95.57 Organización del Computador
Apunte ARM

Introducción **3**

ARM - Arquitectura **4**
- Set de Registros 4
  - Registro de Enlace (R14/LR) 4
  - Contador del Programa (R15/PC) 4
  - Current Program Status Register 5
    - Condition flags 5

ARM - Sintaxis **6**
- Caracteres especiales 6
- Directivas 6

ARM - Set de Instrucciones **8**
- Características principales 8
- Formato de las instrucciones 8
- Ejecución condicional 8
  - Códigos de Condición 9
  - Seteo de Códigos de Condición 10
  - Ejemplo 10
- Load/Store Multiple 11
- Instrucciones Aritméticas 11
  - Operaciones 11
  - Sintaxis 12
  - Ejemplos 12
  - Multiplicación 12
    - Restricciones 12
- Instrucciones Lógicas 12
  - Operaciones 12
  - Sintaxis 13
  - Ejemplos 13
- Instrucciones de Bifurcación 13
  - Bifurcaciones condicionales 14
- Pseudo-Instrucciones 15
- Subrutinas 15
  - Ejemplo 15
    - Programa llamador 15
    - Subrutina 16
- Movimiento de datos 16
  - Operaciones 16

Sintaxis 16
Ejemplos 16
Barrel Shifter 16
Operaciones soportadas por el Barrel Shifter 16
Shift a Izquierda (LSL) 16
Shift Lógico a Derecha (LSR) 17
Shift Aritmético a Derecha (ASR) 17
Rotate Right (ROR) 17
Rotate Right Extended (RRX) 17
Ejemplos 18
Instrucciones Load/Store 18
Transferencia de datos de un registro 18
Ejecución condicional 18
Sintaxis 18
Registro Base 18
Offset desde el Registro Base 19
Direccionamiento Pre-indexado 19
Direccionamiento Post-indexado 20
Pila (Stack) 21
Pilas y subrutinas 21
Interrupción de Software (SWI) 21
SWIs en ARSim# 21
SWI Codes 21

Modos de direccionamiento **23**
Modo pre-indexado 23
Modo pre-indexado con reescritura 23
Modo post-indexado 24

Anexo **25**
Sumario de Set de Instrucciones 25
Introducción a la Familia de Arquitecturas ARM 29
Ejemplo en C 29
¿Por qué enseñar lenguaje assembler y arquitectura de computadoras? 29
¿Por qué aprender ensamblador? 30

Introducción

ARM significa Advanced RISC Machine y es el primer procesador de computadora de set de instrucciones reducida (RISC) para uso comercial. Originalmente fue Acorn RISC Machine, dado que fue concebida originalmente por Acorn Computers para su uso en ordenadores personales.

ARM es una arquitectura RISC (Reduced Instruction Set Computer) de 16 ó 32 bits y, a partir de la versión V8-A, también de 64 Bits.

Un enfoque de diseño basado en RISC permite que los procesadores ARM requieran una cantidad menor de transistores que los procesadores x86 CISC, típicos en la mayoría de ordenadores personales. Este enfoque de diseño nos lleva, por tanto, a una reducción en los costes y energía. Estas características son deseables para dispositivos que funcionan con baterías, como los teléfonos móviles, tablets o netbooks.

La relativa simplicidad de los procesadores ARM los hace ideales para aplicaciones de baja potencia. Como resultado, se han convertido en los dominantes dentro del mercado de la electrónica móvil e integrada, encarnados en microprocesadores y microcontroladores pequeños, de bajo consumo y relativamente bajo costo.

La primera ARM se estableció en la Universidad de Cambridge en 1978. Las computadoras del grupo Acorn desarrollaron el primer procesador comercial RISC de ARM en 1985. ARM se fundó y fue muy popular en 1990. En 2005, alrededor del 98% de los más de mil millones de teléfonos móviles vendidos utilizaban al menos un procesador ARM y en 2007, ARM era utilizada en la gran mayoría de los teléfonos móviles. Desde 2009, los procesadores ARM son aproximadamente el 90% de todos los procesadores RISC de 32 bits integrados.

