Sistemas Informáticos
Facultad de Informática,
Universidad Complutense de Madrid

ARM9Core

Un emulador del procesador ARM9TDMI para PC

Autores:

Sergio Hidalgo Serrano
Alberto Huerta Aranda
Daniel Sañudo Vacas

Profesor director:

Katzalín Olcoz Herrero

Curso Académico:

2006/2007

Resumen

El proyecto consiste en desarrollar el núcleo de un emulador para PC del microprocesador ARM9TDMI usado en sistemas empotrados, dispositivos portátiles, etc... Este proyecto está diseñado como una librería que se puede incorporar a una aplicación que necesite reproducir el comportamiento de un sistema basado en dicho procesador.

Su diseño permite la integración de forma sencilla junto con otros módulos que emulen las distintas partes del sistema, y a su vez facilita la incorporación de nuevas funcionalidades y características.

Dispone también de opciones para su uso en depuración de programas, tales como ejecución paso a paso, visualización y modificación del estado de la máquina, etc...

Dentro del proyecto también implementamos un programa de prueba que utiliza este núcleo para emular un sistema simple de depuración de programas, con el que hemos realizado diversos tests para comprobar el funcionamiento y rendimiento del emulador.

Un segundo objetivo del proyecto era el de estudiar una arquitectura real y las razones de las distintas decisiones tomadas durante su diseño, así como las mejores técnicas de emulación y sus ventajas e inconvenientes.

Summary

Our project focuses in developing the core of an ARM9TDMI emulator for the PC. This microprocessor is commonly used in embedded systems, portable devices, etc... The project is designed to be added as a library into a bigger application which needs to reproduce an ARM-based system's behaviour.

Its design allows easy integration with other modules emulating the different system's parts, as well as simplifying the addition of new features and improved functions.

It also has options for its use while debugging other programs written for the ARM, such as step-by-step execution, full access to the machine's current state, etc...

Within the project we also developed a test program which uses the core to emulate a simple debugging system, which we used to test the proper working and performance of our emulator.

A second of goal for the project was to study a real processor architecture and the reasons behind the different design choices, along with the best emulation techniques and their weak and strong points.

Organización de la memoria

A lo largo de este documento vamos a estudiar la estructura de la arquitectura ARM v4T (en concreto, del procesador ARM9TDMI), y sus características más importantes al nivel del proyecto. Además, explicaremos cuál es el proceso que se ha seguido a la hora de emular cada una de ellas, así como los detalles funcionales del emulador y del programa de prueba implementado.

En el primer capítulo hablaremos de las motivaciones y objetivos de este proyecto, así como las diferencias entre emuladores y simuladores, las principales técnicas de emulación, y las características generales del procesador emulado (ARM) y de la máquina sobre la que se ejecutará nuestro proyecto (x86).

En el segundo capítulo, examinaremos la estructura interna del emulador y las razones que llevaron a tomar las distintas decisiones en su diseño. Aprovecharemos también para comentar en detalle aspectos de la máquina real.

El tercer capítulo describirá a fondo cada uno de los grupos de instrucciones de la máquina real, su funcionamiento y posibles formatos, así como su correspondiente emulación.

En el cuarto capítulo se describen las características que tiene el sistema de memoria y el emulador debe permitir, así como las técnicas que se han usado para reproducir su comportamiento.

El quinto capítulo está dedicado al tratamiento de interrupciones. Explicaremos su funcionamiento en la máquina real y el procedimiento que se ha seguido en su emulación.

El programa de prueba realizado para comprobar la corrección y rendimiento del emulador será descrito en el capítulo seis, que también servirá como ejemplo práctico de cómo usar el ARM9Core en una aplicación mayor.

A continuación, en el séptimo capítulo, veremos un resumen de las distintas pruebas realizadas para medir el rendimiento del emulador bajo distintas configuraciones.

Y por último, el capítulo ocho estará dedicado a comentar las conclusiones obtenidas a lo largo del desarrollo del proyecto, y el trabajo que se podría hacer en el futuro para mejorarlo.

Índice

1. Introducción

1.1 – Motivación

El proyecto consiste en desarrollar un emulador de la arquitectura ARM v4T, en concreto del microprocesador ARM9TDMI tipo RISC de 32 bits que se ejecutará sobre la plataforma Win32 en x86.

Los procesadores ARM son utilizados en sistemas empotrados, tales como dispositivos portátiles, videoconsolas, teléfonos móviles, calculadoras, etc... Actualmente, el 75% de los procesadores de 32 bits en este tipo de sistemas son modelos ARM, debido a su bajo consumo y alto rendimiento.

El mercado de dispositivos empotrados actualmente está en auge, aumentando el número de procesadores que se venden para estos usos cada año. Esta tendencia implica que el uso de los procesadores ARM es cada vez más popular, lo que les está convirtiendo en una de las arquitecturas más comunes en el panorama actual del desarrollo de aplicaciones.

Para el desarrollo de estas aplicaciones para el ARM muchas veces no se dispone de una máquina real donde poder realizar las pruebas necesarias para la depuración de las mismas. Esto significa que muchos desarrolladores deben recurrir a emuladores de la arquitectura para la cual trabajan, en nuestro caso, de la ARM v4T.

La mayoría de emuladores para esta plataforma son propietarios, y los emuladores libres y gratuitos no suelen incluir muchas características para su uso en depuración de programas en ejecución (acceso a la memoria, emulación de interrupciones, etc...). Por eso decidimos realizar un emulador libre diseñado tanto para la ejecución de programas nativos para la máquina real, como para tareas de depuración de nuevos programas (ejecución paso a paso, visualización y modificación de los registros y el estado de la máquina, etc...).

El emulador que desarrollamos es un módulo software que reproduce el comportamiento del core del microprocesador ARM9TDMI, y es capaz de integrarse con otros módulos que reproduzcan el comportamiento de otras partes del sistema (memoria, coprocesadores, interfaces de entrada y salida, etc...).
El objetivo de esto es que cualquier desarrollador pueda utilizar el “core emulado” en el marco de una aplicación mayor que necesite emular una plataforma completa basada en ARM.

El desarrollo de este proyecto también tiene como motivación el conocer en profundidad los detalles de la arquitectura. Además, el conocer una arquitectura real permite entender con más precisión el funcionamiento de un microprocesador, lo que simplifica el proceso de aprendizaje de futuras arquitecturas.

Dentro del proyecto, además del core del emulador, realizaremos un programa de prueba que reproduzca el comportamiento de una máquina genérica ARM y muestre tanto el funcionamiento de las características de depuración más comunes, como la integración del core dentro de una aplicación real.

1.2 – Emulador vs. simulador

Reproducir el comportamiento de un procesador nos plantea un dilema: elegir entre un emulador o un simulador.

Un simulador tiene como objetivo reproducir el comportamiento interno del procesador, normalmente con motivos de aprendizaje de la arquitectura. Por tanto, un simulador intenta crear un "modelo" del procesador.

Por su parte, un emulador implica una visión externa del comportamiento de la máquina a emular. El objetivo en este caso es ejecutar programas reales tal y como lo haría la máquina real. Esto conlleva la necesidad de que el programa emulado se ejecute a la misma velocidad que tendría en el dispositivo físico, lo que impone grandes restricciones en cuanto a eficiencia.

Por tanto, a la hora de programar un emulador no es necesario reproducir la arquitectura física, sino que se considera al procesador como una caja negra: no importa como funcione internamente siempre que su comportamiento externo sea el mismo que el de la máquina real ante el mismo programa.

Por regla general, en un simulador se suele trabajar a nivel de ciclos, reproduciendo la arquitectura segmentada del procesador. Un emulador, en cambio, trabaja con una precisión a nivel de instrucción, por lo que se ignora dicho comportamiento segmentado.

Esto significa que mientras que el simulador muestra las dependencias de datos, paradas y bloqueos del pipeline, etc..., el emulador no se ve afectado por estas circunstancias. Por esto, el emulador es menos preciso que el simulador, pero mucho más rápido, lo que le permite ejecutar programas a la velocidad real.

La elección entre simulador y emulador condiciona por tanto toda la organización y estructura del programa, ya que los requisitos que se persiguen son completamente opuestos. Esto obliga a tener los objetivos muy bien definidos y saber escoger entre ambas perspectivas de acuerdo a lo que se busca.

En nuestro caso, los objetivos consistían en crear un programa capaz de ejecutar código ARM en tiempo real, lo que significa que buscamos eficiencia, aunque ello signifique una pérdida de precisión. Nuestro emulador funciona por tanto a nivel de instrucción, e ignora el comportamiento del pipeline segmentado.

1.3 – Intérprete vs. traducción binaria

A la hora de diseñar un emulador, se presentan dos posibles modelos: el de intérprete o el de traducción binaria.

El primer modelo, más simple, usa el mismo planteamiento que una máquina virtual, es decir, lee cada instrucción original y reproduce su comportamiento a través de una serie de rutinas escritas en el código de la máquina destino.

Por el contrario, la traducción binaria toma el código original y lo convierte en un código funcionalmente equivalente para la máquina destino en un primer paso de preprocesamiento. Posteriormente se ejecuta directamente el código traducido, eliminando así las penalizaciones que son comunes en los intérpretes.

Para realizar una traducción binaria en primer lugar se divide el código en bloques funcionales separados por instrucciones de salto. Cada bloque se ejecutará siempre del mismo modo, de forma que la traducción se realizará sobre bloques completos de código, dado que su comportamiento está ya determinado.

Al traducir un bloque el número de instrucciones obtenido suele ser distinto al número original. Esto es debido a que distintas arquitecturas pueden realizar ciertas operaciones con un número mayor o menor de instrucciones máquina. Sin embargo, al ser los bloques funcionalmente equivalentes, su ejecución dará siempre el mismo resultado.

Los emuladores que se implementan sobre sistemas de traducción binaria suelen también incluir un intérprete, dado que en ciertas ocasiones no es posible la traducción de algunas instrucciones de forma eficiente. Además, debido a las diferencias entre las distintas arquitecturas, resulta una tarea muy compleja desarrollar un traductor binario que genere código eficiente. Actualmente se sigue investigando y avanzando en este campo.

Una opción a medio camino entre interpretación y traducción binaria es la de la pre-decodificación. Este sistema consiste en realizar una primera pasada durante la inicialización del programa y extraer los distintos campos de las instrucciones para su almacenamiento en memoria.

De esta forma, al realizarse la ejecución de las distintas instrucciones, se evita la penalización por la decodificación de las mismas. El inconveniente de este sistema, además de un mayor consumo de memoria, se presenta al tener código que se automodifica.

En estos casos, es necesario llevar un control de las palabras de memoria que corresponden a instrucciones y han sido modificadas, mediante algún tipo de “cache de instrucciones”. Esto significa que en estos casos se producirán penalizaciones en tiempo de ejecución.

En nuestro proyecto optamos por no implementar estos sistemas debido a restricciones de tiempo. Nuestro emulador será por tanto un intérprete con decodificación de instrucciones en tiempo de ejecución.

1.4 – Características del ARM v4T y del x86

A la hora de diseñar un emulador es necesario conocer las principales características tanto de la máquina cuyo comportamiento se desea reproducir (a la que llamaremos "máquina original"), como del procesador sobre el que se ejecutará dicho emulador (al que llamamos "máquina destino").

En nuestro caso, la máquina original se corresponde con la arquitectura v4T del ARM (ARM9TDMI) y la máquina destino con la familia x86 de Intel.

El conocer las características de ambos permite saber qué operaciones son más eficientes en la máquina destino, y por tanto deben ser aprovechadas para implementar el comportamiento de la máquina origen. También con esto se conoce las operaciones más frecuentes en la máquina origen, que deberán ser emuladas de manera eficiente.

En muchos casos se presentan similitudes en el comportamiento de ciertas características en ambas máquinas. Esto significa que si se consigue aprovechar esta similitud será posible una emulación mucho más eficiente que si fuera necesario implementar estos comportamientos manualmente.

Por ejemplo, tanto el ARM como el x86 presentan comportamientos similares en algunos de sus bits de estado (flags). Si en el emulador se consigue aprovechar esta característica se podría evitar el cálculo manual de los bits de estado, que es una operación costosa y común. De esta forma se utilizan los propios flags de la máquina destino como flags de la máquina original.

Características del ARM:

Dispone de dos repertorios de instrucciones distintos: uno de 32 bits (ARM) y otro de 16 (Thumb). De ellos, nuestro emulador sólo implementa el repertorio ARM.

El repertorio ARM permite a un programa tener el máximo rendimiento con el menor número de instrucciones, mientras que el repertorio Thumb es más simple pero ofrece más densidad de código y optimización de espacio.

Se puede cambiar el repertorio durante la ejecución con una instrucción o modificando el flag correspondiente.

A nivel físico, las instrucciones del repertorio Thumb son convertidas en las equivalentes del repertorio ARM.

El ARM tiene 31 registros de propósito general de 32 bits, de los cuales 16 son visibles al programador al mismo tiempo. De entre ellos:

- Registro 13: Puntero a pila (SP).
- Registro 14: Dirección de retorno de subrutina (LR).
- Registro 15: Contador de programa (PC).
- Registro de estado (CPSR) y una copia del mismo (SPSR) por modo de ejecución.

Todos los registros son visibles en ambos modos (ARM o Thumb).

El ARM9TDMI es un procesador segmentado que presenta un pipeline dividido en 5 etapas:

1. Fetch.
2. Decodificación de instrucción y lectura de registros.
3. Ejecución de desplazamientos, ALU, cálculo de direcciones y multiplicaciones.
4. Acceso a memoria y multiplicación.
5. Escritura en registros.

Este procesador no dispone de anticipación de operandos, lo que significa que en el caso de producirse una dependencia de datos el pipeline entero quedará bloqueado.

Sin embargo, debido a que el emulador no tiene precisión a nivel de ciclo, el comportamiento del pipeline no es reproducido, pero es necesario tenerlo en cuenta en algunos momentos para asegurar que ciertas instrucciones se comportan de igual forma.

Todas las instrucciones ARM se ejecutan de forma condicional, dependiendo de un campo de condición que se compara con el valor de los bits de estado durante la fase de decodificación. Si no se cumple la condición, se salta a la siguiente instrucción del código.

Las instrucciones opcionalmente pueden actualizar los registros de condición (flags de la ALU: N, Z, C, V) de acuerdo al resultado.

Una de las principales características del ARM9TDMI es su desplazador de barril, que permite realizar desplazamientos y rotaciones de operandos sin apenas coste adicional. Esto implica que muchas de las instrucciones hacen uso de dicho elemento.

El ARM tiene 5 tipos de excepciones (interrupciones), a cada una de las cuales se le asocia un modo de ejecución. Cada modo dispone de un banco de registros lógicos que se corresponden a distintos registros físicos de entre los 31 disponibles.

Dispone de 4 tipos de instrucciones:

- Aritmético-lógicas
- Load / Store
- Branch (saltos)
- Instrucciones de coprocesador

Características del x86:

Esta arquitectura tiene un repertorio de instrucciones de 32 bits, a partir de i386 y superiores, aunque inicialmente era de 16 bits. Al contrario que el ARM, aquí no se dispone de ejecución condicional debiéndose utilizar instrucciones de salto para emularla. El repertorio ofrece instrucciones aritméticas, lógicas, de transferencia, misceláneas y de salto.

Se dispone de 4 registros de propósito general: EAX (Acumulador), EBX (Base), ECX (Contador) y EDX (Datos). Se proporciona acceso a los 32 bits, a los 16 bits menos significativos y a los dos bytes menos significativos de los registros anteriores.

Hay 4 registros de segmento de 16 bits que sirven para formar direcciones de memoria: CS (Código), SS (Pila), DS (Datos) y ES (Extra).

Existen además 2 registros de 32 bits para acceso indexado: ESI (Índice fuente) y DSI (Índice destino), y otros 2 registros de 32 bits que almacenan un puntero a la pila: ESP y EBP. El registro de 32 bits que apunta a la instrucción actual es el IP (Instruction Pointer). También existen una serie de registros especiales para control, test, debug...

El x86 posee indicadores de estado que toman sus valores según el resultado de las operaciones, siendo los siguientes:

- C: Carry
- O: Overflow
- S: Signo
- Z: Cero
- A: Carry Auxiliar, igual que Carry pero restringido al nibble (4 bits) bajo
- P: Paridad, indica si el resultado tiene una cantidad par de bits a uno

Cada uno de estos flags será modificado dependiendo del tipo al que pertenezca la instrucción x86, sin la posibilidad de indicar que se altere o no como en el ARM. Es decir, la instrucción de suma siempre cambiará el flag de signo según el valor del resultado obtenido.

La memoria usa una ordenación little-endian y permite accesos a direcciones no alineadas de 16 y 32 bits.

En la actualidad la gran mayoría de procesadores x86 incluye también el repertorio de instrucciones MMX, que permite operar con varios datos en una única instrucción (SIMD).

Esta extensión añade 8 registros de 64 bits (MM0-MM7), que pueden contener un dato de 64 bits, dos 32 bits, 4 de 16 bits u 8 de un byte (empaquetamiento). Estos registros realmente están solapados con los de la unidad de coma flotante (FPU), y cualquier modificación en estos alterará los otros.

A diferencia del ARM, el x86 no dispone de un desplazador de barril. Esto significa que la emulación de los desplazamientos de las instrucciones del ARM tendrá una penalización al ejecutarse sobre este procesador.

Debido a que el emulador busca la mayor eficiencia posible, y que se recurre en muchas ocasiones a integrar código en ensamblador junto con el código a alto nivel, es necesario tener en cuenta estas características de la máquina destino a la hora de implementar las distintas rutinas del mismo.

2. Funcionamiento del Emulador

2.1 – Esquema General

Basado en un diseño modular, el emulador ARM9Core puede ser incorporado en una aplicación que necesite reproducir el comportamiento del ARM9TDMI (arquitectura v4T) sobre un sistema Win32 ejecutándose en una plataforma x86.

Este requisito nos obliga a que el ARM9Core se distribuya como una librería para que el programador de la aplicación que lo vaya a usar pueda incluirlo en su propio proyecto. De esta forma, el usuario sólo necesitará programar el comportamiento de los subsistemas específicos de la máquina que desee emular, mientras que el ARM9Core le proporcionará el funcionamiento del microprocesador.

El proyecto está desarrollado para Win32 en Visual C++ 6.0, haciendo uso del ensamblador del x86 para conseguir el mayor rendimiento posible. El diseño está orientado a objetos, y se organiza de la siguiente forma:

ARM9Core:
Esta es la clase principal del proyecto. El usuario tendrá que instanciarla para poder hacer uso de las funciones del emulador. De esta forma esta clase actúa como interfaz con el resto de la aplicación.

ARM9Contexto:
El contexto representa el estado del procesador en un momento dado. El emulador puede reproducir el comportamiento de un sistema multinúcleo, en cuyo caso habría varios objetos de esta clase, donde cada uno representaría el estado de uno de los distintos núcleos.

ARM9RutinasARM:
Esta clase contiene el código de las rutinas que reproducen el comportamiento de las instrucciones de 32 bits de la arquitectura ARM.

ARM9TablaARM:
Código que genera las direcciones de salto a cada una de las rutinas ARM según los campos comunes de un determinado tipo de instrucción.

ARM9MacrosARM:
Código que realiza tareas que tienen en común las diferentes rutinas ARM, tales como: comprobación de condición, obtención de operandos...

Utils:
Contiene definiciones de tipos y estructuras comunes utilizadas por las clases.

2.2 – Contexto

El ARM9TDMI dispone de 7 modos de ejecución distintos:

- **USER:** Modo usuario, es un modo sin privilegios en el que se ejecutan las aplicaciones generales.

- **SVC:** Modo supervisor, es el modo en el que se ejecutan las interrupciones por software (SWI).

- **ABORT:** Este modo se activa para realizar el tratamiento de las excepciones que se producen al haber un fallo de acceso a memoria (DATA_ABORT o PREFETCH_ABORT).

- **UNDEFINED:** Modo indefinido, se activa al producirse una excepción por la lectura de una instrucción no existente en el repertorio.

- **IRQ:** Modo de tratamiento de interrupciones, se activa cuando se recibe una interrupción externa.

- **FIQ:** Modo de tratamiento de interrupciones rápidas, similar al anterior pero al disponer de más registros físicos permite una ejecución más eficiente.

- **SYSTEM:** Modo de sistema, similar al modo usuario pero con privilegios al acceder a regiones de memoria.

El procesador dispone de 31 registros de propósito general de 32 bits de los cuales solamente 16 son accesibles al programador en cada modo. Estos registros están distribuidos en bancos independientes, de forma que en modos distintos el mismo número de registro se corresponde a distintos registros físicos.

Esta disposición permite una ejecución más eficiente dado que al entrar en uno de estos modos con registros replicados no es necesario salvar en memoria los valores almacenados en los registros antiguos, porque estos ya son preservados.

Además de los registros de propósito general el ARM9TDMI dispone de un registro de 32 bits llamado CPSR (Current Program Status Register), y de 5 registros SPSR (Saved Program Status Register) distribuidos entre los distintos modos.

El CPSR guarda el estado actual del programa, con los siguientes campos:

- Modo de ejecución: 5 bits que codifican el modo de ejecución actual.

- Bit de estado: Discrimina entre el modo ARM o modo Thumb.

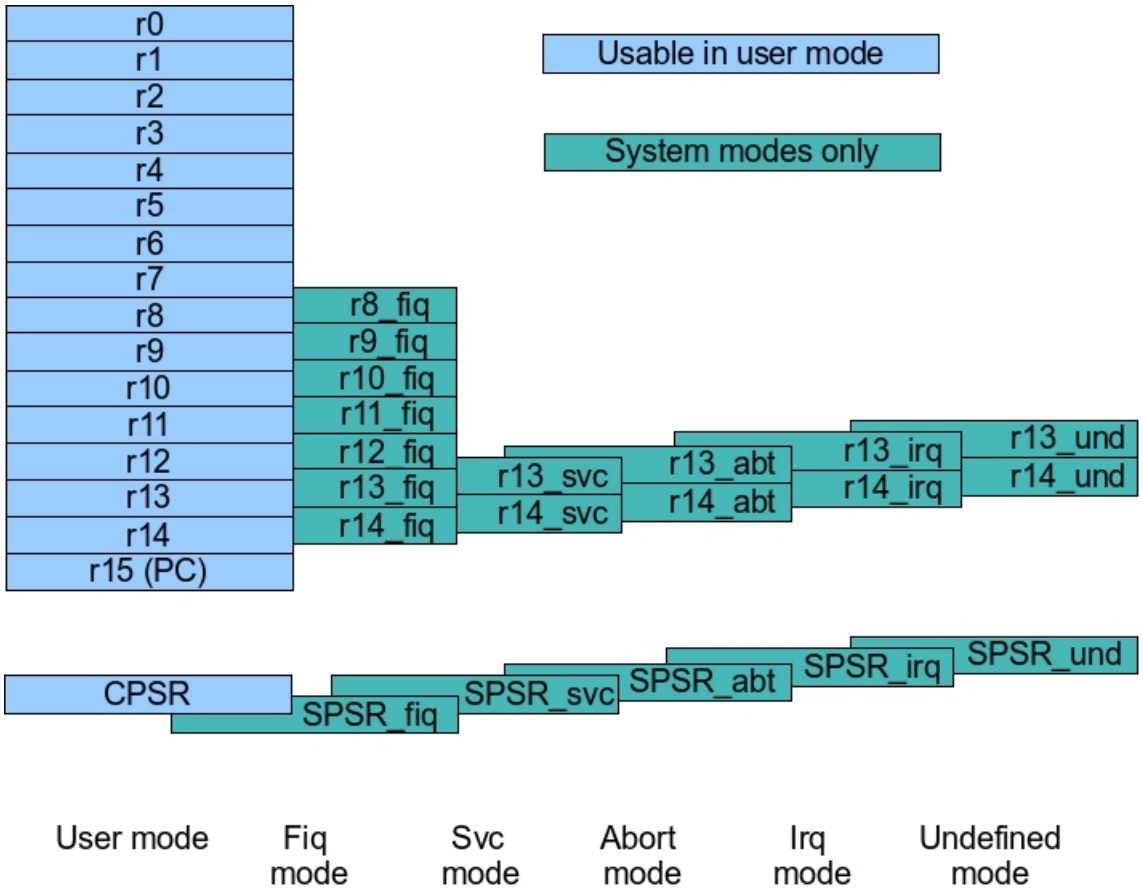
- Bit de desactivación de FIQ: Desactiva las interrupciones rápidas.

