSR_FIQ 00000000
R8FIQ   00000000    R9FIQ   00000000    R10FIQ  00000000    SPSR_IRQ 00000000
R11FIQ  00000000    R12FIQ  00000000    R13FIQ  00000000    SPSR_UND 00000000
R14FIQ  00000000    R13IRQ  00000000    R14IRQ  00000000    SPSR_ABT 00000000
R13SVC  00000000    R14SVC  00000000    R13ABT  00000000    SPSR_SVC 00000000
R14ABT  00000000    R13UND  00000000    R14UND  00000000
busA     00000004    busB     00000000    busC     00000000    Result    00000004
----- NUEVO CICLO -----
R0      00000004    R1      00000004    R2      00000028    R15(PC)  0000004C
R3      23456789    R4      00000045    R5      00000000
R6      00000000    R7      00000000    R8      00000000    CPSR      000000D3
R9      00000000    R10     00000000    R11     00000000
R12     00000000    R13     00000000    R14     00000000    SPSR_FIQ 00000000
R8FIQ   00000000    R9FIQ   00000000    R10FIQ  00000000    SPSR_IRQ 00000000
R11FIQ  00000000    R12FIQ  00000000    R13FIQ  00000000    SPSR_UND 00000000
R14FIQ  00000000    R13IRQ  00000000    R14IRQ  00000000    SPSR_ABT 00000000
R13SVC  00000000    R14SVC  00000000    R13ABT  00000000    SPSR_SVC 00000000
R14ABT  00000000    R13UND  00000000    R14UND  00000000
busA     00000028    busB     00000002    busC     00000000    Result    0000002A
----- NUEVO CICLO -----
R0      00000004    R1      00000004    R2      00000028    R15(PC)  00000050
R3      23456789    R4      00000067    R5      00000000
R6      00000000    R7      00000000    R8      00000000    CPSR      000000D3
R9      00000000    R10     00000000    R11     00000000
R12     00000000    R13     00000000    R14     00000000    SPSR_FIQ 00000000
R8FIQ   00000000    R9FIQ   00000000    R10FIQ  00000000    SPSR_IRQ 00000000
R11FIQ  00000000    R12FIQ  00000000    R13FIQ  00000000    SPSR_UND 00000000
R14FIQ  00000000    R13IRQ  00000000    R14IRQ  00000000    SPSR_ABT 00000000
R13SVC  00000000    R14SVC  00000000    R13ABT  00000000    SPSR_SVC 00000000
R14ABT  00000000    R13UND  00000000    R14UND  00000000
busA     00000028    busB     000000EE    busC     00000000    Result    000000EE
----- NUEVO CICLO -----
R0      00000004    R1      00000004    R2      00000028    R15(PC)  00000010
R3      23456789    R4      00000089    R5      00000000
R6      00000000    R7      00000000    R8      00000000    CPSR      000000D7
R9      00000000    R10     00000000    R11     00000000
R12     00000000    R13     00000000    R14     00000000    SPSR_FIQ 00000000
R8FIQ   00000000    R9FIQ   00000000    R10FIQ  00000000    SPSR_IRQ 00000000
R11FIQ  00000000    R12FIQ  00000000    R13FIQ  00000000    SPSR_UND 00000000
R14FIQ  00000000    R13IRQ  00000000    R14IRQ  00000000    SPSR_ABT 000000D3
R13SVC  00000000    R14SVC  00000000    R13ABT  00000000    SPSR_SVC 00000000
R14ABT  0000004C    R13UND  00000000    R14UND  00000000
busA     00000028    busB     000000EE    busC     00000000    Result    00000000
----- NUEVO CICLO -----

```

Nótese que **el penúltimo ciclo de reloj del listado anterior era 0x00000050, y el último en cambio es 0x00000010**. Se ha producido un salto en el programa a la dirección 0x00000010, que se corresponde con el vector de la excepción ABORT. Luego se tratará de una instrucción de memoria que en (PC)=50 se encontraba en fase de *memory*, y que entró en la *pipeline* en (PC)=44.

También se ha señalado en el último ciclo descrito que en R14_abt se ha almacenado la dirección a partir de la cual se calcula la dirección de retorno, y que el CPSR ha cambiado su valor (hemos entrado en modo ABORT).

Se trata de “LDR pre up word W=1 R2 R5 #2”. El problema está en que el registro base (R2) contiene el valor 0x00000028, y el offset es #2, luego la dirección del acceso es 0x00000028, que está fuera del rango de la RAM de pruebas (la RAM tiene datos entre 0x00000000 y 0x00000020).

Los eventos más importantes, resaltados en negrita, son:

- El gestor de interrupciones ha informado a la *pipeline* de que se ha producido una excepción, mediante la señal *exc_report*, y además le ha indicado que se trata de un ABORT gracias al bus EXCEPT. El *control global* informa a las etapas implicadas de que hay excepción por medio de la señal *exc_happens*. ante una excepción, se realizan muchas acciones que aquí no se aprecian, pero la más importante, es que se activan las señales *jump* y *jump_address*, indicando al controlador del PC de que la próxima instrucción no es la que secuencialmente sigue a la actual, sino la indicada en el bus *jump_address*.
- Se ha producido entonces un salto a (PC)=10 que a su vez contiene otra instrucción de salto a la rutina de atención del ABORT. Lo que pasa es que una instrucción de salto no se hace efectiva hasta que llega a la etapa *execute*. Por ello, hasta (PC)=18, no se activa de nuevo *jump* y *jump_address*
- Al siguiente ciclo de reloj, (PC) = *jump_address*

----- NUEVO CICLO -----							
Inst	E3B0F000	d_inst	EF000000	e_inst	E02D2000	(PC)	00000050
m_inst	E8385260	w_inst	E83840A0	m_data	000000FF	w_data	00000089
m_wbReg	0000002A	w_wbReg	00000004	emw-WBR	010	m_addr	0000002A
eA	0	eB	0	eC	0		
busA	0	busB	0	busC	0		
op A	00000028	op B	000000EE	op C	00000000	AUXBUS	00000
Result	000000EE	(e/m/w)	000	execute	1		
JUMP	1	JUMP_ADDRESS	00000010	WS	0	M_CTRL	0
exc_happens	1	exc_report	1	EXCEPT:	010		
----- NUEVO CICLO -----							
Inst	E3A0F054	d_inst	F3B0F000	e_inst	EF1F0000	(PC)	00000010
m_inst	EF2D2000	w_inst	E8385261	m_data	000000EE	w_data	FFFFFFFF
m_wbReg	0000002A	w_wbReg	0000002A	emw-WBR	000	m_addr	0000002A
eA	0	eB	0	eC	0		
busA	0	busB	0	busC	0		
op A	00000028	op B	000000EE	op C	00000000	AUXBUS	00000
Result	00000000	(e/m/w)	111	execute	1		
JUMP	0	JUMP_ADDRESS	00000000	WS	0	M_CTRL	0
exc_happens	0	exc_report	0	EXCEPT:	000		
----- NUEVO CICLO -----							

```

----- NUEVO CICLO -----
Inst      E3A0F000    d_inst    E3A0F054    e_inst    F02DF000    (PC)    00000014
m_inst    EF1F0000    w_inst    EF2D2000    m_data    000000EE    w_data    000000EE
m_wbReg    0000002A    w_wbReg    0000002A    emw-WBR    000    m_addr    0000002A
eA        0          eB        0          eC        0
busA      0          busB      0          busC      0
op A      00000028    op B      00000000    op C      00000000    AUXBUS    00000
Result    00000000    (e/m/w)    111    execute    0
JUMP      0          JUMP_ADDRESS    00000000    WS        0          M_CTRL    0
exc_happens    0    exc_report    0    EXCEPT:    000
----- NUEVO CICLO -----
Inst      E3A0F000    d_inst    E3A0F000    e_inst    E00DF000    (PC)    00000018
m_inst    FF2DF000    w_inst    EF1F0000    m_data    000000EE    w_data    000000EE
m_wbReg    0000002A    w_wbReg    0000002A    emw-WBR    000    m_addr    0000002A
eA        0          eB        0          eC        0
busA      0          busB      0          busC      0
op A      00000028    op B      00000054    op C      00000000    AUXBUS    00000
Result    00000054    (e/m/w)    011    execute    1
JUMP      1          JUMP_ADDRESS    00000054    WS        0          M_CTRL    0
exc_happens    0    exc_report    0    EXCEPT:    000
----- NUEVO CICLO -----
Inst      E3B02010    d_inst    F3A0F000    e_inst    EF0DF000    (PC)    00000054
m_inst    E00DF000    w_inst    FF2DF000    m_data    00000054    w_data    000000EE
m_wbReg    0000002A    w_wbReg    0000002A    emw-WBR    000    m_addr    0000002A
eA        0          eB        0          eC        0
busA      0          busB      0          busC      0
op A      00000028    op B      00000000    op C      00000000    AUXBUS    00000
Result    00000000    (e/m/w)    101    execute    1
JUMP      0          JUMP_ADDRESS    00000000    WS        0          M_CTRL    0
exc_happens    0    exc_report    0    EXCEPT:    000
----- NUEVO CICLO -----

```

En la rutina de servicio al ABORT se corrige el valor del registro base R2 por el valor 0x00000010, que está dentro del rango de direcciones válidas para la RAM. Además se realiza una operación absurda, “MOVE R13, #E”, para monitorizar el momento posterior a la resolución del problema (esto era útil al principio de las simulaciones, cuando no contábamos con el fichero *pipeline.dat*).