El núcleo del procesador ARM es el motor dentro del sistema que obtiene las instrucciones ARM de la memoria y las ejecuta. Los núcleos ARM son muy pequeños y suelen ocupar unos pocos milímetros cuadrados del área del chip. Debido a su bajo consumo de energía y a que tiene un mejor rendimiento en comparación con otros procesadores, ARM es el procesador utilizado en los productos digitales avanzados, como teléfonos móviles, sistemas de cámaras digitales, redes domésticas, tecnologías inalámbricas y dispositivos portátiles.

ARM - Arquitectura

ARM está basado en una arquitectura load/store, reduciendo así el set de instrucciones; esto significa que el núcleo no puede operar directamente con la memoria. Todas las operaciones de datos deben realizarse mediante registros con la información que se encuentra en la memoria.

Set de Registros

ARM tiene 16 Registros visibles al programador y un Registro de estado del programa actual, CPSR (Current Program Status Register). El detalle de los registros es:

- *R0* a *R12* son los registros de *uso general*
- *R13* está reservado para que el programador lo utilice como *puntero a la pila*
- *R14* o *LR* es el *registro de enlace* que almacena una dirección de retorno de una subrutina
- *R15* o *PC* contiene el contador del programa y es accesible al programador
- *CPSR* es el registro que contiene información sobre el estado actual del programa

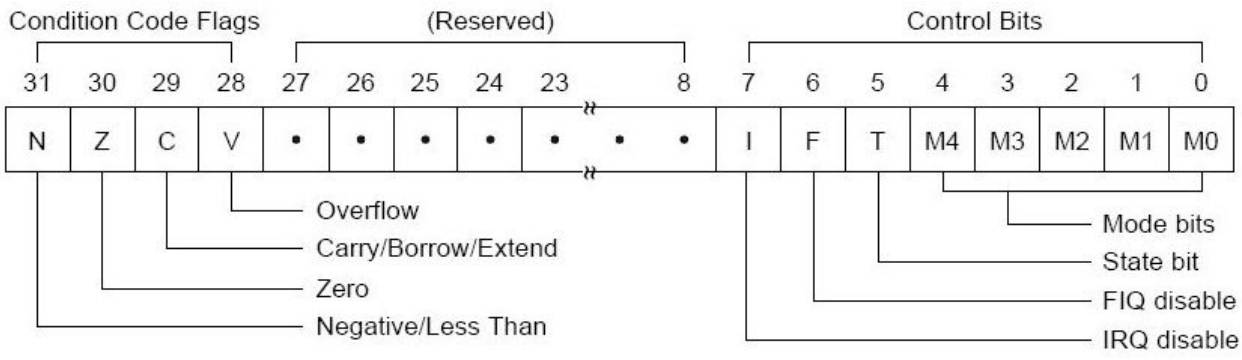
Registro de Enlace (R14/LR)

El *R14* es usado como registro enlace a subrutinas (*LR*) y almacena la dirección de retorno cuando una se realiza una operación *Branch with Link*, la cual se calcula desde el registro *PC*. Siendo así, para volver desde una subrutina puede ejecutarse: `MOV r15, r14` o `MOV pc, lr`.

Contador del Programa (R15/PC)

Como todas las instrucciones tienen longitud de 32 bits y deben estar alineadas a *word* (cada *word* es de 4 bytes, esto significa que cada instrucción comienza en una posición de memoria divisible por 4), el valor almacenado en el registro *Contador del Programa (PC)* es almacenado en los bits [31:2] con los bits [1:0] iguales a 0 (dado que las instrucciones no pueden estar alineadas a *halfword* o *byte*).

Current Program Status Register



Condition flags

Flag	Instrucción Lógica	Instrucción Aritmética
Negative	Sin significado particular	Indica un número negativo en una operación con signo
Zero	Resultado es cero	Resultado es cero
Carry	Luego de una operación Shift: Se cargó '1' en el flag de carry	Resultado es mayor a 32 bits
o**V**erflow	Sin significado particular	Resultado es mayor a 31 bits. Indica una posible corrupción del signo en operaciones con signo

ARM - Sintaxis

Los comentarios en comienzan con `#` y se ignora todo, desde el signo hasta el final de la línea.