- Bit de desactivación de IRQ: Desactiva las interrupciones normales.

- Flags de condición: 4 bits que indican condiciones del resultado de la última operación que los haya activado.
  - Flag N: Resultado negativo.
  - Flag Z: Resultado cero.
  - Flag C: Resultado con acarreo.
  - Flag V: Resultado con desbordamiento en el bit de signo.

Los registros SPSR son similares al CPSR y se usan para preservar el estado de éste al cambiar de modo.

El registro 15 hace la función de contador de programa (PC), aunque también puede ser utilizado como registro de propósito general en la mayoría de las operaciones.



Para implementar el contexto contamos con una estructura ARM9Reg32 que representa un registro físico.
Esta estructura nos permite acceder al mismo en distintos formatos:

- Palabra de 32 bits con o sin signo.
- Media palabra de 16 bits con o sin signo.
- Acceso a los dos bytes más bajos de forma independiente con o sin signo.
- Acceso a los distintos bits del formato del registro CPSR/SPSR.

La implementación en C++ de este sistema se realiza usando la característica “union” que permite solapar distintos modos de acceso sobre el mismo dato físico.

El contexto implementa el banco de registros en su conjunto. En primera instancia se optó por tener 31 registros físicos y utilizar un array de 16 punteros que tomarían las direcciones de memoria de los 16 registros del modo actual. Al realizarse un cambio de modo era necesario actualizar este array con las nuevas direcciones.

Sin embargo, esto complicaba el acceso a los valores de los registros al producirse demasiadas indirecciones. Debido a esto, se decidió implementar otra solución que, si bien tiene una penalización mayor a la hora de cambiar de modo, es más eficiente a la hora de acceder a los datos.

Se disponen de 16 registros físicos ARM9Reg32 que se corresponden a los registros accesibles por el modo actual. Además de estos registros, existen otras tres tablas que guardan los valores de los bancos de registros que no están accesibles en este momento (registros del FIQ, registros 13 y 14, registros SPSR).

Al producirse un cambio de modo, se intercambian los contenidos entre los registros visibles y las tablas de los bancos de registros dependiendo del modo anterior y actual. Esto se implementa en el método cambiarModo de la clase.

Para acceder de forma eficiente a los datos del CPSR, este registro no se implementa como una estructura ARM9Reg32 debido a la penalización que tendría desempaquetar sus bits, dado que se usan con mucha frecuencia.
De este modo cada uno de los campos del CPSR se guarda de forma separada. A la hora de trabajar con el CPSR como un registro de 32 bits es necesario disponer de un método para reconstruirlo a partir de los campos independientes, y otro que haga el proceso inverso. Estos métodos están disponibles en la clase ARM9Contexto.

Por último esta clase también incluye un identificador de la última excepción marcada, para poder realizar las comprobaciones de prioridad.

2.3 – Bucle principal

El ARM9Core funciona de forma secuencial. El flujo de ejecución del emulador se organiza en torno a un bucle que realiza los pasos correspondientes a la búsqueda, decodificación y ejecución de las instrucciones. Este bucle ejecuta grupos de instrucciones consecutivas durante un número determinado de ciclos (*"slice"*). Por cada *slice* se realizan las tareas de tratamiento de excepciones, sincronización y llamadas a funciones externas.

```
while (no fin ejecución){

        numero ciclos restantes = tamaño slice;

        while (numero ciclos restantes > 0){

                ejecutar instrucciones (numero ciclos restantes);
                tratar excepciones ();
        }

        sincronizar ();
        función externa ();
}
```

Como se ve en el pseudocódigo anterior, el bucle más externo se repite hasta que se indica al emulador que termine su ejecución (normalmente desde la aplicación externa). En cada iteración se ejecuta a su vez un bucle interno encargado de la emulación del *slice* de tiempo (este *slice* viene determinado por una cantidad de ciclos que puede especificar el usuario).

La clase ARM9RutinasARM realiza la ejecución de las instrucciones hasta que o bien se acaba el *slice*, o bien se produce una excepción, en cuyo caso se trata y se repite el proceso con los ciclos restantes del *slice*.

Después de ejecutar cada *slice* se sincroniza la velocidad del emulador con la velocidad especificada por el usuario, y posteriormente se hace una llamada a la función externa. Esta función la implementa el usuario en su aplicación, de forma que éste pueda recuperar el control de la ejecución para realizar sus propias tareas (emulación de sistemas de entrada y salida, interrupciones externas, etc...).

La razón de que las instrucciones se ejecuten en *slices* es para conseguir la mayor eficiencia posible. El código de emulación de rutinas es más eficiente cuanto mayor es el grupo de instrucciones consecutivas que se ejecutan, por tanto, la velocidad de ejecución será mayor para valores grandes del *slice*.

Por otra parte, la emulación de las interrupciones externas se realiza en la llamada a la función externa, que se hace únicamente cuando termina cada *slice*. Por tanto, la emulación de estas interrupciones será más precisa cuanto menor sea dicho *slice*.

El usuario tendrá que buscar un balance entre eficiencia y precisión, y ajustar los ciclos del *slice* de acuerdo a sus intereses. Como regla práctica se suele usar la frecuencia de la interrupción más rápida, que puede ser la señal de refresco de la pantalla, un temporizador, etc ...

Por ejemplo, si queremos emular un dispositivo portátil construido sobre un procesador ARM funcionando a una frecuencia de 25 Mhz, y que dispone de una pantalla que se refresca 50 veces por cada segundo (50 Hz), el tiempo de *slice* se podría calcular del siguiente modo:

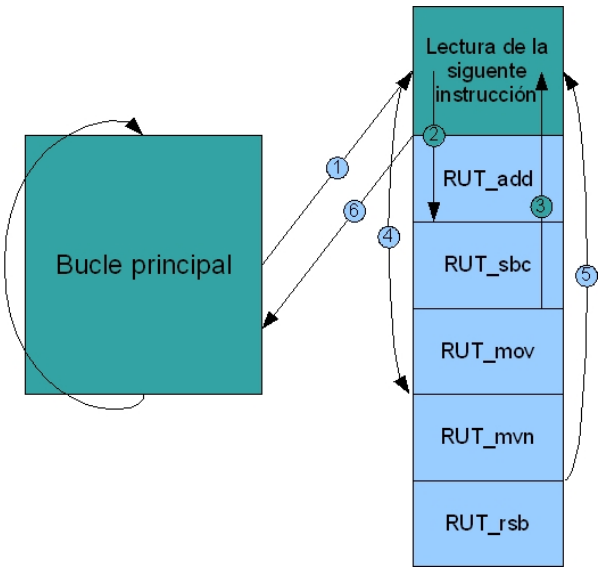
50 Hz -> 1/50 segs entre cada dos actualizaciones de pantalla

25 Mhz = $25*10^6$ ciclos por segundo
$25*10^6 / 50 = 500000$ ciclos de *slice*

Ejecución de instrucciones

El cuerpo interno del bucle principal transfiere el control a la clase de Rutinas encargada de la emulación del repertorio de instrucciones correspondiente.

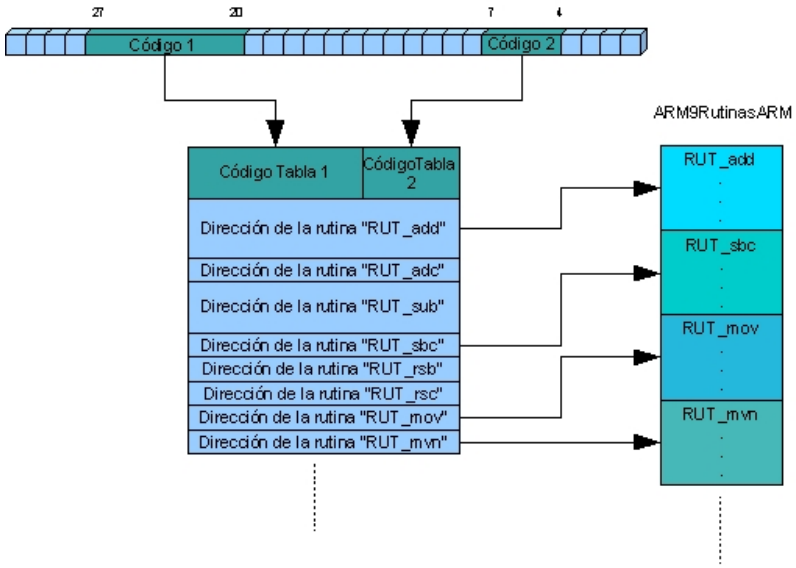
Para ganar eficiencia el flujo de ejecución se mantiene siempre en el cuerpo de una misma y única función que contiene todo el código de todas las rutinas de emulación de las distintas instrucciones. Esto evita las penalizaciones por la creación y destrucción de los registros de activación cada vez que se invoca a una función, al realizar únicamente saltos entre distintos puntos de este código.



El primer paso al entrar en la rutina de tratamiento es la lectura de la siguiente instrucción apuntada por el PC. Aquí se accede a memoria para leer esta instrucción, se incrementa el contador del programa, y se salta al segmento de código encargado de emular dicha instrucción una vez sea identificada a través de sus campos comunes. Al terminar de emularla, se pasa a leer la siguiente mientras queden ciclos en el *slice*.

Dado el complejo repertorio de instrucciones del que dispone la arquitectura ARM v4T, con varios formatos posibles para una misma instrucción y alta carga semántica, la identificación de la instrucción leída se convierte en una tarea costosa. Con el objetivo de evitar el impacto que supondría hacer tantas comprobaciones como instrucciones existen, el ARM9Core dispone de una tabla precalculada de saltos.

Esta tabla almacena las direcciones de salto correspondientes al código de las rutinas que implementan las distintas instrucciones, indexadas a través de los campos de bits que permiten diferenciarlas. De esta forma, el proceso consiste en extraer estos campos de bits, y saltar directamente a la dirección que se almacena en la tabla de saltos para esa entrada.



Esta tabla se inicializa dinámicamente durante la primera ejecución de la primera instrucción a emular del programa. Esto provoca una penalización de tiempo en el primer *slice*, que se compensa más adelante al continuar la ejecución del programa.

La creación de la tabla no se realiza en la fase de compilación debido a la imposibilidad de determinar las direcciones de salto de dichas subrutinas en código C/C++. Para ello es necesario recurrir al ensamblador del x86, que al ejecutarse permite leer las etiquetas de salto y almacenar sus direcciones en la memoria:

```
__asm{
     lea eax, etiqueta
     mov direccion, eax
}
tabla[entrada] = direccion;
```

Como se ve en este ejemplo, la instrucción del x86 “lea” toma la dirección de la etiqueta y la almacena en un registro (eax), que luego es transferido a la variable “direccion”. A partir de aquí, esta variable puede ser manipulada en código C/C++ tradicional.

Ejecución condicional

Todas las instrucciones de la arquitectura ARM permiten la ejecución condicional. Esto es, se incluye un código de condición en cada una que se compara con los bits de estado actuales del procesador. Dependiendo del resultado, la instrucción se ejecuta o no. En caso de no ejecutarse, se trata como una “NOP” (consumiendo un ciclo) y se continúa por la siguiente.

Los posibles códigos de condición para las instrucciones son:

Sufijo	Descripción	Condición
EQ	Equal	Z = '1'
NE	Not Equal	Z = '0'
CS / HS	Unsigned higher or same	C = '1'
CC / LO	Unsigned lower	C = '0'
MI	Minus / Negative	N = '1'
PL	Plus / Positive or zero	N = '0'
VS	Overflow	V = '1'
VC	No overflow	V = '0'
HI	Unsigned higher	C = '1' y Z = '0'
LS	Unsigned lower or same	C = '0' ó Z = '1'
GE	Signed greater than or equal	N = V
LT	Signed less than	N ≠ V
GT	Signed greater than	Z = '0' y N = V
LE	Signed less than or equal	Z = '1' ó N ≠ V
AL	Always	Cualquiera
NV	Never (No usar)	Ninguno

Estos 16 posibles códigos de condición se codifican siempre como los 4 bits más significativos de cada instrucción. A la hora de emular lo primero que debe hacer cada rutina es la comprobación para saber si debe o no ejecutarse.

Para agilizar este proceso, el ARM9Core utiliza una tabla precalculada de condiciones. Esta tabla tiene una entrada por cada posible valor de los flags de estado y de los bits de condición de la instrucción. De esta forma la rutina de emulación sólo tiene que consultar el valor de la tabla para los valores actuales, para saber si se debe ejecutar o no.

Dado que hay 16 posibles combinaciones tanto para los códigos como para los flags, el tamaño de la tabla es de 256 entradas, lo cual no es excesivo para la ganancia en eficiencia que se obtiene.

Esta tabla se genera al inicializar el emulador, por lo que a diferencia de la tabla de saltos, no se incurre en ninguna penalización para la primera ejecución.

Sincronización

Un emulador intenta siempre reproducir el mismo comportamiento externo de la máquina real. Esto quiere decir que un programa diseñado para esta máquina origen

debe ejecutarse también a la misma velocidad al ser emulado. De no ser así, programas que incluyan interfaces para el usuario se volverían inmanejables.

El ARM9Core permite al usuario especificar una frecuencia objetivo a la que quiere que la máquina emulada trabaje. Sabiendo esto y el número de ciclos por *slice*, el emulador es capaz de calcular el tiempo real que debería tardar dicho *slice* en ejecutarse.

La sincronización se hace entonces a nivel de *slice*. Se mide el tiempo que se ha tardado en ejecutarlo, y se compara con el tiempo objetivo deseado. Si se ha tardado menos, el emulador se detiene e introduce una espera hasta que ambos tiempos coincidan. De esta forma, la velocidad de emulación se corresponderá con la velocidad deseada.

En caso de que la frecuencia objetivo introducida sea muy alta, es probable que sea imposible alcanzarla. Si la emulación es más lenta, no se introduce espera alguna y se intenta ir a la velocidad más rápida posible.

Dado que los tiempos con los que se trabaja son del orden de los milisegundos es necesario disponer de un sistema de medida suficientemente preciso. Para ello el ARM9Core hace uso de las funciones de la API de Win32 "QueryPerformanceCounter" y "QueryPerformanceFrequency" que devuelven, respectivamente, el valor del contador de alta precisión del x86 y su frecuencia.

Estas funciones utilizan valores de 64 bits lo que permite obtener precisiones del orden de 10^{-9} segundos (nanosegundos), lo cual es más que suficiente para nuestras necesidades.

El proceso de sincronización se realiza como muestra el siguiente ejemplo:

```
tiempoInicio = calcularTiempoActual();
EjecutarSlice ();
tiempo = calcularTiempoActual()- tiempoInicio;

//Sincronizacion
while (tiempo < tiempoObjetivo) {
      Sleep (0);
      tiempo = calcularTiempoActual() - tiempoInicio;
}
```

A la hora de calcular los ciclos consumidos para obtener el tiempo objetivo que debería ocupar el *slice* es necesario tener en cuenta durante la emulación de las instrucciones el número de ciclos que consume cada una de ellas. En el caso de que se ejecuten más ciclos de los especificados en el tamaño del *slice*, estos serán descontados en la siguiente iteración.

Emulación multinúcleo

El ARM9Core permite también la emulación de un sistema con varios núcleos (cores), cada uno con un contexto independiente. Todos ellos acceden a la misma memoria y se ejecutan a una misma frecuencia. Cuando se detiene el procesador también se detienen cada uno de los núcleos, de forma que siempre se ejecutan a la par.

Sin embargo, dado que el emulador es secuencial es necesario emular cada uno de los núcleos de forma consecutiva. Por tanto se ejecutan tantos *slices* como cores haya definidos. Por ejemplo, si se especifica un *slice* de 100000 ciclos y un procesador con 4 núcleos, cada iteración del bucle representará la emulación de 400000 ciclos (100000 por cada uno).

A la hora de sincronizar, se utiliza la medida del total de ciclos consumidos por todos los núcleos en conjunto.

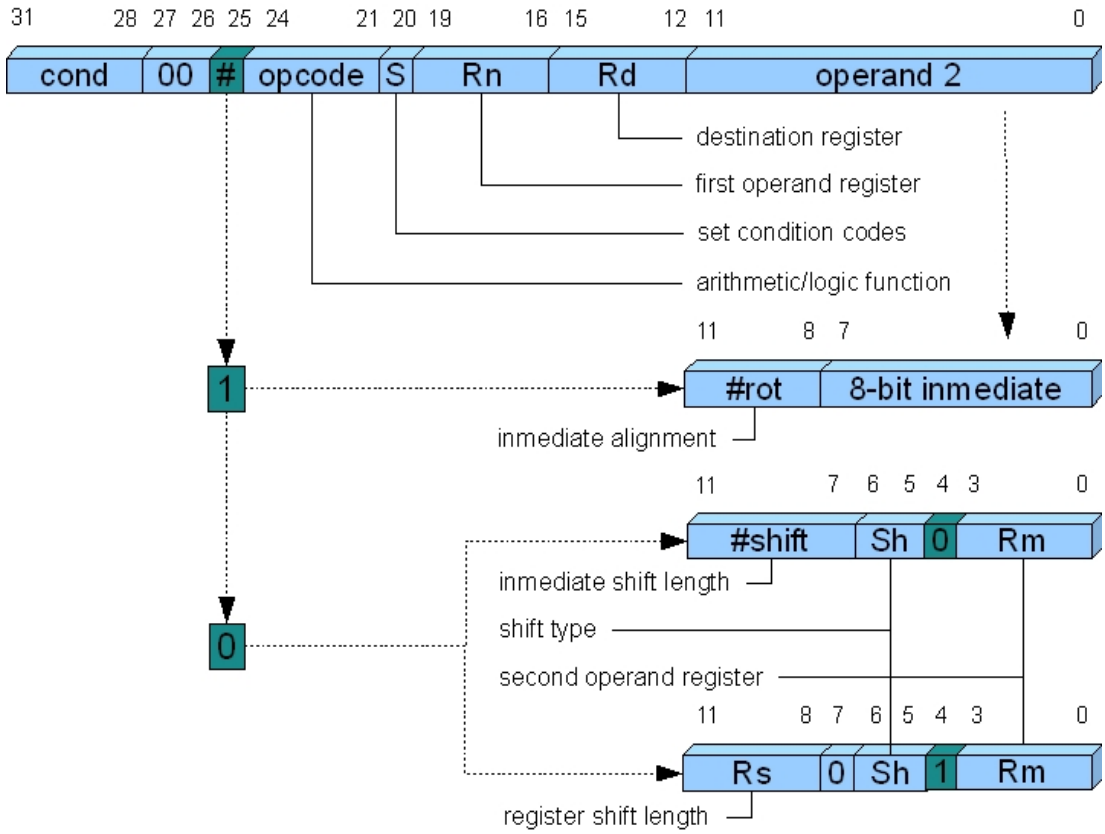
Por último, todos los cores comparten los mismos métodos de tratamiento de excepciones.

3. Instrucciones ARM

3.1 – Instrucciones Aritméticas y Lógicas

Formato:

<op>{<condición>}{S} Rd, Rn, #<valor inmediato de 32 bits>
<op>{<condición>}{S} Rd, Rn, Rm, {<shift>}
<op> puede ser cualquiera de las 16 instrucciones aritméticas
<shift> especifica el tipo de desplazamiento, seguido de la cantidad (valor inmediato de 5 bits o registro)



Descripción:

Las instrucciones aritméticas y lógicas (data processing) en la arquitectura ARM emplean un formato de 3 operandos: un registro destino y dos operandos fuente. De estos dos operandos fuente el primero siempre es un registro (Rn), mientras que el segundo puede ser otro registro (Rm) o un operando inmediato.

En el caso de que el segundo operando sea también un registro, el ARM permite aplicarle un desplazamiento antes de realizar la operación aritmética. Este desplazamiento puede venir dado por un valor inmediato de 5 bits o por un cuarto registro (Rs).

Todas las instrucciones aritméticas generan nuevos valores para los bits de condición, pero estos sólo se guardan si el bit 'S' de la instrucción está activado.

A partir de las posibles configuraciones que puede adoptar el segundo operando se obtienen los tres posibles formatos para las instrucciones aritmético-lógicas, y que aparecen en el mismo orden en la figura anterior:

DPI (Data Processing Inmediate): Operando inmediato
DPIS (D. P. Inmediate Shift): Registro desplazado por valor inmediato
DPRS (D. P. Register Shift): Registro desplazado por valor del registro Rs

Aunque todos los posibles formatos toman un registro destino y dos fuente, no todas las instrucciones aritméticas los usan. Por ejemplo, las instrucciones de transferencia de registros (MOV y MVN) ignoran el primer operando, y las instrucciones de comparación (TST, TEQ, CMP y CMN) no escriben en ningún registro destino.

Y por último, se puede especificar un mismo registro como destino y como fuente al mismo tiempo, así como usarlo simultáneamente para el primer y segundo operando, sin ninguna restricción.

Operando Inmediato:

La arquitectura ARM usa operandos de 32 bits. En el caso de usar un operando inmediato, sólo se disponen de 8 bits para codificar su valor. Pero dado que el ARM dispone de un desplazador de barril que le permite aplicar rotaciones a los operandos con una penalización de tiempo de ejecución nula, la arquitectura permite usarlo también con los operandos inmediatos.

De esta forma, se permite que el operando de 8 bits sea rotado por una cantidad par que también se especifica de forma inmediata en la propia instrucción. El valor por el que se desplaza al operando es el doble del indicado en el campo #rot, lo que permite cubrir los 32 bits posibles.

Esto es, en el ARM se puede especificar como operando inmediato cualquier número de la forma: $[0\text{-}255] \times 2^{2n}$

Registro desplazado:

Como hemos dicho antes, una de las características fundamentales de la arquitectura ARM es su desplazador de barril. El poder aplicar desplazamientos a los valores de los registros en cualquier instrucción aritmética sin apenas ninguna penalización da mucha versatilidad al repertorio, por lo que se convierte en una de las funciones más usadas por los programas diseñados para estos procesadores.

