Microcontroladores ARM

Advanced RISC Machine

Arquitectura ARM

Antecedentes

La arquitectura ARM se diseñó para permitir implementaciones de tamaño muy reducido y de alto rendimiento. Estas arquitecturas tan simples permiten dispositivos con muy bajo consumo de energía. Se caracteriza fundamentalmente por ser una computadora de set de instrucciones reducido (*Reduced Instruction Set Computer*, RISC), como lo indica su propio nombre.

El concepto RISC se originó en los programas de investigación de procesadores de las universidades de Stanford y Berkeley, alrededor de 1980. El único ejemplo de arquitectura RISC fue el de Berkeley, RISC I y II, y Stanford, MIPS (*Microprocessor without Interlocking Pipeline Stages*).

El primer procesador ARM fue desarrollado, entre 1983 y 1985, por *Acorn Computers Limited of Cambridge, England.* Fue el primer microprocesador RISC para uso comercial. Las posteriores arquitecturas RISCs tuvieron diferencias significativas con este primer diseño. Acorn tuvo una posición fuerte en el mercado de las computadoras personales del Reino Unido debido al suceso de la microcomputadora BBC (*British Broadcasting Corporation*). El micro BBC fue una máquina potenciada por un microprocesador de 8 bits, el 6502.

En 1990, ARM *Limited* se estableció como una compañía separada específicamente dedicada a la explotación de la tecnología ARM. El criterio de la empresa fue otorgar la licencia de esta tecnología a varios fabricantes de semiconductores en todo el mundo. Comenzó a establecerse como líder del mercado para aplicaciones embebidas de bajo consumo y costo.

Merchant RISC Microprocessor Shipments (1000s)

thru	1994	1995	1996	1997	1998	1999	2000	2001
ARM/StrongARM	2,170	2,100	4,200	9,800	50,400	152,000	414,000	402,000
MIPS	3,254	5,500	19,200	48,000	53,200	57,000	62,800	62,000
Hitachi SH	2,800	14,000	18,300	23,800	2,600	33,000	50,000	45,000
PowerPC	2,090	3,300	4,300	3,800	6,800	8,300	18,800	23,000
Total	30,499	33,830	58,480	98,220	149,080	262,820	556,800	538,860

Source: Andrew Allison

RISC shipments from 1994 to 2001

Figura extraida del libro "Co-Verification of Hardware And Software- for ARM SoC Design", Jason Andrews.

El ARM soporta una herramienta que incluye un emulador del set de instrucciones para verificación del modelo de *hardware* y el de *software* y *Assembler*, compiladores C y C++, un *linker* y un *debugger* simbólico.

Características heredadas de RISC

La arquitectura ARM incorporó algunas características del diseño RISC de Berkeley, aunque no todas. Las que se mantuvieron son:

- **Arquitectura de carga y almacenamiento** (*load-store*)
  Las instrucciones que acceden a memoria están separadas de las instrucciones que procesan los datos, ya que en este último caso los datos necesariamente están en registros

- **Instrucciones de longitud fija de 32 bits**
  Campos de instrucciones uniforme y de longitud fija, para simplificar la decodificación de las instrucciones.

- **Formatos de instrucción de 3 direcciones**
  Consta de "f" bits para el código de operación, "n" bits para especificar la dirección del 1er. operando, "n" bits para especificar la dirección del 2do. operando y "n" bits para especificar la dirección del resultado (el destino). El formato de esta instrucción en *Assembler*, por ejemplo para la instrucción de sumar dos números para producir un resultado, es:

```
ADD   d, s1, s2            ;d := s1 + s2
```

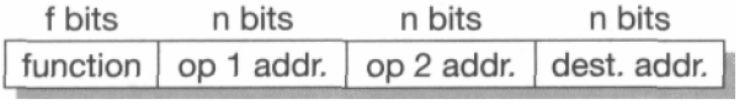


Figura 1.Formato de instrucción de tres direcciones.

En el formato de 3 direcciones de ARM las direcciones se especifican por registros, que se han cargado previamente con el contenido de las direcciones de memoria correspondientes.

En general, el término "**arquitectura de 3 direcciones**" se refiere a un conjunto de instrucciones donde los dos operandos fuente y el destino, se pueden identificar independientemente uno del otro, pero frecuentemente solo dentro de un restringido conjunto de valores posibles.
Esta arquitectura no destructiva (el resultado de la operación no se escribe sobre alguno de los operandos) permite a los compiladores organizar las instrucciones de manera de optimizar el "*pipeline*" (técnica de procesamiento de varias instrucciones en "paralelo")

Características incorporadas por ARM

Características que posteriormente agregó ARM:

- **Todas las instrucciones se ejecutan en un ciclo de reloj**
- **Modos de direccionamiento simples**
  El procesamiento de datos solo opera con contenidos de registros, no directamente en memoria.
- **Control sobre la unidad aritmética lógica (ALU,** *Arithmetic Logic Unit***) y el "***shifter***",** en cada instrucción de procesamiento de datos para maximizar el uso de la ALU y del "*shifter*".
- **Modos de direccionamiento con incremento y decremento automático de punteros,** para optimizar los lazos de los programas.
- **Carga y almacenamiento de múltiples instrucciones,** para maximizar el rendimiento de los datos.
- **Ejecución condicional de todas las instrucciones,** para maximizar el rendimiento de la ejecución.
- **Set de instrucciones ortogonal, regular o simétrico**
  En este tipo de set no hay restricciones en los registros usados en las instrucciones, son todos registros de propósitos generales, con muy pocas excepciones (por ejemplo el contador de programa, PC)
  A los programadores *Assembler* les resulta más fácil aprender un set con estas características. Y también, a los compiladores les resulta más fácil manejarlo. Mientras que la implementación del *hardware* será generalmente más eficiente.
- **Técnica "***pipeline***"**
  Esta técnica consiste en comenzar la próxima instrucción antes de que la actual haya finalizado. El objetivo es economizar tiempo.
- **Excepciones vectorizadas**
  Las excepciones son condiciones inusuales o inválidas asociadas con la ejecución de una instrucción particular.
- **Arquitectura "***Thumb***"**
  Algunos procesadores ARM tienen esta arquitectura para aplicaciones que necesiten mejorar la densidad de código. Consiste en usar un set de instrucciones de 16 bits que es una forma comprimida del set de instrucciones ARM de 32 bits.

Estas mejoras sobre la arquitectura RISC básica permiten a los procesadores ARM adquirir un buen equilibrio entre alto rendimiento, escaso tamaño de código, bajo consumo y poca área de silicio.

Revolución RISC

Reduced Instruction Set

En ese mundo de sets de instrucciones complejos en crecimiento, nació el *Reduced Instruction Set Computer* (RISC). Este concepto fue el que más influencia tuvo en el diseño de los procesadores ARM, inclusive ARM le debe a RISC la letra del medio de su nombre.

Arquitectura RISC

- Tamaño fijo de instrucciones de 32 bits, con pocos formatos
- Arquitectura de almacenamiento y carga (*load-store*) donde las instrucciones que procesan datos lo hacen solamente a través de registros y separadas de las instrucciones que acceden a memoria. Hay instrucciones para cargar los datos de memoria en registros y otras para guardarlos desde registros a memoria.
- Un banco grande de 32 registros. Todos ellos se pueden usar para cualquier propósito, permitiendo que la arquitectura de almacenamiento y carga funcione eficientemente

Estas características simplifican enormemente el diseño del procesador y permiten organizar la arquitectura de tal manera que aumenta el rendimiento de los dispositivos.

Organización RISC

- Lógica de decodificación de instrucción cableada por *hardware*.
- Técnica de "*pipeline*".
- Tiempo de ejecución en un único ciclo. Los procesadores RISCs al ser más simples se pueden diseñar para funcionar a velocidades de reloj que pueden usar todo el ancho de banda disponible de las memorias.

Ninguna de estas dos últimas propiedades es una característica de la arquitectura, pero ambas dependen de la arquitectura, que es lo suficientemente simple como para permitir que se las incorpore.

Ventajas RISC

- **Menor desperdicio de área de silicio**. Un procesador simple economiza transistores y área de silicio. En consecuencia una CPU RISC deja mayor área libre para realizar mejoras de rendimiento, tales como, memoria caché, funciones de manejo de memoria, punto flotante por *hardware,* etc.

- **Menor tiempo de desarrollo**. Un procesador simple tiene menor costo y lleva menos esfuerzo de diseño, además se adapta mejor a los emprendimientos de tecnología de procesos.

- **Mayor rendimiento**. Si se diseña un procesador simple y luego se le agregan instrucciones complejas hará en forma más eficiente varias funciones de alto nivel pero también decaerá un poco el reloj para el conjunto de las instrucciones. Midiendo los beneficios de esta técnica en programas típicos se comprueba que todos los sets de instrucciones complejos hacen que el programa corra a menor velocidad.

Inconvenientes RISC

- **No ejecuta códigos x86.** Pero hay programas de emulación para varias plataformas RISCs.

- **Generalmente tienen una pobre densidad de código comparada con CICS.** Ésta es consecuencia del set de instrucciones de longitud fija y para un gran número de aplicaciones, es bastante serio. En ausencia de memoria caché, esta pobre densidad de código significa que la búsqueda de la instrucción necesita un aumento del ancho de banda de la memoria principal, dando por resultado un mayor consumo.

Categoría de instrucciones

Puede esperarse que un set de instrucciones de propósitos generales incluya instrucciones que se encuentren dentro de alguna de las siguientes categorías:

- Instrucciones de procesamiento de datos, tales como adición, substracción, multiplicación.

- Instrucciones de movimientos de datos, que copian datos de una posición de memoria en otra, o de memoria a registros del procesador, etc.

- Instrucciones de control de flujo, que conmutan la ejecución de una parte del programa por otra, posiblemente dependiendo de los valores de los datos.

- Instrucciones especiales que controlan el estado de la ejecución del procesador, por ejemplo para conmutar a modo privilegiado que lleva a funciones del sistema operativo.

Uso dinámico del tiempo de instrucciones

Un enfoque para diseñar procesadores más rápidos consiste en estudiar cómo los procesadores usan su tiempo.
A nivel del set de instrucciones es posible medir la frecuencia de uso de varias instrucciones. Esto es muy importante para obtener mediciones dinámicas, o sea, aquellas que determinan la frecuencia con que se ejecutan las instrucciones, en contraposición con las mediciones estáticas que cuentan los tipos de instrucciones en la imagen binaria.