Las etiquetas pueden definirse utilizando una secuencia de caracteres alfanuméricos, barras inferiores (_) y puntos (`.`) que no comienzan con un número. Los códigos de operación son palabras reservadas que no pueden ser utilizados como identificadores válidos. Las etiquetas se declaran colocándolas al principio de una línea seguida de dos puntos, por ejemplo:

```
item:
      .word 1
```

Las cadenas de caracteres (strings) deben ir entre comillas dobles (`"`) y los caracteres especiales siguen la convención de C:

```
      newline         \n
      tab             \t
      quote           \"
```

Caracteres especiales

La presencia del caracter `@` en una línea indica el comienzo de un comentario que se extiende hasta el final de la línea.

Si el caracter `#` aparece como el primer caracter de una línea, toda la línea es tratada como un comentario, pero en este caso la línea puede también ser una directiva de lógica numérica o un comando de control para el preprocesador.

El caracter `;` puede ser usado en lugar de una nueva línea para separar sentencias.

Tanto `#` como `$` pueden ser usados para indicar operandos inmediatos.

Directivas

`.equ sym, constant`
Da el nombre simbólico *sym* a una constante *constant*.

`.data <addr>`
Indica que los siguientes ítems son datos y deben almacenarse en el segmento de datos. Si el argumento opcional *addr* está presente, los ítems son almacenados desde la dirección *addr*.

`.align n`
Alinear el siguiente dato en una posición de memoria divisible por 2^n. Por ejemplo, `.align 2` alinea el siguiente valor en una dirección divisible por 4 (alineado a `word`) límite de palabra.

`.align 0` desactiva la alineación automática de las directivas `.half`, `.word`, `.float` y `.double` hasta la siguiente directiva `.data`.

`.ascii str`
Almacena el string en memoria pero no agrega byte nulo al final.

`.asciiz str`
Almacena el string en memoria y agrega byte nulo al final.

`.byte b1, ..., bn`
Almacena los n valores en bytes sucesivos en memoria.

`.half h1, ..., hn`
Almacena los n valores de 16 bits en halfwords sucesivos en memoria.

`.word w1, ..., wn`
Almacena los n valores de 32 bits en words sucesivos en memoria.

`.float f1, ..., fn`
Almacena los n flotantes de precisión simple sucesivos en memoria.

`.double d1, ..., dn`
Almacena los n flotantes de precisión doble sucesivos en memoria.

`.comm sym size`
Aloca *size* bytes en el segmento de datos para el símbolo *sym*.

`.globl sym`
Declara que el símbolo sym es global y que puede ser referenciado desde otros archivos.

`.label sym`
Declara que el símbolo sym es una etiqueta.

`.text <addr>`
Indica que los siguientes ítems en memoria son instrucciones. Si el argumento opcional *addr* está presente, los ítems son almacenados desde la dirección *addr*.

`.end`
Marca el fin del archivo del módulo del programa.

ARM - Set de Instrucciones

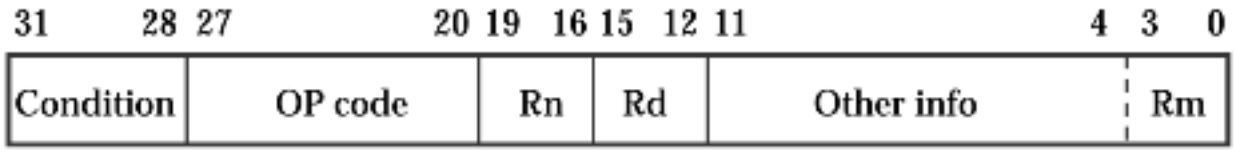
Características principales

- Todas las instrucciones tienen una longitud de 32 bits
- La mayor parte de las instrucciones se ejecutan en un solo ciclo de reloj
- La mayor parte de las instrucciones pueden ser ejecutadas condicionalmente
- Arquitectura Load/Store
    - Las instrucciones de procesamiento de datos actúan únicamente sobre registros
        - Formato de tres operandos
        - ALU y Shifter combinados para alta velocidad en la manipulación de bits
    - Instrucciones específicas de acceso a memoria con potentes modos de direccionamiento de auto indexación
        - Tipos de datos de 32, 16 y 8 bits
        - Instrucciones Load y Store flexibles

Formato de las instrucciones

Cada instrucción es codificada en una 32-bit word.