Estos desplazamientos nos permiten multiplicar por constantes pequeñas de forma mucho más eficiente que las instrucciones MUL. Por ejemplo, para multiplicar R0 por 5, podemos desplazarlo dos posiciones a la izquierda y sumarlo consigo mismo, como hace la siguiente instrucción:

```
ADD  r0, r0, r0, LSL #2        ;r0 := r0*5;
```

En este caso, se trata de una instrucción DPIS, puesto que la cantidad por la que desplazamos viene dada de forma inmediata (#2).

LSL indica que se realiza un desplazamiento a la izquierda. El ARM nos permite especificar distintos tipos de desplazamiento, que se codifican en el campo "Sh" de la instrucción:

- **LSL** (00)**:** Desplazamiento lógico a la izquierda. Rellena con ceros.
- **ASL** (00): Desplazamiento aritmético a la izquierda. Igual que el anterior.
- **LSR** (01): Desplazamiento lógico a la derecha. Rellena con ceros.
- **ASR** (10): Desplazamiento aritmético a la derecha. Rellena con el signo del operando original (0 si positivo, 1 si negativo).
- **ROR** (11): Rota a la derecha. Al desplazar, el bit menos significativo que se pierde se usa para rellenar por la izquierda.
- **RRX** (11): Rota a la derecha una posición. El bit menos significativo se guarda como bit de carry, y el más significativo se rellena con el valor del carry antiguo.

Los desplazamientos ROR y RRX comparten la misma codificación (11). Para distinguirlos es necesario comprobar el valor de la cantidad (#shift). Como RRX siempre desplaza una única posición, no necesita usar este campo, que pone a cero. Si encontramos un valor distinto de cero, sabremos que estamos ante un ROR.

En el caso de que la instrucción no realice ningún tipo de desplazamiento, nos encontraremos con una codificación LSL #0 (es decir, todos los bits a cero). Saber esto es importante a la hora de emular, puesto que nos permite optimizar estos casos.

Uso de r15

Podemos especificar el registro del PC (r15) como operando fuente o destino en cualquiera de las instrucciones aritméticas. Si lo usamos como fuente, su valor será el de la dirección de la instrucción actual + 8, por el comportamiento del pipeline segmentado.

La única restricción es que no podremos tomarlo como segundo operando en el caso de tratarse de una instrucción DPRS.

En el caso de usarlo como destino, la instrucción aritmética se convierte en un salto. Esto también altera el comportamiento del bit 'S'. Si está activado, en lugar de actualizar los flags, el ARM copiará el valor del registro SPSR del modo actual sobre el CPSR.

Esta es una forma común de regresar de subrutinas e interrupciones. Por ejemplo, la siguiente instrucción se usaría para finalizar el tratamiento de una interrupción y restaurar el estado original de la máquina:

```
MOVS pc, r14
```

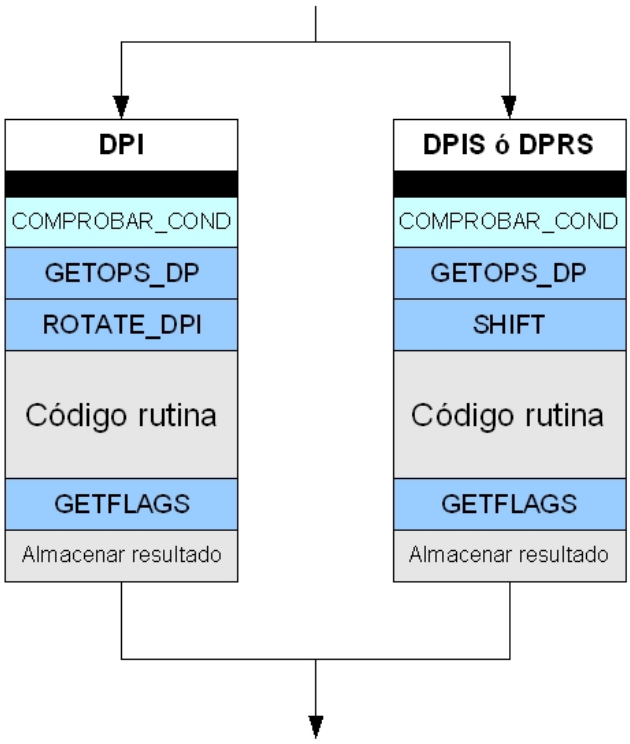
Esto copia el registro r14 (link register) al PC, haciendo que este último apunte de nuevo al lugar del código donde se produjo la interrupción, y restaura el CPSR, cambiando así el modo de ejecución del procesador y los bits de activación de interrupciones a sus valores originales.

Emulación:

A la hora de emular las instrucciones aritméticas del ARM nos encontramos con dos factores a tener en cuenta que toman una especial importancia y que son los que condicionan en gran medida la implementación: por una parte la gran carga semántica de las instrucciones y los distintos formatos posibles, que complican la decodificación, y por otra parte el efecto del desplazador en el rendimiento.

Inicialmente, y dado que los bits que permiten distinguir entre formatos (bits 25 y 4) aparecen en nuestras tablas de saltos, consideramos la opción de crear tres subrutinas para cada instrucción aritmética, una por cada formato posible que esa instrucción puede adoptar.

Tras considerar la complejidad de esa solución, optamos por tener una única subrutina por instrucción, pero con dos flujos de ejecución diferenciados dependiendo del formato: uno para las instrucciones DPI, y otro para las DPIS y DPRS; como se muestra en el siguiente diagrama:



COMPROBAR_COND es la macro encargada de la ejecución condicional, y compara los bits de condición con los flags para determinar si la instrucción se ejecuta. Esto es común para todas las instrucciones.

En azul oscuro se indican las macros específicas de las instrucciones aritméticas. GETOPS_DP es la encargada de decodificar la instrucción leída, y guardar los distintos operandos en variables para su uso posterior. Esta macro es común para ambos formatos, por lo que en ocasiones decodifica campos que no tienen sentido para el formato de la instrucción, aunque esto no supone un gran impacto en el rendimiento.

Los dos flujos de ejecución difieren a la hora de preparar su segundo operando. En el caso de las DPI, se usa la macro ROTATE_DPI, que extrae el operando inmediato y lo rota por la cantidad indicada en la propia instrucción usando instrucciones del ensamblador del x86.

En el caso de las DPIS o DPRS se invoca a la macro SHIFT, que emula el comportamiento del desplazador de barril del ARM. Esta macro toma como entradas un valor (el contenido del registro Rm) y una cantidad (el valor inmediato o el contenido del registro Rs) ambas proporcionadas por GETOPS_DP.

SHIFT comprueba el tipo de desplazamiento de la instrucción (el campo Sh) y ejecuta el código necesario para emular ese desplazamiento en el x86. Dado que las instrucciones del x86 tienen un comportamiento diferente a las del ARM para desplazamientos superiores a 31 bits, es necesario también detectar estos casos especiales y calcular directamente el resultado.

Si bien este tipo de desplazamientos es poco útil en la práctica, decidimos que merecía la pena emularlos correctamente puesto que algunos programas podrían recurrir a ellos. Por tanto, la penalización total de la emulación del desplazador es equivalente a una media de 5 instrucciones del x86 y dos o tres comparaciones condicionales.

Como comentábamos antes, en el caso de encontrarnos con un LSL #0 entendemos que no hay desplazamiento, por lo que no se ejecuta ninguna instrucción adicional. En ese caso, no hay penalización.

Independientemente de cuál de estas dos macros se ejecute, ambas tienen en común que al terminar almacenan el valor del primer operando en el registro EAX del x86, y el valor del segundo operando (debidamente calculado) en EBX. De forma que cada rutina específica para cada instrucción (ADD, SUB, etc...) puede ejecutar la operación correspondiente directamente sobre estos registros sin más complicaciones.

Emulación de los flags

Cualquier instrucción ARM puede modificar el valor de los flags del procesador activando el bit 'S'. La emulación de este comportamiento es crucial para que los programas se ejecuten correctamente. Como calcular manualmente los valores de estos flags para cada operación sería muy costoso, decidimos utilizar los flags del propio x86 para optimizarlo.

En concreto, tras la instrucción x86 que realiza la operación a emular incluimos una instrucción “pushf”, que guarda el valor de los flags en la pila. La macro GETFLAGS es la que se encarga, si el bit 'S' está activado, de extraer estos valores y guardarlos en el contexto del emulador.

Sin embargo existen algunas diferencias entre el comportamiento de los flags en el x86 y el ARM. Por ejemplo, el bit de carry no toma los mismos valores en las operaciones de resta. Por ello, emulamos las restas como un complemento del segundo operando (o del primero) seguido de una suma. El valor del carry obtenido así es correcto, pero nos implica tener que generar el bit de overflow de forma manual, comprobando los signos antes y después de la operación.

Otro comportamiento especial de los flags del ARM está relacionado con el uso del desplazador de barril. Este desplazador puede generar su propio bit de carry, que es ignorado por las instrucciones aritméticas. Las instrucciones lógicas (AND, EOR, TST, TEQ, ORR, MOV, MVN, BIC) en cambio usan este valor como carry final.

Para emular esto, las macros ROTATE_DPI y SHIFT guardan su propio carry en una variable auxiliar, que se escribe en el contexto o no dependiendo del tipo de instrucción.

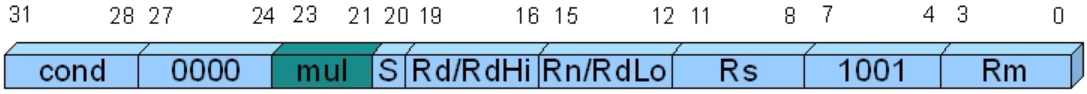
Por último, la macro GETFLAGS también es la encargada de restaurar el CPSR con el valor del SPSR del modo actual si el registro destino es el PC, como comentábamos antes.

3.2 – Multiplicación

Formato:

32 bits:
MUL{<condición>}{S} Rd, Rm, Rs
MLA{<condición>}{S} Rd, Rm, Rs, Rn

64 bits:
<mul>{<condición>}{S} RdHi, RdLo, Rm, Rs
<mul> puede ser: UMULL, UMLAL, SMULL, SMLAL



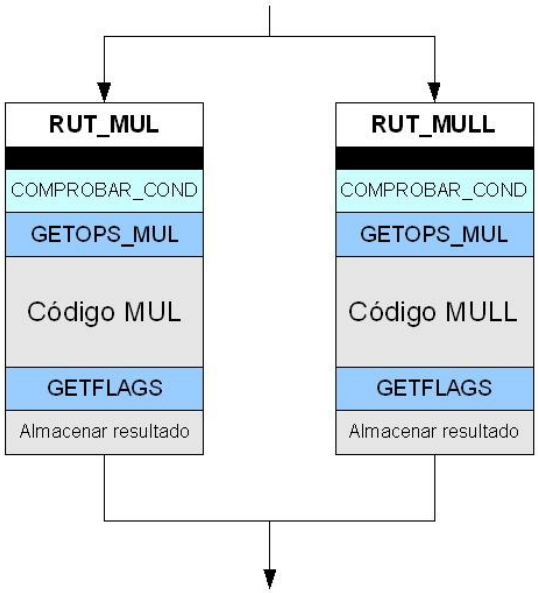
Descripción:

El ARM presenta dos tipos de multiplicaciones, con resultados de 32 bits (se pierden los 32 bits más altos), o de 64 bits (divididos en dos registros de 32, RdHi, el más significativo, y RdLo, el menos significativo).

Estas operaciones además permiten la multiplicación con acumulador (registro Rn para 32 bits o registros RdHi, RdLo para 64), y con o sin signo (sólo para 64 bits).

Emulación:

Decidimos separar la emulación en dos rutinas, una para las multiplicaciones de 32 bits y otra para las de 64:



GETOPS_MUL es una macro común para obtener los operandos. El ensamblador del x86 proporciona una instrucción de multiplicación de 64 bits que, al igual que la del ARM, almacena la parte alta y baja del resultado en dos registros distintos de 32 bits.

Por tanto, el código de la rutina de 32 bits realiza una multiplicación y conserva únicamente el registro bajo del resultado, al que se le suma el valor del acumulador si es necesario.

Las multiplicaciones de 64 bits sin acumulador se emulan igualmente recurriendo a la instrucción del x86, y conservando esta vez ambos registros.
Sin embargo, el x86 no dispone de una suma de 64 bits, por lo que tras examinar varias alternativas, decidimos recurrir a las extensiones MMX, que son comunes en los procesadores actuales, para emular la multiplicación de 64 bits con acumulador.

De esta forma, tras llamar a la instrucción de multiplicación, usamos las MMX para montar los registros de 64 bits a partir de los dos de 32 donde se almacenó el resultado. Después realizamos la suma del acumulador (también montado en otro registro MMX), y desempaquetamos por último el resultado final sobre los dos registros de 32 bits.

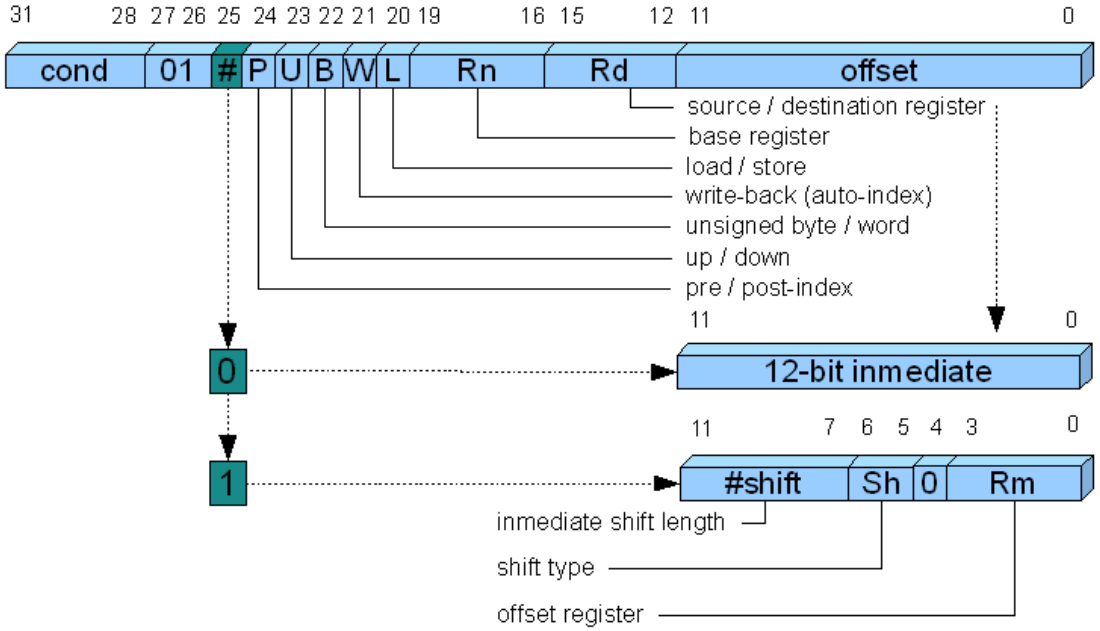
Dado que la multiplicación del x86 también puede trabajar con o sin signo, no es necesario añadir más complejidad a nuestro código para emular este comportamiento.

3.3 – Transferencia de una palabra y byte sin signo

Formato:

Forma pre-indexada:
LDR{<condición>}{B} Rd, [Rn, <offset>]{!}
STR{<condición>}{B} Rd, [Rn, <offset>]{!}

Forma post-indexada:
LDR{<condición>}{B}{T} Rd, [Rn], <offset>
STR{<condición>}{B}{T} Rd, [Rn], <offset>



Descripción:

Estas instrucciones de transferencia de datos construyen la dirección de almacenamiento o carga a partir del registro Base (Rn) sumando o restándole un valor sin signo inmediato o el valor de un registro (Rm) posiblemente escalado.

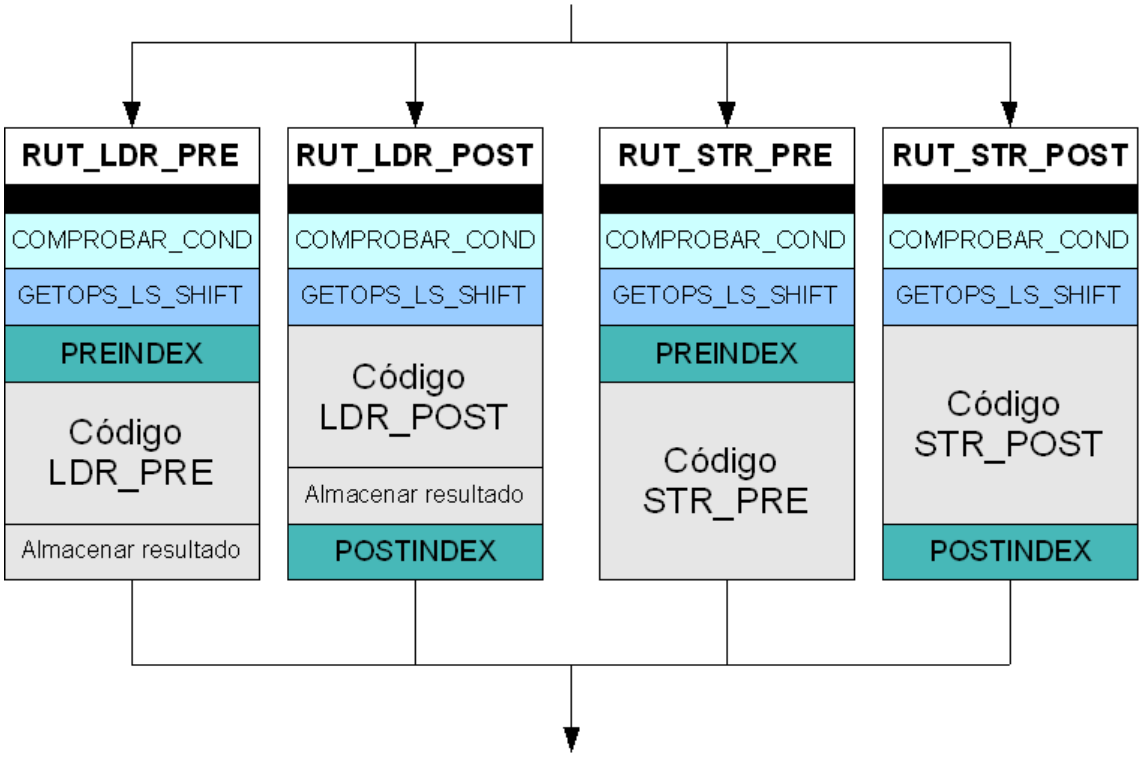
Cuando un byte es cargado en un registro, éste es extendido con 0's hasta los 32 bits, y cuando el byte es guardado en la dirección de almacenamiento en memoria, éste corresponde a los 8 bits menos significativos del registro Rm (escalado o no).

En direccionamiento pre-indexado la dirección de memoria siempre es la combinación del registro Base y del desplazamiento y se calcula antes de realizar la transferencia. El registro Base solamente será actualizado con dicha dirección si el bit Write-Back está activo.

Sin embargo, en post-indexado se utiliza solamente el valor del registro Base para la transferencia pero después siempre se almacena en el registro Base la combinación de su valor y el desplazamiento, independientemente del bit Write-Back.

Emulación:

Para la emulación de estas instrucciones dividimos las rutinas a realizar según el tipo de instrucción (Load/Store) y el modo de direccionamiento (Pre/Post-indexado).



GETOPS_LS_SHIFT es la macro encargada (por medio de la macro interna GETOPS_LS) de obtener el valor del registro Base, el registro destino/fuente, desplazamientos, tipo de la transferencia (palabra o byte), bit Write-Back y tipo de la combinación del registro Base con el desplazamiento, es decir, suma o resta. Si el registro Base es el registro 15 (o PC) el valor utilizado se verá incrementado en 4 bytes.
Además el valor del desplazamiento será escalado (gracias a la macro SHIFT) si se detecta que el bit 25 de la instrucción es 1 o el desplazamiento es un registro.

Según el direccionamiento se llamará a la macro PREINDEX o POSTINDEX. Las dos calcularán la dirección efectiva (para la transferencia en la primera y para actualizar el registro Base en la segunda) sumando o restando el desplazamiento según el bit U de la instrucción. En PREINDEX si el bit Write-Back está activo se almacenará tal dirección en el registro Base.

En el código de la instrucción Load con direccionamiento pre-indexado (y post) si la transferencia es de un byte, cargaremos el byte desde la dirección efectiva de memoria. Para una palabra, alinearemos la dirección efectiva y cargaremos la palabra desde la dirección alineada*. Si la palabra no estuviera alineada entonces la ajustamos, realizando las rotaciones necesarias en ensamblador de x86.
Esto también se hará de igual modo con el registro Base en post-indexado al acceder a memoria, y si se produce una excepción DATA ABORT no se cargará el valor en el registro en ambos casos. Sin embargo el registro Base sí se actualizará aunque se

* Por dirección alineada nos referimos a direcciones que hacen referencia a una palabra de 32 bits o 4 bytes. Cualquier dirección cuyos dos últimos bits tengan un valor distinto de 0 se considerarán direcciones no alineadas.

produzca este tipo de excepción al leer de memoria o suceda un error al escribir en memoria.

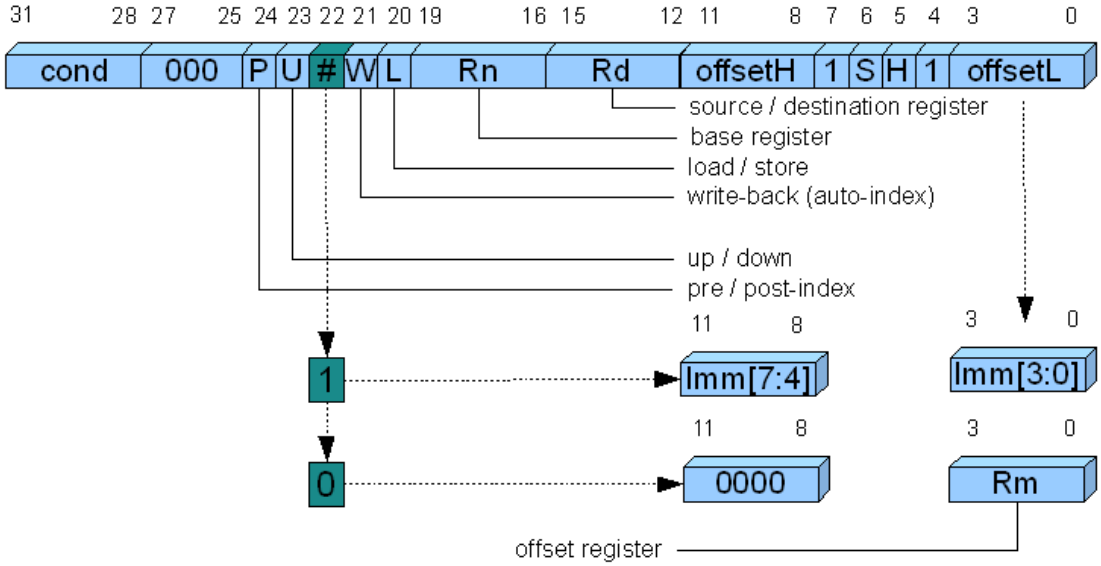
Para las instrucciones Store, si la transferencia es de un byte, almacenaremos en la dirección efectiva de memoria los 8 bits menos significativos del registro fuente.
Si la transferencia es de una palabra, la dirección efectiva será alineada de igual modo que en los Load, de forma que el valor del registro fuente será guardado en esta última dirección de memoria.