Instruction type	Dynamic usage
Data movement	43%
Control flow	23%
Arithmetic operations	15%
Comparisons	13%
Logical operations	5%
Other	1%

Figura 2. Típico uso dinámico de instrucciones

Esta tabla sugiere que para optimizar el uso del tiempo por parte del procesador hay que comenzar por optimizar las instrucciones de movimientos de datos porque éstas son las que le insumen la mayor parte del tiempo. Ocupan casi la mitad del tiempo del total utilizado por todas las instrucciones ejecutadas. Las instrucciones que controlan el flujo, tales como saltos y llamadas a procedimientos, ocupan casi la cuarta parte. En tercer lugar están las operaciones aritméticas, ocupan el 15%.
Sabiendo cómo reparten los procesadores su tiempo se pueden buscar métodos para hacerlos más veloces. Entre estos métodos, el más importante es el "*pipeline*". Otra técnica importante consiste en usar memoria caché.
Un procesador ejecuta una instrucción individual en una secuencia de etapas. Una secuencia típica podría ser:

1. *Búsqueda de código de operación*
   fetch. Buscar en la memoria el código de operación de la instrucción

2. *Decodificación*
   dec. Decodificar para saber de qué tipo de instrucción se trata

3. *Leer registro*
   reg. Acceder a un banco de registro para obtener algún operando

4. *Unidad Aritmético-Lógica*
   ALU Combinar los operandos para obtener el resultado o formar una dirección de memoria.

5. *Acceso a memoria*
   mem. Si es necesario, acceder a memoria para obtener un operando

6. *Escribir resultado*
   res. escribir el resultado en el banco de registros

Nótese que las instrucciones requerirán por lo menos de las primeras etapas (1. y 2.), aunque la mayoría requerirán más etapas. Estas etapas usan partes diferentes del *hardware*. Por ejemplo, la ALU probablemente se use solamente en la etapa 4. Así, si una instrucción comienza recién cuando terminó de ejecutarse su predecesora solamente se estará usando en cada etapa una pequeña parte del *hardware* del procesador.

Arquitectura carga-almacenamiento

Load-store architecture

Como la mayoría de los procesadores RISCs, el ARM emplea una arquitectura carga-almacenamiento. Esto significa que el set de instrucciones solamente procesará (adición, substracción, etc.) valores que estén en los registros o directamente especificados dentro de la instrucción en sí misma y siempre se obtendrá el resultado de tales procesos en un registro. Las únicas operaciones que se aplican a la memoria son aquellas que copian datos de la memoria en los registros (instrucciones de carga) o copian datos de los registros en la memoria (instrucciones de almacenamiento).
ARM no soporta operaciones memoria a memoria. Por lo tanto todas las instrucciones ARM caen en una de las tres categorías siguientes:

1. **Instrucciones que procesan datos**. Solamente usan y modifican valores en registros. Una instrucción, por ejemplo, puede sumar dos registros y ubicar el resultado en otro registro.

2. **Instrucciones de transferencia de datos**. Estas copian los datos de la memoria en registros (instrucciones de carga) o copian los datos de los registros en la memoria (instrucciones de almacenamiento). Una forma adicional, útil solamente en códigos de sistemas, intercambian un dato en memoria con un dato en un registro.

3. **Instrucciones de control de flujo**. Normalmente se ejecutan instrucciones ubicadas en direcciones de memorias consecutivas. Aunque frecuentemente el control del flujo de las instrucciones ocasiona que la ejecución conmute en una dirección diferente, ya sea en forma permanente (instrucciones de salto) o guarde una dirección de retorno para recuperar la secuencia original (instrucciones de salto y retorno) o ejecute un código de llamadas al supervisor del sistema, instrucciones tipo *trapping*, "atrapadas".

Pipeline

Se llama "*pipeline*" a la técnica que aprovecha un método obvio de optimizar los recursos de *hardware* y también el rendimiento del procesador. Consiste en comenzar a procesar una instrucción antes de que se haya finalizado de procesar la actual.
En la siguiente figura se ilustra la ejecución de instrucciones con la técnica *pipeline*. Tomando la secuencia de operaciones a partir de la instrucción "1", el procesador se organiza de tal manera que tan pronto como haya completado la primera etapa de esa instrucción, "*fetch*" y haya avanzado hacia la segunda etapa, comenzará la primera etapa, "*fetch*", de la próxima instrucción. En principio, de esta manera el tiempo de ejecución debería ser hasta seis veces más veloz que el que corresponde a instrucciones no superpuestas pero, como veremos luego, en la práctica no ocurre así.

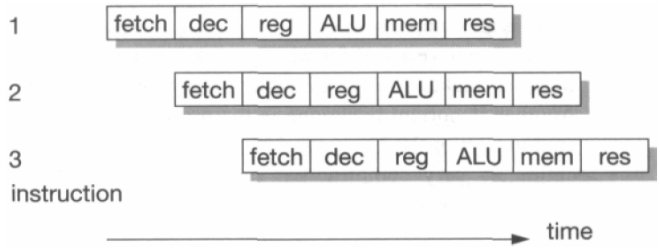


Figura 3, extraída de "ARM. System-on-Chip Architecture", Steve Furber.
Ejecución de instrucciones con la técnica *pipeline*

Pérdida de la ventaja del *Pipeline* (*Pipeline hazards*)

Riesgo de pérdida de la ventaja de *pipeline*, lectura después de escritura *(read-after-write pipeline hazard)*

En los programas típicos para computadoras, es muy frecuente que el resultado de una instrucción sirva de operando para la siguiente instrucción. Cuando sucede esto, se rompe la secuencia de *pipeline* mostrada en la figura anterior, ya que el resultado de la instrucción 1 no está disponible en el momento en que la instrucción 2 reúne sus operandos. Entonces, la instrucción 2 debe detenerse hasta que el resultado esté disponible, obteniéndose el comportamiento mostrado en la siguiente figura.

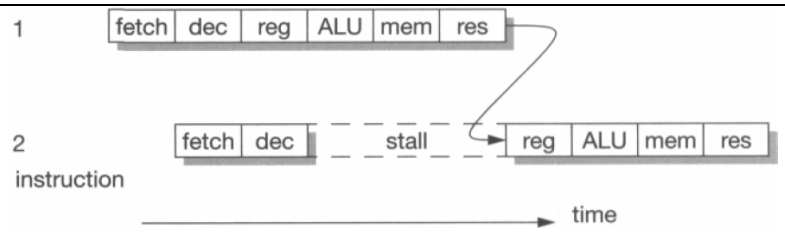


Figura 4, extraída de "ARM. System-on-Chip Architecture", Steve Furber.
Riesgo de pérdida de ventaja de *pipeline*, lectura después de escritura
(read-after-write pipeline hazard)

Las instrucciones de salto dan también por resultado una pérdida del comportamiento *pipeline* ya que se afecta la etapa de búsqueda de código de operación (*fetch*) de la siguiente instrucción y por lo tanto, se la debe diferir. Mientras se decodifica la instrucción de salto y antes de que se la reconozca como tal, se efectuarán las búsquedas de los códigos de operación (*fetches*) de las siguientes instrucciones en la secuencia. Si se ejecuta el salto, estas operaciones no tendrán sentido y se descartarán, perdiéndose las ventajas de esta técnica.

Si, por ejemplo, el cálculo de la dirección destino del salto se realiza en la etapa de ALU de la técnica *pipeline* de la primera figura, se buscarán los códigos de operación y comenzarán las distintas etapas de la ejecución de tres instrucciones de la vieja sucesión antes de que esté disponible la dirección destino del salto. Esto se ilustra en la siguiente figura. Es mejor calcular el destino del salto lo antes posible en la técnica *pipeline*, aún cuando esto requiera *hardware* dedicado. Si las instrucciones de salto tuvieran un formato fijo podría calcularse el destino en forma especulativa

(esto es, antes de que se determine que la instrucción es, efectivamente, de salto) durante la etapa de decodificación, **dec**, reduciendo la latencia del salto a un único ciclo. Aún en este *pipeline* de destino especulativo puede haber riesgo de pérdida de las ventajas de esta técnica, por ejemplo en los saltos condicionales dado que la dirección destino depende del resultado de la condición de la instrucción precedente a la de salto.

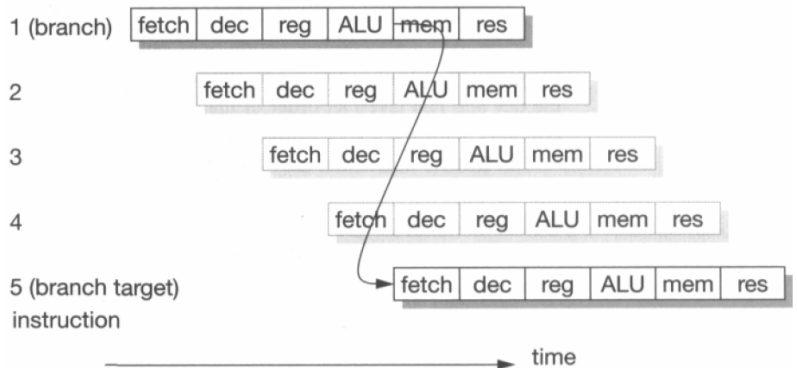


Figura 5, extraída de "ARM. System-on-Chip Architecture", Steve Furber.
Comportamiento de un salto usando la técnica *pipeline*

Hay técnicas para reducir el impacto de estos desperdicios de *pipeline* pero no se pueden evitar todos en forma simultánea.

Organización Pipeline de tres etapas

Esta organización se muestra en la siguiente figura. Los principales componentes son:

- El banco de registros (*register bank*), que almacena el estado del procesador. Tiene dos puertas de lectura y una de escritura que se pueden usar para acceder a cualquier registro, además tiene, una puerta adicional de lectura y otra puerta adicional de escritura que dan acceso especial al contador de programa, r15.
- El *barrel shifter*, que permite desplazar o rotar un operando un número determinado de bits.
- La ALU, que ejecuta las funciones aritméticas y lógicas requeridas por el set de instrucciones.
- Los registros de direcciones (*address register*) y el secuenciador que las incrementa (*incrementer*), seleccionan y retienen todas las direcciones de memoria y, cuando se lo requiere, generan direcciones secuenciales.
- Los registros de datos (*data registers*), que contienen los datos que pasan a la memoria o vienen desde la memoria.
- El decodificador de instrucciones (*instruction decoder*) y la lógica de control asociada (*associated control logic*).

En una instrucción de procesamiento de datos de un solo ciclo, se accede a dos operandos, que están en sendos registros. Se desplaza hacia la ALU, el valor en el bus B y se combina con el valor en el bus A, luego el resultado se

escribe en el banco de registros. El valor del contador de programa está en el registro de direcciones, desde donde se lo lleva al dispositivo "*incrementer*". Luego el valor incrementado se copia nuevamente en el registro r15 del banco de registros y también en el registro de dirección, para que posteriormente se lo use como dirección de la próxima búsqueda de código de operación.