El formato de codificación básico para instrucciones de carga, almacenamiento, aritméticas y lógica es:



Una instrucción especifica un código de ejecución condicional (*Condition*), el código OP (*OP code*), dos o tres registros (*Rn*, *Rd* y *Rm*) y alguna otra información adicional.

Ejecución condicional

Una característica distintiva y algo inusual de los procesadores ARM es que todas las instrucciones se ejecutan condicionalmente dependiendo de una condición especificada en la instrucción.

La instrucción es ejecutada sólo si el estado actual del flag del código de condición del procesador satisface la condición especificada en los bits b_{31}-b_{28} de la instrucción. Por lo tanto, las instrucciones cuya condición no se ve satisfecha en el flag de código de condición del

procesador no se ejecutan. Una de las condiciones se utiliza para indicar que la instrucción siempre se ejecuta.

Esta característica elimina la necesidad de utilizar muchas bifurcaciones. El costo en tiempo de no ejecutar una instrucción condicional es frecuentemente menor que el uso de una bifurcación o llamado a una subrutina que, de otra manera, sería necesaria.

Códigos de Condición

Code [31:28]	Mnemonic	Interpretación	Status flag state required
0000	EQ	Igual / Igual a cero	Z seteado
0001	NE	Distinto	Z vacío
0010	CS/HS	Carry seteado / ≥ sin signo	C seteado
0011	CC/LO	Carry vacío / < sin signo	C vacío
0100	MI	Menor / negativo	N seteado
0101	PL	Mayor / positivo o cero	N vacío
0110	VS	Overflow	V seteado
0111	VC	No overflow	V vacío
1000	HI	> sin signo	C seteado y Z vacío
1001	LS	≤ sin signo	C vacío y Z seteado
1010	GE	≥ con signo	N igual a V
1011	LT	< con signo	N distinto de V
1100	GT	> con signo	Z vacío y N igual a V
1101	LE	≤ con signo	Z seteado o N distinto de V
1110	AL	Siempre	Cualquiera
1111	NV	Nunca	Ninguno

Para que una instrucción sea ejecutada condicionalmente se le agrega el sufijo con la condición apropiada. Por ejemplo, una instrucción de suma tiene la siguiente forma:

```
ADD r0, r1, r2
```

y para ejecutarla sólo si el flag cero está seteado:

```
ADDEQ r0, r1, r2
```

Esto mejora la densidad del código y la performance reduciendo el número de instrucciones de bifurcación. Ej.:

```
        CMP   r3, #0
        BEQ   skip
        ADD   r0, r1, r2
.skip:  ...
```

```
        CMP   r3, #0
        ADDNE r0, r1, r2
```

Seteo de Códigos de Condición

Algunas instrucciones, como Compare, dadas por `CMP Rn, Rm` que realiza la operación `[Rn]-[Rm]` tienen como único propósito establecer los flags de código de condición en función del resultado de la resta.