```

----- NUEVO CICLO -----
R0        00000004    R1        00000004    R2        00000028    R15(PC)    00000058
R3        23456789    R4        00000089    R5        00000000
R6        00000000    R7        00000000    R8        00000000    CPSR        000000D7
R9        00000000    R10       00000000    R11       00000000
R12       00000000    R13       00000000    R14       00000000    SPSR_FIQ    00000000
R8FIQ     00000000    R9FIQ     00000000    R10FIQ    00000000    SPSR_IRQ    00000000
R11FIQ    00000000    R12FIQ    00000000    R13FIQ    00000000    SPSR_UND    00000000
R14FIQ    00000000    R13IRQ    00000000    R14IRQ    00000000    SPSR_ABT    000000D3
R13SVC    00000000    R14SVC    00000000    R13ABT    00000000    SPSR_SVC    00000000
R14ABT    0000004C    R13UND    00000000    R14UND    00000000
busA      00000028    busB      00000000    busC      00000000    Result      00000000
----- NUEVO CICLO -----
R0        00000004    R1        00000004    R2        00000028    R15(PC)    0000005C
R3        23456789    R4        00000089    R5        00000000
R6        00000000    R7        00000000    R8        00000000    CPSR        000000D7
R9        00000000    R10       00000000    R11       00000000
R12       00000000    R13       00000000    R14       00000000    SPSR_FIQ    00000000
R8FIQ     00000000    R9FIQ     00000000    R10FIQ    00000000    SPSR_IRQ    00000000
R11FIQ    00000000    R12FIQ    00000000    R13FIQ    00000000    SPSR_UND    00000000
R14FIQ    00000000    R13IRQ    00000000    R14IRQ    00000000    SPSR_ABT    000000D3
R13SVC    00000000    R14SVC    00000000    R13ABT    00000000    SPSR_SVC    00000000
R14ABT    0000004C    R13UND    00000000    R14UND    00000000
busA      00000028    busB      00000010    busC      00000000    Result      00000010

```

----- NUEVO CICLO -----							
R0	00000004	R1	00000004	R2	00000028	R15(PC)	00000060
R3	23456789	R4	00000089	R5	00000000		
R6	00000000	R7	00000000	R8	00000000	CPSR	000000D7
R9	00000000	R10	00000000	R11	00000000		
R12	00000000	R13	00000000	R14	00000000	SPSR_FIQ	00000000
R8FIQ	00000000	R9FIQ	00000000	R10FIQ	00000000	SPSR_IRQ	00000000
R11FIQ	00000000	R12FIQ	00000000	R13FIQ	00000000	SPSR_UND	00000000
R14FIQ	00000000	R13IRQ	00000000	R14IRQ	00000000	SPSR_ABT	000000D3
R13SVC	00000000	R14SVC	00000000	R13ABT	00000000	SPSR_SVC	00000000
R14ABT	0000004C	R13UND	00000000	R14UND	00000000		
busA	00000028	busB	0000000E	busC	00000000	Result	0000000E
----- NUEVO CICLO -----							
(Nótese que en R14ABT se ha almacenado la dirección a partir de la cual se calcula la dirección de vuelta del ABORT, y que se ha cambiado a modo abort)							
----- NUEVO CICLO -----							
R0	00000004	R1	00000004	R2	00000028	R15(PC)	00000064
R3	23456789	R4	00000089	R5	00000000		
R6	00000000	R7	00000000	R8	00000000	CPSR	000000D7
R9	00000000	R10	00000000	R11	00000000		
R12	00000000	R13	00000000	R14	00000000	SPSR_FIQ	00000000
R8FIQ	00000000	R9FIQ	00000000	R10FIQ	00000000	SPSR_IRQ	00000000
R11FIQ	00000000	R12FIQ	00000000	R13FIQ	00000000	SPSR_UND	00000000
R14FIQ	00000000	R13IRQ	00000000	R14IRQ	00000000	SPSR_ABT	000000D3
R13SVC	00000000	R14SVC	00000000	R13ABT	00000000	SPSR_SVC	00000000
R14ABT	0000004C	R13UND	00000000	R14UND	00000000		
busA	0000004C	busB	00000008	busC	00000000	Result	00000044
----- NUEVO CICLO -----							
R0	00000004	R1	00000004	R2	00000010	R15(PC)	00000044
R3	23456789	R4	00000089	R5	00000000		
R6	00000000	R7	00000000	R8	00000000	CPSR	000000D3
R9	00000000	R10	00000000	R11	00000000		
R12	00000000	R13	00000000	R14	00000000	SPSR_FIQ	00000000
R8FIQ	00000000	R9FIQ	00000000	R10FIQ	00000000	SPSR_IRQ	00000000
R11FIQ	00000000	R12FIQ	00000000	R13FIQ	00000000	SPSR_UND	00000000
R14FIQ	00000000	R13IRQ	00000000	R14IRQ	00000000	SPSR_ABT	000000D3
R13SVC	00000000	R14SVC	00000000	R13ABT	00000000	SPSR_SVC	00000000
R14ABT	0000004C	R13UND	00000000	R14UND	00000000		
busA	0000004C	busB	00000006	busC	00000000	Result	00000006
----- NUEVO CICLO -----							
R0	00000004	R1	00000004	R2	00000010	R15(PC)	00000048
R3	23456789	R4	00000089	R5	00000000		
R6	00000000	R7	00000000	R8	00000000	CPSR	000000D3
R9	00000000	R10	00000000	R11	00000000		
R12	00000000	R13	00000000	R14	00000000	SPSR_FIQ	00000000
R8FIQ	00000000	R9FIQ	00000000	R10FIQ	00000000	SPSR_IRQ	00000000
R11FIQ	00000000	R12FIQ	00000000	R13FIQ	00000000	SPSR_UND	00000000
R14FIQ	00000000	R13IRQ	00000000	R14IRQ	00000000	SPSR_ABT	000000D3
R13SVC	0000000E	R14SVC	00000000	R13ABT	00000000	SPSR_SVC	00000000
R14ABT	0000004C	R13UND	00000000	R14UND	00000000		
busA	0000004C	busB	00000007	busC	00000000	Result	00000007
----- NUEVO CICLO -----							
R0	00000004	R1	00000004	R2	00000010	R15(PC)	0000004C
R3	23456789	R4	00000089	R5	00000000		
R6	00000000	R7	00000000	R8	00000000	CPSR	000000D3
R9	00000000	R10	00000000	R11	00000000		
R12	00000000	R13	00000000	R14	00000000	SPSR_FIQ	00000000
R8FIQ	00000000	R9FIQ	00000000	R10FIQ	00000000	SPSR_IRQ	00000000
R11FIQ	00000000	R12FIQ	00000000	R13FIQ	00000000	SPSR_UND	00000000
R14FIQ	00000000	R13IRQ	00000000	R14IRQ	00000000	SPSR_ABT	000000D3
R13SVC	0000000E	R14SVC	00000000	R13ABT	00000000	SPSR_SVC	00000000
R14ABT	0000004C	R13UND	00000000	R14UND	00000000		
busA	00000010	busB	00000002	busC	00000000	Result	00000012
----- NUEVO CICLO -----							

Después de resolver el problema se vuelve de nuevo a (PC)=44, que es la instrucción que causó el ABORT, y tras cuatro ciclos de reloj, realiza de nuevo el acceso a memoria, ésta vez con éxito. En R5 vemos el valor almacenado.

Como la dirección de acceso era HEX(10+2), tenemos que se accede a la dirección hex(10) que está alineada con un word, y como tenemos 2 bytes de desalineación, se intercambian los dos bytes menos significativos por los dos bytes más significativos, así que en lugar de almacenar FDB97531, que es el valor almacenado en la posición 0x00000010 de memoria, vemos que R5 contiene el valor 7531FDB9.

----- NUEVO CICLO -----							
R0	00000004	R1	00000004	R2	00000010	R15(PC)	00000050
R3	23456789	R4	00000089	R5	00000000		
R6	00000000	R7	00000000	R8	00000000	CPSR	000000D3
R9	00000000	R10	00000000	R11	00000000		
R12	00000000	R13	00000000	R14	00000000	SPSR_FIQ	00000000
R8FIQ	00000000	R9FIQ	00000000	R10FIQ	00000000	SPSR_IRQ	00000000
R11FIQ	00000000	R12FIQ	00000000	R13FIQ	00000000	SPSR_UND	00000000
R14FIQ	00000000	R13IRQ	00000000	R14IRQ	00000000	SPSR_ABT	000000D3
R13SVC	0000000E	R14SVC	00000000	R13ABT	00000000	SPSR_SVC	00000000
R14ABT	0000004C	R13UND	00000000	R14UND	00000000		
busA	00000010	busB	000000EE	busC	00000000	Result	000000EE
----- NUEVO CICLO -----							
R0	00000004	R1	00000004	R2	00000010	R15(PC)	00000054
R3	23456789	R4	00000089	R5	00000000		
R6	00000000	R7	00000000	R8	00000000	CPSR	000000D3
R9	00000000	R10	00000000	R11	00000000		
R12	00000000	R13	00000000	R14	00000000	SPSR_FIQ	00000000
R8FIQ	00000000	R9FIQ	00000000	R10FIQ	00000000	SPSR_IRQ	00000000
R11FIQ	00000000	R12FIQ	00000000	R13FIQ	00000000	SPSR_UND	00000000
R14FIQ	00000000	R13IRQ	00000000	R14IRQ	00000000	SPSR_ABT	000000D3
R13SVC	0000000E	R14SVC	00000000	R13ABT	00000000	SPSR_SVC	00000000
R14ABT	0000004C	R13UND	00000000	R14UND	00000000		
busA	00000010	busB	000000EE	busC	00000000	Result	00000000
----- NUEVO CICLO -----							
R0	00000004	R1	00000004	R2	00000012	R15(PC)	00000008
R3	23456789	R4	00000089	R5	7531FDB9		
R6	00000000	R7	00000000	R8	00000000	CPSR	000000D3
R9	00000000	R10	00000000	R11	00000000		
R12	00000000	R13	00000000	R14	00000000	SPSR_FIQ	00000000
R8FIQ	00000000	R9FIQ	00000000	R10FIQ	00000000	SPSR_IRQ	00000000
R11FIQ	00000000	R12FIQ	00000000	R13FIQ	00000000	SPSR_UND	00000000
R14FIQ	00000000	R13IRQ	00000000	R14IRQ	00000000	SPSR_ABT	000000D3
R13SVC	0000000E	R14SVC	00000050	R13ABT	00000000	SPSR_SVC	000000D3
R14ABT	0000004C	R13UND	00000000	R14UND	00000000		
busA	00000010	busB	00000000	busC	00000000	Result	00000000
----- NUEVO CICLO -----							

Nos llama la atención el hecho de que se produzca otro salto a otro vector de interrupción, esta vez el 0x00000008, que se corresponde con el vector de la excepción SWI.

Esto es debido a que la instrucción correspondiente a (PC) = 4C es una llamada al sistema SWI.

La rutina de atención a la excepción realiza las siguientes operaciones:

- MOVE R6, #6
- MOVE R7, #7
- MOVE R8, #8

Omitimos los ciclos referentes a los saltos, primero a 0x00000008 (vector de excepción SWI), que a su vez contiene una instrucción de salto a 0x00000060, pues son procedimientos similares a los explicados con ABORT.

El caso es que hasta (PC)=74 no se escribe el registro R6, un ciclo después R7, y finalmente en el siguiente ciclo R8.

Posteriormente se devuelve el control, volviendo a la instrucción que inmediatamente sigue a la instrucción SWI (causante de la excepción), y que se encuentra en 0x00000050.

Esta instrucción resulta ser "MOVS R15, #0" que es un salto al vector de reset (inicio del programa). El salto no se ejecuta de forma efectiva hasta (PC)=58 y entonces, el programa comenzaría de nuevo su ejecución.

----- NUEVO CICLO -----						
R0	00000004	R1	00000004	R2	000000EE	R15(PC) 00000074
R3	23456789	R4	00000089	R5	7531FDB9	
R6	00000006	R7	00000000	R8	00000000	CPSR 000000D3
R9	00000000	R10	00000000	R11	00000000	
R12	00000000	R13	00000000	R14	00000000	SPSR_FIQ 00000000
R8FIQ	00000000	R9FIQ	00000000	R10FIQ	00000000	SPSR_IRQ 00000000
R11FIQ	00000000	R12FIQ	00000000	R13FIQ	00000000	SPSR_UND 00000000
R14FIQ	00000000	R13IRQ	00000000	R14IRQ	00000000	SPSR_ABT 000000D3
R13SVC	0000000E	R14SVC	00000050	R13ABT	00000000	SPSR_SVC 000000D3
R14ABT	0000004C	R13UND	00000000	R14UND	00000000	
busA	00000010	busB	00000050	busC	00000000	Result 00000050
----- NUEVO CICLO -----						
R0	00000004	R1	00000004	R2	000000EE	R15(PC) 00000050
R3	23456789	R4	00000089	R5	7531FDB9	
R6	00000006	R7	00000007	R8	00000000	CPSR 000000D3
R9	00000000	R10	00000000	R11	00000000	
R12	00000000	R13	00000000	R14	00000000	SPSR_FIQ 00000000
R8FIQ	00000000	R9FIQ	00000000	R10FIQ	00000000	SPSR_IRQ 00000000
R11FIQ	00000000	R12FIQ	00000000	R13FIQ	00000000	SPSR_UND 00000000
R14FIQ	00000000	R13IRQ	00000000	R14IRQ	00000000	SPSR_ABT 000000D3
R13SVC	0000000E	R14SVC	00000050	R13ABT	00000000	SPSR_SVC 000000D3
R14ABT	0000004C	R13UND	00000000	R14UND	00000000	
busA	00000010	busB	00000000	busC	00000000	Result 00000000
----- NUEVO CICLO -----						
R0	00000004	R1	00000004	R2	000000EE	R15(PC) 00000054
R3	23456789	R4	00000089	R5	7531FDB9	
R6	00000006	R7	00000007	R8	00000008	CPSR 000000D3
R9	00000000	R10	00000000	R11	00000000	
R12	00000000	R13	00000000	R14	00000000	SPSR_FIQ 00000000
R8FIQ	00000000	R9FIQ	00000000	R10FIQ	00000000	SPSR_IRQ 00000000
R11FIQ	00000000	R12FIQ	00000000	R13FIQ	00000000	SPSR_UND 00000000
R14FIQ	00000000	R13IRQ	00000000	R14IRQ	00000000	SPSR_ABT 000000D3
R13SVC	0000000E	R14SVC	00000050	R13ABT	00000000	SPSR_SVC 000000D3
R14ABT	0000004C	R13UND	00000000	R14UND	00000000	
busA	00000010	busB	00000000	busC	00000000	Result 00000000
----- NUEVO CICLO -----						
R0	00000004	R1	00000004	R2	000000EE	R15(PC) 00000058
R3	23456789	R4	00000089	R5	7531FDB9	
R6	00000006	R7	00000007	R8	00000008	CPSR 000000D3
R9	00000000	R10	00000000	R11	00000000	
R12	00000000	R13	00000000	R14	00000000	SPSR_FIQ 00000000
R8FIQ	00000000	R9FIQ	00000000	R10FIQ	00000000	SPSR_IRQ 00000000
R11FIQ	00000000	R12FIQ	00000000	R13FIQ	00000000	SPSR_UND 00000000
R14FIQ	00000000	R13IRQ	00000000	R14IRQ	00000000	SPSR_ABT 000000D3
R13SVC	0000000E	R14SVC	00000050	R13ABT	00000000	SPSR_SVC 000000D3
R14ABT	0000004C	R13UND	00000000	R14UND	00000000	
busA	00000010	busB	00000000	busC	00000000	Result 00000000
----- NUEVO CICLO -----						
R0	00000004	R1	00000004	R2	000000EE	R15(PC) 00000000

Accesos a Memoria

A continuación vamos a incluir el fichero *memory.dat* completo, pues es bastante corto, y muy explicativo.

Listamos las instrucciones de memoria que aparecían en el programa:

- (PC) = 30 ldr, pre, up, word, W=1 R0 R3 #4
- (PC) = 34 ldr, pre, up, byte, W=0 R0 R4 #3 (es la que provocó el interlocking con la anterior)
- (PC) = 38 ldr, pre, up, byte, W=0 R0 R4 #2
- (PC) = 3C ldr, pre, up, byte, W=0 R0 R4 #1
- (PC) = 40 ldr, pre, up, byte, W=0 R0 R4 #0
- (PC) = 44 ldr, pre, up, word, W=1 R2 R5 #2 (la primera vez provoca ABORT)

Es importante destacar cómo accede el CORE del ARM a la memoria en lectura.

- La dirección siempre está alineada con WORD
- Siempre lee la palabra completa (32 bits)
- Si la lectura era de un byte, se elige el byte correspondiente a los últimos dos bits de la dirección (es decir, el número de bytes que está desalineada la dirección)
- Si la lectura era de un halfword, se carga el correspondiente al penúltimo bit (es decir, si estamos alineados con el word, o estamos un bytes desalineados, cogemos el halfword menos significativo, y si estamos desalineados dos o tres bytes, cogemos el halfword más significativo).
- En cambio, si la lectura es de un word, se lee el word completo igualmente, pero en el registro se carga rotado tantos bytes como desalineada esté la dirección.

(PC) = 30 ldr, pre, up, word, W=1 R0 R3 #4
Dirección Calculada 4
Dirección de Acceso 4