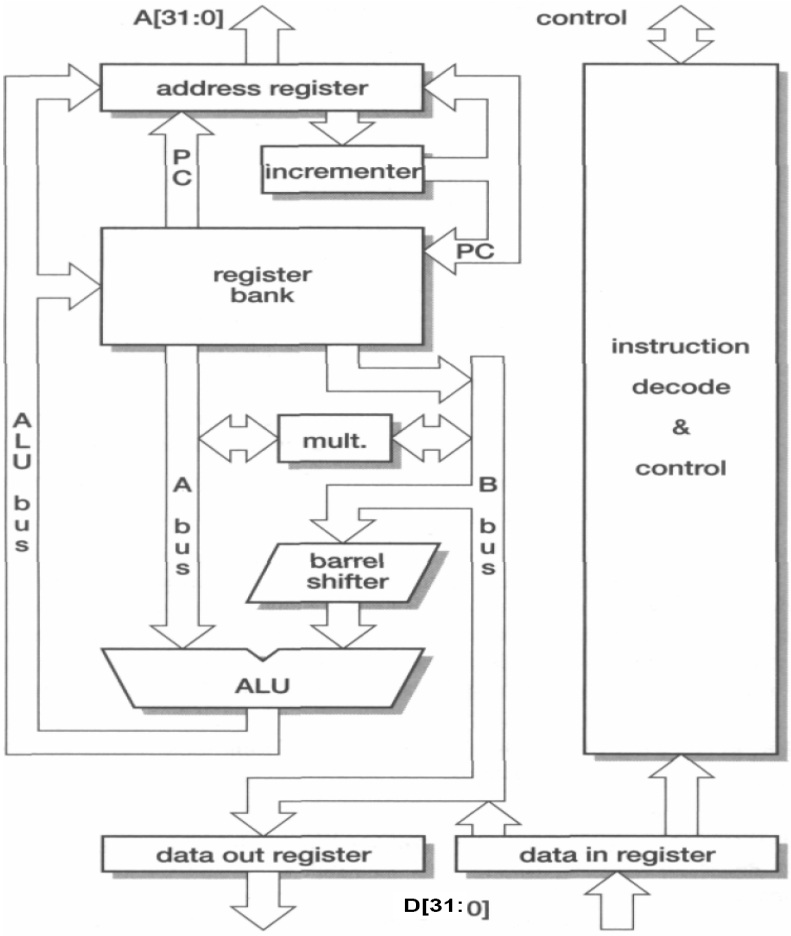


Figura 6, extraída de "ARM. System-on-Chip Architecture", Steve Furber.
Organización *pipeline* de tres etapas

Pipeline de tres etapas

Los procesadores ARM hasta el ARM7, emplean un *pipeline* simple de tres etapas. Éstas son:

- **Búsqueda de código de operación (*fetch*).** Se busca el código de operación en la memoria y se lo ubica en la instrucción *pipeline*.
- **Decodificación (*decode*).** Se decodifica la instrucción y se preparan las señales de control de recorrido del dato (*datapath*) para el próximo ciclo. En esta etapa se decodifica la instrucción pero no el recorrido del dato.
- **Ejecución (*execute*).** La instrucción "reconoce" ("*owns*") su recorrido de datos. Se lee el banco de registros, se desplaza un operando, la ALU genera un resultado y lo escribe en el registro

destino.

Cada una de estas etapas la pueden ocupar tres instrucciones a la vez, por lo tanto, el *hardware* de cada etapa tiene la capacidad de operaciones independientes.
Cuando el procesador está ejecutando instrucciones simples de procesamiento de datos, la técnica *pipeline* lo habilita para que complete una instrucción en cada ciclo de reloj. Completar una instrucción individual lleva tres ciclos de reloj, por lo tanto tiene tres ciclos de latencia, pero con esta técnica se logra una instrucción por ciclo.
En la siguiente figura se muestra la técnica *pipeline* de tres etapas, para instrucciones de un solo ciclo.

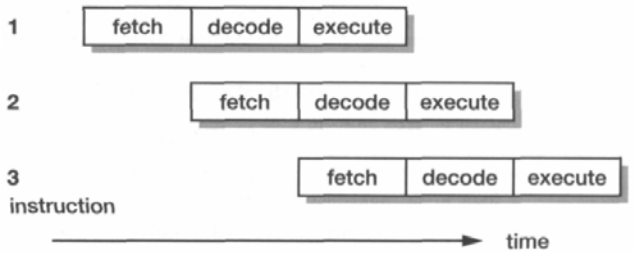


Figura 7, extraída de "ARM. System-on-Chip Architecture", Steve Furber.
Técnica *pipeline* de tres etapas para instrucciones ARM de un solo ciclo

El flujo es más irregular cuando se ejecutan instrucciones de múltiples ciclos. Esto se ilustra en la siguiente figura.

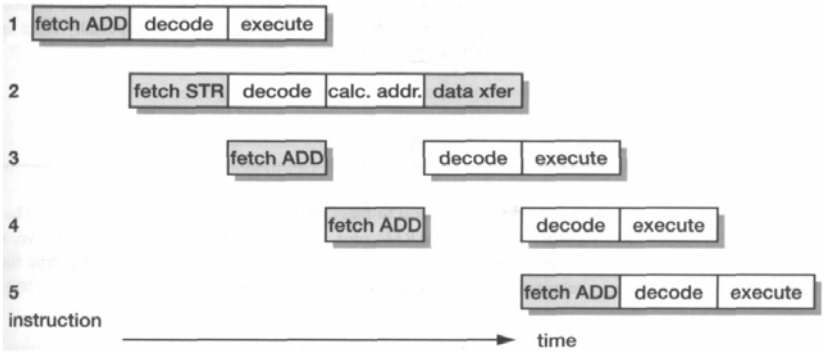


Figura 8, extraída de "ARM. System-on-Chip Architecture", Steve Furber.
Técnica *pipeline* de tres etapas para instrucciones ARM de múltiples ciclos.

Esta figura muestra una secuencia de instrucciones de adición, ADD, de un solo ciclo y una instrucción de almacenamiento de datos, STR (*store*), que ocurre después de la primera instrucción ADD. Se muestran, sobre fondo gris, los ciclos en los cuales se accede a la memoria principal, así se puede ver que la memoria se usa en cada ciclo. También el recorrido de los datos (*datapath*) se usa en cada ciclo y está involucrado en todos los ciclos de ejecución, cálculo de direcciones y transferencia de datos. La lógica de decodificación siempre genera las señales de control para el recorrido de los datos que se va a usar en el próximo ciclo. Además de los ciclos de decodificación explícitos también se genera el control para la transferencia de datos durante los ciclos de cálculo de la dirección de la instrucción STR.
Así, en la secuencia de esta instrucción, todas las partes del procesador están activas en cada ciclo y el factor limitante es la memoria, que define el número de ciclos que tomará la secuencia.
Una manera simple de ver los quiebres en la técnica *pipeline* ARM es observar que:

- Todas las instrucciones ocupan el camino de los datos por uno o más ciclos adyacentes.
- Por cada ciclo en que la instrucción ocupó el camino de los datos, en el ciclo inmediatamente anterior, lo ocupó la lógica de decodificación.
- Durante el primer ciclo de recorrido de datos, cada instrucción emite una búsqueda de código de operación para la siguiente instrucción, pero solo una.
- Las instrucciones de salto fluyen repentinamente y rellenan la instrucción *pipeline*.

Organización Pipeline de cinco etapas

La técnica *pipeline* de tres etapas que usa el corazón ARM hasta el ARM7 es buena en costo-eficiencia pero los requerimientos de alto rendimiento hicieron que se vuelva a considerar la organización de los procesadores.

El tiempo, T_{prog}, que se requiere para ejecutar un programa dado, se obtiene de la siguiente ecuación:

$$T_{prog} = \frac{N_{inst} \times CPI}{f_{clk}}$$

donde:

N_{inst}	es el número de instrucciones ARM ejecutadas durante el curso del programa
CPI	es el promedio de ciclos de reloj por instrucción
f_{clk}	es la frecuencia de reloj del procesador

Ya que N_{inst} es constante para un programa dado (compilado con un compilador dado, usando un determinado set de instrucciones, etc.) hay solo dos modos de aumentar el rendimiento:

- Aumentar la velocidad de reloj, f_{clk}. Esto requiere que se simplifique la lógica en cada etapa del *pipeline* y, además, que se aumente el número de *pipeline*.
- Reducir el promedio de número de ciclos de reloj por instrucción, **CPI**. Esto requiere que las instrucciones que ocupaban más que un ranura (*slot) pipeline* en el *pipeline* de tres etapas se vuelva a implementar de tal manera que ocupe pocas ranuras (*slots)*, o que se reduzcan las obstrucciones de *pipeline* causadas por la dependencia entre instrucciones o, una combinación de ambas.

Cuello de botella de memoria

El problema principal para reducir el CPI relativo a corazones de tres etapas está relacionado con el cuello de botella Von Neumann – un programa de computadora almacenado en una memoria única para instrucciones y datos tendrá su rendimiento limitado por el ancho de banda de la memoria disponible. Un corazón ARM de tres etapas, accede a la memoria en casi cada ciclo de reloj, ya sea para buscar un código de operación como para transferir datos. Unos pocos ciclos donde no se usa la memoria, redituarán en una pequeña ganancia del rendimiento. Para que un sistema de memoria adquiera un mejor promedio de ciclos de reloj por instrucción, CPI, tiene que entregar más de un valor en cada ciclo de reloj o dar más de 32 bits por ciclo desde una única memoria o tener memorias separadas para acceder a instrucciones y datos.
Como resultado de este análisis, los corazones de los ARM de alto rendimiento emplean *pipeline* de 5 etapas y tienen memorias separadas para instrucciones y para datos, o sea que son de arquitectura Harvard.

Si se parte la ejecución de las instrucciones en cinco componentes en vez de en solo tres, se reduce el trabajo máximo que tiene que completarse en un ciclo de reloj y en consecuencia se permite usar una mayor frecuencia de reloj (provista por otros componentes del sistema).
Además la memoria de instrucciones también se rediseñó para funcionar a esta mayor velocidad de reloj. Las memorias de instrucciones y de datos separadas (las que pueden ser memorias caché conectadas a una instrucción unificada y a la memoria de datos principal) permiten una reducción significativa del promedio de ciclos de reloj por instrucción del corazón del ARM.

El ARM9TDMI emplea una técnica *pipeline* típica de 5 etapas.

Densidad de código ARM y "*Thumb*"

Se llama densidad de código (*code density*) a la medida de cuánta memoria necesita un sistema embebido para contener instrucciones. Frecuentemente en los sistemas embebidos hay una limitación al tamaño de la memoria. Esto es especialmente cierto para los sistemas con la memoria dentro del chip, en los cuales la memoria normalmente ocupa más espacio en el chip que la propia CPU, por ejemplo la memoria *caché.*

La arquitectura del procesador ARM se basa en los principios RISCs aunque tiene aún mejor densidad de código que la mayoría de los procesadores RISCs. Sin embargo, su densidad de código todavía no alcanza a ser tan buena como la de varios de los procesadores CISCs.
Para las aplicaciones en que es primordial la importancia de la densidad del código, ARM incorporó un novedoso mecanismo llamado arquitectura "*Thumb*". El set de instrucciones *Thumb* es una forma comprimida a 16 bits del set de instrucciones ARM de 32 bits original y emplea *hardware* de descompresión dinámica en la instrucción *pipeline*, para descomprimir las instrucciones de 16 a 32 bits. Por supuesto, la etapa extra requerida para manejar instrucciones de 16 bits afecta el rendimiento.
La densidad del código *Thumb* es mejor que la alcanzada por la mayoría de los procesadores CISCs.

El set de instrucciones *Thumb* (sub set del de 32 bits) usado para obtener alta densidad de código en varios procesadores ARM utiliza, predominantemente, una arquitectura de dos direcciones.