3.4 – Transferencia de media-palabra y byte con signo

Formato:

Forma pre-indexada:
LDR{<condición>}H|SH|SB Rd, [Rn, <offset>]{!}
STR{<condición>}H|SH|SB Rd, [Rn, <offset>]{!}

Forma post-indexada:
LDR{<condición>}H|SH|SB Rd, [Rn], <offset>
STR{<condición>}H|SH|SB Rd, [Rn], <offset>



Descripción:

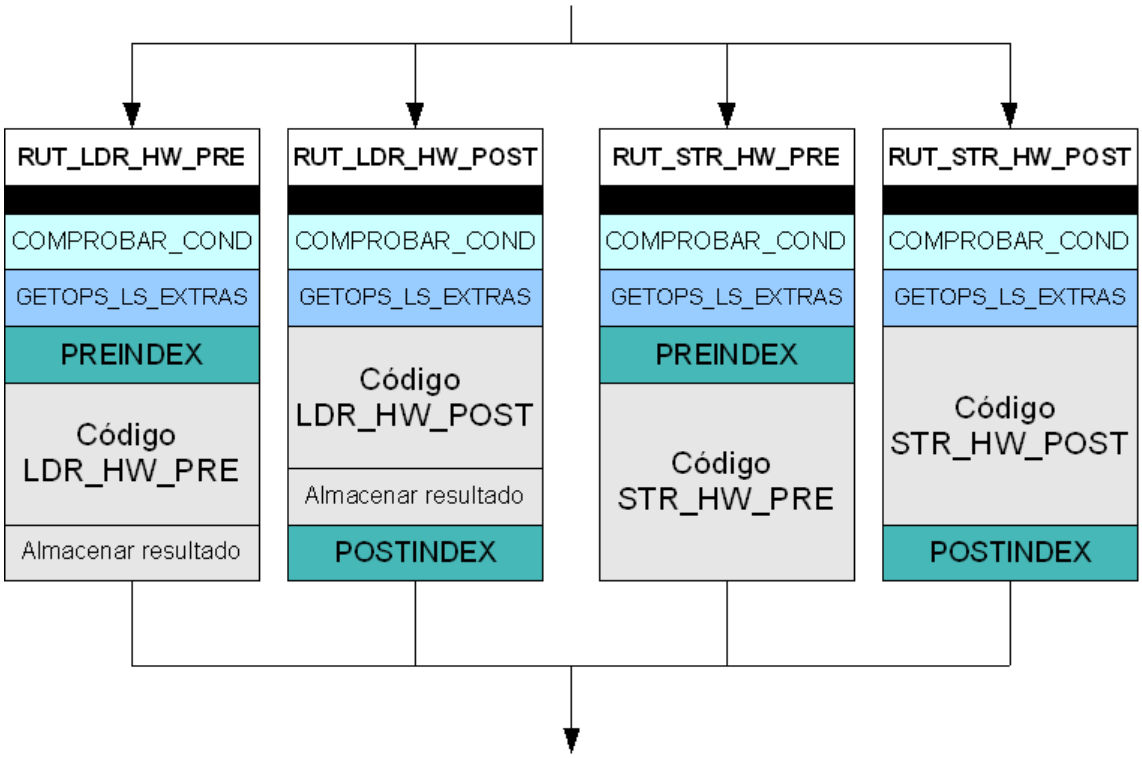
Este tipo de instrucciones es muy parecido a las de transferencia de datos de palabras y bytes sin signo pero en este caso el desplazamiento inmediato es reducido a sólo 1 byte y en el caso de que el desplazamiento sea un registro, su valor no puede ser escalado.

Para los Load de este tipo el dato puede ser byte con signo y media-palabra con y sin signo. Para los Store no importa el signo y se almacena media-palabra.

Cuando una media-palabra sin signo es cargada en un registro, éste es extendido con 0's hasta los 32 bits, y cuando es cargado un byte o media-palabra con signo, el registro es extendido hasta los 32 bits con el bit más significativo del dato.

Emulación:

Al igual que la emulación de las instrucciones de transferencia de palabras y bytes sin signo, aquí también dividimos las rutinas a realizar según el tipo de instrucción (Load/Store) y el modo de direccionamiento (Pre/Post-indexado).



GETOPS_LS_EXTRAS es la macro encargada de obtener los campos comunes a todas las instrucciones (por medio de la macro GETOPS_LS). En esta macro también se obtiene si el tipo de dato es byte con signo o media-palabra con y sin signo por medio de los bits 6 y 5 (S y H) para posteriormente acceder a memoria de forma apropiada. Si el bit 22 de la instrucción vale 1 entonces el desplazamiento es un valor inmediato de 1 byte que montamos mediante desplazamientos de bits usando los bits 11 a 8 como parte alta y los bits 3 a 0 como baja. Si es 0, el desplazamiento es el valor del registro Rm.

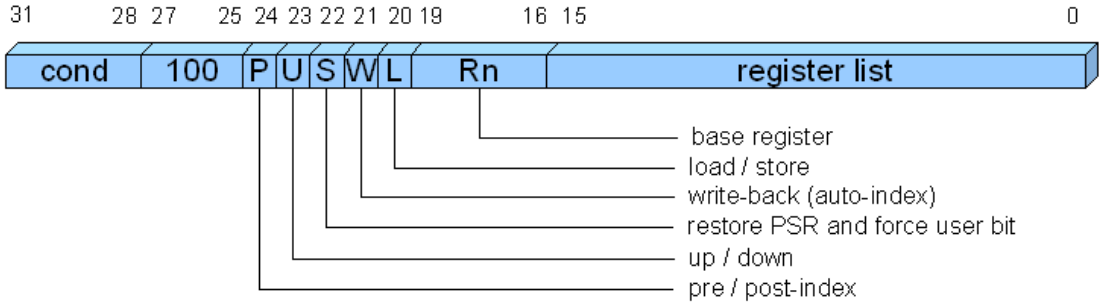
Para el código de la instrucción Load con direccionamiento pre-indexado se carga el byte con signo o media-palabra con o sin signo desde la dirección efectiva de memoria obtenida en la macro PREINDEX.
Con direccionamiento post-indexado se hará de igual modo desde la dirección de memoria indicada por el valor del registro Base.
Si el acceso a memoria provoca una excepción DATA ABORT entonces no se cargará el valor en el registro destino, pero sí se actualizará el registro Base.

Para las instrucciones Store simplemente almacenamos la media-palabra (con o sin signo) en la dirección efectiva de memoria o en la dirección de memoria indicada por el registro Base, según el tipo de direccionamiento.

3.5 – Transferencia de registro multiple

Formato:

LDM{<condición>}<modo direccionamiento> Rn{!}, <registros {+ pc}>{^}
STM{<condición>}<modo direccionamiento> Rn{!}, <registros>{^}
<modo direccionamiento> puede ser: IA, IB, DA, DB, FD, ED, FA, EA.



Descripción:

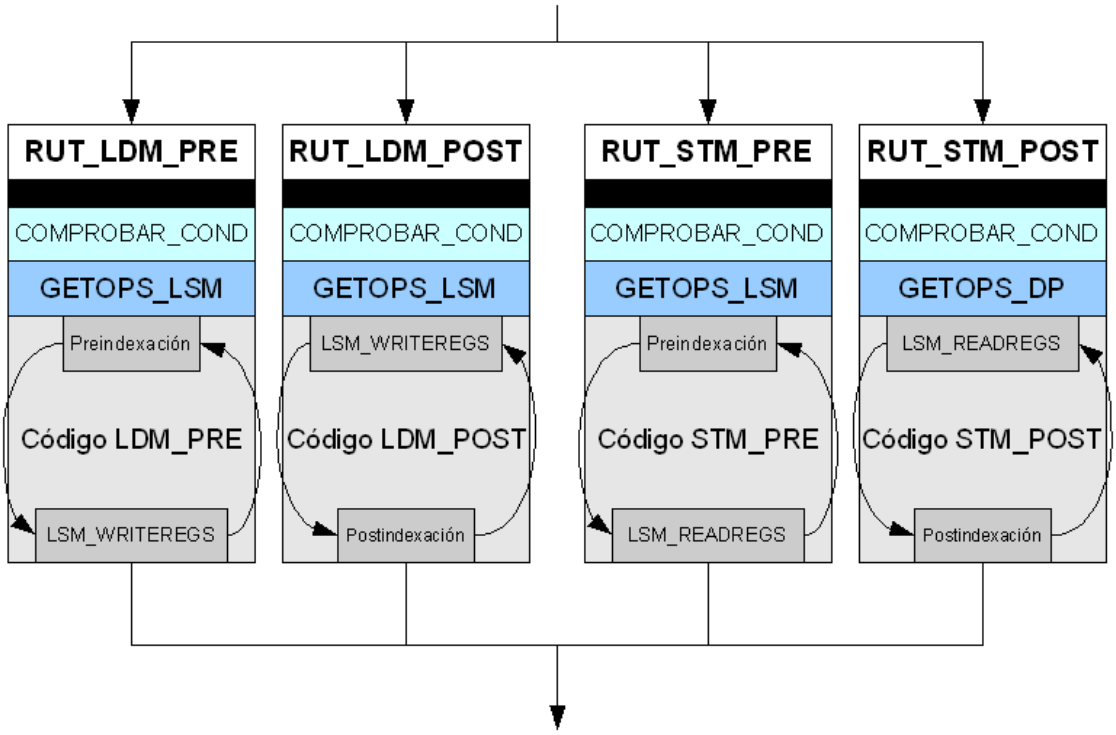
En este tipo de instrucciones cada bit de la lista de registros indica si el registro con índice igual a la posición de ese bit en la instrucción interviene en la transferencia. Es decir, si los bits 2, 5 y 8 de la lista de registros tienen valor 1, entonces los registros 2, 5 y 8 serán usados en la transferencia.

Los registros indicados en la lista serán cargados con las palabras que forman el bloque contiguo de memoria que comienza en la dirección del registro Base y se desplaza según el modo de direccionamiento en el caso de una operación Load. Para una operación Store las palabras de los registros indicados será almacenado en un bloque contiguo de memoria que comenzará en la dirección del registro Base y se desplazará según el direccionamiento. El valor del registro Base aumentará o decrementará antes o después de que una palabra se transfiera, y si el bit W está activo el registro Base se incrementará o decrementará con el número de bytes transferidos cuando se complete la instrucción.

Si el registro 15 (o PC) está en la lista de registros de un Load y el bit S está fijado entonces el SPSR del modo actual será copiado en el CPSR. Si el PC no está en la lista en un Load o Store ejecutados en modos distintos al modo usuario y el bit S está activo entonces se transfieren los registros del modo usuario.

Emulación:

La emulación de estas instrucciones al igual que las de transferencias de registro 'simple' se dividen según el tipo de instrucción (Load/Store) y el modo de direccionamiento (Pre/Post-indexado).



La macro GETOPS_LSM se encarga (por medio de la macro interna GETOPS_LS) de obtener el valor del registro Base y el bit U. Si el registro Base es el registro 15 (o PC) el valor utilizado se verá incrementado en 4 bytes. Aquí el offset siempre es el mismo y es el tamaño de una palabra, 4 bytes. También obtenemos la lista de los registros que actuarán en la transferencia y alineamos el valor del registro Base si no lo está. Además comprobamos si el PC está en la lista y el bit S está activo para uso posterior.

Independientemente de que la instrucción sea Load o Store, de que el direccionamiento sea pre/post-indexado y de que la dirección efectiva se incremente o decremente después de cada transferencia, el registro con menor índice que tenga bit a 1 en la lista estará asociado con la dirección efectiva más baja que se produzca en la instrucción.

En el caso de Load esto indica que en dicho registro se cargará la palabra del bloque contiguo con dirección efectiva de memoria más baja. Y para el Store indica que la palabra del bloque contiguo con dirección efectiva de memoria más baja contendrá el valor de dicho registro.

De esta forma anterior es como vamos seleccionando y operando con cada uno de los registros destinos que tienen el bit a 1 en la lista.

Para los Load pre-indexados sumamos o restamos (según el bit U) el offset para calcular la dirección efectiva y con esta dirección obtenemos la palabra de memoria.

Justo aquí se llama a la macro LSM_WRITEREGS que cargará la palabra obtenida en el registro destino actual. El registro destino actual puede ser el del modo actual u otro dependiendo de la siguiente política:

- PC no está en lista, bit S activo e índice del registro mayor igual que 8:
  - Índice menor que 13 y modo actual FIQ -> Registro del banco FIQ.
  - Índice menor que 13 -> Registro físico asociado al modo actual.
  - Índice mayor igual que 13 -> Registro del banco del modo usuario.

- PC no está en lista ó bit S no activo -> Registro físico asociado al modo actual.

Si se ha producido una excepción DATA ABORT al obtener la palabra o el registro destino actual es el registro Base entonces no se escribirá sobre el destino. El registro Base siempre se actualiza con el valor del desplazamiento, y por tanto no se puede escribir sobre él.

En caso de Load post-indexado, realizamos la suma o resta del desplazamiento para calcular la dirección efectiva después de la macro LSM_WRITEREGS.

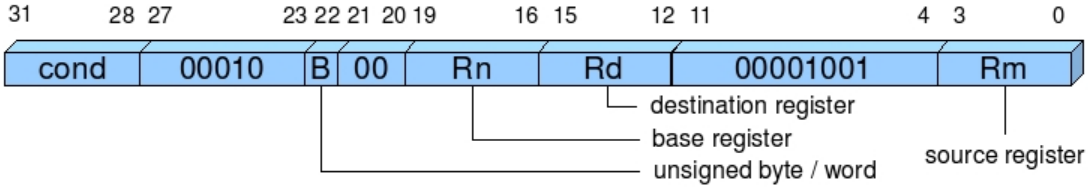
Una vez hecha la carga de la palabra, repetiremos el proceso anterior hasta que hayamos examinado totalmente la lista de registros de la instrucción.
Finalmente si el bit W está a 1, actualizamos el registro Base con el número de bytes transferidos, y si el PC está en la lista y el bit S a 1, copiamos el SPSR del modo actual en el CPSR.

En los Store la situación es similar, salvo que almacenamos en memoria el valor de los registros que tengan bit a 1 en la lista, llamamos a la macro LSM_READREGS para obtener valores de dichos registros y que sigue la misma política que la macro LSM_WRITEREGS. Y en este caso no se copia el SPSR en el CPSR. Si sucedierá un error al grabar en memoria el registro Base sería igualmente actualizado.

3.6 – Intercambio entre memoria y registro

Formato:

SWP{<condición>} {B} Rd, Rm, [Rn]



Descripción:

Las instrucciones de Swap combinan una instrucción de Load y otra de Store sobre una palabra, o sobre un byte sin signo.

La instrucción carga una palabra si el bit B = 0, o un byte sin signo si B = 1 de la dirección de memoria indicada en el registro Rn sobre el registro Rd, y guarda usando el mismo tipo de dato, el valor del registro Rm sobre la dirección de memoria indicada en Rn.

El registro 15 (PC) no debe utilizarse en ninguna de las codificaciones posibles de los 3 registros usados. El registro base Rn no debe ser el mismo que el registro Rm o el registro Rd.

Emulación:

La emulación de esta instrucción se implementa sobre una misma rutina.

GETOPS_LS es la macro encargada de obtener el valor del registro base (Rn), el valor del registro destino (Rd), el registro fuente (Rm), y el tipo de dato a transferir (bit B).

En el código de la rutina SWAP diferenciamos dos casos :

- B = 1 :

  - Cargamos el byte sin signo de la dirección de memoria referenciada por Rn.

  - Comprobamos si se ha producido un DATA_ABORT, en cuyo caso pararíamos la ejecución de la instrucción y proseguiríamos con el tratamiento de dicha excepción.

  - Guardamos en memoria el valor del registro Rm sobre la dirección de memoria referenciada por Rn.

- B = 0 :

  - Alineamos la dirección de memoria obtenida de Rn y cargamos la palabra referenciada.

  - Comprobamos si se ha producido un DATA_ABORT, en cuyo caso pararíamos la ejecución de la instrucción y proseguiríamos con el tratamiento de dicha excepción.

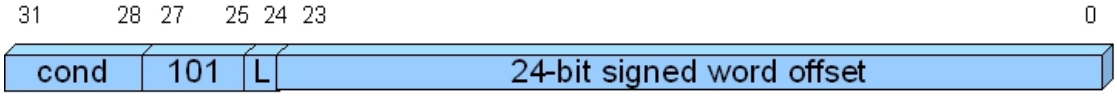
  - Guardamos en memoria el valor del registro Rm sobre la dirección de memoria referenciada por Rn.

Por ultimo guardamos el valor cargado de memoria en el registro Rd.

3.7 – Branch y Branch with link

Formato:

B{L}{<condicion>} <dirección destino>



Descripción:

Esta instrucción provoca que el PC salte a otra posición de memoria y continue ejecutando a partir de ella. La dirección se calcula a partir del operando inmediato extendido y desplazado a la izquierda de forma que apunte a una palabra alineada, que se suma al valor actual del contador de programa.

Cuando el bit L está activo, la dirección siguiente al salto es almacenada en el registro 14 del modo actual del procesador (link register).

La instrucción siguiente a la instrucción de salto no se ejecuta, ya que el ARM no dispone del modelo de salto retardado.

Emulación:

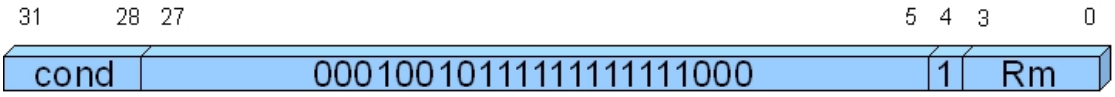
El ARM9Core emula esta instrucción con una única rutina. A diferencia de otros procesadores, las instrucciones de salto del ARM no necesitan comprobar los códigos de condición para decidir si se realiza el salto, ya que se utiliza la ejecución condicional para simular este comportamiento.

Esto simplifica la tarea de emulación, de forma que lo único que hay que hacer es calcular la nueva dirección y guardar el registro link si el bit L está activo. Es necesario también corregir la dirección restando 4 bytes a la que calcula el ARM por motivo de la segmentación.

3.8 – Branch with exchange

Formato:

BX {<condición>} Rm



Descripción:

Esta instrucción está permitida únicamente en ARMs que soportan el formato de instrucción THUMB (16-bit) y permite cambiar el formato de instrucción a ejecutar entre ARM y THUMB.

El bit 0 del registro Rm es copiado al bit T del registro CPSR. Si Rm[0] = 1 cambia a modo THUMB y copia el valor del registro Rm con Rm[0] = 0 al PC, de esta forma provoca un salto a una dirección alineada a media palabra. Si Rm[0] = 1 cambia a modo ARM y copia el valor del registro Rm con Rm[1] = 0 al PC, de esta forma provoca un salto a una dirección alineada a una palabra.

Emulación:

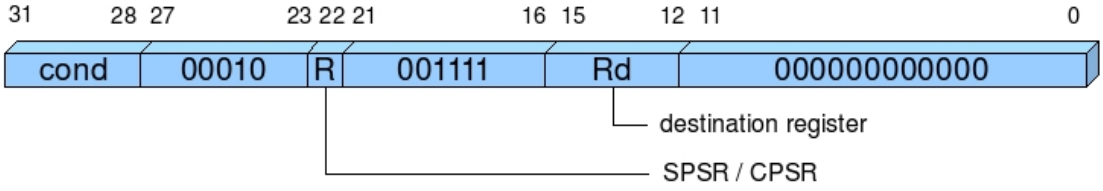
La emulación de esta instrucción se ejecuta sobre una misma rutina sin uso de macros externas encargadas de partes de la implementación de la instrucción.

Primero obtenemos el valor del registro que contiene la dirección del salto y después saltamos a la dirección de memoria indicada. Dado que nuestro emulador no soporta el modo Thumb, esta instrucción nunca cambia de modo.

3.9 – Transferencia de CPSR/SPSR a registro general

Formato:

MRS{<condición>} Rd, CPSR | SPSR



Descripción:

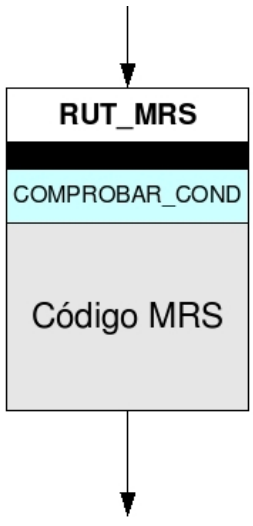
Esta instrucción copia el valor del CPSR (R = 0), o el valor del SPSR (R = 1) a un registro general (Rd).

No es posible usar la transferencia del SPSR a un registro general en modo usuario, o en modo sistema, ya que estos modos carece de un SPSR.

Estas instrucciones se usan comúnmente para guardar los valores del CPSR y SPSR del modo actual a un registro general y poder modificarlos, para después restaurar el valor con la instrucción MSR.

Emulación:

La emulación de esta instrucción se ejecuta sobre una misma rutina, sin uso de macros externas encargadas de partes de la implementación de la instrucción.



En primer caso si el bit R = 0, almacenamos en el registro destino (Rd) el valor del CPSR. Para leer el CPSR hacemos uso del método leerCPSR. Éste compone los valores

de los campos del CPSR del contexto actual sobre un registro ARM9Reg32 para después realizar operaciones sobre él tratándolo como un entero.

En caso de que el bit R = 1, comprobamos que no nos encontramos en modo usuario, o en modo sistema, para después almacenar en el registro destino (Rd) el valor correspondiente al SPSR del modo actual.

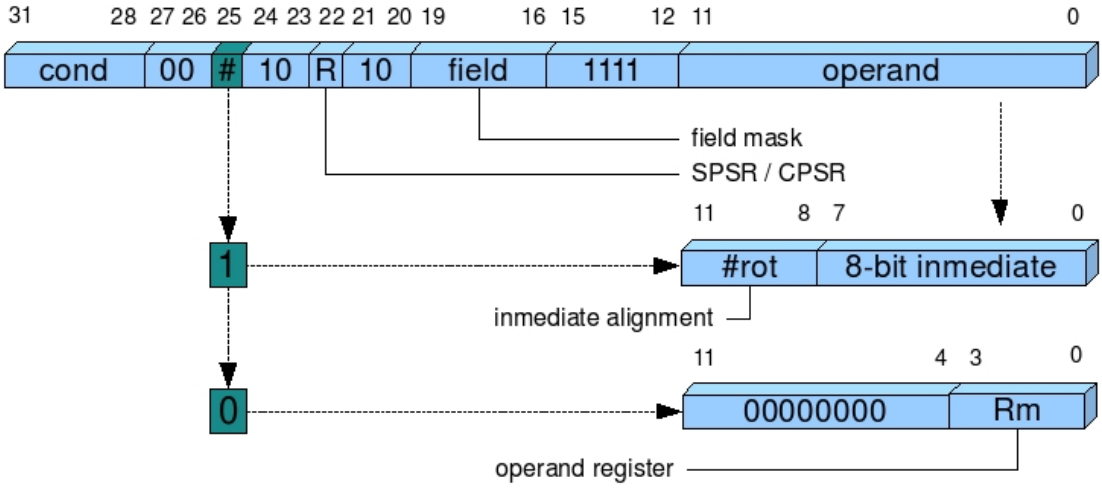
3.10 – Transferencia de registro general a CPSR/SPSR

Formato:

MSR{<condición>} CPSR_f | SPSR_f, #<32-bit inmediate>
MSR{<condición>} CPSR_<field mask> | SPSR_<field mask>, Rm

donde <field mask> es :

c – campo de control – *PSR[7:0].
x – campo de extensión – *PSR[15:8]. (No se usa actualmente).
s – campo de estado – *PSR[23:16]. (No se usa actualmente),
f – campo de flags – *PSR[31:24].