```
----- NUEVO CICLO -----
VAddress 00000004  SIZE    11                (PC)    00000038
WData     00000000  w_enable 0
RData     23456789  r_enable 1
----- NUEVO CICLO -----
```

(PC) = 34 ldr, pre, up, byte, W=0 R0 R4 #3
Dirección Calculada 7
Dirección de Acceso 4
Byte Cargado en Registro 4° (MSB)

```
----- NUEVO CICLO -----
VAddress 00000004  SIZE    01                (PC)    00000040
WData     00000000  w_enable 0
RData     23456789  r_enable 1
----- NUEVO CICLO -----
```

```
(PC) = 38  ldr, pre, up, byte, W=0  R0  R4 #2
Dirección Calculada ..... 6
Dirección de Acceso ..... 4
Byte Cargado en Registro ..... 3°
```

```
----- NUEVO CICLO -----
VAddress 00000004  SIZE    01          (PC)    00000044
WData    00000000  w_enable 0
RData    23456789  r_enable 1
----- NUEVO CICLO -----
```

```
(PC) = 38  ldr, pre, up, byte, W=0  R0  R4 #1
Dirección Calculada ..... 5
Dirección de Acceso ..... 4
Byte Cargado en Registro ..... 2°
```

```
----- NUEVO CICLO -----
VAddress 00000004  SIZE    01          (PC)    00000048
WData    00000000  w_enable 0
RData    23456789  r_enable 1
----- NUEVO CICLO -----
```

```
(PC) = 38  ldr, pre, up, byte, W=0  R0  R4 #0
Dirección Calculada ..... 4
Dirección de Acceso ..... 4
Byte Cargado en Registro ..... 1° (lsb)
```

```
----- NUEVO CICLO -----
VAddress 00000004  SIZE    01          (PC)    0000004C
WData    00000000  w_enable 0
RData    23456789  r_enable 1
----- NUEVO CICLO -----
```

```
(PC) = 44  ldr, pre, up, word, W=1  R2  R5 #2
Dirección Calculada ..... 2A
Dirección de Acceso ..... 28
GENERA ABORT (NO ES UNA DIR. VALIDA)
```

```
----- NUEVO CICLO -----
VAddress 00000028  SIZE    11          (PC)    00000050
WData    00000000  w_enable 0
RData    FFFFFFFF  r_enable 1
----- NUEVO CICLO -----
```

```
(PC) = 44  ldr, pre, up, word, W=1  R2  R5 #2
Dirección Calculada ..... 12
Dirección de Acceso ..... 10
N° de bytes a rotar ..... 2
```