Se puede llevar a cabo un nuevo ajuste en el número de bits que se requiere para almacenar una instrucción haciendo que el registro de destino coincida

con alguno de los registros fuente. El formato en lenguaje *Assembler* podría ser:

```
ADD     d,   s1          ;d := d + s1
```

La representación binaria, ahora reducida se muestra en la siguiente figura.

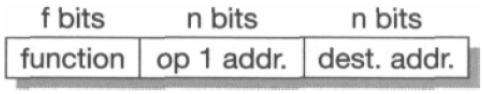


Figura 9, extraída de "ARM. System-on-Chip Architecture", Steve Furber.
Formato de instrucción de dos direcciones.

Características del set de instrucciones ARM

Todas las instrucciones ARM son de 32 bits (excepto las instrucciones comprimidas *Thumb* de 16 bits) y se alinean en bandas de 4 bytes en memoria. Las características más notables del set de instrucciones ARM son:

- La arquitectura carga-almacenamiento
- Instrucciones de procesamiento de datos de 3 direcciones (se especifican en forma independiente los dos registros con los operandos fuentes y el registro resultado)
- La ejecución condicional para cada instrucción
- La inclusión de instrucciones muy potentes de carga y almacenamiento de múltiples registros
- La habilidad de ejecutar una operación general de desplazamiento y una operación general ALU en una instrucción única que se ejecuta en un único ciclo de reloj
- Extensiones abiertas del set de instrucciones a través del set de instrucciones del coprocesador, incluyen el agregado, al modelo del programador, de nuevos registros y tipos de datos
- Arquitectura *Thumb*, representación de 16 bits del set de instrucciones, comprimida y muy compacta

Excepciones

La arquitectura ARM soporta un rango de interrupciones, *instrucciones indefinidas* y llamadas al supervisor, todas agrupadas bajo el encabezamiento general de **excepciones**. Generalmente se las usa para manejar eventos inesperados que ocurren durante la ejecución de un programa, tales como interrupciones o fallas de la memoria. En la arquitectura ARM este término también se usa para las interrupciones de *software*, `SWI`, las instrucciones indefinidas *traps* (que realmente no califican como "inesperadas") y las funciones de reset del sistema que

lógicamente aparecen antes de la ejecución de un programa (a pesar de que el procesador puede necesitar que se lo resetee nuevamente mientras está corriendo). Todos estos eventos están agrupados bajo el nombre "excepción" porque todos ellos usan el mismo mecanismo básico dentro del procesador.
Las excepciones ARM se pueden clasificar en tres grupos:

1. Excepciones generadas como efecto directo de la ejecución de una instrucción.
   Dentro de éstas están:

   - **Interrupciones de** *software, software interrupts*, SWI
     Se pueden usar para hacer llamadas al sistema operativo.

   - **Indefinidas**, *undefinided*
     Intento de ejecución de una instrucción indefinida.
     Incluye instrucciones del coprocesador cuando éste está ausente.

   - **Abortos de memoria**, *Memory Abort*
     Abortos de memoria antes de la pre búsqueda del código de operación.
     Instrucciones que son inválidas debido a fallas en la memoria ocurridas durante la búsqueda del código de operación.
     Se pueden usar para implementar protecciones de memoria o memoria virtual.

2. Excepciones generadas como un efecto colateral de una instrucción.

3. Excepciones generadas externamente, no relacionadas con el flujo de instrucciones.

   Caen en esta categoría:

   - *Reset*

   - FIQ (*Fast Interrupt Request*)
     Solicitud de interrupciones de alta prioridad

   - IRQ (*Interrupt Request*)
     Solicitud de interrupción

Cuando aparece una excepción, ARM completa la instrucción en curso, (excepto excepciones de reset que terminan inmediatamente la instrucción en curso) y luego sale de la secuencia de esa instrucción para manejar la excepción.

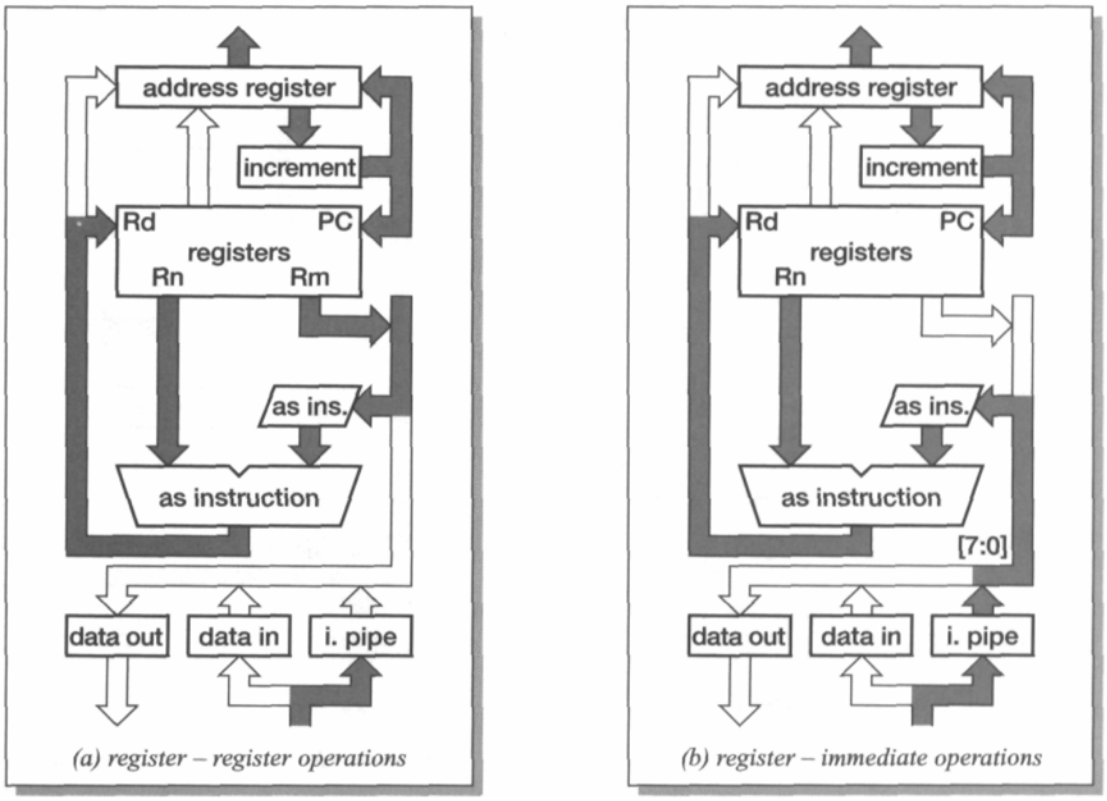
Ejecución de instrucciones ARM

Instrucciones de procesamiento de datos

Una instrucción que procesa datos requiere de dos operandos, uno de ellos

siempre es un registro y el otro es un segundo registro o un valor inmediato. El segundo operando pasa a través del registro de desplazamiento *barrel shifter*, donde sufre un desplazamiento, luego se combina con el primer operando en la ALU. Finalmente, el resultado de la ALU se escribe en el registro destino (y se puede actualizar el registro de código de condición).

Todas estas instrucciones tienen lugar en un único ciclo de reloj, como se exhibe en la siguiente figura. Nótese, también, cómo el valor del contador de programa, PC, en el registro de direcciones, se incrementa y se copia nuevamente, tanto en el registro de direcciones como en r15 del banco de registros. La próxima instrucción se carga al final del *pipeline* de instrucciones (*i. pipe*). Cuando se requiere el valor inmediato se lo extrae de la instrucción actual, al tope del *pipeline* de instrucciones. Solamente se usan los ocho bits finales, bits [7:0], de la instrucción para el procesamiento de instrucciones con el segundo operando dado con un valor inmediato.



i. pipe (*instruction pipeline*) = "*pipeline*" de instrucción.

Figura 10, extraída de "ARM. System-on-Chip Architecture", Steve Furber.
Actividad del camino de datos de las instrucciones de procesamiento de datos

Instrucciones de transferencia de datos

Una instrucción de transferencia de datos (carga o almacenamiento) calcula una dirección de memoria de una manera muy similar a la que lo hacen las instrucciones de procesamiento de datos. Se usa un registro como dirección base, al cual se le agrega (o se le substrae) un desplazamiento que puede ser otro registro o un valor inmediato. Se envía la dirección al registro de direcciones y tiene lugar un segundo ciclo de transferencia de datos. En lugar de dejar el trayecto de los datos (*datapath*) inactivo durante la transferencia de los datos, la ALU retiene las direcciones desde el primer

ciclo y está disponible por si se requiere usarlas para calcular una modificación de auto indexación del registro base. Si no se requiere la auto indexación, en el segundo ciclo no se escribe el valor calculado en el registro base.

En la siguiente figura se muestran los dos ciclos de la operación del trayecto de los datos de una instrucción de almacenamiento (*store*, STR) con un desplazamiento inmediato. Nótese como el valor incrementado del contador de programa, PC, se almacena en el banco de registros al finalizar el primer ciclo, así el registro de direcciones está libre para aceptar la dirección del dato transferido del segundo ciclo, luego, al finalizar el segundo ciclo, el contador de programa, PC, se carga nuevamente en el registro de direcciones para permitir que continúe la pre búsqueda del código de operación de la instrucción.

Nótese, que en esta etapa el valor enviado al registro de direcciones en un ciclo es el valor usado para el acceso a memoria en el ciclo próximo. En efecto, el registro de direcciones es un registro *pipeline* entre el trayecto de los datos del procesador y la memoria externa.

Cuando se lo desea, el registro de direcciones puede producir una dirección de memoria para el próximo ciclo, un poquito antes del final del ciclo actual. Esto puede permitir que funcionen con alto rendimiento varios dispositivos de memoria. Por ahora, veremos el registro de direcciones como un registro *pipeline* a la memoria.

Cuando la instrucción especifica el almacenamiento de un tipo de datos byte, el bloque de datos ***data.out*** extrae un byte desde el final del registro y lo repite cuatro veces a través del bus de datos de 32 bits. Luego, la lógica de control de la memoria externa, puede usar los dos bits del bus de direcciones del final para activar el byte apropiado dentro del sistema de memoria.

Las instrucciones de carga siguen un patrón similar excepto porque el dato se adquiere desde la memoria solo a través del registro ***data.in*** en el segundo ciclo y se necesita un tercer ciclo para transferir el dato desde allí al registro destino.