Descripción:

Esta instrucción carga en el CPSR o SPSR un operando inmediato de 8 bits rotado (la rotación sigue el mismo proceso que para las instrucciones aritméticas y lógicas), o el valor de un registro (Rm), los cuales han sufrido un proceso de enmascarado anteriormente.

El campo <field mask> determina cuales de los 4 bytes del PSR serán actualizados con los valores del registro Rm. En caso de usar un operando inmediato solo será posible actualizar los bits [31:24] del PSR. El bit 16 corresponde al PSR[7:0], el bit 17 corresponde al PSR[15:8], el bit 18 corresponde al PSR[23:16], y el bit 19 corresponde al PSR[31:24].

Los bits del PSR[31:24] son los únicos que pueden ser actualizados en modo usuario.
El registro de estado a actualizar será el CPSR si R = 0, o el SPSR si R = 1.
No se producirá ningún cambio sobre el CPSR o SPSR en caso de estar en modo usuario e intentar actualizar algún bit del 23 al 0.

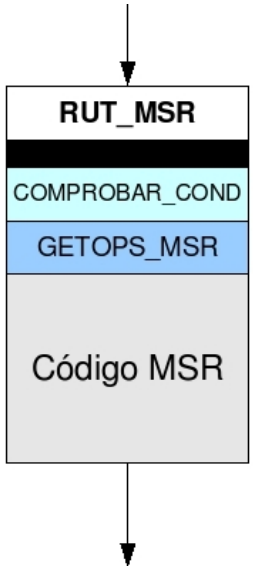
* PSR: representa tanto al CPSR como al SPSR.

No se permite modificar el SPSR en modo usuario, o modo sistema, ya que estos modos carecen de un SPSR.

Estas instrucciones se usan comúnmente para guardar en los registros de estado los registros de propósito general, modificados después de usar una MRS.

Emulación:

La emulación de esta instrucción se implementa sobre una misma rutina.



GETOPS_MSR es la macro encargada de obtener el valor del registro operando Rm si el bit 25 = 0, o de rotar el valor inmediato, que será usado como operando, si el bit 25 = 1. También obtiene el valor del bit R y el valor del campo <field mask>.

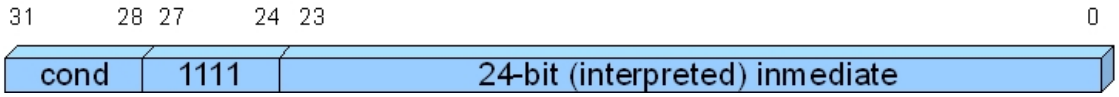
Para R = 0, cargamos el valor del CPSR y actualizamos los 4 bytes del CPSR uno a uno, dependiendo del valor del campo <field mask> y el valor del operando. Una vez actualizados los 4 bytes del valor cargado del CPSR, sobrescribimos el CPSR con el nuevo valor. A la hora de escribir el valor del CPSR usamos el método escribirCPSR. Este método transforma el valor recibido a un registro ARM9Reg32 a partir del cual extraemos los valores correspondientes a los campos del CPSR del contexto actual. Después se procede a cambiar al nuevo modo.

En caso de que R = 1, comprobamos que no nos encontramos en modo usuario, o en modo sistema, y repetimos el mismo proceso que para R = 0, solo que esta vez actualizamos el valor correspondiente del SPSR del modo actual.

3.11 – Interrupción por software

Formato:

SWI{<condición>} <24-bit inmediate>



Descripción:

La instrucción SWI provoca una interrupción por software usadas por el sistema operativo comúnmente llamadas "supervisor call". Cambian el modo del procesador a modo supervisor y comienzan a ejecutar instrucciones a partir de la dirección 0x08. El campo inmediato de 24 bits no influye en las operaciones de la instrucción pero puede ser interpretado por el código del sistema.

En caso de que se cumpliera el código de condición :

1. Guarda la dirección de la instrucción siguiente a la SWI en r14_SVC.
2. Guarda el CPSR en el SPSR_SVC.
3. Entra en modo supervisor y deshabilita las IRQs cambiando el CPSR[4:0] a 10011 y el CPSR[7] a 1.
4. Cambia el PC a 08 y comienza a ejecutar instrucciones a partir de esta dirección.

Después de ejecutar el código de la SWI la rutina del sistema restaura el valor del PC con r14_SVC y el CPSR con SPSR_SVC.

Emulación:

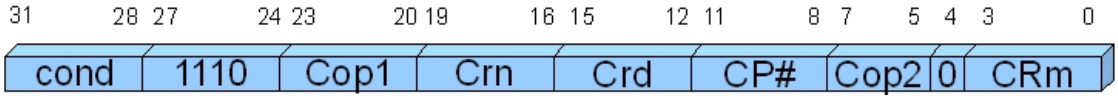
La emulación de esta instrucción se ejecuta sobre una misma rutina sin uso de macros externas encargadas de partes de la implementación de la instrucción.
Primero se obtiene el valor del operando inmediato de la instrucción y después se hace uso del método marcarExcepcion (EXCEP_SWI, operando) que provocará el tratamiento de la SWI. La documentación referente al tratamiento de todas las interrupciones se encuentra en el apartado 5.

3.12 – Instrucciones de coprocesador

Formato:

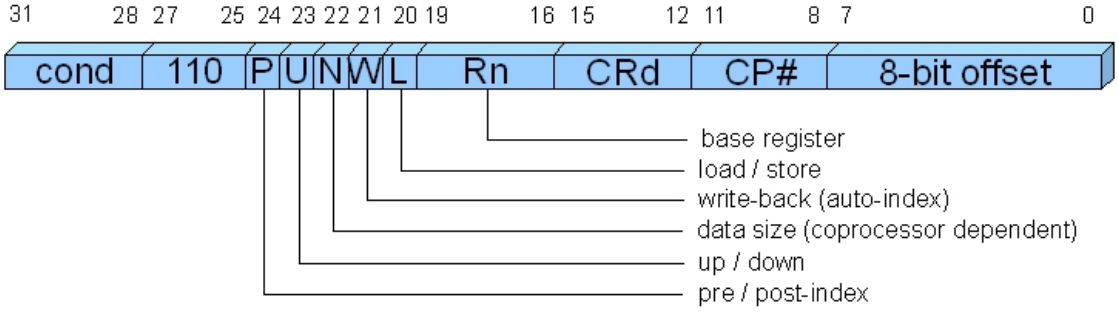
Operaciones de datos de coprocesador :

CDP {<condición>} <CP#>, <Cop1>, CRd, CRn, CRm{, <Cop2>}



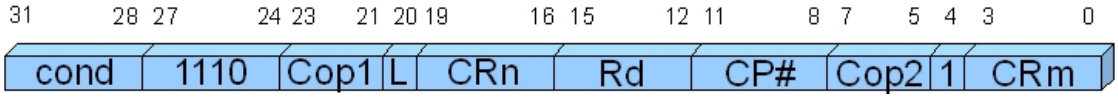
Transferencia de datos de coprocesador :

Pre-indexed
LDC | STC {<condición>} {L} <CP#>, CRd, [Rn, <offset>] {!}
Post-indexed
LDC | STC {<condición>} {L} <CP#>, Crd, [Rn], <offset>



Transferencias de registros de coprocesador :

De coprocesador a registros de ARM
MRC {<condición>} <CP#>, <Cop1>, Rd, CRn, CRm{, <Cop2>}
De registros de ARM a coprocesador
MRC {<condición>} <CP#>, <Cop1>, Rd, CRn, CRm{, <Cop2>}



Descripción:

Las operaciones de datos de los coprocesadores son completamente internas al coprocesador y provocan modificaciones sobre sus registros (sumas, restas, multiplicaciones...). Un uso común son los coprocesadores de punto flotante encargados de realizar estas operaciones tan costosas.

Las operaciones de transferencia de datos de los coprocesadores permiten cargar o leer datos de la memoria del coprocesador. Desde que los coprocesadores pueden soportar su propio tipo de datos, el número de palabras transferidas son determinadas por el propio coprocesador. El ARM calcula la dirección de memoria y el coprocesador transfiere las palabras necesarias.

Las operaciones de transferencia de registros de los coprocesadores permiten transferir registros del ARM al coprocesador y viceversa.

Emulación:

Las instrucciones de coprocesador han sido emuladas para que el propio usuario pudiera implementarlas por su cuenta. De esta forma tenemos tres funciones handler, una para cada tipo de instrucción de coprocesador

Disponemos de tres rutinas para las instrucciones de coprocesador, una para cada tipo de operación que anteriormente hemos descrito. Una vez dentro de una de estas tres rutinas obtenemos el número del coprocesador que se usará y llamamos al handler encargado de ejecutar esta operación si ha sido implementado (pasándole el número del coprocesador). Si el handler no ha sido implementado se producirá una interrupción (EXCEP_UNDEFINED).

Esta interrupción es similar a la que se produciría en una máquina real en caso de encontrarse con una instrucción que hace referencia a un coprocesador inexistente. Esto permite que el comportamiento de la instrucción se emule por software a través de la interrupción.

4. Sistema de Memoria

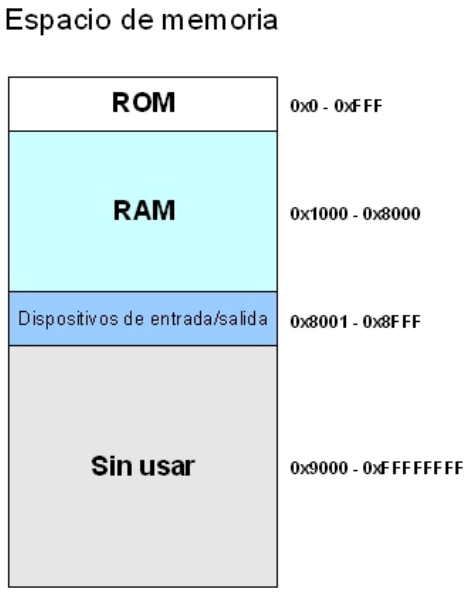
Descripción

El procesador ARM9TDMI es capaz de direccionar un espacio de memoria de hasta 4 Gigabytes de manera directa. La mayoría de los sistemas empotrados que usan este procesador necesitan espacios de memoria más pequeños, del orden de Megabytes.

El sistema de memoria realiza los accesos a nivel de byte y además de proporcionar lectura y escritura de un dato en una dirección del espacio de memoria, debe permitir, según las necesidades del sistema, características como:

- Acceso a diferentes tipos de memoria o dispositivos de entrada/salida.
- Acceso restringido a determinadas regiones de memoria.
- Acceso a diferentes tamaños de datos: palabras (32 bits), medias palabras (16 bits) y bytes (8 bits).
- Manejo de direcciones virtuales, permitiendo mayores espacios de memoria.
- Prevención de errores y manejo de excepciones de memoria.
- Uso de caches, write-buffers y otras técnicas.

Un posible espacio de memoria con algunas de las características anteriores sería el siguiente:



Como se puede observar tenemos varias regiones en el espacio de memoria. La primera de ellas está asociada a una memoria ROM con un tamaño 4 Kilobytes. En la región de esta memoria tan sólo se pueden hacer accesos de lectura y en caso de que se intente escribir en ella, el sistema de memoria provocará una excepción.

En cambio, cualquier intento de lectura o escritura en el rango de direcciones de la memoria RAM sí estará permitido. El acceso a la posición 0x1000 correspondería a la primera posición (posición 0) de esta memoria, cuyo tamaño es de 28 Kilobytes.
La región de memoria siguiente a la RAM será utilizada para comunicarse con dispositivos de entrada y salida, con permisos de lectura y/o escritura según la funcionalidad de estos.

Finalmente vemos que el resto del espacio no está asociado a ninguna memoria y/o dispositivo. Y un intento de acceso a esta región, ya sea de lectura o escritura, provocará un error o excepción.

Como hemos visto anteriormente, disponemos de varios modos de ejecución que se dividen entre aquellos con privilegios y el modo usuario, sin privilegios. En las regiones de memoria podemos requerir que sea necesario que el modo actual del procesador sea privilegiado o no, para permitir el acceso a las mismas.

Emulación

Para poder permitir un sistema de memoria con las características comentadas y obtener la mayor eficiencia posible definimos una región de memoria con la estructura ARM9MemRegion que contiene los siguientes campos:

```
uint32 inferior;          //Dirección inferior del bloque de memoria
uint32 superior;          //Dirección superior

uint8 *buffer;            //Buffer de memoria

//Punteros a funciones manejadoras (lectura y escritura)
void *handlerReadX;       //X puede ser 8, 16 o 32 bits
void *handlerWriteX;      //X puede ser 8, 16 o 32 bits

//Permisos
int permisos;//(wp, rp, wu, ru) (Write/Read Privilegiado/Usuario)
```

Los campos inferior y superior configuran el rango de direcciones de la región de memoria y la resta de ambos nos determinan el tamaño de la misma.

El campo buffer apunta a un array de bytes (bloque de memoria), cuyo tamaño debe ser igual al de la región. Si el campo es distinto de NULL, el sistema de memoria del ARM9Core actuará sobre el buffer, siempre y cuando el acceso a memoria esté en el rango de direcciones de la región.

En caso de que no se asigne un buffer a la región, el usuario deberá asignar a los campos handler punteros a las funciones (creadas por el usuario) encargadas de gestionar* la región. Hay dos tipos de handler, uno para lecturas y otro para escrituras, y para cada tipo de handler se disponen de tres campos para cada tamaño posible del acceso: byte, media palabra y palabra. Los tipos de punteros a las funciones que se asocian a los handler (de lectura y escritura) tienen la siguiente estructura:

```
uint32 (*readHandlerT)(uint32 dir); //T puede ser 32, 16 u 8 bits
void (*writeHandlerT)(uintT datos, uint32 dir);
```

* Por gestionar queremos decir que por lo general, la función sólo se encarga de obtener o almacenar el dato, dado que el cálculo de la dirección relativa a la región y la comprobación de los permisos de acceso son realizados por el propio emulador.

Estos punteros a funciones se ofrecen para que el usuario pueda implementar sistemas de memoria más complejos y/o personalizados con manejo de direcciones virtuales, uso de caches, write-buffers... o para poder trabajar sobre estructuras de memoria distintas al posible buffer asociado a la región.

El campo "permisos" de 4 bits (tipo ARM9Permission) permite definir y combinar los siguientes accesos a la región:

```
USR_READ_PERM = 0x1,  //Permiso de acceso a lectura para modo usuario
USR_WRITE_PERM = 0x2, //Permiso de acceso a escritura para modo usuario
PRIV_READ_PERM = 0x4, //Permiso de acceso a lectura con privilegios
PRIV_WRITE_PERM = 0x8 //Permiso de acceso a escritura con privilegios
```

Estas propiedades para las regiones permiten, entre otras opciones, que sea posible que dos regiones (con rangos distintos de direcciones) puedan trabajar con el mismo buffer, y sin embargo tengan permisos distintos de acceso. Esto puede ser útil en un programa en el que un núcleo escriba información y otro núcleo sólo deba leerla, donde la región a la que tiene acceso este último sólo permita accesos de lectura, impidiendo la modificación de la misma por su parte.

La clase ARM9Core dispone de un puntero a las regiones de memoria, las cuales son definidas por el usuario con las propiedades que desee. El usuario debe asegurar que los rangos de direcciones entre dos o más regiones no se solapen.

Para que el emulador funcione correctamente tiene que existir al menos una región de memoria con permisos de lectura, para poder leer instrucciones de un programa, y evitar errores al intentar obtenerlas.

Para añadir todas las regiones de memoria se dispone de la función "addMemoria" que recibe el número de regiones y la lista con las mismas. Al añadirlas se eliminan todas las regiones que anteriormente se tenían y si alguna de las regiones nuevas se solapa con otra se producirá un error, devolviendo falso la función. Esto se debe a que si sucede un acceso a memoria en una dirección común a dos o más regiones no es posible determinar con cuál de ellas operar.

La búsqueda de instrucciones se realiza con la función "readInstruccion" que recibe como parámetro una dirección, el contador de programa (PC). La secuencia de acciones a seguir es la siguiente:

1. Buscamos en las regiones de memoria si la dirección del PC se encuentra en el rango de direcciones de alguna de ellas.

2. Si encontramos una región, vamos a 3. Si no, marcamos una excepción PREFECTH_ABORT, que indica un error en la obtención de instrucción, y devolvemos como instrucción 0x00000000 que a efectos prácticos indica una instrucción indefinida, lo que provocará una parada del procesador.

3. Comprobamos que el permiso de lectura de la región coincide con el del modo actual, es decir, que la región tiene permiso de lectura privilegiado y el modo actual también lo es, o si no es privilegiado entonces el procesador está en el modo usuario. En caso afirmativo vamos a 4, y si no lanzamos una excepción PREFETCH_ABORT.

4. Si la región de memoria no tiene definida un buffer se devuelve la instrucción que retorna la función asociada al handler, que recibe por parámetro la dirección relativa de la región, es decir, la dirección del PC menos el valor del campo inferior de la región. En caso contrario se devuelve la palabra almacenada (en la posición relativa de la región) en el buffer.

Para poder obtener o almacenar datos en memoria según el tipo de acceso (lectura o escritura) y tipo de dato (palabra, media palabra y byte) de la transferencia, se emplean las diferentes instrucciones "readMemoria" o "writeMemoria" de la clase ARM9Core.

Estas funciones siguen la misma secuencia de pasos que la función que se encarga de leer las instrucciones de programa, salvo que las excepciones que se marcan son del tipo DATA_ABORT, que indica un error en la obtención o almacenamiento de un dato.

5. Interrupciones

5.1 – Tipos de Interrupciones

Las interrupciones permiten al procesador realizar el tratamiento correspondiente a los eventos que pueden producirse durante la ejecución de un programa, que pueden ser originados tanto de forma interna (instrucciones indefinidas, errores de acceso a memoria, etc...) como señalizados desde el exterior (IRQs, FIQs, ...).

En la arquitectura ARM v4T existen 7 tipos posibles de interrupciones que reciben distintas prioridades. Ordenados de menor a mayor prioridad, son los siguientes:

Instrucción indefinida (UNDEFINED)

Se da cuando la última instrucción leída no se corresponde a ninguno de los formatos existentes, y por tanto no es posible su decodificación. También puede activarse si una instrucción del coprocesador hace referencia a un coprocesador inexistente en el sistema.

Interrupción por Software (SWI)

Tiene la misma prioridad que la anterior. Es activada por el propio programa en ejecución al llamar a una instrucción SWI, y normalmente se utiliza para implementar llamadas al sistema.

Prefetch Abort

Se activa cuando se produce un error al acceder a memoria para realizar la lectura de la siguiente instrucción (por ejemplo, si se da un fallo de página). Será necesario corregir el problema que causó el error y posteriormente reintentar la ejecución de la instrucción que no se pudo leer.

Interrupción externa (IRQ)

Se activa desde el exterior a través de una línea de interrupción específica. Puede ser usada, por ejemplo, para realizar interfaces de comunicación con los dispositivos de entrada y salida, por un temporizador para indicar que ha transcurrido un periodo de tiempo, etc... Esta interrupción puede ser desactivada (enmascarada) modificando el bit correspondiente en el CPSR.

Interrupción rápida (FIQ)

Similar a la anterior. La diferencia está en que el modo de ejecución FIQ tiene más registros físicos propios, por lo que no es necesario preservar todo el contexto en memoria. Es recomendable usar las FIQ para las interrupciones más frecuentes. Al igual que las IRQ, también es posible enmascararlas modificando el bit del CPSR.

Data Abort

Se activa cuando se produce un error en cualquier acceso a memoria (tanto en lectura como en escritura). El error puede ser debido a un fallo de página, pero también a intentos por parte del programa de acceder a zonas de memoria inexistentes, o regiones protegidas sin disponer de los permisos de acceso, etc...

Reset

Activada desde el exterior, la señal de Reset detiene inmediatamente la ejecución y restaura el procesador a un estado inicial conocido. Se usa antes de comenzar la ejecución de programas y se trata de la interrupción con mas prioridad.

5.2 – Procesamiento de Interrupciones

El orden de prioridad de las distintas interrupciones determina cuál de ellas se ejecutará antes en el caso de que se activen varias al mismo tiempo. El tratamiento que hace el ARM cuando se presenta una interrupción es el siguiente:

1. Completa la ejecución de la instrucción activa lo mejor posible, y posteriormente abandona la secuencia de código actual. En el caso de una señal de reset, la ejecución se detiene inmediatamente.

2. Cambia el modo de ejecución al correspondiente dependiendo del tipo de interrupción.

3. Guarda el valor del PC actual en el registro 14 (link register) del nuevo modo activado.

4. Guarda el valor del CPSR actual en el SPSR del nuevo modo activado.

5. Se enmascaran las IRQs activando el bit correspondiente del CPSR. Si la interrupción es una FIQ, se enmascaran estas también.

6. Fuerza el valor del PC al del vector de interrupción correspondiente.

La siguiente tabla muestra los distintos modos y vectores de interrupción que se usan para cada tipo:

Tipo de Interrupción	Modo	Vector
Reset	SVC	0x00000000
Instrucción Indefinida	UNDEF	0x00000004
Software Interrupt	SVC	0x00000008
Prefetch Abort	ABORT	0x0000000C
Data Abort	ABORT	0x00000010
IRQ	IRQ	0x00000018
FIQ	FIQ	0x0000001C

Como se vio anteriormente, cada modo de ejecución privilegiado (exceptuando el modo SYSTEM) dispone de dos registros físicos propios. Estos registros se suelen denominar “link register” (registro de enlace) y “stack pointer” (puntero a pila), dado que se usan para guardar la dirección de retorno a la instrucción que fue interrumpida y el puntero de la pila de datos, respectivamente.

La pila de datos se usa para preservar los valores antiguos de los registros al entrar en la rutina de tratamiento de la interrupción. Las FIQ no necesitan guardar todos los registros, ya que disponen de varios de ellos en un banco independiente, por lo que son más eficientes.