```
----- NUEVO CICLO -----
VAddress 00000010  SIZE    11          (PC)    00000050
WData    00000000  w_enable 0
RData    FDB97531  r_enable 1
----- NUEVO CICLO -----
```


6.8 Simulaciones del prototipo

No vamos a entrar en detalle, pues el capítulo siguiente se refiere íntegramente al proceso de diseño, simulación, implementación y prueba del prototipo.

Simplemente mencionar que se realizó una simulación a escala temporal, es decir, cada uno de los bucles que componía el programa se repitió un número de veces 100 veces inferior al real, puesto que de lo contrario necesitaríamos horas de simulación para obtener un resultado visible.

Lo que se hizo fue comprobar que realmente el prototipo funcionaba correctamente antes de abordar todo el trabajo de implementación y prueba en la FPGA.

Queremos resaltar una situación que se manifestó en las simulaciones del controlador de la placa de displays:

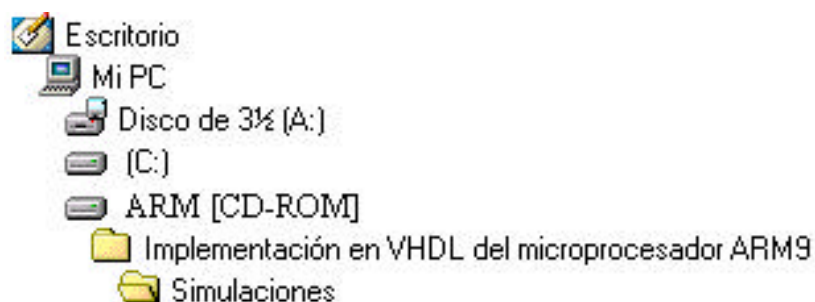
- Resulta que las señales que activan el display a controlar son activas a nivel alto
- En cambio, las señales que iluminan un segmento concreto dentro de un display, son activas a nivel bajo.
- Para que la RAM que contenía los valores a representar fuese más genérica, se uniformizó el criterio, de modo que los valores representados en la RAM fueron calculados suponiendo que todas las señales se activaban a nivel bajo.
- Por tanto, se hace necesario que el controlador (*driver*) de la placa, implementado junto con el CORE en la FPGA, invierta los 8 bits menos significativos de su registro de datos, para adaptar las señales de control al formato esperado por la placa de *displays*.
- En la simulación, se detectó que este último paso se había omitido

Además existía un problema más difícil de detectar, referente a la dirección de memoria asignada al periférico:

- El periférico tenía asignada inicialmente la dirección 0x000000FF, utilizada para escribir el registro de datos del display a través de una instrucción **STR** de tamaño WORD (es decir, al realizar un acceso **en escritura** a la dirección asignada, se escribe el registro del periférico)
- Resulta que el CORE asegura que la dirección en la interfaz de memoria esté alineada con WORD, de modo que fuerza los dos bits menos significativos a “00”. Esto implica que cuando se intenta acceder a la dirección 0x000000FF, el CORE utiliza la dirección 0x000000FC, que sí está alineada.
- Esto significa que el periférico no se enteraría de que se intenta escribir en su registro interno de datos, y no se mostraría nada por pantalla.
- El problema se solucionó simplemente asignándole la dirección 0x000000FC, en lugar de la 0x000000FF.

6.9 Ubicación y descripción de los archivos de tests

Todos los archivos referentes a las simulaciones y test, se incluyen en el CD-ROM. La ruta de acceso es la siguiente:

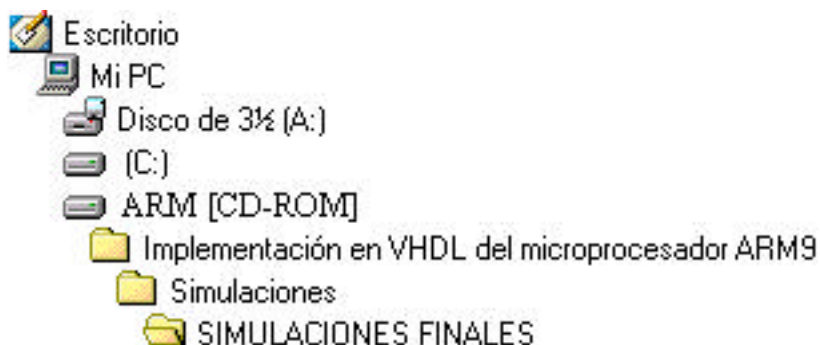


Dentro del directorio “Simulaciones” se pueden encontrar dos subdirectorios. En ellos se encuentran almacenados todos los ficheros de entrada y de salida utilizados en todas y cada una de las simulaciones que se han llevado a cabo en el desarrollo de este proyecto.

Todos y cada uno de los subdirectorios correspondientes a una simulación en concreto, cuentan al menos con los siguientes ficheros:

- Fichero de Entrada (.txt): a partir de él se puede generar una rom (siempre debe llamarse ROM.vhd), utilizando la aplicación **Txt2vhd.exe**, creada específicamente para este proyecto
- Ficheros de Salida (.dat): al menos siempre aparecerán *traza.dat* y *pipeline.dat*, aunque normalmente también nos encontraremos con el fichero *memory.dat*

Las simulaciones definitivas explicadas a lo largo del Capítulo 6, se encuentran ubicadas en la siguiente ruta:



A continuación se adjunta una tabla, donde se detallan los subdirectorios almacenados en esta ruta, y la descripción de cada uno de los ficheros que contiene.

DIRECTORIO	ARCHIVO	DESCRIPCIÓN
DATA_PSR	data_psr.txt	fichero fuente para la generación de la ROM
	traza.dat	muestra el contenido del banco de registros
	pipeline.dat	muestra el valor de los registros internos de la pipeline
EXCEPTION	exception.txt	fichero fuente para la generación de la ROM
	traza.dat	muestra el contenido del banco de registros
	pipeline.dat	muestra el valor de los registros internos de la pipeline
	memory.dat	muestra la interfaz de memoria en cada acceso
HALFWORD	halfword.txt	fichero fuente para la generación de la ROM
	traza.dat	muestra el contenido del banco de registros
	pipeline.dat	muestra el valor de los registros internos de la pipeline
	memory.dat	muestra la interfaz de memoria en cada acceso
LDR_STR	ldr_str.txt	fichero fuente para la generación de la ROM
	traza.dat	muestra el contenido del banco de registros
	pipeline.dat	muestra el valor de los registros internos de la pipeline
	memory.dat	muestra la interfaz de memoria en cada acceso
MULTIPLY	multiply.txt	fichero fuente para la generación de la ROM
	traza.dat	muestra el contenido del banco de registros
	pipeline.dat	muestra el valor de los registros internos de la pipeline
MULTLONG	multlong.txt	fichero fuente para la generación de la ROM
	traza.dat	muestra el contenido del banco de registros
	pipeline.dat	muestra el valor de los registros internos de la pipeline
SALTOS	saltos.txt	fichero fuente para la generación de la ROM
	traza.dat	muestra el contenido del banco de registros
	pipeline.dat	muestra el valor de los registros internos de la pipeline
SHIFTER	shifter.txt	fichero fuente para la generación de la ROM
	traza.dat	muestra el contenido del banco de registros
	pipeline.dat	muestra el valor de los registros internos de la pipeline
SWP	swp.txt	fichero fuente para la generación de la ROM
	traza.dat	muestra el contenido del banco de registros
	pipeline.dat	muestra el valor de los registros internos de la pipeline
	memory.dat	muestra la interfaz de memoria en cada acceso
COMPLETO	completo.txt	fichero fuente para la generación de la ROM
	traza.dat	muestra el contenido del banco de registros
	pipeline.dat	muestra el valor de los registros internos de la pipeline
	memory.dat	muestra la interfaz de memoria en cada acceso
PROTOTIPO	ROM.txt	fichero fuente para la generación de la ROM
	ROM.vhd	ROM generada a partir del fichero ROM.txt
	ROM2.vhd	ROM de datos necesaria para la implementación
	prototip.dat	Muestra la interfaz con el periférico en cada acceso
MSR_MRS	msr_mrs.txt	fichero fuente para la generación de la ROM
	traza.dat	muestra el contenido del banco de registros
	pipeline.dat	muestra el valor de los registros internos de la pipeline
EXCEPTION2	serie FIQ,IRQ	múltiples ficheros para comprobar el control de excepciones
	serie ABORT	verifica el comportamiento ante secuencias críticas de inst.

7

Implementación