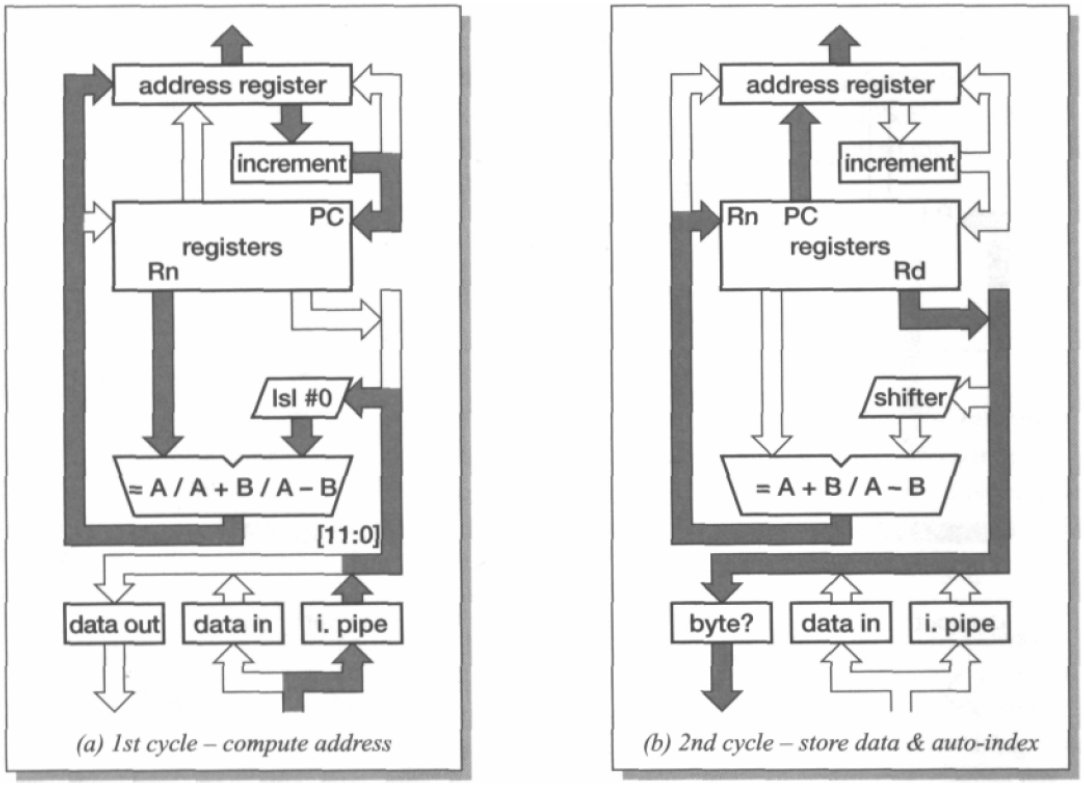


Figura 11, extraída de "ARM. System-on-Chip Architecture", Steve Furber.
Actividad del trayecto de los datos en STR (*store register*).

Instrucciones de salto

Las instrucciones de salto calculan la dirección destino en el primer ciclo, como se muestra en la siguiente figura. Se extrae un campo inmediato de 24 bits de la instrucción, luego se desplaza dos posiciones de bits hacia la izquierda para dar un desplazamiento alineado por palabras, que se agrega al contador de programa, PC. El resultado es emitido como una dirección de búsqueda de código de operación de una instrucción, y mientras se rellena el *pipeline* de instrucciones, si se lo requiere se copia la dirección de retorno en el registro *link*, r14, es decir, la instrucción es un salto con retorno.
El tercer ciclo, que se requiere para completar el rellenado del *pipeline*, también se usa para hacer una conexión pequeña al valor almacenado en el registro *link* a fin de que éste apunte directamente a la instrucción que sigue al salto. Esto es necesario porque r15 contiene PC+8, mientras que la dirección de la próxima instrucción es PC+4.

Otras instrucciones ARMs funcionan de manera similar a éstas descriptas.

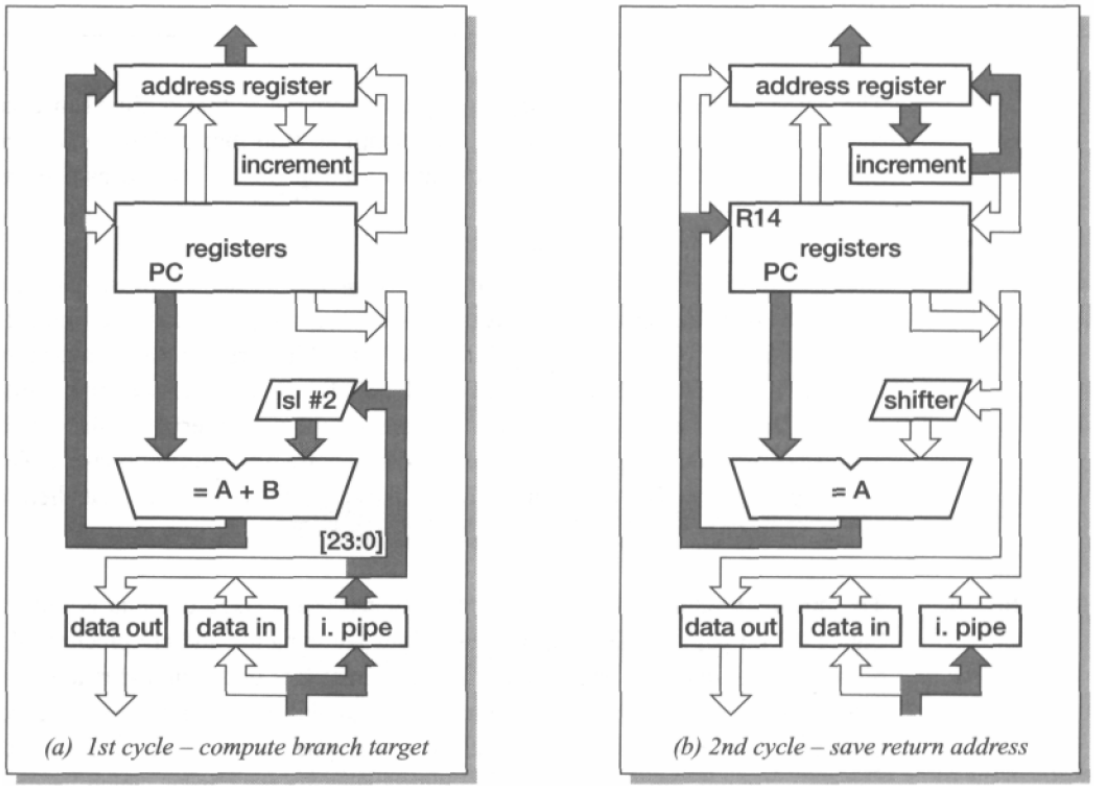


Figura 12, extraída de "ARM. System-on-Chip Architecture", Steve Furber.
Los primeros dos (o tres) ciclos de una instrucción de salto

El "barrel shifter"

La arquitectura ARM soporta instrucciones que ejecutan operaciones de desplazamiento en serie con una operación de la Unidad Aritmético–Lógica, ALU. De esta manera se vuelve crítica la eficacia del "*shifter*", ya que el tiempo de desplazamiento contribuye directamente al tiempo de los ciclos del trayecto de los datos, como se muestra en el siguiente diagrama temporal.

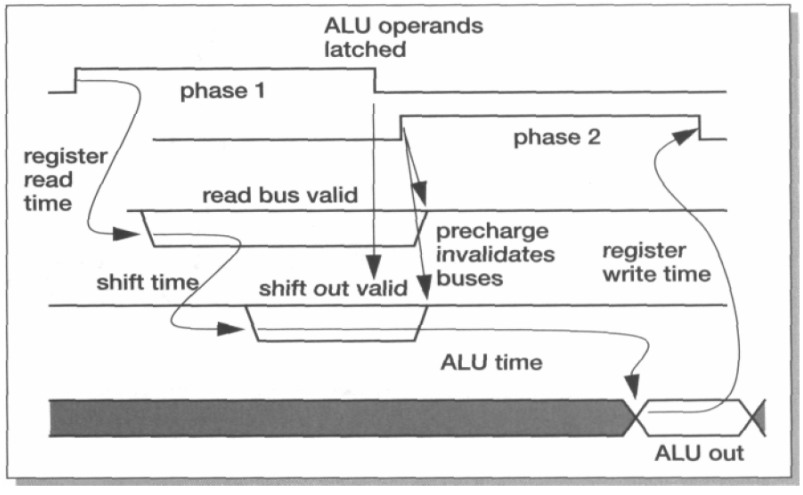


Figura 13, extraída de "ARM. System-on-Chip Architecture", Steve Furber.
Temporización del trayecto de los datos (*pipeline* de 3 etapas).

A fin de minimizar el retraso a través del "*shifter*", se usa una diagonal de matriz de interruptores para llevar cada entrada a la salida apropiada. El principio del travesaño de matriz de interruptores se ilustra en la siguiente figura para una matriz de 4 x 4 (el procesador ARM usa una matriz de 32 x 32). Cada entrada se conecta a cada salida a través de un interruptor. Si se usa lógica precargada dinámica, como la que se utiliza en el trayecto de los datos de ARM, se puede implementar cada interruptor como un único transistor NMOS.
Se implementan las funciones de desplazamiento conectando los interruptores a lo largo de la diagonal a una entrada de control común.

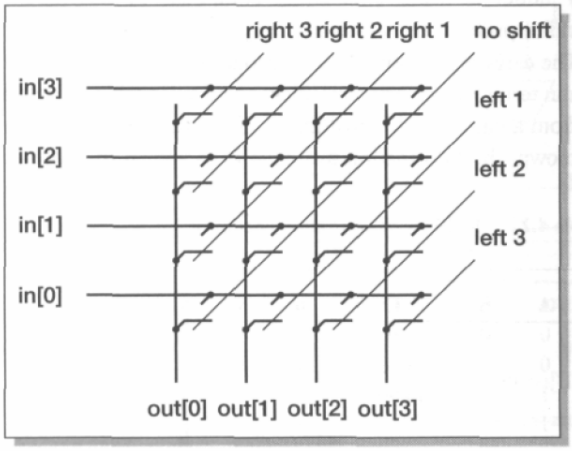


Figura 14, extraída de "ARM. System-on-Chip Architecture", Steve Furber.
El principio de la diagonal ("travesaño") de matriz de interruptores del "*barrel shifter*".

Modos de funcionamiento del procesador ARM

Tiene varios modos básicos de funcionamiento. El cambio de modo se puede hacer bajo control del *software* o puede ser causado por interrupciones externas.
La mayoría de los programas de aplicaciones se ejecutan en modo usuario. Mientras el procesador está en modo usuario, el programa que se está ejecutando no puede acceder a recursos protegidos del sistema o cambiar de modo. Esto permite escribir un sistema operativo adecuado para controlar el uso de los recursos del sistema.

Modo usuario, *USR, User Mode*

Sin privilegios, bajo el cual corren la mayoría de las aplicaciones. Modo de ejecución normal de los programas. Los programadores a nivel de usuario deben tener en cuenta que solo pueden acceder a las funciones a nivel del sistema a través de llamadas del supervisor. Generalmente estas funciones incluyen cualquier acceso a los registros periféricos del *hardware* y operaciones frecuentes, tales como entrada y salida de caracteres.
Los restantes modos son privilegiados y se llaman excepciones.

Modos "excepciones"

- **Modo supervisor,** *SVC Supervisor Mode,* entra en este modo con el reset o cuando se ejecuta una interrupción de *software, SWI, Software Interrupt*. Modo protegido para sistema operativo. El mecanismo de protección asegura que el código del usuario no

pueda obtener los privilegios del supervisor sin los controles apropiados a fin de asegurar que este código no intente operaciones ilegales.