Exceptuando a las instrucciones de comparación, las operaciones de procesamiento de datos no afectan a los *condition flags*. Para que los *condition flags* se vean afectados, el bit *S* de la instrucción necesita estar seteado. Esto se hace agregando el sufijo *S* a la instrucción (y a cualquier código de condición). Por ejemplo, para restar uno al R1 y afectar los *condition flags* en un loop:

```
.loop:  ...
        subs   r1, r1, #1
        bne    loop
```

Ejemplo

```
        ldr    r1, n
        ldr    r2, puntero
        mov    r0, #0
.loop:
        ldr    r3, [r2], #4
        add    r0, r0, r3
        subs   r1, r1, #1
        bgt    loop
        str    r0, suma
```

Suma los n enteros desde la posición apuntada por puntero y almacena el resultado en suma
`GT`: comparación por mayor con signo
`BGT`: bifurca si si `Z=0` y `N=0`

Load/Store Multiple

En los procesadores ARM, hay dos instrucciones para cargar y almacenar múltiples operandos y son las llamadas instrucciones de transferencia de bloques. Cualquier subconjunto de los registros de propósito general se puede cargar o almacenar. Solo se permiten operandos *word* y los códigos OP utilizados son `LDM` (Load Multiple) y `STM` (Store Multiple).

Los operandos de memoria deben estar en ubicaciones de *words* sucesivas. Todas las formas de pre- y post-indexación con y sin reescritura están disponibles. Operan en un registro base *Rn* especificado en la instrucción y el desplazamiento es siempre 4:

```
ldmia R10!, {R0,R1,R6,R7}
```

IA: "Increment After" corresponde a post-indexación

El Registro Base puede ser actualizado si se le agrega el signo `!`

Instrucciones Aritméticas

La expresión básica para instrucciones aritméticas es `OPcode Rd, Rn, Rm`

Ejemplos básicos:

```
add r0, r2, r4  @ realiza la operación r0⇐[r2]+[r4]
sub r0, r6, r5  @ realiza la operación r0⇐[r6]-[r5]
add r0, r3, #17 @ realiza la operación r0⇐[r3]+17
```

El segundo operando puede ser shifteado o rotado antes de ser usado en la operación. Por ejemplo:

`add r0, r1, r5, lsl #4` opera del siguiente modo: el segundo operando almacenado en `r5` es shifteado a izquierda 4 bits (equivalente a `[r5]x16`), y se le suma el contenido de `r1`; la suma es almacenada en `r0`.

Operaciones

```
add   operand1 + operand2              @ suma
adc   operand1 + operand2 + carry      @ suma con acarreo
sub   operand1 - operand2              @ resta
sbc   operand1 - operand2 + carry - 1  @ resta con acarreo
rsb   operand2 + operand1              @ resta inversa
```

```
rsc   operand2 - operand1 + carry - 1  @ resta inversa con acarreo
```

Sintaxis

```
<operation>{<cond>}{S} Rd, Rn, operand2
```

Ejemplos

- `add    r0, r1, r2`
- `subgt  r3, r3, #1`
- `rsbles r4, r5, #5`

Multiplicación

ARM básico provee dos instrucciones de multiplicación:

```
mul{<cond>}{S} Rd, Rm, Rs        @ Rd = Rm * Rs
mla{<cond>}{S} Rd, Rm, Rs, Rn    @ Rd = (Rm * Rs) + Rn
```

Restricciones

- `Rd` y `Rm` no pueden ser el mismo registro
- No puede usarse el registro `PC`

Instrucciones Lógicas

Las operaciones lógicas AND, OR, XOR, y Bit-Clear son implementadas con instrucciones con los códigos de operación AND, ORR, EOR, y BIC. Por ejemplo:

`and r0, r0, r1` realiza la operación `r0⇐[r0]+[r1]`

La instrucciones Bit-Clear (BIC) está estrechamente relacionada con la instrucción AND: complementa cada bit del operando `Rm` antes de aplicarle AND con los bits del registro `Rn`. Por ejemplo:

`bic r0, R0, r1`, siendo `r0=02FA62CA` y `r1=0000FFFF`, el resultado de la instrucción es `r0=02FA0000`.