Para completar un trabajo de diseño de las características y dimensiones del presente proyecto, es necesario realizar un prototipo de pruebas, que demuestre físicamente que el funcionamiento del CORE es correcto. En este apartado se expone todo lo referente al prototipo de pruebas realizado, abarcando su concepción, diseño, simulación, implementación y test de campo.

7.1 Comentarios sobre el tamaño inicial

7.2 Planteamiento del prototipo

7.3 Diseño hardware del prototipo

7.4 Realización del Programa

7.4.1 Memoria de Datos

7.4.2 Calculando los parámetros del programa

7.5 Implementación en la Virtex 800

7.6 Prueba final

7.1 Comentarios sobre el tamaño inicial

La primera implementación del CORE se realizó para la Virtex 300, y tuvo un carácter meramente informativo. Simplemente se pretendía tener una idea aproximada del porcentaje de ocupación sobre la FPGA, y determinar así el espacio disponible para la implementación de ROMs en la propia FPGA y/o la adición de un periférico para la realización del prototipo final.

Solamente la ALU ocupaba aproximadamente un 70% de la Virtex 300, **sin incluir el multiplicador combinacional**. Lo elevado del porcentaje de ocupación se puede explicar si se considera el hecho de que estamos utilizando buses de 32 bits, y la complejidad de la ALU del ARM8/9 es bastante elevada: nótese que posee un shifter operando en serie con los sumadores, y que se realizan operaciones de todo tipo, tanto aritméticas como lógicas.

El diseño completo ocupaba inicialmente un 266% de la FPGA, pero éste porcentaje se debía principalmente a un error en el código VHDL que hacía que el software implementase dos COREs, cada uno de los cuales escribía un registro distinto, y los multiplexase, en lugar de multiplexar los dos registros.

Tras solucionar el problema, se realizaron diversas simplificaciones, consiguiendo así una ocupación de aproximadamente el 110% de la Virtex 300, pero sin ROM y sin periférico.

Realmente el diseño no es “tan grande”; lo que ocurre es que estamos intentando implementar una arquitectura de 32 bits, que contiene buses de 32 bits, multiplexores de dichos buses, registros de 32 bits, etc. Al “rutar” todas estas entidades anteriormente mencionadas en el dispositivo, el diseño es bastante ineficiente (no existen multiplexores de 32 bits en la FPGA, luego no se pueden aprovechar los que contiene ya hechos, y se utilizan muchos SLICES para crear multiplexores de 32 bits). Podríamos decir que la implementación de un diseño con arquitectura de 32 bits en una FPGA de estas características, resulta bastante ineficiente.

Finalmente se optó por llevar a cabo la implementación en la Virtex800. Las primeras pruebas con el CORE únicamente, estuvieron en torno al 38% de ocupación de SLICES (que resulta ser el recurso limitante de nuestro diseño).

Así que contábamos con espacio de sobra para implementar ROMs, periféricos, etc.

De todos modos, sería bastante aclarativo el realizar una implementación en una Virtex 2, que está ideada para soportar arquitecturas con buses mayores a los que soporta la Virtex 800, dando así lugar a implementaciones más óptimas. Incluye multiplexores propios de más bits, de modo que no se vería obligada a utilizar SLICES para crear multiplexores, etc.

7.2 Planteamiento del prototipo

Nuestra idea inicial consistía en dotar al CORE de un periférico que le permitiese mostrar resultados al exterior.

Además, se requería que presentase una circuitería externa sencilla (pues el alcance de este proyecto no incluye realizar ninguna placa), aunque el controlador podría ser todo lo complejo que se desee, pues se puede implementar en la FPGA.

Teniendo en cuenta el hardware disponible en lo que a tarjetas se refiere, surge la idea de crear un periférico que muestre un mensaje a través de una serie de displays de 7 segmentos con punto decimal.

Para implementar este periférico se utiliza:

- ❑ una placa que contiene cuatro displays de 7 segmentos, y cuyo funcionamiento describiremos a continuación
- ❑ un controlador que se añadirá al CORE (se escribirá un nuevo fichero VHDL)

Además del periférico, necesitamos dos ROMs

- ❑ ROM: memoria de instrucciones que contiene el programa a ejecutar
- ❑ ROM2: contiene una tabla con los caracteres que deseamos imprimir en el periférico, y dos constantes que el programa utilizará para dividir la frecuencia del reloj y ajustar así el refresco de los displays de modo que el mensaje mostrado sea inteligible

El periférico deberá ir mapeado en memoria, de manera que debe asignársele una dirección de 32 bits fuera del rango de la memoria de datos utilizada por el programa para realizar sus cálculos (es decir, ROM2).

Funcionamiento de la placa de displays

La placa cuenta con cuatro displays, cada uno de los cuales tiene siete segmentos y un punto decimal.

Para controlar el encendido de cada uno de los segmentos y el punto, existe un bus de 8 bits, que es el mismo para todos los displays, luego en un instante determinado, sólo puede encenderse un display.

Para seleccionar el display a controlar por el bus anterior, existen 4 pines. Estos pines son activos a nivel alto. En cambio, los segmentos y el punto de los displays son activos a nivel bajo.

Por tanto, si queremos escribir algo en las cuatro pantallas, hay que refrescarlas constantemente para que dé la sensación de que imagen fija.

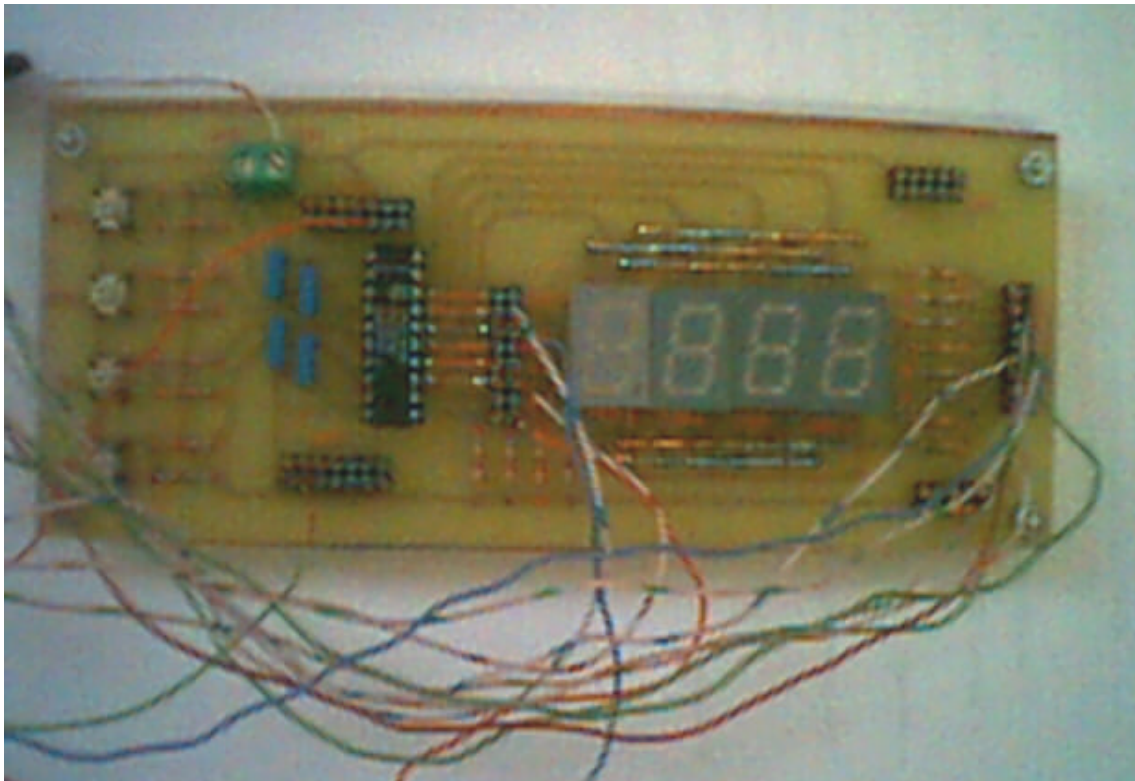


Figura 7-1: vista de la placa de displays

Para utilizar la placa de displays, únicamente es necesario construir un bus de 12 bits, *ENTRADA[11:0]*, y tener claro qué segmento controla cada uno de los 8 bits menos significativos, y qué pantalla selecciona cada uno de los 4 bits más significativos. Es importante recordar que las señales a,...,h son activas a nivel bajo, mientras que las señales d3...d0 lo son a nivel alto

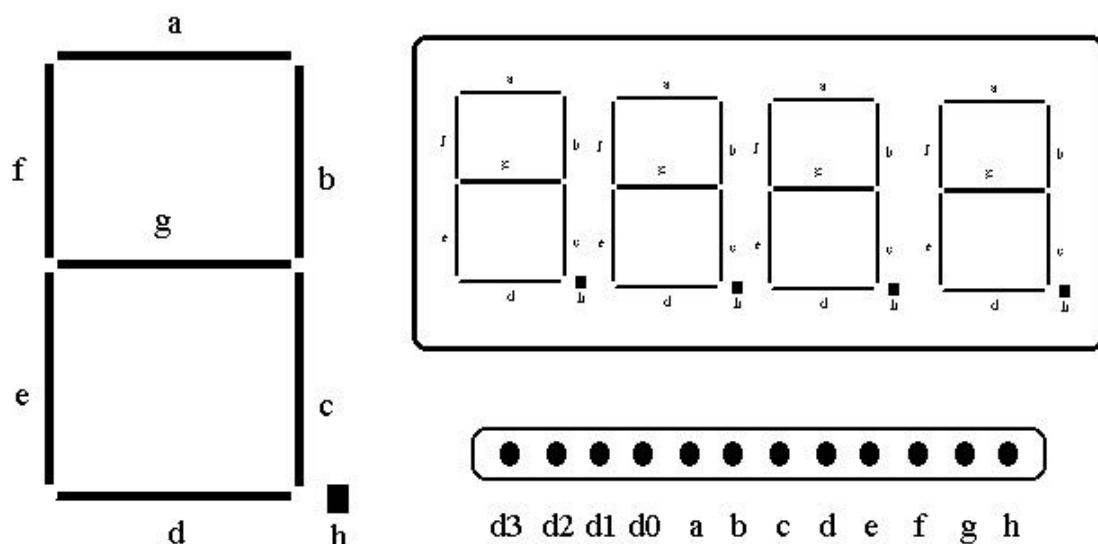


Figura 7-2: numeración de los segmentos y descripción de pines

Planteamiento del Refresco de los Displays para el diseño del Periférico

Nuestro objetivo es poder mostrar un mensaje utilizando como pantalla los cuatro displays de los que disponemos. Pero las señales de control de la placa sólo nos permite tener activo un único display en un instante determinado.

Por tanto, nos vemos obligados a realizar un refresco, de manera que cada cierto tiempo cambiaremos de un display a otro mediante las señales $ds3\dots ds0$, y mientras tanto, mantendremos controlado el display actual, activando las señales convenientes para iluminar los segmentos que conforman el carácter deseado.

Ésto se repetirá un número finito de veces, hasta que transcurrido cierto tiempo, debemos sustituir en cada display el símbolo por el del display de su izquierda, de forma que habrá que introducir un nuevo carácter en el display más a la izquierda, y desaparecerá el carácter que mostraba el display más a la derecha. Hemos, por tanto, completado el desplazamiento de un carácter del mensaje.

Si todo el proceso anterior se repite un número de veces suficiente, tendremos un periférico que es capaz de mostrar cualquier mensaje con sólo cuatro displays