- **Modo instrucción indefinida,** *UND, Undefined Mode*, se usa este modo para manejar instrucciones indefinidas. Soporta emulación de *software* de coprocesadores de *hardware*.

- **Modo: interrupciones de alta prioridad,** *FIQ Fast Interrupt Request Mode,* entra en este modo cuando aparece una interrupción de alta prioridad (*fast*). Soporta un proceso de transferencia de datos o canal de alta velocidad

- **Modo: interrupciones de baja prioridad o normales,** *IRQ, Normal interrupt request Mode*, entra en este modo cuando aparece una interrupción de baja prioridad (normal). Usado para manejar interrupciones de propósitos generales.

- **Modo aborto,** *ABT, Abort Mode*, se usa para manejar violaciones de acceso a memoria. Aborto por una fallo de memoria al leer una instrucción (*instruction fetch memory fault*) o al acceder a un dato (*data abort*)

- **Modo sistema,** *SYS, System Mode*, modo privilegiado que usa los mismos registros que el modo usuario. Corre tareas privilegiadas del sistema operativo (arquitectura ARM versión 4 y superiores)

Se entra en ellos cuando aparece una excepción específica. Tienen acceso a todos los recursos y pueden cambiar libremente de modo. Cada uno de ellos tiene registros adicionales para evitar que cuando ocurre una excepción se arruinen los registros del modo usuario.

Al modo sistema, presente solamente a partir de la arquitectura ARM 4, no se entra con cualquier excepción y participa de los mismos registros que el modo usuario. Sin embargo, es un modo privilegiado y, por lo tanto, no está sujeto a las restricciones del modo usuario. Se entra en él para usar tareas del sistema operativo que necesitan acceder a recursos de éste, pero deseando evitar el uso de registros adicionales asociados con el modo de excepción. Así se evita que el estado de la tarea no se corrompa por la aparición de cualquier excepción.

Exception	Mode	Vector address
Reset	SVC	0x00000000
Undefined instruction	UND	0x00000004
Software interrupt (SWI)	SVC	0x00000008
Prefetch abort (instruction fetch memory fault)	Abort	0x0000000C
Data abort (data access memory fault)	Abort	0x00000010
IRQ (normal interrupt)	IRQ	0x00000018
FIQ (fast interrupt)	FIQ	0x0000001C

Figura 15, extraída de "ARM. System-on-Chip Architecture", Steve Furber.
Direcciones de los vectores de excepción

El modelo de

programación ARM

El set de instrucciones del procesador define las operaciones que el programador puede usar para cambiar el estado del sistema. Normalmente este estado comprende los valores de los datos en los registros visibles del procesador y de la memoria del sistema. Cada instrucción se puede ver como una transformación definida desde el estado anterior de la instrucción que se está ejecutando hacia el estado posterior, cuando se complete la instrucción. Nótese que, a pesar de que un procesador tendrá típicamente muchos registros invisibles involucrados en la ejecución de la instrucción, el valor de dichos registros antes y después de que la instrucción se ejecute no es significativo, solamente el valor en los registros visibles tiene algún significado.

Registros del procesador

ARM tiene 37 registros, 31 de ellos son registros de 32 bits de propósitos generales, los 6 restantes son registros de estado. Estos registros también son de 32 bits pero solo necesitan implementarse 12 bits.
Los registros se ubican en bancos parcialmente superpuestos, con un banco diferente para cada modo del procesador.
El conjunto de registros a los cuales el usuario tiene acceso está determinado por el modo de funcionamiento del procesador.
En cualquier momento se puede acceder por *software* a 16 registros, que van de **`r0`** a **`r15`**, o sea que 16 registros son visibles. Los otros registros se usan para acelerar el proceso de excepción. Todos los registros especificados en una instrucción ARM se pueden "direccional" con cualquiera de los 16 registros visibles.
Todos los códigos del modo usuario usan el banco principal de 16 registros Son los registros del modo usuario. Este modo es diferente de los otros modos porque no tiene privilegios, esto significa:

- Es el único modo que no puede conmutar en otro modo del procesador sin generar una excepción.
- Tiene menos acceso a la memoria del sistema y funciones del coprocesador que los modos privilegiados.

El registro **`r15`** es el contador de programa (PC, *Program Counter*).

El registro **`r14`** (LR, *Link Register*) además de usarse como registro de propósitos generales se usa para guardar la dirección de la siguiente instrucción cuando se lleva a cabo una instrucción de salto y *link* (instrucción que se ejecuta cuando se llama a una subrutina).

Otro registro visible para el usuario es el registro de estado del programa (**CPSR**, *Current Program Status Register*). En general, a los registros que guardan los estados del programa, se los llama, registros de estado del programa (**PSR**, *Program Status Register*). En total hay 6 de estos registros.

Cuando se escriben programas a nivel de usuario, solamente se deben considerar los 15 registros de 32 bits, de propósitos generales, el contador de programa y el registro de estado actual del programa, o sea:

- **r0 … r13**
- **r14** (LR, *Link Register*)
- **r15** (PC, *Program Counter*)
- **CPSR** (*Current Program Status Register*)

Solamente los usan los programas a nivel del sistema.
Los registros pueden cambiar su significado de acuerdo con el modo de funcionamiento del procesador.
En la siguiente figura se exhiben los registros visibles del procesador ARM.

fiq : *Fast interrupt request*
irq : *Normal interrupt request*
svc : *Supervisor*
CPSR : *Current Program Status Register*
LR : *Link Register*
PC : *Program Counter*
abt : *Abort*
und : *Undefined*

Figura 16. Registros visibles del ARM.

Registro contador de programa, PC

Program Counter Register
Una consecuencia del modelo de ejecución *pipeline* es que el PC tiene que apuntar más adelante de la instrucción en curso. Una vez que se busca el

código de operación de una instrucción el contador de programa, PC, tiene que apuntar 8 bytes (dos instrucciones) más adelante de la instrucción en curso.

Registro de estado actual del programa, CPSR

Current Program Status Register
Este registro se usa en programas a nivel de usuario para almacenar los bits de condición del código. Estos bits se usan, por ejemplo, para registrar el resultado de una operación de comparación y controlar si tendrá o no lugar un salto condicional. Normalmente, a los programadores a nivel de usuario no les concierne cómo está configurado este registro, de todos modos se lo exhibe en la siguiente figura.

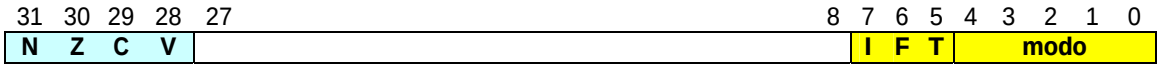


Figura 17. Formato del registro de estado actual del programa, CPSR (*Current Program Status Register)*

Indicadores de condiciones del código

Condition code flags
Son los 4 bits de la parte alta del registro. El procesador puede modificar estos indicadores del CPSR (*Current Program Status Register*) como resultado de una operación aritmética o lógica y todas las instrucciones pueden verificarlos para determinar si se debe o no ejecutar la instrucción.
Tienen el siguiente significado:

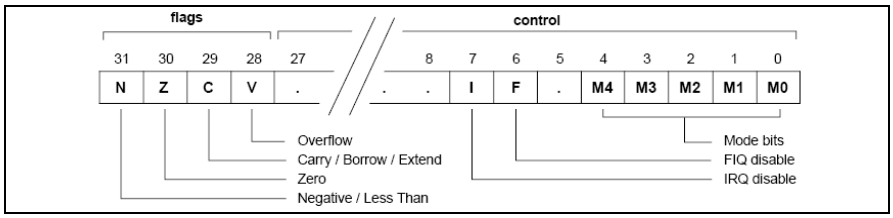
- N: *Negative*; negativo, la última operación de la ALU que cambió los indicadores produjo un resultado negativo (el bit más significativo del resultado de 32 bits fue un uno).
- Z: *Zero,* cero, la última operación de la ALU que cambió los indicadores produjo un resultado cero (cada bit de los 32 bits del resultado fue cero).
- C: *Carry*. Acarreo, la última operación de la ALU que cambió los indicadores generó un acarreo de salida, tanto como resultado de una operación aritmética en la ALU como por un desplazamiento.
- V: *Overflow*, desborde, la última operación de la ALU que cambió los indicadores generó un desborde en el bit de signo.

Bits remanentes

Son los bits de la parte baja del registro. Están protegidos contra cambios a nivel del programa del usuario. Tienen el siguiente significado:

- "I", *Normal Interrupt Disable*, inhabilitación de interrupciones normales, cuando I=1
- "F", *Fast Interrupt Disable*, inhabilitación de interrupciones de mayor prioridad, cuando F=1
- "T", *Instruction Set*, set comprimido de instrucciones de 16 bits

- *Mode*, modo, control del modo del procesador



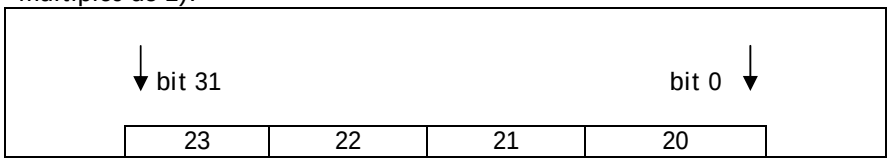
Se llama, en forma colectiva, **bits de control** a los 28 bits inferiores de un PSR *(Program Status Register*). Estos bits de control incluyen a los bits **`I, F`** y **`M4…M0.`** Estos bits se modificarán cuando ocurra una excepción. Además, cuando el procesador esté en algún modo privilegiado, se los podrá modificar por *software*.
Los bits no usados en el PSR están reservados para futuros procesadores, por este motivo se los debe preservar cuando se quiera modificar algún bit de condición o de control.
Los cinco últimos bits que van de **`M4 a M0`** controlan el modo en el cual funcionará el procesador. No se han definido todas las combinaciones posibles de los bits de modos de manera que no son todas válidas. En la siguiente tabla se muestra la interpretación de estos bits para los modos de funcionamiento que hemos vistos.

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

Tabla 1 extraída del libro " ARM Architectura Reference Manual. ARM Limited". Interpretación de los modos del procesador (bits 4…0 de un *Program Status Register*, PSR) y conjunto de registros accesibles para cada modo

El sistema de memoria

Además del estado del registro del procesador, un sistema ARM tiene estado de la memoria. La memoria se puede ver como un arreglo lineal de bytes numerados desde cero hasta $2^{32} - 1$. Los datos pueden ser de bytes (8 bits), de medias palabras (16 bits) o palabras (32 bits). Las palabras están siempre alineadas en bandas de 4 bytes (esto es, los 2 bits de direcciones menos significativos son cero, porque son múltiplos de 4) y las medias palabras están alineadas en bandas de bytes pares (porque son múltiplos de 2).