La instrucción Move Negative complementa los bits del operando fuente y almacena el resultado en `Rd`. Por ejemplo:

`mvn r0, r3`

Operaciones

```
and   operand1 AND operand2
```

```
eor   operand1 EOR operand2

orr   operand1 OR operand2

orn   operand1 NOR operand2

bic   operand1 AND NOT operand2
```

Sintaxis

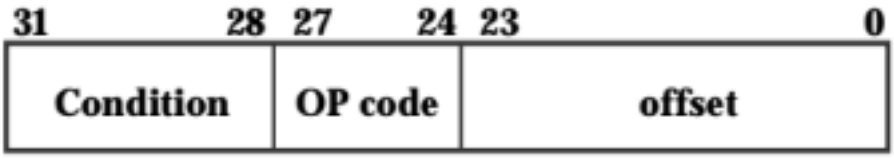
```
<operation>{<cond>}{S} Rd, Rn, operand2
```

Ejemplos

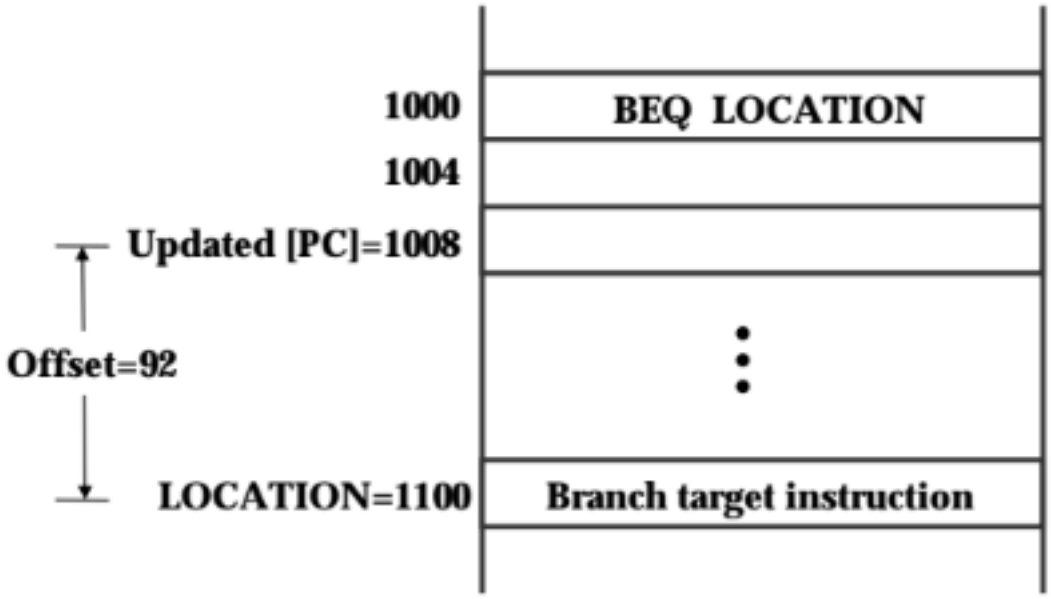
- `and   r0, r1, r2`
- `biceq  r3, r3, #1`
- `eors   r4, r5, #5`

Instrucciones de Bifurcación

Las instrucciones de bifurcación condicionales contienen un desplazamiento de 24 bits con signo que se agrega al contenido actualizado del *PC* para generar la dirección de destino de la bifurcación. El formato de las instrucciones de bifurcación es el que se muestra a continuación:



Por ejemplo, la instrucción BEQ (Bifurca si igual a 0) bifurca si el flag Z está seteado en 1:



Bifurcaciones condicionales

Mnemonic	Interpretación	Aplicación
B	Incondicional	Siempre bifurcar
BAL	Siempre	Siempre bifurcar
BEQ	Igual	Comparación igual o resultado es cero
BNE	Distinto	Comparación distinta o resultado no es cero
BPL	Positivo	Resultado positivo o cero
BMI	Negativo	Resultado negativo
BCC	Flag Carry vacío	Operación aritmética no resultó en acarreo hacia afuera
BLO	Menor	Comparación sin signo con resultado menor
BCS	Flag Carry seteado	Operación aritmética resultó en acarreo hacia afuera
BHS	Mayor o igual	Comparación sin signo con resultado mayor o igual
BVC	Flag Overflow vacío	Operación de entero con signo: resultado sin overflow
BVS	Flag Overflow seteado	Operación de entero con signo: resultado con overflow
BGT	Mayor	Comparación de entero con signo: mayor