Una vez terminado el tratamiento de la interrupción es necesario restaurar el estado antiguo del procesador y retomar la ejecución del programa en el punto en que se abandonó. Esto significa:

- Restaurar los registros a partir de los valores almacenados en la pila.
- Restaurar el valor del CPSR con el SPSR.
- Restaurar el PC al valor guardado en el registro de enlace.

Estos dos últimos pasos han de ejecutarse de forma atómica, puesto que si primero se restaurase el CPSR, el registro de enlace dejaría de ser accesible, pero si se hiciese al contrario, actualizando primero el PC, la rutina de tratamiento perdería el control y no podría modificar el CPSR.

Tal y como se vio en las instrucciones aritmético-lógicas, se puede activar el flag 'S' para que el ARM restaure el CPSR al modificar el valor del PC, realizando las dos operaciones de forma atómica. Por ejemplo, para volver de una SWI o una instrucción indefinida, se haría:

```
MOVS PC, r14
```

En el caso de tratarse de una IRQ, FIQ, o Prefetch Abort deberemos volver a la instrucción anterior a la guardada en el PC, es decir, a la misma instrucción que fue interrumpida y cuya ejecución no se terminó:

```
SUBS PC, r14, #4
```

Y por último, en las Data Abort deberemos volver dos instrucciones antes de la que señala el registro de enlace, puesto que tenemos que ejecutar de nuevo la instrucción que provocó el fallo de acceso:

```
SUBS PC, r14, #8
```

Otra posible forma de volver de la rutina de tratamiento de la interrupción es la instrucción de carga múltiple. Podemos almacenar la dirección del PC en la pila en lugar de en el registro de enlace, y aprovechar esta instrucción para restaurarlo junto a los demás registros.

Esta instrucción también permite copiar el SPSR al CPSR de forma atómica si se activa el flag correspondiente.

5.3 – Emulación de Interrupciones

Reproducir correctamente el comportamiento de las interrupciones es un requisito básico si se quiere que un emulador pueda ejecutar programas reales diseñados para la máquina original. Sin embargo, dada la naturaleza secuencial del ARM9Core y la necesidad de eficiencia, la forma de emular este comportamiento es completamente distinta al funcionamiento real. Esto convierte a las interrupciones en un ejemplo que pone de manifiesto las diferencias entre un emulador y un simulador.

Funciones "handler"

Por ejemplo, podemos encontrarnos con una SWI que se encarga de realizar una división (recordemos que el repertorio ARM no dispone de instrucciones de división). A la hora de emularla, la primera opción sería simular el comportamiento real, saltar al vector de interrupción y ejecutar un código ARM que realice la operación.

Sin embargo sería mucho más eficiente si pudiéramos emular la SWI directamente con una línea de código C que realizara la división, ya que el x86 si que dispone de esa instrucción en su repertorio. Además, haciéndolo así no sería necesario reproducir todos los cambios de modo, salvar los registros en la pila y restaurarlos, etc... por lo que la diferencia de velocidad es muy significativa.

El ARM9Core permite ambas posibilidades. El usuario puede programar en su aplicación una serie de funciones "handler" en código C/C++ (una por cada tipo de interrupción) que se encarguen de la emulación de las mismas, y enlazarlas al Core en la etapa de inicialización.

Cuando el emulador se encuentre con una interrupción, comprobará si hay algún handler definido para la misma. En caso de haberlo, invocará a la función y continuará posteriormente su ejecución. Si no hay ningún handler definido simulará el comportamiento real del ARM, cambiando de modo y saltando al vector de interrupción.

Estas funciones están definidas de la forma:

```
void handler (int n);
```

Para enlazarlas, es necesario hacer una llamada al método "setExcepHandler" de la clase ARM9Core indicando como parámetros el tipo de interrupción y el puntero a la función que se encargará de tratarla.

La interrupción Reset es un caso especial que no dispone de función handler ya que el propio Core proporciona un método específico para el caso de que sea necesario resetear el estado del emulador.

Por último, el parámetro n que reciben las funciones handler es un valor que depende del tipo de interrupción, con el objetivo de que ese dato común sea accesible de forma aún más rápida para el código que realiza la emulación:

Tipo de Interrupción	Valor de n
Data Abort	Dirección accedida
FIQ	N / A
IRQ	N / A
Prefetch Abort	Dirección accedida
Instrucción Indefinida	Operando 1 *
Software Interrupt	Código de SWI

Señalización y ejecución de interrupciones

El sistema de señalización y ejecución de interrupciones es distinto dependiendo de si éstas son internas (Aborts, SWIs, e instrucciones indefinidas) o externas (IRQs, FIQs, y resets), y constituye otra diferencia con el comportamiento de la máquina real.

En el caso de las interrupciones internas, éstas son generadas durante la emulación de las instrucciones (es decir, durante el "*slice*" de ciclos emulados). En el momento en que se produce la interrupción, ésta es señalizada. El método de señalización recuerda el tipo de interrupción y tiene en cuenta las prioridades, quedándose siempre con la más prioritaria en el caso de que haya más de una.

Al empezar la ejecución de la siguiente instrucción se comprueba también la marca de interrupciones. En el caso de que haya alguna, se interrumpe en este punto la emulación y se cede el control al método de tratamiento de interrupciones, que llamará al handler correspondiente o modificará el contexto del emulador de acuerdo a lo explicado anteriormente. Después se ejecutan los ciclos restantes del *slice*.

El caso de las interrupciones externas es diferente. Dado que el emulador sigue una estructura secuencial, y que este tipo de excepciones sólo puede ser señalizado desde la aplicación externa, eso significa que no pueden producirse durante el *slice* de emulación del Core. El único momento en que pueden activarse es al recuperar el control la aplicación, durante la llamada a la "función externa" al término de cada *slice*.

Como en este punto la aplicación externa ya tiene el control, no es necesario que señalice las interrupciones de forma explícita; puede limitarse a tratarlas directamente y modificar la memoria y el contexto del emulador de acuerdo a sus necesidades.

Por tanto, el método más eficiente de emular las IRQs y FIQs es a través de la función externa. Como esta función sólo es llamada una vez por cada *slice*, el programador de la aplicación tendrá que elegir un tamaño de *slice* adecuado a la frecuencia con que quiera comprobar la presencia o no de dichas interrupciones.

Normalmente se recurre a la frecuencia de la interrupción más rápida para ello (temporizadores, refresco de pantalla, etc...).

* El valor de Operando 1 sólo tiene sentido en el caso de que la interrupción Indefinida se produzca a raíz de la ejecución de una instrucción del coprocesador para la cual no exista un coprocesador asignado.

De forma opcional, el ARM9Core permite también la señalización explícita de interrupciones IRQ, FIQ y Reset (siendo ésta última equivalente a una llamada directa al método de reset de la clase), de modo similar a como se realiza la señalización de las interrupciones internas.

Si bien, como se ha visto, éste método no es recomendable por su poca eficiencia, se ofrece aún así por motivos de compatibilidad, dado que en casos concretos el usuario puede estar interesado en emular código ARM de tratamiento de interrupciones de forma explícita.

6. Programa de prueba

Dado que el ARM9Core es una librería que debe ser usada en el contexto de una aplicación mayor para funcionar, nos vimos en la necesidad de crear un programa simple que incluyese la librería y ejecutase la emulación del core para poder realizar sobre él las pruebas y mediciones de rendimiento necesarios.

Dicho programa consistía inicialmente en una memoria sencilla (emulada mediante un array), conectada directamente a una instancia del ARM9Core. Esto nos permitió cargar instrucciones en la memoria, emularlas, y comprobar los resultados.

Según avanzó el proyecto y fuimos añadiendo nuevas características al emulador, fue necesario también ampliar el programa de prueba. Por ejemplo, hubo que añadir soporte para interrupciones, funciones handler, regiones de memoria, etc... En las últimas fases del proyecto se incorporó también la posibilidad de cargar programas en archivos en formato .ELF y emularlos.

Si bien al empezar el desarrollo del proyecto uno de nuestros objetivos era el de utilizar la librería como núcleo de una aplicación que emulase una máquina real (incluyendo memoria, sistemas de entrada y salida, etc...) no disponíamos de suficiente tiempo para su desarrollo. Por tanto, decidimos en su lugar ampliar el programa de prueba que ya teníamos para convertirlo en un emulador genérico (que no se correspondía a ninguna máquina real), pero con las suficientes opciones como para ser útil en la prueba y depuración de programas compilados para ARM.

En este capítulo explicaremos las características que ofrece el programa de prueba en su versión final, y lo usaremos también como ejemplo de cómo se debe enlazar, inicializar y usar el ARM9Core como componente de una aplicación final.

El programa de prueba funciona en modo consola mediante una serie de comandos. La lista completa de comandos está disponible en el apéndice B.

Enlazado e inicialización

El primer paso a realizar si se quiere compilar una aplicación que haga uso de la librería es enlazarla con la misma. En nuestro caso, el enlace es estático. En primer lugar, debemos incluir el archivo de cabecera en el código:

```
#include "ARM9Core.h"
```

Para que el compilador enlace la librería, es necesario también añadirla a sus opciones de enlazado. En nuestro caso, se añade el archivo "ARM9Core.lib" a la lista de librerías a enlazar.

Dependiendo del compilador, será también necesario configurar los directorios de archivos de cabecera (.h) y librerías (.lib) para que encuentre los dos archivos nuevos con los que estamos trabajando.

Una vez hecho esto, podemos inicializar el ARM9Core creando una instancia de la clase y llamando a los siguientes métodos:

```
//Establece un sólo núcleo a emular
core.setCores(1);

//Asigna las regiones de memoria
core.addMemoria (nRegiones, regionesMemoria);

//Asigna los handlers para las interrupciones
core.setExcepHandler (EXCEP_SWI, &swi);
core.setExcepHandler (EXCEP_UNDEFINED, &undef);
core.setExcepHandler (EXCEP_DATA_ABORT, &abort);
core.setExcepHandler (EXCEP_PREFETCH_ABORT, &abort);

//Asigna los handlers para los coprocesadores (si los hay)
core.setCopro (0, &coDP, &coLS, &coRT);

//Asigna el handler de la función externa
core.setFuncionExterna (&fexterna);

//Fija la frecuencia de reloj a 50 Mhz
core.setFrecuenciaARM (50.0);

//Resetea el core y lo prepara para la ejecución
core.reset ();
```

Regiones de memoria

El primer dato de cierta complejidad que debemos pasarle al ARM9Core es la lista de regiones de memoria. Esto se hace a través del método:

```
bool addMemoria (int nRegiones, ARM9MemRegion *lista);
```

Este método recibe como parámetros el número total de regiones, seguido de una lista (array) de identificadores de región. El valor devuelto es “true” si la inicialización fue correcta, o “false” en el caso de detectarse algún error en la lista dada (por ejemplo, regiones solapadas).

Cada identificador de región contiene la información de las direcciones de inicio y final de la misma, así como punteros a los handlers o al buffer dependiendo de su tipo. También incluye la información relativa a los permisos de acceso. Por ejemplo, el siguiente código se usa para inicializar una región desde la dirección 0 a la 0x8000, y asociarla a un buffer de datos, con todos los permisos de acceso disponibles:

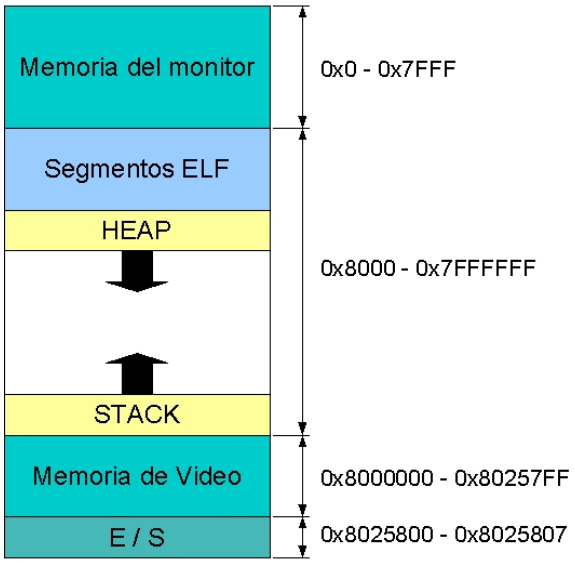
```
regionesMemoria[0].inferior = 0;
regionesMemoria[0].superior = 0xFFFF;
regionesMemoria[0].buffer = (uint8*)memoria;

regionesMemoria[0].handlerRead32 =
      regionesMemoria[0].handlerRead16 =
      regionesMemoria[0].handlerRead8 = NULL;
regionesMemoria[0].handlerWrite32 =
      regionesMemoria[0].handlerWrite16 =
      regionesMemoria[0].handlerWrite8 = NULL;
```

```
regionesMemoria[0].permisos = USR_READ_PERM |
                              USR_WRITE_PERM |
                              PRIV_READ_PERM |
                              PRIV_WRITE_PERM;
```

En el caso de usar una función handler en lugar de un buffer de datos, a este último se le daría valor NULL, mientras que a los distintos handlers se les asignarían los punteros a las funciones correspondientes.

El mapa de memoria del programa de prueba es el siguiente:



La primera región (Memoria del Monitor) se corresponde a la memoria reservada para el programa monitor en una máquina real. Dado que en nuestro caso la emulación de interrupciones se hace mediante handlers, esta memoria no se usa.

A continuación se sitúa la memoria de aplicación, desde la dirección 0x8000 (normalmente los compiladores para ARM asumen por defecto que la memoria de aplicación empieza siempre en esta dirección), hasta la 0x7FFFFFF. En la zona más baja se situán los segmentos del programa compilado (el .ELF), seguidos del Heap. En la zona más alta se sitúa la pila descendente.

Por último, existen otras dos regiones. La primera es un buffer de vídeo (si se activa la salida de vídeo con el comando correspondiente todo lo que haya aquí se representará gráficamente en pantalla). La segunda se usa para entrada / salida y guarda algunas variables de estado.

Handlers

El core se enlaza con varios handlers que se encargan del tratamiento de interrupciones, instrucciones del coprocesador, etc... De ellos los más importantes son los handlers para interrupciones y la función externa.

Los handlers de interrupciones se asocian mediante el método:

```
void setExcepHandler (ARM9Exception ex,
                      void (*handler)(int n));
```

El primer parámetro es un identificador del tipo de excepción. El segundo es un puntero a la función que la tratará. Por ejemplo, la función encargada de las SWI puede ser la siguiente:

```
void swi (int n) {
      if (n == 0x123456) {
            //Protocolo ANGEL
            angelSyscall();
      } else {
            core.stop();
            printf ("\nSWI numero %X\n\n", n);
      }
}
```

El protocolo ANGEL es el usado por las llamadas al sistema del programa monitor, y viene explicado en el apéndice F.

La función externa es un handler distinto, dado que es invocado siempre al término de cada *slice*. Esta función sirve para devolver el control al programa principal y que éste pueda realizar las tareas que considere necesarias, así como la emulación de interrupciones IRQ y FIQ. En nuestro caso se realiza la actualización de la salida de vídeo (si está activada), y la emulación de la interrupción debido al refresco de dicha pantalla.

Sincronización y ejecución

Por último, es necesario especificar la frecuencia (50 Mhz en este caso), y proceder a la ejecución del core. La ejecución se realiza mediante el método:

```
core.run (nCiclos);
```

Que activa el emulador indicando un tamaño de *slice* de nCiclos. La emulación continúa hasta que desde la aplicación se invoque a:

```
core.stop ();
```

Otro modo de emulación, útil si se desea depurar programas, es el modo “paso a paso”. Este método sólo ejecutará la siguiente instrucción, y después devolverá el control:

```
core.step ();
```

Por último, el programa de prueba realiza sus propias mediciones de tiempo utilizando el mismo tipo de contador de alta precisión que se usa en el propio ARM9Core para la sincronización. Estas medidas se usan para determinar el rendimiento de las distintas configuraciones.

7. Tests y Estadísticas

Los tests han sido realizados con pequeños trozos de código y programas en ensamblador ARM para observar la velocidad de emulación y comparar el tiempo empleado por las instrucciones emuladas. Para la medición se ha utilizado el programa de prueba que hace uso de la librería ARM9Core.

Para obtener unas medidas fiables y realistas, las instrucciones que forman parte del código del programa se ejecutan un número elevado de veces para diferentes tamaños del *slice*. A su vez se emplea una frecuencia alta para el procesador ARM para evitar que los resultados obtenidos en la máquina destino, donde se realiza la emulación, se vean afectados por esperas de sincronización.

Las pruebas han sido realizadas en un ordenador con procesador Intel Celeron M a 1,6 Ghz, 1 Gb de memoria RAM y sistema operativo Microsoft Windows XP.

Test 1

Ejecución de 1 millón de ciclos con un tamaño de *slice* entre 50 y 1000, aumentando en 100 ciclos.

Código ADD:

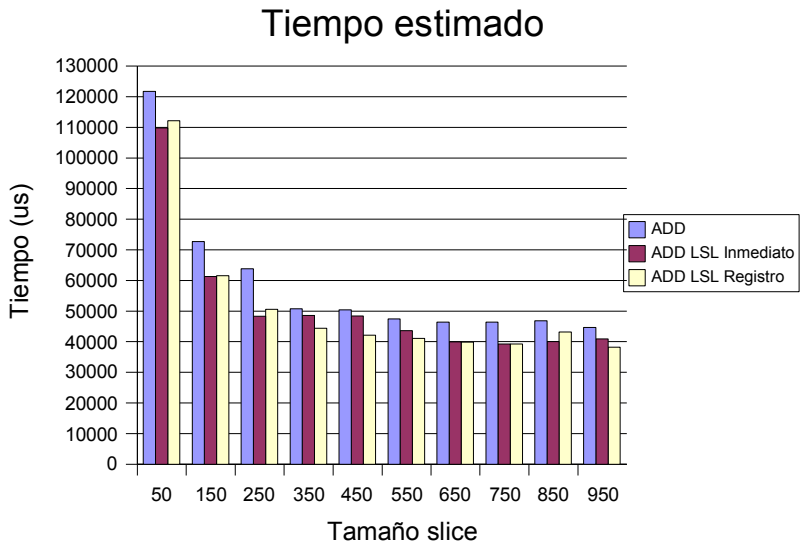
```
mov r2, #4
add r0, r0, #4
```

Código ADD LSL Inmediato:

```
mov r2, #4
add r0, r0, r1, LSL #4
```

Código ADD LSL Registro:

```
mov r2, #4
add r0, r0, r1, LSL r2
```



Vemos que las instrucciones aritméticas con desplazamiento son emuladas un poco más rápido que las que no, dado que la expansión del operando inmediato es más costosa. La diferencia entre desplazamiento con operando inmediato y registro es prácticamente inapreciable.

Observamos que por norma general un mayor tamaño del *slice* también favorece al tiempo de ejecución, con la ya mencionada pérdida de precisión en las interrupciones.

Esto es debido a que las instrucciones del código no requieren de muchos ciclos por cada pasada, y en este caso a partir de un *slice* mayor que 1000 la variación es cada vez menor.

Para tamaños pequeños del *slice* el rendimiento es muy bajo respecto a tamaños altos, por lo que utilizar valores pequeños sólo es recomendable si deseamos mucha precisión en el tratamiento de interrupciones.

Test 2

Ejecución de 2 millones de ciclos con un tamaño de *slice* entre 50 y 2000, aumentando en 200 ciclos.

Código MUL:

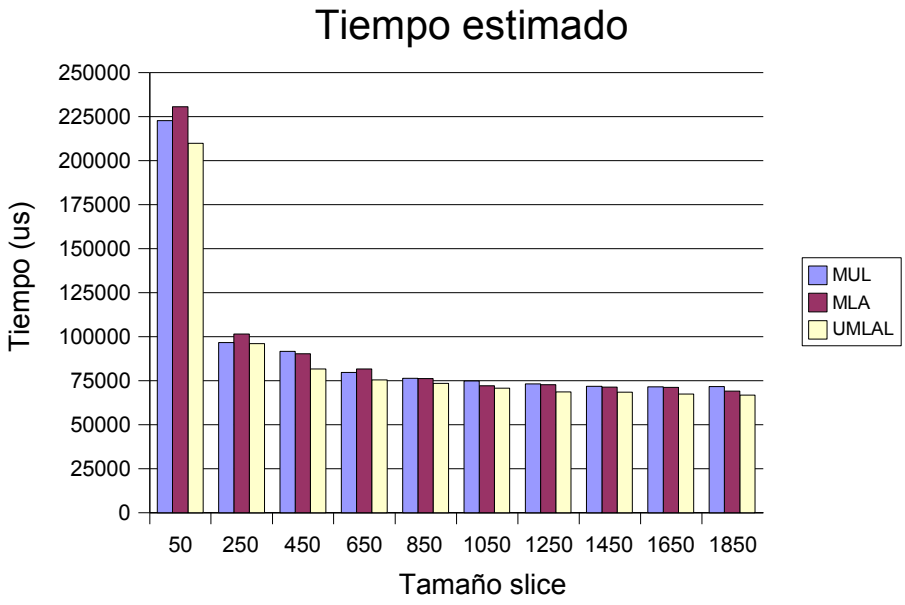
```
mov r1, #256
mov r2, #32
mov r3, r2
mul r4, r1, r2
```

Código MLA:

```
mov r1, #256
mov r2, #32
mov r3, r2
mla r4, r1, r2, r3
```

Código UMLAL:

```
mov r1, #131072
mov r2, #65536
mov r3, r2
umlal r4, r5, r1, r2
```



La instrucción que requiere normalmente más tiempo para ejecutarse es la multiplicación con acumulación (MLA) y la multiplicación sin acumulación (MUL) la sigue muy de cerca e incluso necesita más tiempo para algunos tamaños del *slice*.

Por otra parte, la instrucción de multiplicación de 64 bits con acumulación (UMLAL) a pesar de ser más compleja y necesitar más instrucciones para ser emulada, se ve beneficiada por el uso de las instrucciones MMX y obtiene la mayor parte de las veces el menor tiempo de ejecución.

Volvemos a observar que tamaños del *slice* muy pequeños aumentan el tiempo de ejecución, y para tamaños grandes disminuye notablemente con apenas variación entre ellos.

Test 3

Ejecución de 4 millones de ciclos con un tamaño de *slice* entre 25 y 1000, aumentando en 100 ciclos.