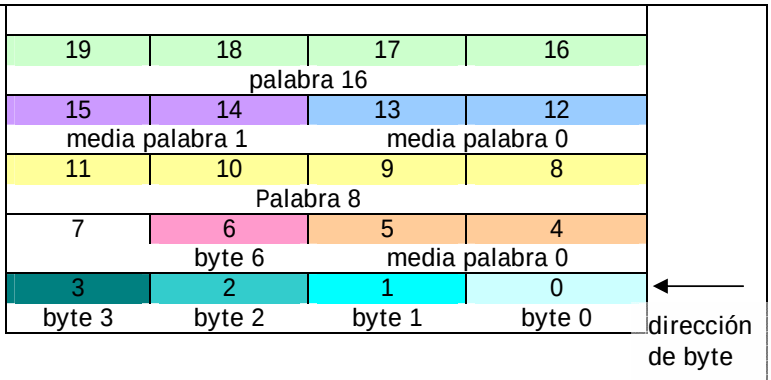


Figura 18. Organización de la memoria ARM, *little endian*.

La figura anterior muestra una pequeña área de memoria donde la posición de cada byte tiene un único número. Un byte puede ocupar cualquiera de estas posiciones de memoria. El dato de tamaño de una palabra debe ocupar un grupo de posiciones de cuatro bytes que comienzan en una dirección que es un múltiplo de cuatro y tiene sus cuatro bits menos significativos en **0**. En la figura se muestran algunos ejemplos. Las medias palabras ocupan posiciones de dos bytes comenzando en direcciones pares.

Esta es la organización de memoria *little-endian* usada por Intel y por ARM. Algunos ARM se puede configurar para trabajar como *big-endian*, que es la configuración adoptada por Motorota y por los protocolos TCP, entre otros, donde los bytes se escriben en el orden natural en que se los lee.

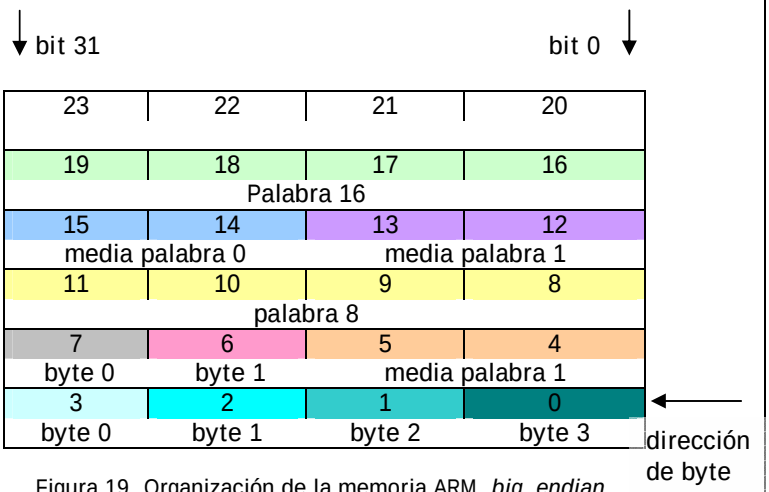


Figura 19. Organización de la memoria ARM, *big endian*.

En la forma *big endian*, al tener primero el byte de mayor peso, se puede saber rápidamente si el número es positivo o negativo sólo comprobando el estado del bit más significativo del primer byte (recordemos que el signo se almacena en el bit más significativo) y sin necesidad de saber la longitud del número. Esta forma de representación coincide con el orden en que se escriben los números, de modo que las rutinas de conversión entre sistemas de numeración son más eficientes que si se realizaran en *little endian*.

Nota:

Como curiosidad, el nombre *big endian* y *little endian* se tomó irónicamente

de "Los viajes de Gulliver", de Jonathan Swift, novela en la que los habitantes de los imperios de Lilliput y Blefuscu libran una encarnizada guerra por una disputa sobre si los huevos hervidos debería empezar a comerse abriéndolo por su extremo pequeño o por su extremo grande.

Manejo de las excepciones ARM

El modo general en que se las maneja es el mismo en todos los casos, excepto para el reset:

1. Se finaliza la ejecución de la instrucción en curso.
2. Se entra en el estado ARM. El modo de operación del procesador se cambia al modo apropiado de la excepción.
3. El estado actual del PC se guarda copiándolo en r14.exc y el CPSR en el SPSR.exc (donde exc representa el tipo de excepción al que se ha entrado).
4. Se inhabilitan las excepciones IRQ y si la fuente de la excepción era una FIQ también éstas se inhabilitan.
5. Se guarda la dirección de retorno en el registro *link*, LR_<mode>.
6. El PC se fuerza a un valor entre 0x00 y 0x1C que corresponde a una dirección de la tabla de vectores. El valor particular depende del tipo de excepción.

La instrucción que está en la posición en la que se forzó al PC (la dirección del vector) usualmente contendrá un salto a una rutina de atención de la excepción conocida por su nombre en inglés: *handler*. El *handler* de la excepción usará r13_exc para guardar varios registros del usuario y usar como registros de trabajo. Este registro normalmente está inicializado para apuntar a una pila dedicada en la memoria.

Para retornar:

1. Se recupera el CPSR del SPSR_<*mode*>
2. Se recupera el CPSR del SPSR_<*mode*>

Esto solamente se puede realizar en el estado ARM. El retorno al programa del usuario se lleva a cabo restaurando los registros del usuario y luego usando una instrucción para restaurar automáticamente el PC y el CPSR. Esto puede involucrar varios ajustes de los valores guardados del PC en r14_exc para compensar el estado de *pipeline* cuando apareció la excepción.

Cuando se produce un reset:

1. Se activa el modo supervisor
2. Se pasa al estado ARM
3. Se inhabilitan las excepciones IRQ y FIQ
4. Se carga en el PC la dirección del vector del reset, 0x00000000.

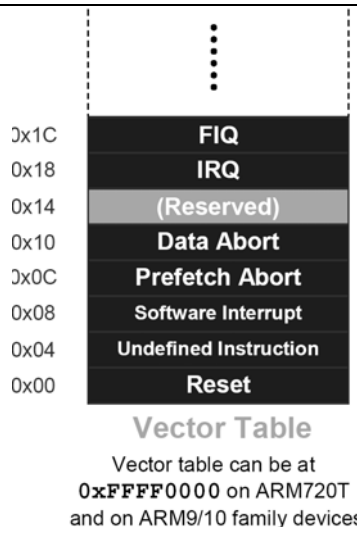


Figura 20, extraída de ARM Architectura Reference Manual. ARM Limited.
Direcciones de los vectores de excepciones

Sistema de entrada/salida

El ARM maneja periféricos de E/S (tales como controladores de disco, interfaces de red, etc.) como dispositivos "mapeados" como memoria, con soporte de interrupciones. Los registros internos de estos dispositivos aparecen como posiciones direccionables dentro del mapa de memoria del ARM y se pueden leer y escribir usando las mismas instrucciones (carga-almacenamiento) como cualquier otra posición de memoria.
Los periféricos puede llamar la atención del procesador haciendo un pedido de interrupción usando:

- La interrupción normal (IRQ, *Interrupt request*)
- La interrupción de alta prioridad (FIQ, *Fast Interrupt Request*)

Ambas entradas de interrupción son sensibles a nivel y enmascarables. Normalmente la mayoría de las fuentes de interrupción comparten la entrada IRQ.
Algunos sistemas pueden incluir *hardware* externo para acceso directo a memoria (DMA, *Direct Memory Access*) para que el procesador maneje el tráfico de E/S de banda ancha.

Un dispositivo periférico como, por ejemplo, el controlador de línea serie, contiene un número determinado de registros. En un sistema "mapeado" como memoria cada uno de estos registros aparecen como posiciones de memoria en una dirección particular (una alternativa es una organización del sistema con funciones de E/S en un espacio de direcciones separado del de los dispositivos de memoria). Un controlador de línea serie puede tener un conjunto de registros como los siguientes:

- Un registro de transmisión de datos (de escritura solamente), los datos que se escriban en este registros se enviarán por la línea serie.

- Un registro de recepción de datos (de lectura solamente), este es el destino de los datos que llegan por línea serie.
- Un registro de control (lectura y escritura), este registro "setea" la velocidad de los datos y controla la señal de solicitud de envío (*request to send*) y otras señales similares.
- Un registro de habilitación de interrupciones (de lectura y escritura), este registro controla qué eventos del *hardware* generarán una interrupción.
- Un registro de estado (de lectura solamente), este registro indica cuándo hay un dato disponible para leer, cuándo el *buffer* de escritura está vacío, etc.

Para recibir los datos el *software* debe inicializar apropiadamente al dispositivo, usualmente para generar una interrupción cuando hay un dato disponible o cuándo se detectó una condición de error. Luego, la rutina de atención de la interrupción debe copiar el dato en un *buffer* y verificar las condiciones de error para saber cuál es la que lo produjo y solucionarlo si es posible.

Programación ARM en Assembler

Características de las Instrucciones de procesamiento de datos

Algunas reglas aplicables a las instrucciones de procesamiento de datos:

- Todos los operandos son de 32 bits y están en registros o se especifican literalmente en la misma instrucción.
- Si hay un resultado es de 32 bits y está en un registro. Como excepción está la variante ARM "M" del set de instrucciones que contempla 4 instrucciones que ejecutan:

  32 x 32 → 64 bits

  32 x 32 + 64 → 64 bits

  operaciones de multiplicación **long** que producen un resultado de 64 bits. La segunda multiplica y acumula.
- Cada registro operando y el registro resultado se especifican en forma independiente en la instrucción. ARM usa el formato de 3 direcciones para estas instrucciones.
- **"Seteo" de bits de condición**

  Solamente las operaciones de comparación modifican los indicadores de condición (*flags*), en este caso no hay opciones pero para todas las otras instrucciones de procesamiento de datos se

debe tener en cuenta que los indicadores de condición no se ven afectados a menos que se lo requiera específicamente.
A nivel de lenguaje Assembler este requerimiento se indica agregando una **s** al código de operación, la letra viene por *Set condition codes*. Cualquier instrucción de procesamiento de datos puede setear los códigos de condición (**N, Z, C** y **V** en el registro **CPSR**) si el programador lo desea.

- **Operandos**

  1. **Registros**

     Una instrucción típica de procesamiento de datos ARM se escribe en Assembler así:

```
ADD   r0, r1, r2              ; r0 := r1 + r2
```

     El punto y coma indica que todo lo que haya a la derecha de éste es un comentario y el *Assembler* lo va a ignorar. Es imprescindible incluir los comentarios en el código fuente en *Assembler* para hacerlo fácilmente entendible. El comentario anterior lo puse para conocer la instrucción pero cuando se incluya un comentario no es para repetir lo que hace la instrucción sino para aclara el objetivo que tuvo en el programa en particular.
     Este simple ejemplo toma los valores de dos registros, **r1** y **r2**, los adiciona y ubica el resultado en un tercer registro, **r0**. Tanto los registros fuente como el registro destino son de 32 bits y se los puede considerar enteros sin signo o enteros signados en complemento a dos.
     Nótese que al escribir el código fuente en lenguaje *Assembler* hay que tener cuidado de escribir los operandos en el orden correcto, primero el registro resultado luego el primer operando y por último el segundo operando (a pesar de que para las operaciones conmutativas no es significativo el orden del primero y del segundo operando).
     Cuando se ejecuta esta instrucción el único cambio en el estado del sistema es el valor del registro destino **r0,** a menos que se quiera alterar los indicadores ya que la adición puede producir un acarreo o, en el caso de valores signados en complemento a dos, se puede producir un desborde interno en el bit de signo (*overflow*). Según vimos en este caso se agrega una **s** al código de operación.