Luego debemos realizar tres bucles:

- ❑ un bucle que mantiene activo un display durante el tiempo correspondiente.
- ❑ un segundo bucle que repite el bucle anterior para cada display, durante un cierto tiempo.
- ❑ un tercer bucle infinito, para que el funcionamiento del display se mantenga indefinidamente

7.3 Diseño hardware del prototipo

Esta fase del diseño se refiere al controlador del periférico que se implementará junto con el CORE del ARM9 en la FPGA.

El objetivo del controlador es gobernarla placa de displays, por lo que las características expuestas en el apartado anterior se convierten automáticamente en las especificaciones de diseño del controlador.

Los elementos que componen el controlador serán:

- Un bus de entrada de datos de 32 bits
- Un bus de direcciones de 32 bits
- Una señal `w_enable`
- Un registro de escritura de datos de 32 bits
- Un registro que contiene una dirección, `D_ADDRESS`, que es parametrizable y se puede cambiar para cada implementación.
- Una lógica de comparación: si el valor del bus de direcciones coincide con `D_ADDRESS` y `w_enable=1`, se carga en el registro de datos el contenido del bus de entrada de datos

El diseño del controlador es bastante simple, lo que se traducirá en un aumento de la complejidad del *driver software*, es decir, del programa que gobierna el funcionamiento de todo el periférico.

Para que el diseño sea lo más genérico posible, la salida del controlador también es un bus de 32 bits, aunque el bus de *ENTRADA* de la placa de displays cuente únicamente con 12 bits (ver Figura 7-2).

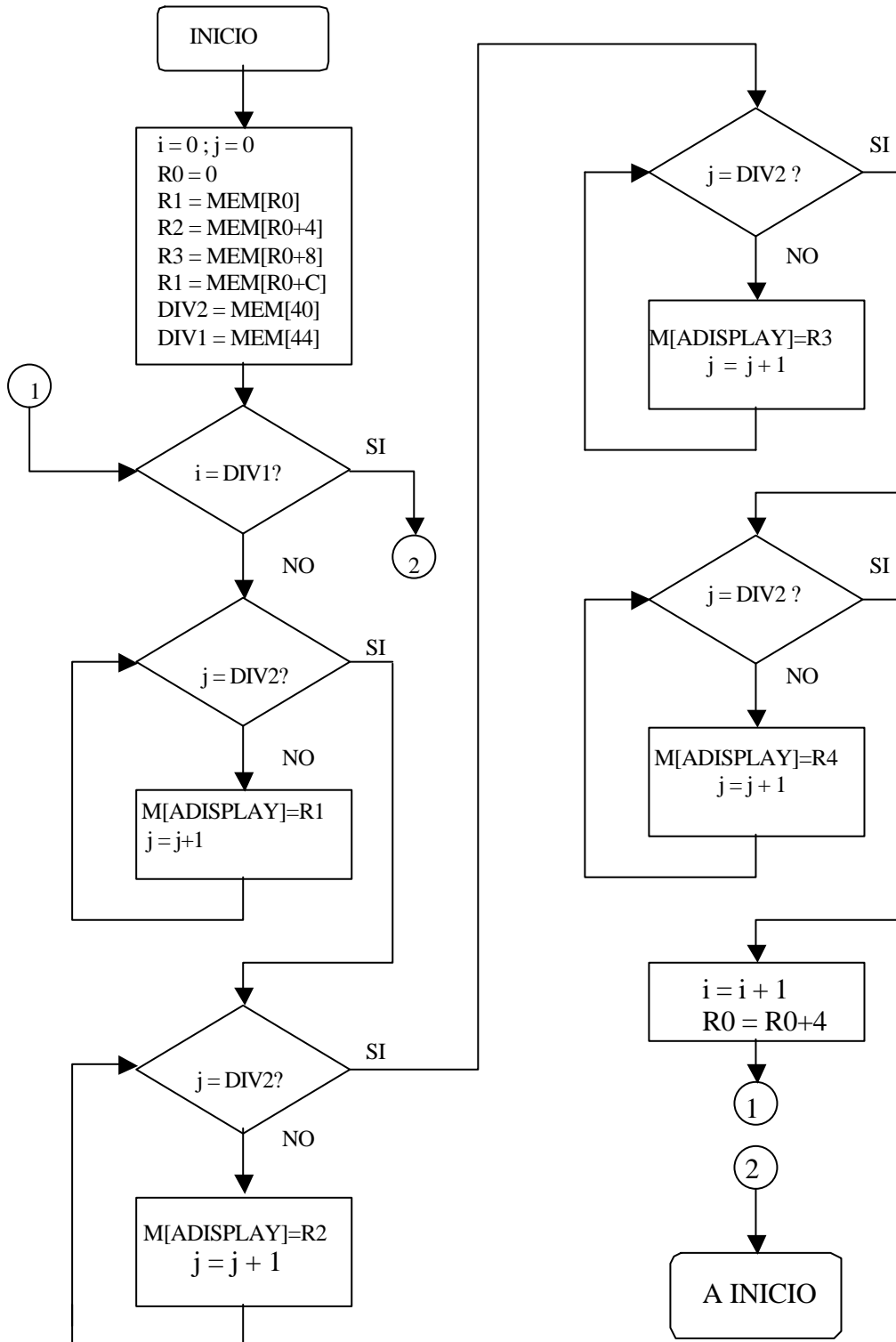
La forma de programar el dispositivo es bastante sencilla: teniendo en cuenta qué es lo que controla cada pin del bus *ENTRADA[11:0]* (figura 7-2), y sabiendo que cada uno de los doce bits menos significativos escritos en `D_ADDRESS` se corresponden con un pin de dicho bus, está claro que el dato escrito en el periférico se corresponderá con la iluminación de una serie de segmentos de un display.

NOTA IMPORTANTE

Aunque los segmentos y el punto decimal de los displays son activos a nivel bajo, y las señales de selección de display *d3*, ... ,*d0* son activas a nivel alto, ***el programador ha de considerar que todas las señales son activas a nivel alto, pues el controlador se encargará de invertir los bits que estime oportuno.***

7.4 Realización del Programa

A continuación se adjunta el diagrama de bloques del programa que controla el periférico.



7.4.1 Memoria de Datos

Mensaje Objetivo: HASTA LUEGO!

Se pretende mostrar el mensaje anterior por la pantalla de displays. Si observamos el programa, vemos que utiliza dos constantes para dividir la frecuencia, DIV2, y DIV1, que están en posiciones fijas de la memoria (0x000040 y 0x00000044).

Esto es debido a que cada carácter del mensaje objetivo ocupa una posición de memoria (desde 0x00000000 a 0x0000002C), y además se incluyen cuatro espacios en blanco para borrar completamente la pantalla y que el mensaje empiece de nuevo (posiciones 0x00000030 a 0x0000003c), por lo que las dos últimas posiciones se corresponden con las ocupadas por DIV1 y DIV2.

Para más detalles, examinar el fichero ROM2.vhd, que no es más que una ROM descrita en VHDL que contiene los caracteres del mensaje objetivo codificados según las especificaciones del periférico a utilizar.

7.4.2 Calculando los parámetros del programa

Los únicos parámetros del diseño son las constantes DIV2 y DIV1. Cada una de ellas controla respectivamente la frecuencia de refresco de los displays, y el tiempo que se va a mantener un carácter fijo antes de realizar un desplazamiento del mismo a la pantalla contigua.

Para modificarlos, basta con editar el fichero ROM2.vhd y sustituir los dos valores antiguos por los deseados.

Vamos a realizar los cálculos para una **frecuencia objetivo de 10 MHz**.

La frecuencia de refresco debe ser superior a 50 Hz. Supongamos **DIV2=5000**. Cada uno de los bucles internos codificado en código máquina del ARM necesitaría 7 ciclos de reloj (ver programa ROM.txt), que multiplicado por DIV2 y dividido por la frecuencia objetivo, nos daría el tiempo que se mantiene cada bucle. Basta multiplicar este valor por 4 bucles, para obtener el Periodo de Refresco:

$$T_{\text{refresco}} = (7 \cdot 5000) / 10^7 \text{ s/bucle} \cdot 4 \text{ bucles} = 0.014 \text{ s} \Rightarrow f_{\text{refresco}} = 71.43 \text{ Hz}$$

El periodo de refresco multiplicado por DIV1 nos da el tiempo que transcurre antes de que se produzca un desplazamiento. Ajustaremos DIV1 para que dicho tiempo se encuentre entre 1 sg y 1.5 sg.

$$T_{\text{refresco}} \cdot \text{DIV1} = [1 \text{ sg}, 1.5 \text{ sg}] \Rightarrow \text{DIV1} = 96 \Rightarrow T_{\text{display}} = 1.344 \text{ sg}$$

- **DIV1 = 0x00000060** (96)
- **DIV2 = 0x00001388** (5000)

7.5 Implementación en la Virtex 800

La última fase en la creación del prototipo es la implementación del mismo en la Virtex 800.

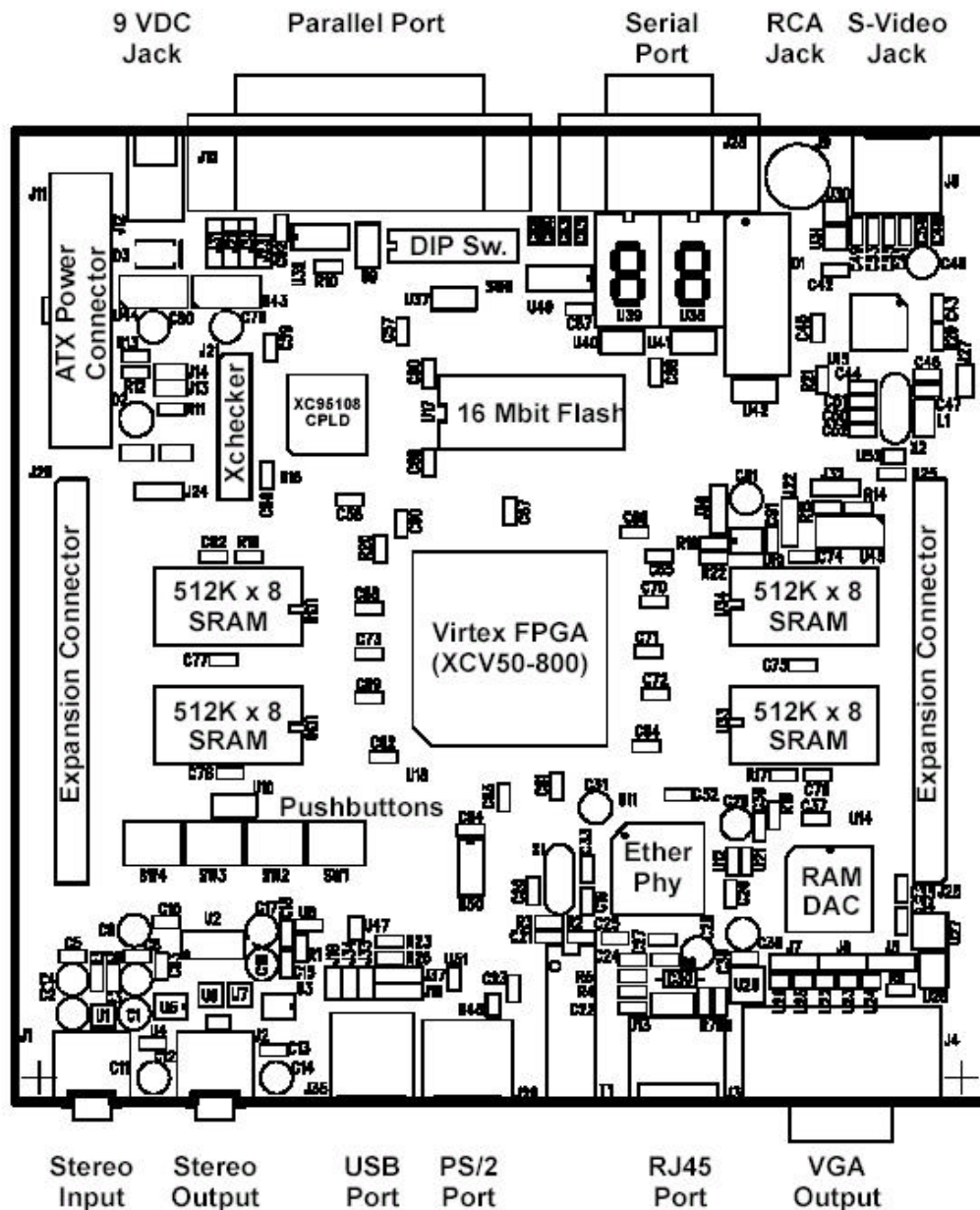


Figura 7-3: diagrama de la placa XSV

Antes de comenzar la implementación, debemos asignar los pines físicos a las distintas señales externas de nuestro prototipo.

Para ello, debemos acudir al catálogo de la placa XSV, y examinar las tablas de Entrada/Salida que vienen incluidas.

Los pines escogidos para nuestro diseño son los siguientes:

NOMBRE	Pin FPGA	Pin Conector	Ubicación Conector (**)
clk	89	****	*****
nReset	216	28	IZQUIERDO
salida[11]	55	5	DERECHO
salida[10]	63	9	DERECHO
salida[9]	66	13	DERECHO
salida[8]	70	17	DERECHO
salida[7]	73	21	DERECHO
salida[6]	79	25	DERECHO
salida[5]	82	29	DERECHO
salida[4]	86	33	DERECHO
salida[3]	94	37	DERECHO
salida[2]	97	41	DERECHO
salida[1]	101	45	DERECHO
salida[0]	107	49	DERECHO

(**) Ver Figura 7-3: la placa cuenta con dos conectores, uno a la derecha y otro a la izquierda

El pin 89 es el reloj; la placa cuenta con un oscilador de 100 MHz programable a frecuencias múltiplos de la misma.

Para programarla, basta con configurar los “jumpers” de la placa en modo programación, y utilizando el software de programación del oscilador, incluir un factor divisor de 10, con lo cual obtenemos una frecuencia de reloj de 10 MHz, que recordemos fue la frecuencia objetivo en el cálculo de los parámetros del periférico.

Nota: la tabla anterior muestra la asignación de pines realizada en nuestro prototipo para realizar los test de campo, y se encuentra en el fichero **arm.ucf**, que se incluye en el mismo directorio que los FICHEROS VHDL. Si se desea realizar una asignación de pines distintas, basta con consultar el manual de la placa de la Virtex 800, incluido en la DOCUMENTACIÓN adjunta en el CD-ROM.

7.6 Prueba Final

Para concluir este capítulo, mostramos una fotografía del prototipo en funcionamiento.

En el CD-ROM se han incluido, además de esta y otras fotografías, dos vídeos ilustrando el funcionamiento del prototipo de pruebas.