Código LDR/STR Pre:

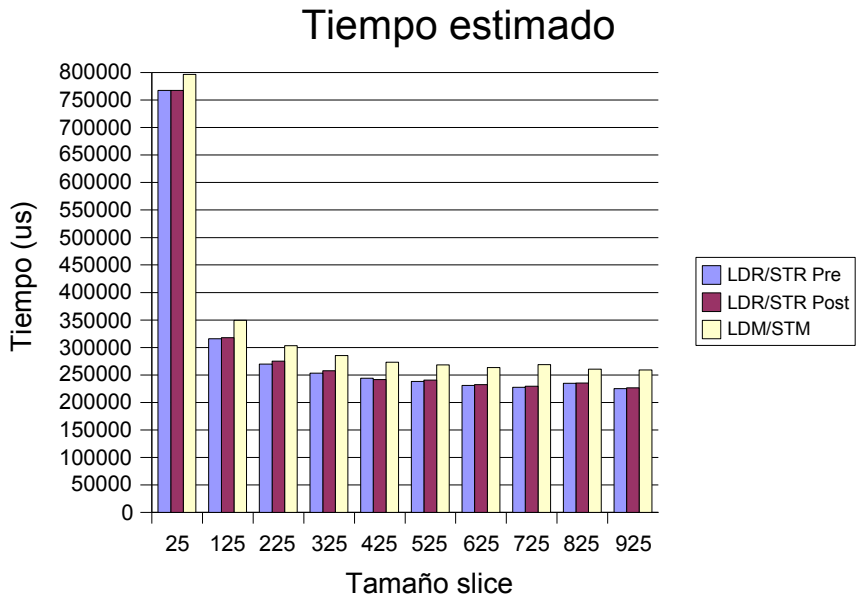
```
mov r1, #0x4000
ldr r5, [r1, #16]!
and r5, r1, r2 ROR #2
mvn r2, #4
add r1, r1, r2
str r5, [r1, #16]!
```

Código LDR/STR Post:

```
mov r1, #0x4000
ldr r5, [r1], #16
and r5, r1, r2 ROR #2
mvn r2, #4
add r1, r1, r2
str r5, [r1], #16
```

Código LDM/STM:

```
mov r1, #0x4000
ldmia r1!, {r5-r8}
and r5, r1, r2 ROR #2
mvn r2, #4
add r1, r1, r2
stmia r1!, {r5-r8}
```



En este test se puede apreciar que en las instrucciones de transferencia de una sola palabra no influye el modo de direccionamiento y los tiempos de ejecución para pre-indexado y post-indexado son prácticamente iguales.

Para las transferencias de múltiples registros el tiempo es mayor, como sucede en el ARM, debido a que se ha de cargar o escribir en varios registros. Aún así la diferencia respecto a las instrucciones que transfieren una única palabra es pequeña y no se ve casi afectada al cambiar el tamaño del *slice.* Esta diferencia será mayor si en estas transferencias se carga o escribe en más de 4 registros. Cabe destacar también la ganancia de rendimiento al utilizar un tamaño de *slice* mayor e igual a 100.

Test 4

Ejecución de 1 millón de ciclos con un tamaño de *slice* entre 10 y 100, aumentando en 10 ciclos.

Código Sin Condición:

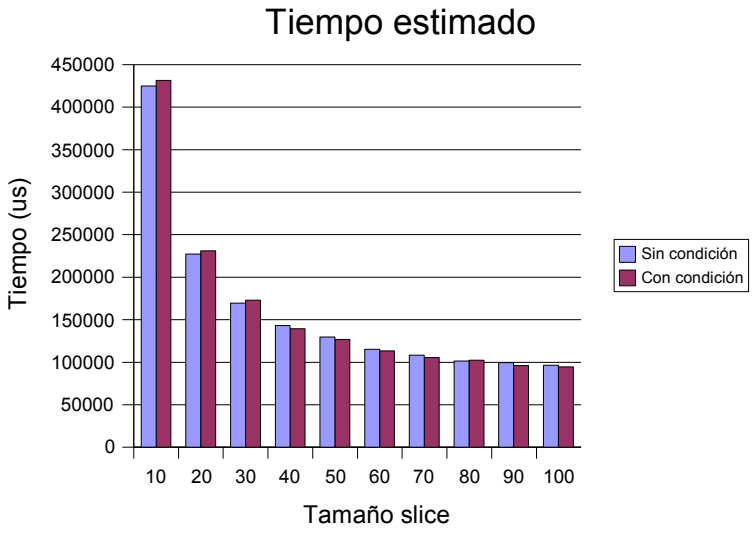
```
mov r0, #0
sub r0, r0, #0
mrs r1, cpsr
and r1, r1, #0x40000000
```

sub r2, r1, #32

Código con Condición:

```
mov r0, #0
subs r0, r0, #0
mrseq r1, cpsr
ands r1, r1, #0x40000000
```

subne r2, r1, #32



Para el código con condición se necesita de más tiempo de ejecución para *slices* bajos puesto que debe actualizar los bits de estado (CPSR) según el resultado de las 2 instrucciones que posee, y que tienen el bit S activado. El otro código no tiene ninguna instrucción de este tipo y se ahorra las instrucciones de emulación para realizar la actualización de los bits de estado.

Las 2 instrucciones que para emularse han de cumplir una condición en el código de condición, no influyen en la diferencia de tiempo entre los dos códigos, ya que en este ejemplo siempre se van a ejecutar al cumplirse sus condiciones. Si no fuera así, habría que tenerlo en cuenta debido a que posiblemente consumirían menos ciclos al no ejecutarse.

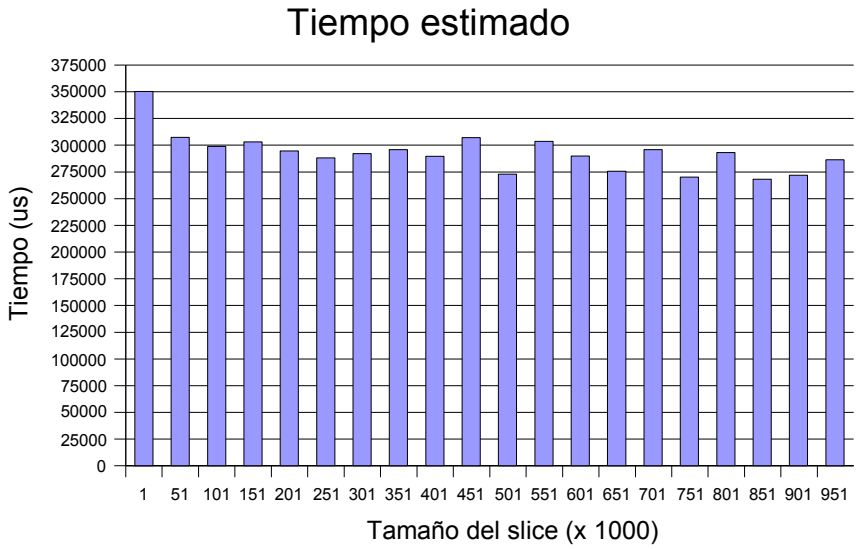
Con valores más altos del *slice* podemos ver que la diferencia de tiempo entre los dos códigos va reduciéndose progresivamente hasta llegar a ser inexistente. Un tamaño del *slice* mayor de 10 ciclos mejora considerablemente el rendimiento de ambos códigos y valores mayores e iguales a 60 permiten que la emulación de ambos códigos tenga un tiempo de ejecución similar.

Test 5

Ejecución de 5 millones de ciclos con un tamaño de *slice* entre 1000 y 1000000, aumentando en 50000 ciclos.

Código División:

```
                mov     r0, #7
                mov     r1, #2
                mov     r2, #1
divasm1:        cmp     r1, #-2147483648
                cmpcc   r1, r0
                movcc   r1, r1, lsl #1
                movcc   r2, r2, lsl #1
                bcc     <divasm1>
                mov     ip, #0
divasm2:        cmp     r0, r1
                subcs   r0, r0, r1
                addcs   ip, ip, r2
                movs    r2, r2, lsr #1
                movne   r1, r1, lsr #1
                bne     <divasm2>
                mov     r0, ip
```



Como sucede en los anteriores tests cuanto mayor es el tamaño del *slice* mayor es la disminución del tiempo de ejecución. Entre 1000 y 50000 ciclos la reducción es muy significativa, y a partir de 50000 sigue habiéndo algunas reducciones aunque de menor tamaño.

Luego tales valores favoreceran a la velocidad del emulador en detrimento de la precisión de las interrupciones. A su vez, el emulador (núcleo) cederá el control al programa (función externa) de forma más tardía, como vamos a ver en el siguiente test.

Test 6

Ejecución de 5 millones de ciclos con un tamaño de *slice* entre 25000 y 700000, aumentando en 25000 ciclos.

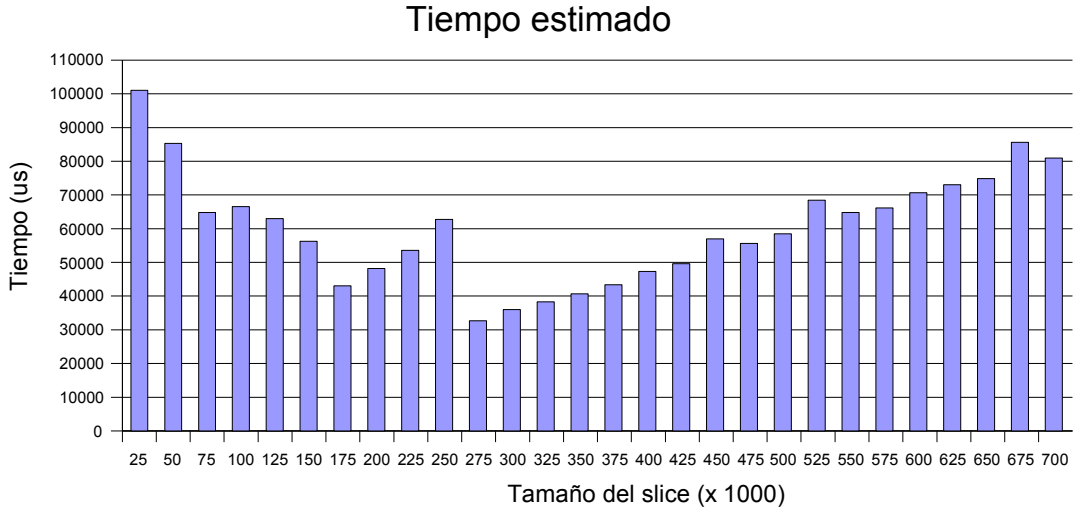
Código Buffer de pantalla de video:

```
                mov     ip, sp
                stmdb   sp!, {fp, ip, lr, pc}
                sub     fp, ip, #4
                sub     sp, sp, #12
                mov     r3, #0
                str     r3, [fp, #-24]
                mov     r3, #0
                str     r3, [fp, #-20]
                mov     r3, #134217728
                str     r3, [fp, #-16]
                b       <salto1>
salto1: mov     r3, #0
                str     r3, [fp, #-24]
                b       <salto2>
salto3: ldr     r3, [fp, #-24]
                mov     r3, r3, lsl #2
                mov     r2, r3
                ldr     r3, [fp, #-16]
                add     r2, r2, r3
                ldr     r3, [fp, #-20]
                str     r3, [r2]
                ldr     r3, [fp, #-24]
                add     r3, r3, #1
                str     r3, [fp, #-24]
salto2: ldr     r2, [fp, #-24]
                mov     r3, #38400
                sub     r3, r3, #1
                cmp     r2, r3
                ble     <salto3>
```

```
ldr     r3, [fp, #-20]
add     r3, r3, #21
str     r3, [fp, #-20]
ldr     r3, [fp, #-20]
cmp     r3, #255
bne     <salto1>
mov     r3, #0
str     r3, [fp, #-20]
b       <salto1>
```

Este código modifica el buffer de la pantalla de video y cuando el núcleo del emulador finaliza un *slice*, a través de la función externa se actualiza la pantalla mostrando los cambios realizados en el buffer como ya se explicó en el programa de prueba.

Como vemos en la siguiente gráfica y a diferencia de los anteriores tests, la disminución del tiempo de ejecución no siempre es creciente a medida que se aumenta el tamaño del *slice*.



En efecto, a medida que aumentamos el tamaño del *slice* desde 25000 ciclos, el tiempo de ejecución disminuye progresivamente, pero a partir de 300000 ciclos el tiempo vuelve a aumentar hasta volver a los valores iniciales.

Como ya hemos comentado esto es provocado por la actualización o refresco de la pantalla que se realiza en la función externa y que es invocada por el núcleo del emulador para cederle el control al finalizar un *slice*. El retardo que en dicha función se produzca también favorecerá o no a la velocidad de emulación dependiendo del tamaño del *slice*.

Así para valores pequeños del tamaño la pantalla será refrescada muchas veces pero habrá menos cambios en el buffer de video por refrescos, y si los valores son altos favorecemos que haya más cambios en el buffer pero también habrá menos refrescos de pantalla provocando saltos o transiciones bruscas en la imagen de video.

Por ello y tal como muestra la gráfica, el tamaño del *slice* adecuado será un valor intermedio que mantega un equilibrio entre el refresco y los cambios provocados en el buffer.

De igual modo esta idea es aplicable para que el rendimiento del emulador sea el mejor posible cuando se empleen programas que hagan uso de interrupciones externas.

8. Conclusiones

Nuestro proyecto, tal como explicamos al principio, estaba motivado por el desarrollo del core de un emulador que fuera posible utilizar en aplicaciones que necesitaran reproducir las funcionalidades de un ARM. Por otra parte, otro objetivo del proyecto consistía en realizar un documento donde se estudiara una arquitectura real y qué aspectos de ella era necesario tener en cuenta en la implementación del emulador.

Habiendo concluido el desarrollo del proyecto, podemos decir que el resultado en ambos objetivos ha sido el que se deseaba. Por una parte, se consiguió completar el desarrollo de la librería cumpliendo con la especificación, y comprobar su correcto funcionamiento en la ejecución de programas reales.

Además, durante el proceso hemos analizado las ventajas e inconvenientes de las distintas técnicas de construcción de emuladores. Así mismo, hemos adquirido un conocimiento de las arquitecturas en procesadores empotrados en general, y del ARM en concreto.

Los primeros meses del proyecto se concentraron en obtener documentación referente tanto a desarrollo de emuladores como a la arquitectura ARM. En este punto decidimos la estructura que tendría nuestro programa, así como la plataforma y el lenguaje sobre los que se construiría.

Decidimos también emular el procesador ARM9TDMI (arquitectura v4T) dado que la mayor cantidad de documentación disponible hacía referencia a éste.

Inicialmente pensamos en implementar un sistema de pre-decodificación de instrucciones, pero debido a que la ganancia en eficiencia no era muy significativa no compensaba su desarrollo.

En cuanto a las dificultades encontradas durante el desarrollo, cabe destacar las siguientes:

La complejidad de la integración de código ensamblador y código fuente en C/C++ con el entorno de desarrollo Microsoft Visual C++ 6.0, que nos obligó a reestructurar parte del código y de las macros utilizadas, dado que una mala organización podía provocar comportamientos inesperados.

La emulación del comportamiento de los bits de estado del ARM. Dado que su comportamiento no era en todos los casos igual a los del x86, fue difícil determinar las diferencias entre ambos lo que provocó errores en su implementación que no fueron detectados hasta las primeras ejecuciones de prueba.

Por otra parte, al no disponer de un procesador ARM real y tener que realizar todas las pruebas sobre un emulador / simulador más limitado, esto complicó conocer el funcionamiento real de ciertos detalles, que dicho emulador no reproducía.

En los inicios del proyecto no realizamos una evaluación realista de la complejidad del desarrollo del emulador. Por tanto, ciertas características que esperábamos incluir se quedaron fuera por falta de tiempo. Por ejemplo, una aplicación que usara el core para emular una máquina real, o el repertorio Thumb.

Uno de los problemas al que nos enfrentamos era la falta de especificación detallada de algunas de las características del ARM. Esto nos obligó a realizar búsquedas de documentación usando distintas fuentes, y en algunos casos a tomar decisiones basadas en el comportamiento del emulador / simulador que usábamos durante el desarrollo.

Por otra parte, tuvimos éxito en cumplir los requisitos de velocidad, uno de los principales objetivos de cualquier emulador. Actualmente se puede ejecutar código real a una frecuencia suficientemente alta como para emular la mayoría de máquinas que usan el ARM.

Si bien no implementamos todo el protocolo de la librería estándar de C en nuestro programa de prueba, sí que incluimos lo suficiente para ejecutar programas simples, y las bases para expandirlo en un futuro a programas más complejos.

La estructura del proyecto se mostró también adecuada a la hora de integrar la librería en una aplicación de forma rápida y fácil, sin tener que realizar grandes modificaciones sobre la misma.

Algunos objetivos interesantes que no se alcanzaron durante el transcurso del proyecto pero que se podrían realizar en el futuro son los siguientes:

Integración del modo Thumb al core. Actualmente el diseño toma en cuenta esta posibilidad, por lo que sería fácil añadir soporte para este modo de forma paralela al que ya existe para el modo ARM.

Desarrollo de una aplicación más avanzada que el programa de prueba actual, basada en una plataforma real. Por ejemplo, sería interesante emular un dispositivo móvil, o una consola de videojuegos portátil (ej: GameBoy Advance) que usen el ARM como base.

Soporte para pre-decodificación de instrucciones. Si bien el programa no está originalmente diseñado para esta posibilidad, sería posible hacer las modificaciones necesarias para incorporar esta características.

Adaptación del emulador a otros sistemas operativos, como puede ser Linux. En este caso, sería necesario adaptar el formato del código ensamblador de Intel a AT&T, que se utiliza en los compiladores de esta plataforma.

Al término del proyecto, decidimos liberar el código del mismo bajo licencia LGPL para que así otros desarrolladores puedan contribuir a continuar el proyecto añadiendo nuevas características.

9. Bibliografía

- Steve Furber, "ARM System-on-Chip Architecture (2nd Edition)". Addison-Wesley Professional, 2000.
- James E. Smith y Ravi Nair, "Virtual Machines, Versatile Platforms for Systems and Processes". Elsevier, 2005.
- Victor Moya del Barrio, "Study of the techniques for emulation programming", 2001.
- "ARM Architecture Reference Manual", http://www.arm.com/
- "ARM9TDMI Technical Reference Manual", http://www.arm.com/
- "ARM Software Development Toolkit User's Guide", http://www.arm.com/
- "ARM Developer Suite Version 1.2 Debug Target Guide", http://www.arm.com/
- "ARM ELF Specification", http://www.arm.com/
- "ARM ELF File Format", http://www.arm.com/
- "ARM Monitor, Program Loading and Initialization". Intel Lectures, Introduction to Embedded Systems.
- Daniel Boris, "How do I write an emulator?", 1999.
- "ARM Assembler", http://heyrick.co.uk/assembler
- Paul A. Carter, "Lenguaje Ensamblador para PC", 2006
- Ciriaco García de Celis, "El universo digital del IBM PC, AT y PS/2", http://www.gui.uva.es/udigital/
- Roger Jegerlehner, "Intel Assembler 80x86 Codetable, Overview of Instructions", http://jegerlehner.ch/intel/
- http://www.gnuarm.com/
- http://simit-arm.sourceforge.net
- http://en.wikipedia.org

Apéndice A - Listado de las instrucciones del ARM

MOV	Guarda un operando en un registro.
MVN	Guarda un operando con los bits negados en un registro.
MRS	Guarda el CPSR/SPSR en un registro.
MSR	Guarda un registro en el CPSR/SPSR.
ADD	Suma de un operando y registro.
ADC	Suma de un operando y registro con carry.
SUB	Resta de un registro con un operando.
SBC	Resta de un registro con un operando menos carry negado.
RSB	Resta de un operando con un registro.
RSC	Resta de un operando con un registro menos carry negado.
MUL	Multiplicación de dos registros.
MLA	Multiplicación de dos registros con acumulador.
UMULL	Multiplicación sin signo de dos registros con resultado de 64 bits.
UMLAL	Multiplicación sin signo de dos registros con acumulador con resultado de 64 bits.
SMULL	Multiplicación con signo de dos registros con resultado de 64 bits.
SMLAL	Multiplicación con signo de dos registros con acumulador con resultado de 64 bits.
CMP	Comparación de un registro con operando (Registro – Operando).
CMN	Comparación de un registro con operando (Registro + Operando).
TST	Comparación de un registro con operando (Registro AND Operando).
TEQ	Comparación de un registro con operando (Registro EOR Operando).
AND	Registro AND Operando.
EOR	Registro EOR Operando.
ORR	Registro OR Operando.
BIC	Registro AND NOT Operando.
B	Salto.
BL	Salto con Link.
BX	Salto con cambio de modo de ejecución.

LDR	Carga de memoria a un registro.
LDM	Carga múltiple de memoria a registros.
STR	Almacenamiento de un registro en memoria.
STM	Almacenamiento múltiple de registros a memoria.
SWP	Intercambio de valores entre registros y memoria.
CDP	Operaciones sobre datos del coprocesador.
MRC	Almacena un registro del ARM en un registro del coprocesador.
MCR	Almacena un registro del coprocesador en un registro del ARM.
LDC	Load del coprocesador.
STC	Store del coprocesador.
SWI	Interrupción por software.

Apéndice B - Comandos del programa de prueba

Esta es la lista de comandos que acepta el programa de prueba. Entre paréntesis se indican los tipos de los parámetros: números en decimal (d), en hexadecimal (x) o cadenas de texto (t). Los parámetros entre llaves {} son opcionales:

Comando	Descripción
sr *k*	Muestra el contenido del registro 'k'.
mr *k v*(x/d)	Modifica el contenido del registro 'k' con el valor 'v'.
smb *i*(x) *s*(x)	Muestra el contenido de la memoria desde la dirección 'i' hasta la 's'.
mmb *d*(x) *v*(x)	Modifica el contenido de la dirección de memoria 'd' con el valor 'v'.
emb *d*(x)	Activa el modo de edición de memoria desde la dirección 'd'. Para finalizar, introducir el caracter '.'
sf	Muestra el valor de los bits de estado.
mf *nzcv*	Modifica los bits de estado con los valores 'nzcv'.
sc	Muestra el modo de ejecución del procesador.
mc *k*(d)	Cambia el modo de ejecución al indicado por 'k'.
ss	Muestra los valores de todos los registros SPSR.
spc	Muestra el contenido del PC y de la dirección de memoria a la que éste apunta.
elf *arch*(t)	Carga el archivo indicado como ejecutable ELF.
txt *arch*(t) *d*(x)	Carga el archivo de texto indicado a partir de la dirección de memoria 'd'.
bin *arch*(t) *d*(x)	Carga el archivo binario indicado a partir de la dirección de memoria 'd'.
vid	Activa y desactiva la salida de vídeo.
rst	Resetea el emulador a su estado inicial.
shz	Muestra la frecuencia actual del emulador (en Mhz).
mhz *f*(d)	Modifica la frecuencia al valor 'f' (en Mhz).
slc	Muestra el tamaño del *slice* actual (en ciclos).

mlc *s*(d)	Modifica el tamaño del *slice* al valor 's' (en ciclos).
run {*d*(x)}	Ejecuta a partir de la dirección 'd', o del valor del PC que hubiese almacenado si no se especifica parámetro.
step	Ejecuta la siguiente instrucción apuntada por el PC.
bench *n*(d)	Ejecuta 'n' *slices* consecutivos y muestra el tiempo que se ha tardado.
h	Muestra la ayuda del programa.
q	Sale del programa.