BGE	Mayor o igual	Comparación de entero con signo: mayor o igual
BLT	Menor	Comparación de entero con signo: menor
BLE	Menor o igual	Comparación de entero con signo: mayor o igual
BHI	Mayor	Comparación de entero sin signo: mayor
BLS	Menor o igual	Comparación de entero sin signo: menor o igual

Pseudo-Instrucciones

La pseudo-instrucción `adr Rd, direccion` mantiene la dirección del valor de 32 bits en `Rd` Esta instrucción no es realmente una instrucción de máquina sino que el ensamblador elige instrucciones de máquina reales apropiadas para implementar pseudo-instrucciones. Por ejemplo, la combinación de la instrucción de máquina `ldr r2, puntero` y la directiva de declaración de datos `puntero dcd num1` es una forma de implementar la pseudo-instrucción `adr r2, num1`.

ALIGN: ajusta el contador de dirección a palabra (word)

END: nada más que ensamblar

EQU: reemplazo de símbolo

```
loopcnt EQU 5
```

Subrutinas

Se utiliza una instrucción de branch and link (`BL`) para llamar a una subrutina.
La dirección de retorno se carga en el registro `R14`, que actúa como un *link register*. Cuando las subrutinas están anidadas, el contenido del *link register* debe ser guardado en una pila por la misma subrutina. El registro `R13` se usa como puntero para esta pila.

Ejemplo

Programa llamador

```
ldr   r1, n
ldr   r2, puntero
bl    sumalista
str   r0, suma
...
```

Subrutina

```
.sumalista:
      @ salva r3 y dir. de retorno en r14 en la pila usando r13 como puntero a pila
      stmfd r13, {r3, r14}
      mov   r0, #0
.loop:
      ldr   r3, [r2], #4
      add   r0, r0, r3
      subs  r1, r1, #1
      bgt   loop
      @ restaura r3 y carga la dir. de retorno en r15
      ldmfd r13!, {r3, r15}
```

Movimiento de datos

Operaciones

```
mov   operand1 <= operand2
```

```
mvn   operand1 <= NOT operand2
```

Sintaxis

```
<operation>{<cond>}{S} Rd, operand2
```

Ejemplos

- `mov    r0, r1`
- `movs   r2, #10`
- `mvneq  r1, #0`

Barrel Shifter

ARM no tiene instrucciones de shift. En su lugar tiene un barrel shifter que provee un mecanismo que lleva a cabo shifts como parte de otras instrucciones.

Operaciones soportadas por el Barrel Shifter

Shift a Izquierda (LSL)

Shift a la izquierda según la cantidad especificada (multiplica por potencias de 2). Por ejemplo:

```
LSL # 5 @ multiplica por 32
```

Shift Lógico a Derecha (LSR)

Shift a la derecha según la cantidad especificada (divide por potencia de 2). Por ejemplo:

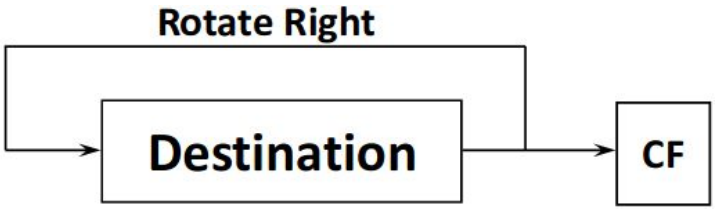
```
LSR # 5 @ divide por 32
```

Shift Aritmético a Derecha (ASR)

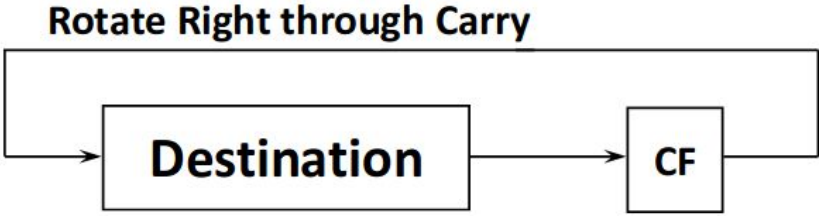
Shift a la derecha según la cantidad especificada (divide por potencia de 2) preservando el bit de signo. Por ejemplo:

```
ASR # 5 @ divide por 32
```