Se puede consultar la ubicación de estos ficheros en el Apéndice A.

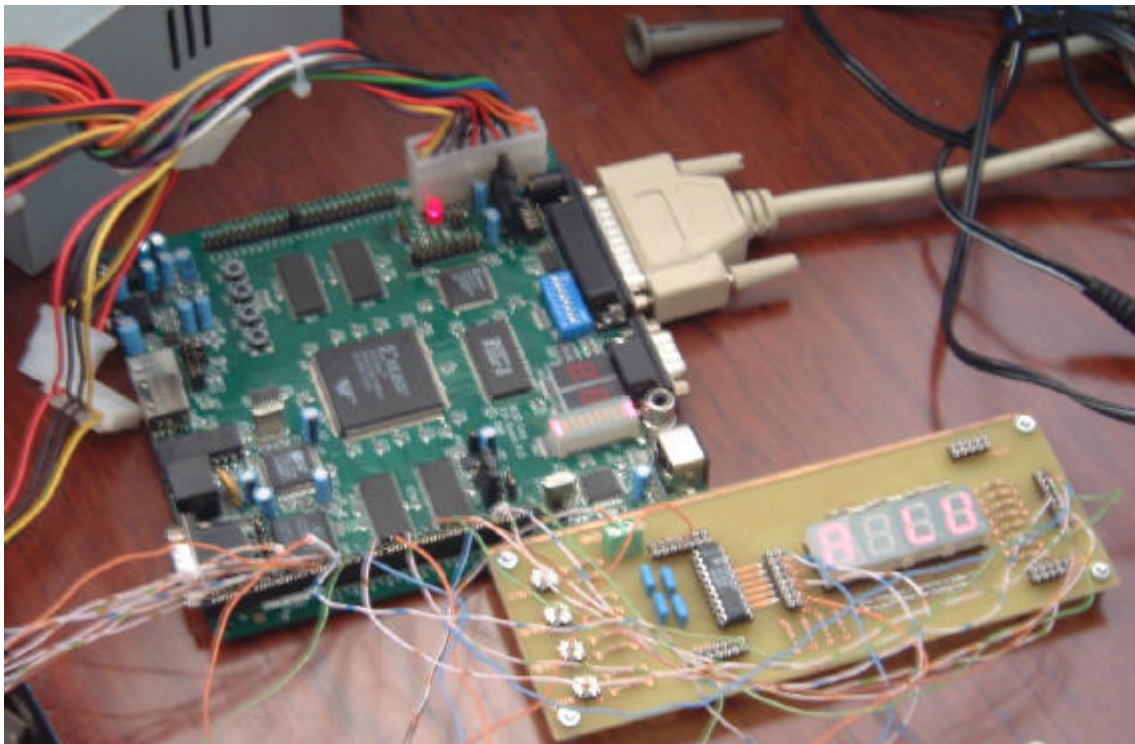


Figura 7-4: prototipo de pruebas en funcionamiento

A

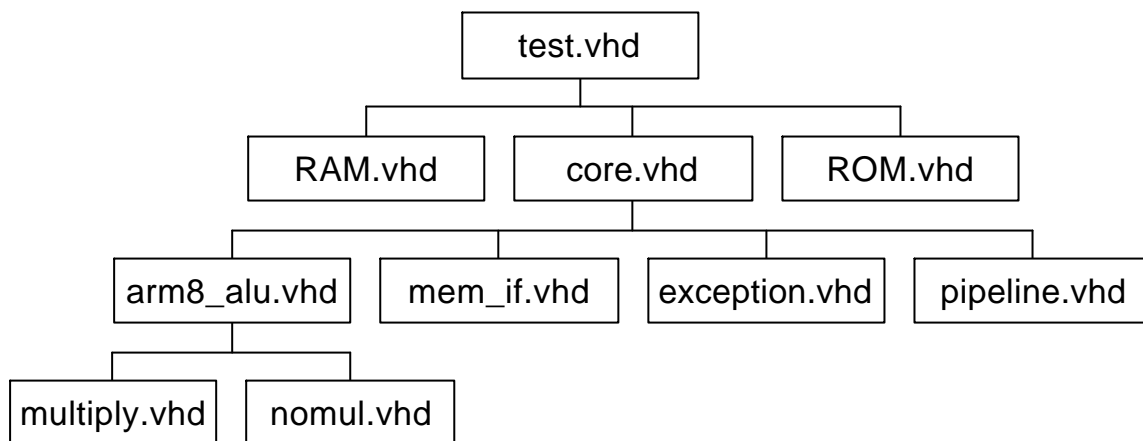
Ficheros

En este apéndice se detallan todos los ficheros relacionados con el diseño e implementación del CORE del ARM8/9, así como su ubicación dentro del CD-ROM que se incluye, y la forma de utilizarlos a la hora de generar sistemas de nivel jerárquico superior que incorporen al CORE.

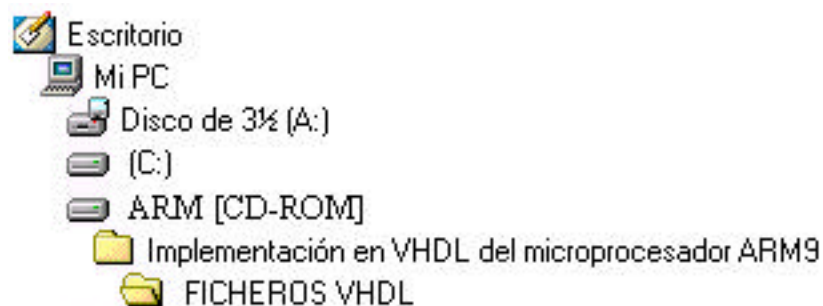
- A.1 Ficheros de Simulación del CORE
- A.2 Ficheros de Simulación del Prototipo
- A.3 Ficheros para Implementación
- A.4 Ficheros VHDL
- A.6 Organización del CD-ROM

A.1 Ficheros de Simulación del CORE

Para abordar la realización de todas las simulaciones, necesitamos en primer lugar crear un directorio de trabajo. En él, debemos cargar todos los ficheros presentes en la siguiente jerarquía:



La ubicación de los archivos anteriores en el CD-ROM, es la siguiente:



También será necesario copiar en el mismo directorio de trabajo los ficheros “programa.bat” y “ejecuta.bat”, que se también se encuentran ubicados en la ruta anterior.

Necesitaremos instalar dos aplicaciones, que también se incluyen en el CD-ROM (ver apartado 8.6).

- **TXT2VHDL:** a partir de un fichero .txt que contiene una secuencia de instrucciones de ARM en hexadecimal, genera una ROM escrita en VHDL con el mismo programa
- **VHDL-Simili:** el fichero de instalación del software de simulación también se incluye en el CD-ROM.

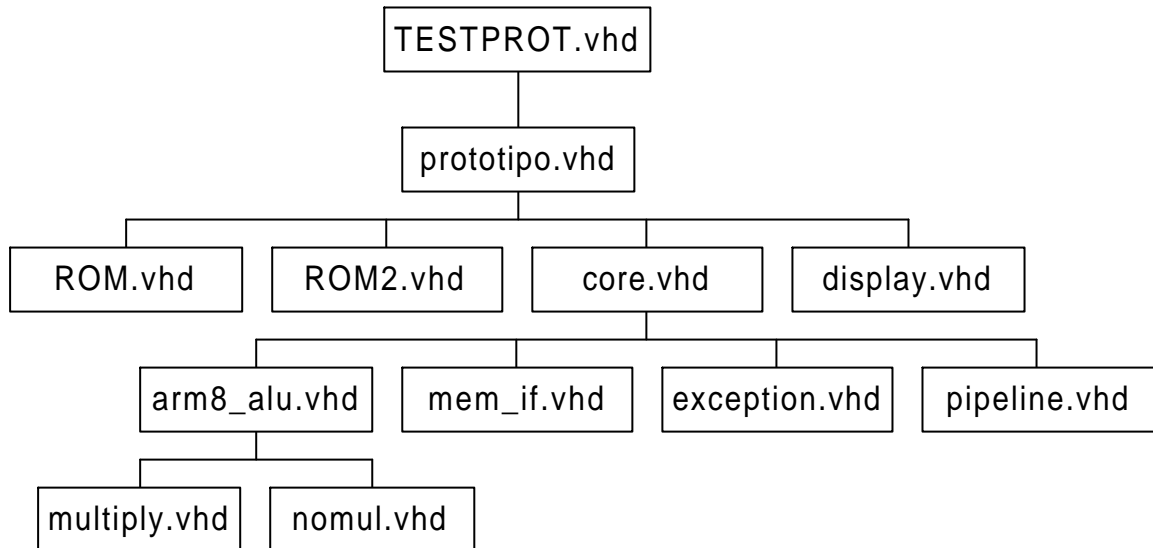
El fichero ROM.vhd debe ser generado a partir del fichero .txt que contiene el programa a ejecutar en la simulación. En los directorios de simulaciones, se incluyen multitud de estos ficheros, que pueden ser utilizados para verificar el funcionamiento del simulador.

Si el proceso de simulación concluye con éxito, deben generarse al menos dos de los siguientes ficheros:

- traza.dat
- pipeline.dat
- memory.dat (sólo en algunas simulaciones)

A.2 Ficheros de Simulación del prototipo

La jerarquía de ficheros a utilizar para la simulación del prototipo es la siguiente:

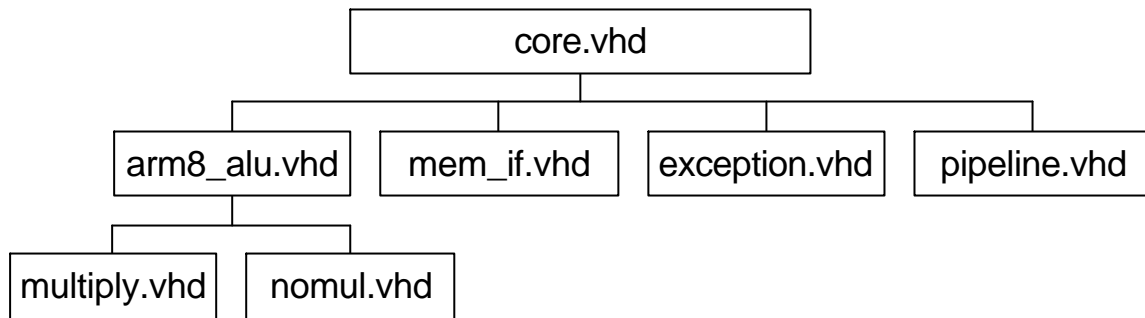


La ubicación de los ficheros y el modo de simulación es idéntico al expuesto en el apartado anterior, con las siguientes salvedades:

- Los ficheros de proceso por lotes a utilizar son *prototipo.bat* y *ejecuta.bat* (por ese orden)
- El simulador escribe como salida un único fichero, denominado *prototip.dat*

A.3 Ficheros para la Implementación

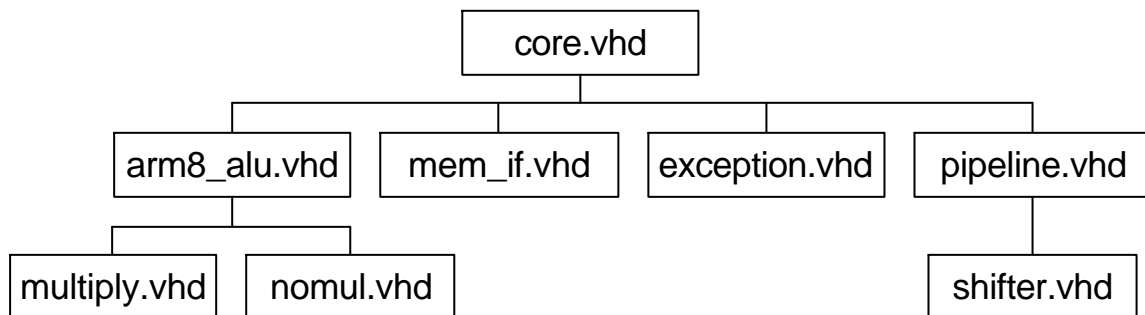
Para la implementación del CORE, se han de utilizar los siguientes ficheros VHDL:



En el propio directorio del CD-ROM donde están ubicados los ficheros VHDL originales, se encuentra también un subdirectorio denominado **VERSION 2** que contine la descripción VHDL de un nuevo diseño optimizado con la finalidad de obtener una frecuencia de reloj mayor en la implementación de la macrocelda del CORE.

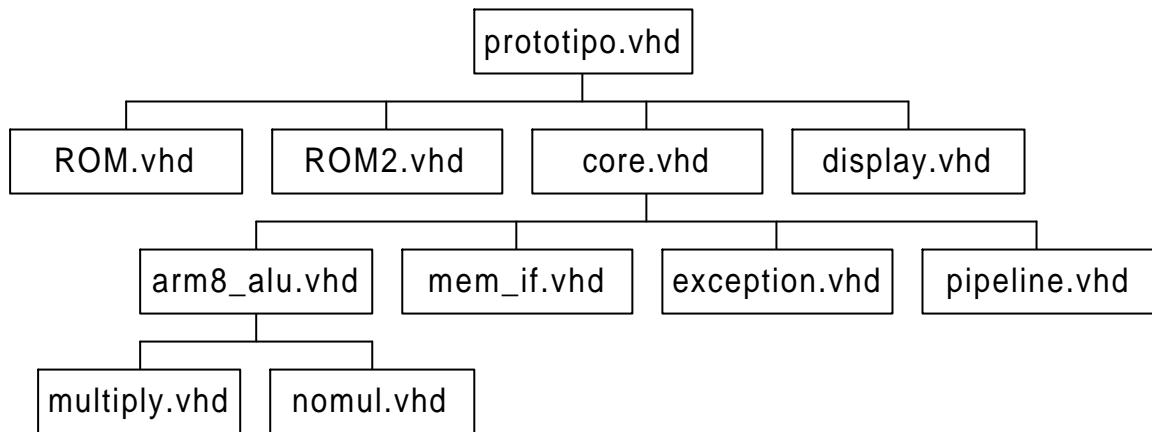
La jerarquía de ficheros es bastante similar a la versión anterior, con la única diferencia de que el *shifter* aparece ahora como un bloque adicional. Además, el fichero `arm8_alu.vhd` no contiene ningún bloque funcional que realice desplazamientos.

Además, el fichero `pipeline.vhd` ha sufrido unas ligeras variaciones, con el fin de poder operar con el ALU y el *shifter* como bloques independientes, y además trabajar con cada uno de ellos en etapa diferentes (*execute* y *decode* respectivamente).



Prototipo de Pruebas

Por otro lado, para la implementación del prototipo de pruebas, la jerarquía a utilizar es algo más compleja:



El resultado de la implementación será un *fichero.bit*, que se utilizará para programar la FPGA.

El fichero generado en el desarrollo de nuestro proyecto, se encuentra almacenado en el CD-ROM. La ubicación del mismo, se muestra en el último apartado del presente capítulo.

Nota: sería conveniente en la implementación, utilizar el fichero **arm.ucf**, que contiene la asignación de pins de entrada/salida utilizada en nuestra implementación, y descrita en el apartado 7.5.

A.4 Ficheros VHDL

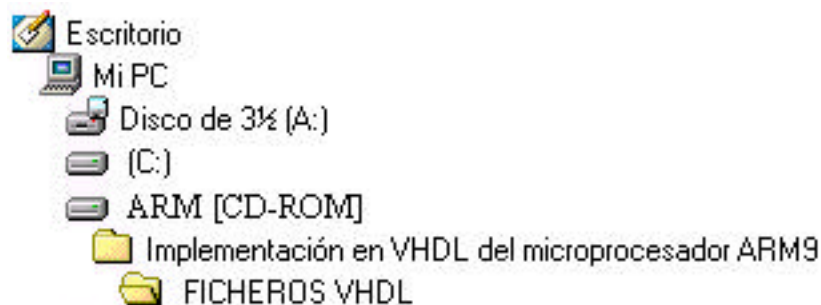
NOMBRE	DESCRIPCIÓN
arm8_alu.vhd	Contiene la descripción VHDL de la ALU del ARM8, que se ha utilizado como ALU en el diseño del CORE del ARM9
core.vhd	Fichero a utilizar como TOP en la implementación del CORE del microprocesador ARM9. Contiene un GENERIC denominado NOMULT (1: no implementar multiplicador; 0: implementar multiplicador)
display.vhd	Descripción VHDL del controlador hardware de la placa de displays
exception.vhd	Priorización de Interrupciones: informa al CORE de qué interrupción debe atender, en función de las prioridades expuestas en el Modelo del Programador
instruction_set.vhd	Package del diseño: contiene la declaración de todas las constantes utilizadas en el diseño, los vectores de test y las funciones creadas específicamente para el diseño
mem_if.vhd	Interfaz de Memoria del Propio CORE. Realiza la conversión de big-endian a little-endian (en caso de que la señal BIGEND esté activa), y asegura la correcta alineación de la dirección de acceso a memoria.
memperso.vhd	Interfaz de Memoria Personalizada (diseño expuesto en el apartado 5.7.1)
Multiply	Fichero de Emulación del Multiplicador (multiplicador de 32x32 bits con Resultado en 48 bits)
Nomul	Escribe 0x00000000 en la salida del multiplicador
Pipeline	Diseño de la arquitectura VHDL de 5 etapas utilizada en el diseño del CORE, descrita en VHDL
prototipo.vhd	Fichero a utilizar como TOP en la implementación del prototipo de pruebas
rom.vhd	Fichero ROM generado mediante la aplicación Txt2vhd.exe a partir de un programa escrito en formato .txt (no es un fichero concreto, sino que en cada caso contiene el programa a simular)
rom2.vhd	Contiene la memoria de datos que almacena los caracteres que conforman el mensaje para la realización del prototipo
ram.vhd	Contiene la memoria de datos utilizada en las simulaciones de evaluación del CORE
test.vhd	Fichero de test para evaluación del CORE. Es una entidad sin puertos de entrada y salida, cuya única finalidad es ser compilada por VHDL-Simili (no se puede implementar)
testprot.vhd	Fichero de test para evaluación del Prototipo. Es una entidad sin puertos de entrada y salida, cuya única finalidad es ser compilada por VHDL-Simili (no se puede implementar)

A.5 Organización del CD-ROM

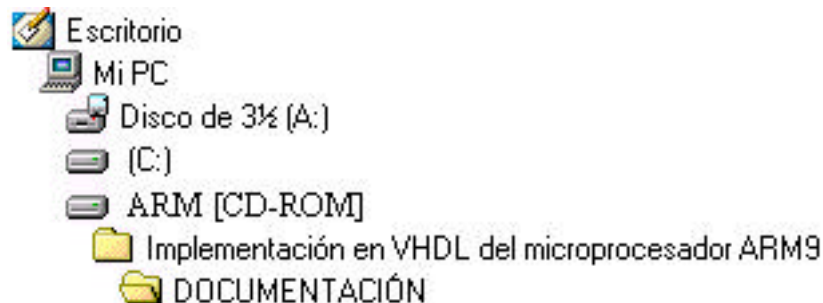
Ficheros VHDL

Este directorio incluye además de todos los ficheros VHDL utilizados en el diseño, el fichero *arm.ucf* que contiene la asignación de pines de entrada/salida realizada para la implementación de nuestro prototipo.

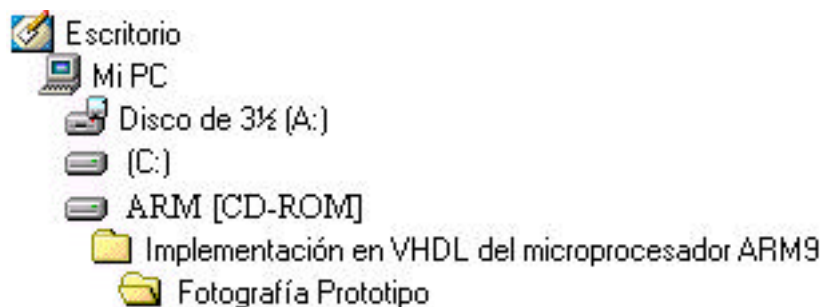
Además, existe un subdirectorio denominado **VERSION 2**, en el cual se almacena una versión optimizada del diseño.