Apéndice C – Tipos de datos del ARM9Core

Tipos de datos comunes :

```
typedef unsigned char  uint8;    //Byte sin signo
typedef signed char  int8;       //Byte con signo
typedef unsigned short  uint16;  //Media palabra sin signo
typedef short  int16;            //Media palabra con signo
typedef unsigned int  uint32;    //Palabra sin signo
typedef int  int32;              //Palabra con signo
typedef unsigned __int64 uint64; //Doble palabra sin signo
typedef __int64 int64;           //Doble palabra con signo
```

Tipo de dato correspondiente a los distintos tipos de acceso a un registro de 32 bits:

```
typedef union{

	//Acceso a la palabra (32 bits) sin signo y con signo
	uint32 u_word;
	int32 s_word;

	//Acceso a la media palabra (16 bits) sin signo y con signo
	uint16 u_halfword;
	int16 s_halfword;

	//Acceso a los bytes de la media palabra sin signo
	struct {
		uint8 lByte; //Byte bajo
		uint8 hByte; //Byte alto
	} u_hw;

	//Acceso a los bytes de la media palabra con signo
	struct {
		int8 lByte;  //Byte bajo
		int8 hByte;  //Byte alto
	} s_hw;

	uint8 u_byte;	//Acceso al byte sin signo
	int8 s_byte;	//Acceso al byte con signo

	//Acceso a los bits del CPSR
	struct{
		unsigned int mode:5;	//Mode bits
		unsigned int state:1;	//State bit
		unsigned int fiqDis:1;	//FIQ disable
		unsigned int irqDis:1;	//IRQ disable
		unsigned int reservados:20;
		unsigned int v:1;	//Overflow
		unsigned int c:1;	//Carry / Borrow / Extend
		unsigned int z:1;	//Zero
		unsigned int n:1;	//Negative / Less Than
	} CPSR;

} ARM9Reg32;
```

Tipo de dato enumerado correspondiente a los distintos modos de ejecución:

```
typedef enum {
	USER_MODE = 0,		//User mode
	SVC_MODE,		//Supervisor mode
	ABORT_MODE,		//Abort mode
	UNDEF_MODE,		//Undefined mode
	IRQ_MODE,		//Interrupt mode
	FIQ_MODE,		//Fast Interrupt mode
	SYSTEM_MODE		//System mode
} ARM9Mode;
```

Tipo de dato enumerado correspondiente a los distintos tipos de interrupciones:

```
typedef enum {
	EXCEP_NONE = 0,			//No hay excepcion
	EXCEP_UNDEFINED,		//Instruccion indefinida
	EXCEP_SWI,			//Software interrupt
	EXCEP_PREFETCH_ABORT,		//Error de memoria al leer instruccion
	EXCEP_IRQ,			//Interrupcion estandar
	EXCEP_FIQ,			//Interrupcion rapida
	EXCEP_DATA_ABORT,		//Error de memoria
	EXCEP_RESET			//Reinicio del procesador
} ARM9Exception;
```

Tipos de punteros a funciones handler de acceso a memoria:

```
typedef uint32 (*readHandler32)(uint32 dir);
typedef uint32 (*readHandler16)(uint32 dir);
typedef uint32 (*readHandler8)(uint32 dir);
typedef void (*writeHandler32)(uint32 datos, uint32 dir);
typedef void (*writeHandler16)(uint16 datos, uint32 dir);
typedef void (*writeHandler8)(uint8 datos, uint32 dir);
```

Tipo de dato enumerado correspondiente a los distintos permisos de memoria:

```
typedef enum {

	USR_READ_PERM = 0x1,		//Acceso a lectura para usuario
	USR_WRITE_PERM = 0x2,		//Acceso a escritura para usuario
	PRIV_READ_PERM = 0x4,		//Acceso a lectura para modo privilegiado
	PRIV_WRITE_PERM = 0x8		//Acceso a escritura para modo privilegiado

} ARM9MemPermission;
```

Tipo de dato correspondiente a una región de memoria:

```
typedef struct {

	uint32 inferior;	//Dirección inferior del bloque de memoria
	uint32 superior;	//Dirección superior
	uint8 *buffer;		//Buffer de memoria

	//Punteros a funciones manejadoras de acceso a memoria
	void *handlerRead8;
	void *handlerRead16;
	void *handlerRead32;
	void *handlerWrite8;
	void *handlerWrite16;
	void *handlerWrite32;

	int permisos;		//Permisos de acceso a memoria

} ARM9MemRegion;
```

Tipos de punteros a funciones handler del coprocesador:

```
typedef int (*coproDP)(uint32 instruccion);
typedef int (*coproLS)(uint32 instruccion);
typedef int (*coproRT)(uint32 instruccion);
```

Tipo de dato de las instrucciones ARM de 32 bits:

```
typedef union {

	//Acceso a la palabra (32 bits) sin signo y con signo
	uint32 u_word;
	int32 s_word;

	//Acceso a los campos comunes de cada instrucción. Son usados para
	//decodificar la instrucción en la tabla de saltos

	struct {
		unsigned int relleno:4;		//No usar
		unsigned int codigoTabla2:4;	//Segundo codigo en la tabla de
						//saltos
		unsigned int relleno2:12;	//No usar
		unsigned int codigoTabla:8;	//Codigo en la tabla de saltos
		unsigned int cond:4;		//Codigo de condicion
	} comunes;

	//Acceso a los campos por tipo de instrucción

	//Data processing inmediate shift
	struct {
		unsigned int Rm:4;		//2do operando
		unsigned int bit4:1;		//Distingue entre DPIS y DPRS
		unsigned int shift:2;		//Shift
		unsigned int shifta:5;		//Shift amount
		unsigned int Rd:4;		//Registro destino
		unsigned int Rn:4;		//1er operando
		unsigned int S:1;		//Set condition codes
		unsigned int opcode:4;		//Opcode
		unsigned int relleno:7;		//No usar
	} DPIS;

	//Seguirían todos los tipos de instrucción restantes del repertorio ARM.

	...

} ARM9Instruccion32;
```

Apéndice D - Lista de métodos del ARM9Core

Función que reinicia el estado del emulador:

```
void reset();
```

Función que define el número de núcleos del emulador:

```
void setCores(int numeroCores);
```

Función que contiene el bucle principal del ARM9Core, realiza la sincronización, la ejecución de los *slices* de cada núcleo, el tratamiento de interrupciones y la llamada a la función externa:

```
void run(int nCiclos);
```

Función que ejecuta una única instrucción:

```
void step();
```

Función que detiene la ejecución de instrucciones:

```
void stop();
```

Función que señaliza una excepción si es de mayor prioridad que la actual para que después sea tratada:

```
void marcarExcepcion (ARM9Exception excepcion, int param=0);
```

Función que trata las distintas excepciones que se puedan producir:

```
void tratarExcepciones ();
```

Función que inicializa las regiones de memoria definidas por el usuario:

```
bool addMemoria (int nRegiones, ARM9MemRegion *lista);
```

Función que lee la siguiente instrucción a ejecutar teniendo en cuenta los permisos de acceso a memoria y las regiones de memoria:

```
uint32 readInstruccion(uint32 direccion);
```

Función que lee de su correspondiente región de memoria una palabra teniendo en cuenta los permisos de acceso a memoria:

```
uint32 readMemoriaWord(uint32 direccion);
```

Función que lee de su correspondiente región de memoria un byte teniendo en cuenta los permisos de acceso a memoria:

```
uint8 readMemoriaByte(uint32 direccion);
```

Función que lee de su correspondiente región de memoria media palabra teniendo en cuenta los permisos de acceso a memoria:

```
uint16 readMemoriaHalfword(uint32 direccion);
```

Función que escribe en su correspondiente región de memoria una palabra teniendo en cuenta los permisos de acceso a memoria:

```
void writeMemoriaWord(uint32 datos, uint32 direccion);
```

Función que escribe en su correspondiente región de memoria un byte teniendo en cuenta los permisos de acceso a memoria:

```
void writeMemoriaByte(uint8 datos, uint32 direccion);
```

Función que escribe en su correspondiente región de memoria media palabra teniendo en cuenta los permisos de acceso a memoria:

```
void writeMemoriaHalfword(uint16 datos, uint32 direccion);
```

Función que define la frecuencia en Mhz del emulador:

```
void setFrecuenciaARM (double f);
```

Función que calcula la frecuencia de la máquina en que se ejecuta el emulador:

```
void calcularFrecuenciaX86 ();
```

Función que devuelve el tiempo en un instante de la ejecución del programa:

```
double calcularTiempoX86 ();
```

Función que enlaza la función externa que deberá implementar el usuario del emulador para el tratamiento de interrupciones FIQ, IRQ, y para sus propios usos:

```
void setFuncionExterna (void (*func)(void));
```

Función que enlaza los handlers para las interrupciones que el usuario puede implementar:

```
void setExcepHandler (ARM9Exception ex, void (*handler)(int n));
```

Función que enlaza los handlers para las interrupciones del coprocesador que el usuario puede implementar:

```
void setCopro (int cpn, coproDP fdp, coproLS fls, coproRT frt);
```

Apéndice E - El formato ELF

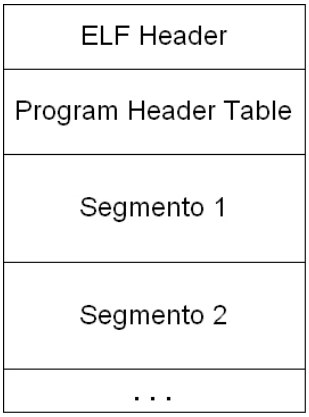
El formato de archivo .ELF (Executable and Linking Format) es utilizado por el SDK oficial de ARM como formato de archivo ejecutable genérico. Esto quiere decir que al compilar un programa para un procesador ARM (usando un compilador como el GNUARM) lo que obtendremos será un archivo de este tipo, con toda la información necesaria para cargar y ejecutar el programa.

Por tanto, y dado que en nuestro proyecto nos marcamos como objetivo el ejecutar programas reales compilados para la máquina original, era necesario poder cargar este tipo de archivos para realizar las pruebas sobre ellos.

Este formato es genérico, y se usa también para otras arquitecturas. Además, no sólo sirve para almacenar la información del programa final, sino que se suele usar también en los pasos intermedios de la compilación y enlazado del mismo como "archivo de objeto". En nuestro caso, haremos un resumen únicamente de los campos que son utilizados cuando se usa como archivo de programa final en la arquitectura ARM.

En nuestro proyecto usamos la librería ELFIO (http://elfio.sourceforge.net) para simplificar el proceso de carga. Sin embargo, sigue siendo necesario conocer la distribución del archivo y sus distintos campos para saber cómo se deben tratar cada uno de ellos.

Cuando se usa un .ELF como programa ejecutable ("executable file"), el archivo se organiza en los siguientes bloques:



El primer bloque, "ELF Header" guarda la información de cabecera del programa. A continuación se sitúa la tabla de cabeceras del programa, que tiene la información relativa a cada uno de los segmentos que componen el mismo. Por último, se almacenan los datos de los segmentos.

A la hora de cargar un .ELF debemos en primer lugar comprobar varios campos de la cabecera del archivo para asegurarnos que se cumple la especificación ARM y el archivo es válido:

- **e_ident**[EI_CLASS]: Debe ser “ELFCLASS32” (32 bits).

- **e_ident**[EI_DATA]: Debe ser “ELFDATA2LSB” para little-endian.

- **e_machine:** Indica el tipo de procesador, debe ser “EM_ARM”.

- **e_entry:** Si el archivo es válido, este campo indica la dirección de memoria a partir de la cual se empieza a ejecutar su código. A la hora de cargar, el PC deberá a puntar a este valor. Normalmente suele ser superior a 0x8000.

Una vez comprobados estos campos, el proceso de carga recorre la tabla de cabeceras con información de cada sección o segmento del programa. Un archivo ELF por lo general se divide en varios segmentos distintos, unos con código o datos del programa, otros con información de depuración o relativa al enlazado, y otros que simplemente describen cómo ha de inicializarse cierta zona de la memoria, pero que no se corresponden con datos físicos del archivo.

A la hora de cargar desde el archivo será necesario comprobar la información de la cabecera para saber de qué tipo de segmento se trata y cuál es su tamaño, posición en memoria, etc... Sólo será necesario tener en cuenta aquellas secciones que sean relativas al programa ejecutable, mientras que todos los segmentos de enlazado, depuración, etc... son ignorados.

Los campos que es necesario consultar son:

sh_type:
Indica el tipo del segmento, los valores que nos interesan son:

- SHT_PROGBITS: Este segmento contiene datos físicos (bits) del archivo, que deberán ser copiados a la memoria de la máquina.

- SHT_NOBITS: Este segmento no ocupa espacio en el archivo, pero hace referencia a una zona de la memoria real que deberá ser inicializada a cero.

sh_flags:
Bits de estado del segmento, sólo nos interesan los valores siguientes:

- SHF_WRITE: Indica que la región de memoria a la que el segmento hace referencia deberá tener permisos de escritura, ya que el programa intentará guardar datos en ella.

- SHF_ALLOC: Indica que el segmento hace referencia a una región de memoria real. Este es el flag más importante, porque sólo trataremos con aquellos segmentos que lo tengan activo, ignorando al resto.

- SHF_EXECINSTR: Indica que el segmento contiene instrucciones del programa. En nuestro caso no realizamos ningún tratamiento especial, pero dependiendo del emulador puede que sea necesario manipular estos datos de forma distinta.

sh_addr:
Dirección de memoria a partir de la cual se sitúa el segmento.

sh_size:
Tamaño del segmento en bytes. Por regla general, el tamaño del segmento una vez cargado en la memoria y el de los datos físicos del archivo es el mismo. En el caso de que sea de tipo SHT_NOBITS, el segmento no ocupa un espacio físico del archivo, aunque este campo tenga valores distintos de cero (el tamaño que ocupará en la memoria).

sh_offset:
Indica a partir de qué punto del archivo se encuentran los datos físicos del segmento.

Los segmentos que deberemos cargar serán por tanto los que vayan señalizados con SHF_ALLOC. Dependiendo de si son SHT_PROGBITS o SHT_NOBITS cargaremos los datos desde el archivo o simplemente inicializaremos la región de memoria, respectivamente. El pseudocódigo sería:

```
si es SHF_ALLOC {
        si es SHT_PROGBITS {
                copiar sh_size bytes del archivo a la memoria (dir. sh_addr)
        }
        si es SHT_NOBITS {
                inicializar a cero sh_size bytes de la memoria (dir. sh_addr)
        }
        si SHF_WRITE {
                activar el permiso de escritura en la región de memoria.
        }
}
```

Apéndice F - El protocolo ANGEL

ARM facilita una herramienta de depuración (ANGEL) que hace las veces de programa "monitor". Este programa se instala tanto en la máquina ARM en la que se va a ejecutar el programa que queremos depurar (máquina target), como en el ordenador desde el cual trabajaremos (host).

Cuando el programa que estamos depurando se ejecuta utiliza el mecanismo de interrupciones por software para comunicar mensajes al programa monitor. Éste se conecta con el host y realiza las acciones necesarias. Con este mecanismo, por ejemplo, un programa puede escribir mensajes en la pantalla del host directamente, evitando la necesidad de conectar una pantalla al ARM y desarrollar todo el código necesario para hacerla funcionar.

Al compilar un programa con un compilador ARM genérico (como GNUARM) la implementación de la librería C utilizada normalmente está pensada para ejecutarse sobre un sistema ANGEL. Por tanto, si queremos ejecutar dichos programas en un emulador, será necesario incluir el soporte para las llamadas del sistema más frecuentes.

El protocolo de interrupciones de ANGEL utiliza siempre la SWI 0x123456. El identificador de la llamada vendrá dado por el valor del registro r0. Las llamadas del sistema son:

SWI	**Descripción**
SYS_OPEN (0x01)	Abre un fichero en el host.
SYS_CLOSE (0x02)	Cierra un fichero en el host.
SYS_WRITEC (0x03)	Escribe un caracter por la consola.
SYS_WRITE0 (0x04)	Escribe una cadena terminada en cero por la consola.
SYS_WRITE (0x05)	Escribe en un fichero del host.
SYS_READ (0x06)	Lee el contenido de un fichero en un buffer.
SYS_READC (0x07)	Lee un byte desde la consola.
SYS_ISERROR (0x08)	Determina si un valor devuelto es un código de error.
SYS_ISTTY (0x09)	Determina si un descriptor de fichero se corresponde a un terminal interactivo.
SYS_SEEK (0x0A)	Se desplaza a una posición del archivo.
SYS_FLEN (0x0C)	Devuelve el tamaño de un archivo.
SYS_TMPNAM (0x0D)	Devuelve el nombre temporal de un archivo.
SYS_REMOVE (0x0E)	Borra un archivo del host.
SYS_RENAME (0x0F)	Renombra un archivo del host.

SWI	Descripción
SYS_CLOCK (0x10)	Devuelve el tiempo desde el principio de la ejecución del programa, medido en centésimas de segundos.
SYS_TIME (0x11)	Devuelve el número de segundos transcurridos desde el 1 de Enero de 1970.
SYS_SYSTEM (0x12)	Envía un comando de consola al host.
SYS_ERRNO (0x13)	Devuelve el valor de la variable "errno" de la librería estándar de C.
SYS_GET_CMDLINE (0x15)	Devuelve la línea de comandos usada al ejecutar el programa.
SYS_HEAPINFO (0x16)	Devuelve el descriptor de la pila y el heap.
reason_EnterSVC (0x17)	Cambia a modo supervisor.
reason_ReportExceptio n (0x18)	Señaliza una excepción. Si el código de excepción es "ADP_Stopped_ApplicationExit" significa que el programa ha terminado su ejecución.
SYS_ELAPSED (0x30)	Devuelve el número de ticks desde el principio de la ejecución.
SYS_TICKFREQ (0x31)	Determina la frecuencia de los ticks.

Aunque para dar un soporte completo a la librería estándar de C sería necesario emular el comportamiento correcto de todas estas llamadas de sistema, por lo general basta con unas pocas para poder ejecutar los programas más simples y probar si el emulador funciona correctamente.

El conjunto mínimo que cualquier programa compilado en C usa son: SYS_OPEN, SYS_HEAPINFO, SYS_GET_CMDLINE, y reason_ReportException para señalizar el fin de la ejecución.

Si se desean emular programas más complejos será necesario implementar el comportamiento de más llamadas del sistema. Por ejemplo, si queremos que el programa pueda hacer uso de la función "printf" de C tendremos que implementar las llamadas del sistema relacionadas con la escritura de caracteres en consola (SYS_WRITE y similares).

Los detalles completos del protocolo están disponibles en la documentación oficial de ARM, "Debug Target Guide", (consultar la bibliografía).

Apéndice G - Herramientas GNUARM y Simlt-ARM

GNUARM

Es un conjunto de herramientas y utilidades, para Windows y Linux, que incluye principalmente un ensamblador y compilador C/C++ (GCC) para arquitecturas ARM. La librería utilizada por el compilador C/C++ es la Newlib, que incluye las funciones estándar de C. Estas utilidades tienen licencia GPL y su documentación y descarga pueden obtenerse desde http://www.gnuarm.org.

Mediante el ensamblador y compilador mencionado se genera el código objeto o ejecutable en formato ELF. El código ELF utilizado en las pruebas es de 32 bits y little-endian (el byte menos significativo está en la posición más baja de memoria). Para generar este código se puede partir de un archivo fuente con código ensamblador ARM o código C/C++ que emplee funciones dadas por la librería Newlib.

Según el tipo de código del archivo fuente la herramienta a utilizar será distinta. Usando la utilidad "arm-elf-as" creamos un archivo ELF a partir de código ensamblador ARM del modo siguiente:

```
arm-elf-as archivoEnsambladorARM -mcpu=arm9tdmi -o archivoELF
```

En el caso de partir de un archivo fuente en código C/C++ utilizaremos "arm-elf-gcc" de la siguiente forma:

```
arm-elf-gcc archivoC -mcpu=arm9tdmi -o archivoELF
```

Para ambos casos podemos generar el archivo ELF con diferentes opciones: incluir información de depuración, definir variables y símbolos, etc.

El archivo ELF que se obtiene del fuente en ensamblador ARM sólo contiene las secciones correspondientes al código ejecutable y datos, y debemos enlazarlo para poder hacerlo ejecutable mediante la utilidad "arm-elf-ld". En esta operación tanto los datos como el código ejecutable se situarán en las direcciones que se indiquen o que la utilidad tome por defecto. Por ejemplo si no se indica la dirección de comienzo (símbolo _start) de las instrucciones, ésta será la dirección 0x8000.
La orden siguiente enlazará el código objeto:

```
arm-elf-ld archivoELF -o archivoELFejecutable
```

El contenido del archivo ELF puede ser visualizado con la utilidad "arm-elf-objdump". Según las opciones utilizadas con esta herramienta es posible ver las distintas secciones que forman el archivo ELF en formato hexadecimal. Las instrucciones ARM además se muestran en ensamblador y con su dirección en hexadecimal. La siguiente orden mostraría todo el contenido del archivo ELF:

```
arm-elf-objdump -xD archivoELF
```

Para más información sobre el uso de estas y otras utilidades incluidas consultar la documentación en la dirección web mencionada.

SimIt-ARM

De entre los diversos emuladores ARM libres, SimIt-ARM es de los más sencillos y funcionales. Este emulador y simulador, para Linux, es capaz de ejecutar programas (en formato ELF) que utilizan únicamente el repertorio de instrucciones ARM, dado que no es capaz de emular el repertorio Thumb.

SimIt-ARM permite una ejecución paso a paso de los programas así como la visualización de los valores de los registros y de memoria. Además puede conocerse el tiempo que se tardó en ejecutar un programa y en el caso del simulador, también permite saber los ciclos empleados por cada instrucción, pero éste no ha sido utilizado dada la naturaleza de nuestro proyecto.

El uso del emulador del SimIt-ARM nos ha permitido comprobar el correcto funcionamiento de nuestro emulador, a través de varios programas de prueba y ejecución individual de la mayoría de las instrucciones del repertorio ARM.

Para ejecutar un programa usando el emulador del SimIt-ARM basta la siguiente orden:

```
ema -d archivoELF
```

Cabe recordar que para la correcta ejecución del archivo ELF, éste debe haber sido previamente enlazado si fue generado mediante la utilidad “arm-elf-as”.

Con el parámetro “-d” procederemos a realizar una ejecución paso a paso del programa. En este modo podemos visualizar los registros y memoria, ejecutar un número dado de instrucciones, indicar que el programa se ejecute hasta llegar a una determinada dirección de memoria, reiniciar los registros y el contador de programa, etc...

Si no indicamos el parámetro “-d”, el emulador nos devolverá el tiempo que se tardó en ejecutar el programa.

A la hora de ejecutar un programa en este emulador debemos tener en cuenta las siguientes desventajas e inconvenientes que observamos durante el desarrollo del proyecto:

- El espacio de memoria sólo puede estar formado por una única región.
- SimIt-ARM trata la memoria como un array, es decir, no se preocupa por el alineamiento.
- No se disponen de registros banqueados para cada modo de ejecución, teniendo sólo 16 registros (incluidos el SP, LR y PC), el CPSR y un único SPSR.
- Los modos de ejecución (bits de estado del CPSR) no influyen en la ejecución.
- Las excepciones no están implementadas.
- Los valores de los registros y de la memoria no pueden ser modificados.

SimIt-ARM dispone de licencia GPL y puede ser encontrada más información sobre su uso y descarga en la dirección web http://simit-arm.sourceforge.net/.

Los autores de la presente memoria autorizan a la Universidad Complutense a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la propia memoria, como el código, la documentación y/o el prototipo desarrollado.

Firmas de los autores:

Sergio Hidalgo Serrano

Alberto Huerta Aranda

Daniel Sañudo Vacas