  2. **Inmediatos**

     Si, en lugar de sumar el contenido de dos registros, simplemente deseamos adicionarle una constante a un registros podemos reemplazar el segundo operando fuente por un valor inmediato, que literalmente es una constante precedida por el signo **#.**

```
ADD    r3, r3, #1     ; r3 := r3 + 1
AND    r8, r7, #&ff   ; r8 := r7(7:0)
```

     El primer ejemplo sirve también para ver que a pesar de que el formato de tres direcciones permite que los operandos, fuente y destino, se especifiquen separadamente no se requiere que los registros sean distintos.
     El segundo ejemplo muestra que el valor inmediato se puede

especificar en notación hexadecimal (base 16) poniendo **&** después de #. También admite la sintaxis 0xff.

3. Registros con una condición

Un tercer modo se especificar una operación de datos es similar a la primera pero permite que el segundo operando se supedite a una operación de desplazamiento antes de combinarlo con el primer operando. Por ejemplo:

```
ADD   r3, r2, r1, LSL #3      ;r3 := r2 + 8 x r1
```

Nótese que todavía esto es una simple instrucción ARM, ejecutada en un único ciclo de reloj. La mayoría de los procesadores ofrecen operaciones de desplazamiento como instrucciones separadas pero ARM las combina con una operación general ALU en una única instrucción.
Aquí **LSL** indica *logical shift left by the specified number of bits*, desplazamiento lógico a la izquierda en un número de bits especificado, que en este ejemplo es **3**. Se puede especificar cualquier número entre **0** y **31**, especificar **0** es equivalente a omitir el desplazamiento. Como se mencionó el símbolo # indica una cantidad inmediata, o sea una constante. Más adelante se especifican otras condiciones.

Instrucciones

▪ Operaciones aritméticas (elementales)

Estas instrucciones ejecutan aritmética binaria (adición, substracción y substracción inversa, que es una substracción con los operandos en orden inverso) con operandos de 32 bits. Los operandos pueden ser enteros sin signo o enteros signados en complemento a dos. Cuando se lo requiere se afecta el bit de acarreo interno que corresponde al valor actual del bit C en el registro **CPSR**.

```
ADD         r0,  r1,  r2          ; r0 := r1 + r2
ADC         r0,  r1,  r2          ; r0 := r1 + r2 + C
SUB         r0,  r1,  r2          ; r0 := r1 – r2
SBC         r0,  r1,  r2          ; r0 := r1 – r2 + C – 1
RSB         r0,  r1,  r2          ; r0 := r2 – r1
RSC         r0,  r1,  r2          ; r0 := r2 + r1 + C – 1
```

ADD adición
ADC adición con acarreo
SUB substracción
SBC substracción con acarreo
RSB substracción inversa ¿Por qué inversa? Esta instrucción se usa cuando se desea afectar de alguna condición al registro r1 (1er. operando) en lugar de al registro r2 (2do. operando). Como veremos más adelante, sólo se pueden especificar condiciones para el último operando escrito. Como la substracción no es una operación conmutativa no tendríamos una forma sencilla de solucionar el problema. ARM tomará, entonces, la condición, sobre el primer operando, éste será el que pasará por el *barrel shifter*:

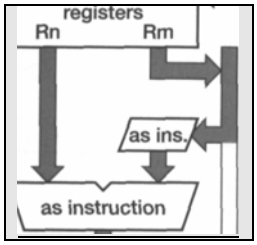


Figura 21, parte de ella extraída de "ARM. System-on-Chip Architecture", S. Furber
Rn sería **r2** y **Rm** sería **r1**.

RSC substracción inversa con acarreo

▪ Operaciones lógicas de bit

Estas instrucciones ejecutan las especificadas operaciones lógicas Booleanas sobre cada par de bits de los operandos de entrada, así, en el primer caso *r0[i] := r1[i] AND r2[i]* para cada valor de *i* desde 0 a 31 inclusive, donde *r0[i]* es el bit i-ésimo de *r0*.

```
AND          r0,  r1,  r2          ; r0 := r1 and r2
ORR          r0,  r1,  r2          ; r0 := r1 or  r2
EOR          r0,  r1,  r2          ; r0 := r1 xor r2
BIC          r0,  r1,  r2          ; r0 := r1 and not r2
```

AND AND
ORR OR
EOR XOR
BIC *bit clear*, (limpia bit) cada **1** en el segundo operando pone en **0** el bit correspondiente del primer operando.
Not, en el comentario de la operación **BIC** significa que invierte cada bit en el siguiente operando.

Podemos encontrar estas operaciones a nivel de compuertas *hardware*.

▪ Operaciones de movimiento entre registros

Estas instrucciones ignoran el primer operando, que se omite en el formato del lenguaje Assembler y simplemente mueven el segundo operando al destino. Permiten invertir primero los bits antes de moverlos.

```
MOV          r0,  r2               ; r0 := r2
MVN          r0,  r2               ; r0 := not  r2
```

MOV copia el contenido de **r2** en el contenido de **r0**
MVN *move negated,* mueve el contenido de **r2** complementado a **1** a **r0**, o sea que deja en el registro resultado el valor obtenido invirtiendo cada bit del operando fuente.

▪ Operaciones de comparación

Estas operaciones no producen un resultado (por eso hay un operando menos en la instrucción). Se usan para alterar los bits de condiciones (**N,**

Z, C y **V**) en el registro **CPSR** de acuerdo con la operación y posteriormente según el valor de un determinado bit de condición proceder o no a ejecutar un salto condicional.

```
CMP         r1,  r2          ; set cc en r1 – r2
CMN         r1,  r2          ; set cc en r1 +  r2
TST         r1,  r2          ; set cc en r1 and r2
TEQ         r1,  r2          ; set cc em r1 xor r2
```

set cc Quiere decir que afecta los indicadores *(condition code, cc)*

CMP *compare*, compara **r1** con **r2**, es SUB pero no se escribe el resultado, a **r1** le resta **r2.** Modifica los indicadores.

CMN *compare negated*, es ADD pero no se escribe el resultado, a **r1** le suma **r2**. Compara el negativo de un valor aritmético con el valor de un registro. Modifica los indicadores.

TST *test*, AND pero no se escribe el resultado. Modifica los indicadores pero no afecta al indicador de desborde (*overflow*). Se puede usar para determinar si son **0** varios bits de un registro o para saber si al menos un bit de un registro es **1**.

TEQ *test equal*, XOR pero no se escribe el resultado. Modifica los indicadores pero no afecta al indicador de desborde (*overflow*). Se puede usar para determinar si dos valores tienen el mismo signo.

▪ Operaciones de desplazamiento (*Barrel shifter operations*)

El corazón del ARM contiene al *barrel shifter*. Sus datos de entrada son:

- El tipo de desplazamiento o rotación que tiene que ejecutar
- El valor sobre el cual tiene que operar
- La cantidad de bits que va a desplazar o rotar

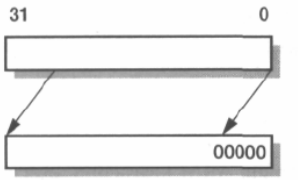
Las instrucciones ARM usan el *barrel shifter* para ejecutar una sola instrucción lo que normalmente requeriría instrucciones complejas.

El número de bits en que se desplazará el segundo operando se puede expresar en forma inmediata y también en un registro. Por ejemplo:

```
ADD   r5, r5, r3, LSL r2          ; r5 := r5 + r3 x 2^r2
```

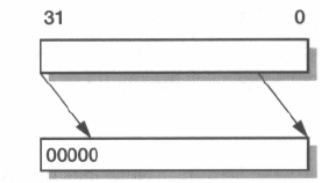
Esta es una instrucción de 4 direcciones. Solamente son significativos los últimos 8 bits del registro **r2**, pero ya que el desplazamiento por más de 32 bits no es muy útil para la mayoría de las aplicaciones esta limitación no es importante.
Estas instrucciones no llevan mucho tiempo a menos que la cantidad de bits a desplazar o rotar se indique en un registro. Para que se complete la instrucción en este caso se necesita un ciclo extra.

LSL desplazamiento lógico a la izquierda de **0** a **31** posiciones. Se carga **0** en los bits menos significativos de la palabra.
LSL de **n** bits significa multiplicar por $\mathbf{2^n}$



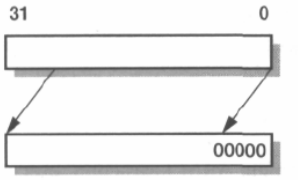
LSL #5

LSR desplazamiento lógico a la derecha de **0** a **31** posiciones. Se carga **0** en los bits más significativos de la palabra.
LSR de **n** bits significa dividir por 2^n (división sin signo)



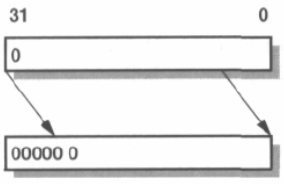
LSR #5

ASL desplazamiento aritmético a la izquierda, es sinónimo de **LSL.**
ASL de **n** bits significa multiplicar por 2^n

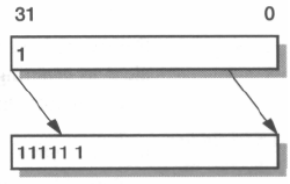


ASL #5

ASR Desplazamiento aritmético a la derecha de **0** a **31** posiciones. Si el operando era positivo se cargan **0** en los bits más significativos de la palabra. Si el operando era negativo se cargan **1** en los bits más significativos de la palabra. Esto es para mantener el signo.
Significa división con signo. Se realimenta el valor del bit más significativo.
ASR de **n** bits significa dividir por 2^n (división com signo)

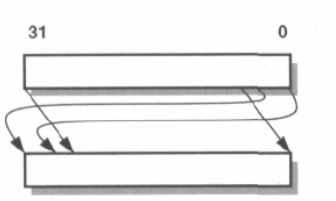


ASR #5, positive operand



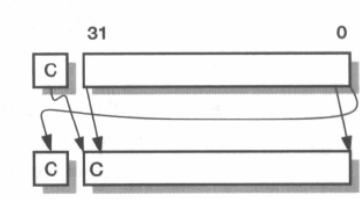
ASR #5, negative operand

ROR Rotación a la derecha de **0** a **31** posiciones, los bits que salen del extremo menos significativo de la palabra se usan en orden, para llenar los bits más significativos de la palabra que van quedando vacíos. El bit menos significativo se copia en el bit de acarreo, **C**.
Rotación de **32** bits.



ROR #5

RRX Rotación a la derecha de **0** a **32** posiciones. Rotación de 33 bits, el 33avo. bit es el de acarreo, **C**. Explicación para una rotación de un solo bit: el bit más significativo que quedó vacío (bit **31**) se llena con el valor viejo del indicador de acarreo, **C**, y el operando se desplaza a la derecha. Mientras que el bit menos significativo (bit **0**) se copia en el bit de acarreo, **C**.
Con el uso apropiado de los códigos de condición se puede ejecutar una rotación de 33 bits del operando y el indicador de acarreo.



RRX