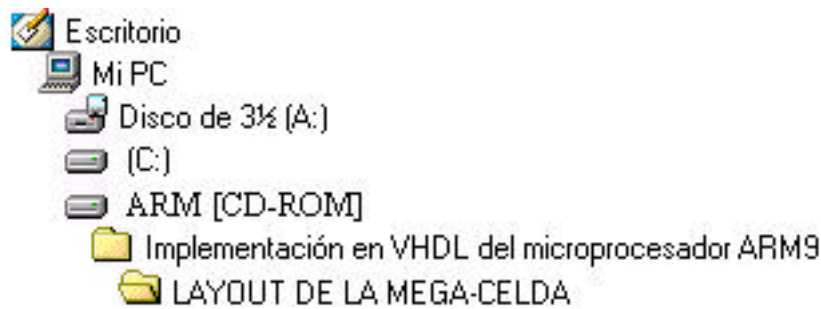
DOCUMENTACIÓN ADJUNTA



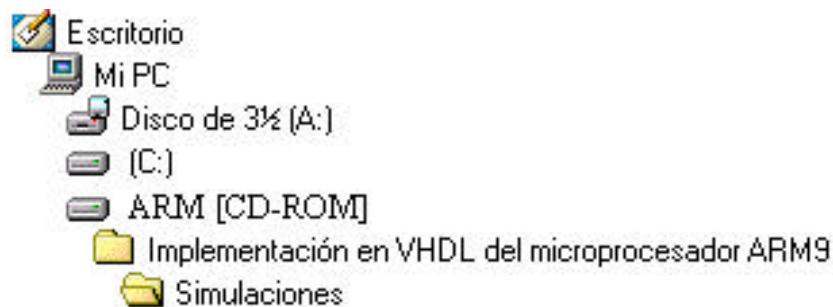
ILUSTRACIONES DEL PROTOTIPO FUNCIONANDO



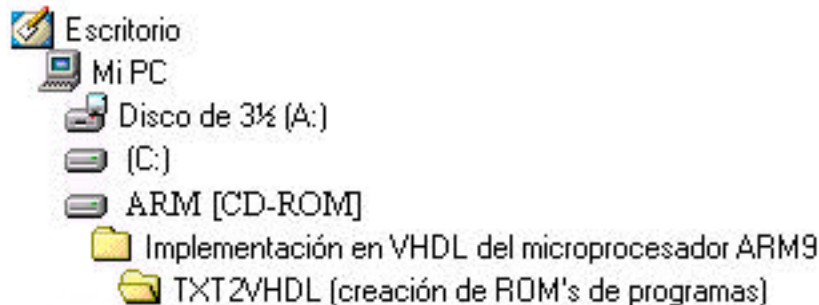
LAYOUT DE LA MEGA-CELTA DEL PROCESADOR ARM9



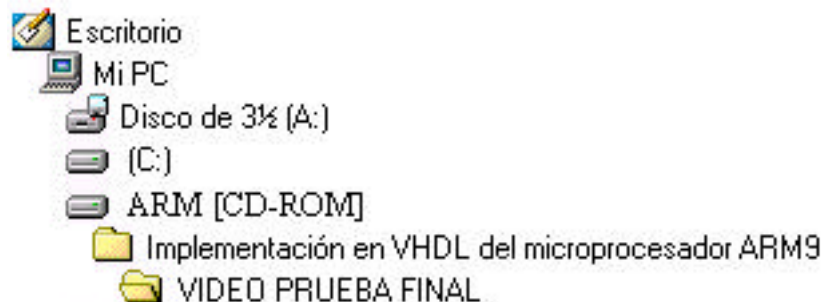
SIMULACIONES



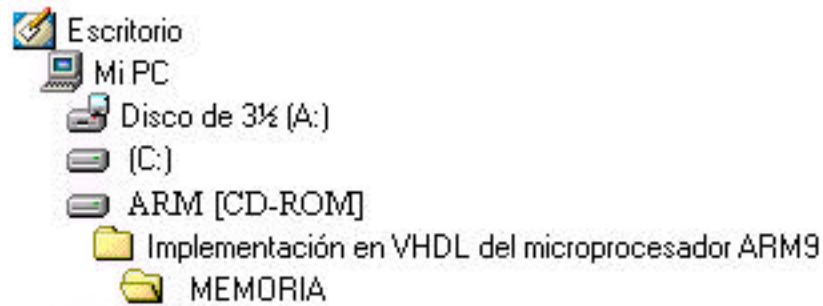
APLICACIÓN DE CREACIÓN DE ROMs PARA ARM (Txt2vhd.exe)



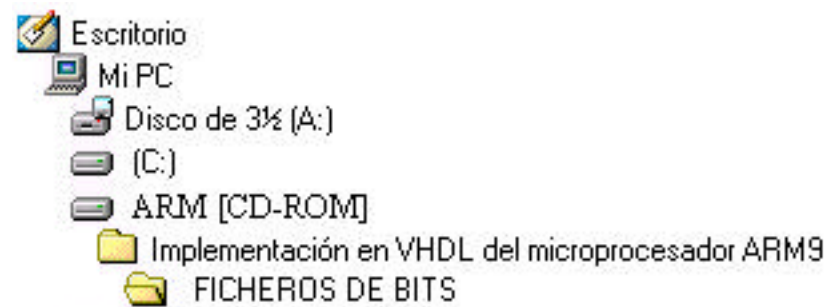
VIDEO DEL PROTOTIPO FUNCIONANDO



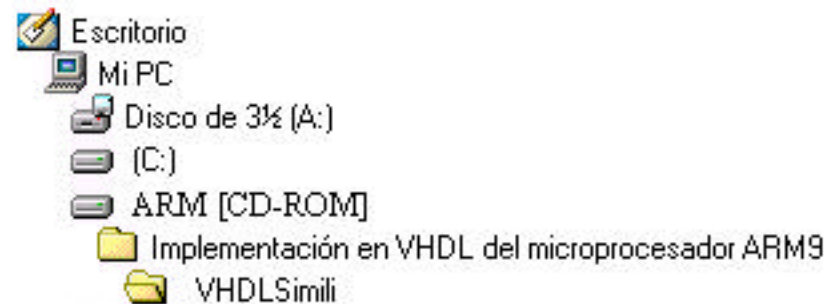
UBICACIÓN DEL PRESENTE DOCUMENTO EN EL CD-ROM



FICHEROS DE PROGRAMACIÓN DE LA FPGA (.BIT)



FICHERO DE INSTALACIÓN DE VHDL-Simili



B

Uso del diseño

El objetivo del presente anexo es informar al futuro usuario del CORE de aquellas señales que obligatoriamente debe configurar externamente, y de ciertos factores que ha de tener en cuenta a la hora de trabajar con el diseño.

- B.1 Entradas a configurar
- B.2 Alineación de direcciones
- B.3 Señal de salida *Privileged*

B.1 Entradas a configurar

Las entradas del dispositivo son las siguientes:

SEÑAL	PINES	E/S	DESCRIPCIÓN
clk	clk	E	Reloj Global del Sistema
nReset	nReset	E	Reset Global (activo a nivel bajo)
nFIQ	nFIQ	E	Petición de Interrupción Rápida (idem)
nIRQ	nIRQ	E	Petición de Interrupción (activa a nivel bajo)
nABORT	nABORT	E	Fallo de Memoria (activa a nivel bajo)
BIGEND	BIGEND	E	0: little-endian - 1: big-endian
WS	ws0, ws1	E	Petición para la generación de estados de espera
INST	inst0...inst31	E	BUS de Instrucciones
Rdata	Rdata0....RData31	E	BUS de lectura de datos de memoria

Señales de Interrupción

nReset, nFIQ, nIRA y nABORT son activas a nivel bajo. Por tanto, en caso de no utilizarlas, deberán estar conectadas a alimentación, o de lo contrario el contador de programa (PC) quedará deshabilitado, y el CORE no será capaz de ejecutar instrucciones.

Señal BIGEND

La señal BIGEND se utiliza para indicar al CORE que la numeración de memoria utilizada se corresponde con el formato big-endian, de modo que si se desea utilizar este formato, será necesario conectar el pin BIGEND a alimentación. En el caso de que el formato utilizado sea little-endian, será necesario conectar BIGEND a tierra.

Entrada WS

Se utiliza para configurar el número de estados de espera deseados en el acceso a memoria, conforme a lo expuesto en la siguiente tabla:

ws[1:0]	Nº estados de espera	Ciclo de Acceso
“00”	0	El acceso dura 1 ciclo de reloj
“01”	1	El acceso dura 2 ciclos de reloj
“10”	2	El acceso dura 3 ciclos de reloj
“11”	3	El acceso dura 4 ciclos de reloj

Es importante configurar SIEMPRE esta señal. Si no se desean estados de espera, se deben fijar los dos pines a GND (nunca dejarlos sin conectar, pues en el momento en el que se produzca un acceso a memoria, el CORE alcanzará un estado inconsistente).

B.2 Alineación de Direcciones

Además de la conversión big-endian => little-endian, la interfaz realiza una alineación de la dirección de memoria, de modo que:

- ❑ En Lectura: la dirección siempre está alineada con un Word, o lo que es lo mismo, siempre es múltiplo de cuatro (los dos bits menos significativos de VAddress se fuerzan a “00”)
- ❑ En Escritura: dependá de *size* el que VAddress esté alineada con un *Word*, con un *Halfword* (únicamente el bit menos significativo de VAddress se fuerza a ‘0’), o que se mantenga intacta en el caso de un *Byte*.

Bits 1 y 0 de la Dirección Calculada	Vaddress [1:0]			
	r_enable = 1	w_enable=1		
		byte	halfword	word
00	00	00	00	00
01	00	01	00	00
10	00	10	10	00
11	00	11	10	00

Cuando se realice una asignación de direcciones, es de vital importancia tener en cuenta lo expuesto en la tabla anterior.

Imaginemos, por ejemplo, que se desea acceder a un periférico utilizando la interfaz de memoria. El periférico cuenta con un registro interno de 32 bits, que podrá ser modificado escribiendo en una dirección de memoria que nosotros asignamos manualmente.

Supongamos que elegimos la dirección 0x000000FF. Esta dirección no está alineada, de modo que al intentar escribir 32 bits (WORD) en dicha dirección, el sistema de memoria va a intentar alinearla, forzando los dos bits menos significativos de la dirección a “00”, de manera que la dirección con la que realmente accede al exterior será 0x000000FC; dirección que el periférico no reconoce como propia, y no permitirá la escritura en su registro interno.

Basta con tener en cuenta las reglas de alineación descritas para evitar este tipo de situaciones no deseables.

B.3 Señal de salida *Privileged*

Este pin de salida ha sido introducido en el diseño para permitir la creación de un sistema de memoria protegido por hardware.

Cuando el CORE está funcionando en modo privilegiado (modo distinto de *user*) esta señal estará activa.

Existen ciertos patrones de bits en las instrucciones de acceso a memoria que desactivan esta señal, aunque el procesador esté trabajando en modo privilegiado, para poder así emular un acceso a memoria en modo *user*. Este mecanismo resulta necesario en el caso de la emulación software de una instrucción no definida, la cual debería ejecutarse en modo *user* aunque la rutina de atención a la excepción que la emula funcione en modo *system* (veanse los apartados 4.8 y 4.9 referentes a las instrucciones de acceso a memoria).



Bibliografía

En este apéndice se incluye una bibliografía básica relacionada con los temas tratados en este documento.

- C.1 Información referente al ARM
- C.2 Manuales de Xilinx
- C.3 Artículos y Publicaciones on-line
- C.4 Otras obras relacionadas

C.1 Información referente al ARM

www.arm.com

ARM Architectural Reference Manual - PRENTICE HALL - ARM DDI 0100B

ARM8 DATASHEET -- ARM DDI 0080C

ARM9TDMI -- ARM DDI0180A-9TDMI

C.2 Manuales de Xilinx

FOUNDATION SERIES 3.1i User Guide

XSV BOARD V1.1 MANUAL -- Copyright (C)1999-2001 XESS Corp.

C.3 Artículos y Publicaciones on-line

Computer Architecture. The anatomy of modern processors (C) John Morris,1998

X-Files NEWS FROM THE LEADING PROVIDER OF FPGA AND DESKTOP ASIC

SYNTHESIS SOLUTIONS (C) 2000

Computer Architecture Tutorial by Gurpur M.

FOLDOC FREE ONLINE COMPUTER DICTIONARY

RISC-CISC John Mashey

C.4 Otras obras relacionadas

Tanembaum, Andrew S. (1992)

"Organización de Computadoras. Un Enfoque Estructurado"

Ed. Prentice Hall.

Rolf Jurgen B.

"Del CISC al RISC: Aumento explosivo de la potencia en los microprocesadores"

Revista Siemens Año 51 Enero/Marzo 1991. Siemens Aktiengesellschaft. Munich, RFA.

Hernández, Luis.

"¿RISC O CISC?"

PC/TIPS BYTE. Año 5 No. 50 Marzo de 1992.



Código VHDL

En este apéndice se adjunta el código VHDL completo de todos los ficheros relacionados con el desarrollo de este proyecto.

No se incluye el listado de los ficheros correspondientes a la Version 2 del diseño, por quedar fuera del alcance del proyecto (aunque se pueden consultar en el CD-ROM adjunto).