Rotate Right (ROR)



Rotate Right Extended (RRX)



Ejemplos

```
mov r2, r0, lsl #2          @ R2 = R0x4
add r9, r5, r5, lsl #3      @ R9 = R5+R5x8 ó R9=R5x9
rsb r9, r5, r5, lsl #3      @ R9 = R5x8-R5 ó R9=R5x7
sub r10, r9, r8, lsr #4     @ R10 = R9-R8/16
mov r12, r4, ror r3         @ R12 = R4 rotado der. por el valor en R3
```

Instrucciones Load/Store

Transferencia de datos de un registro

```
ldr         @ Load Word
str         @ Store Word
ldrb        @ Load Byte
strb        @ Store Byte
ldrh        @ Load Halfword
strh        @ Store Halfword
ldrsb       @ Load Signed Byte (load and extent sign to 32 bits)
ldrsh       @ Load Signed Halfword (load and extent sign to 32 bits)
```

Ejecución condicional

Estas instrucciones pueden ser ejecutadas condicionalmente insertando el condition code apropiado luego de `LDR` / `STR`. Ejemplo: `LDREQB`

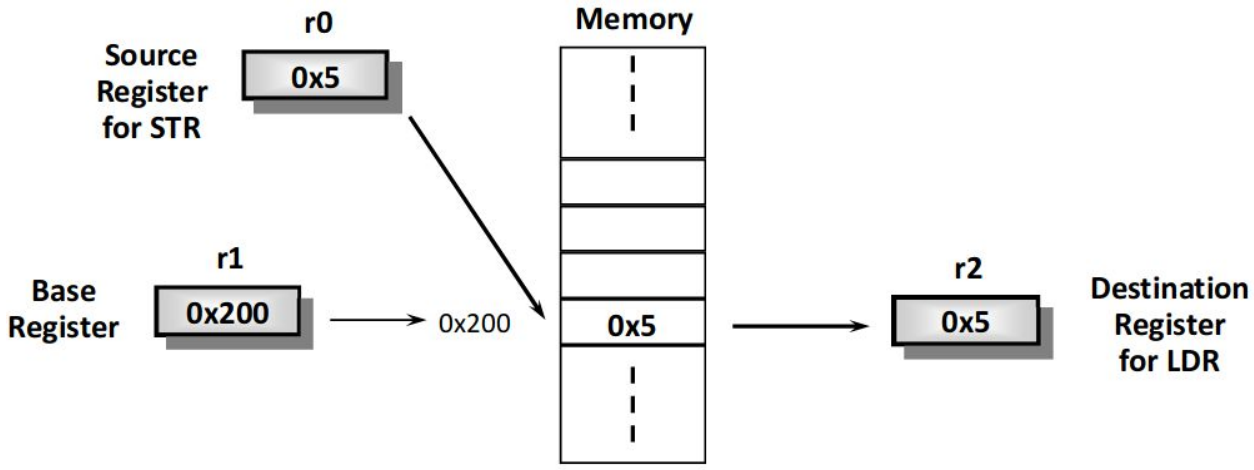
Sintaxis

```
<LDR|STR>{<cond>}{<size>} Rd, <address>
```

Registro Base

La dirección de memoria accedida está contenido en un registro base

```
str r0, [r1] @ Almacena (r0) en la memoria apuntada por r1
ldr r2, [r1] @ Carga en (r2) el valor apuntado por r1
```



Offset desde el Registro Base

Las instrucciones Load/Store pueden acceder la dirección contenida en el registro base así como una dirección offset desde el Registro Base. Este offset puede ser:

- Valor inmediato: BPF s/s de 12 bits
- Registro (opcionalmente shifteado por un valor inmediato)

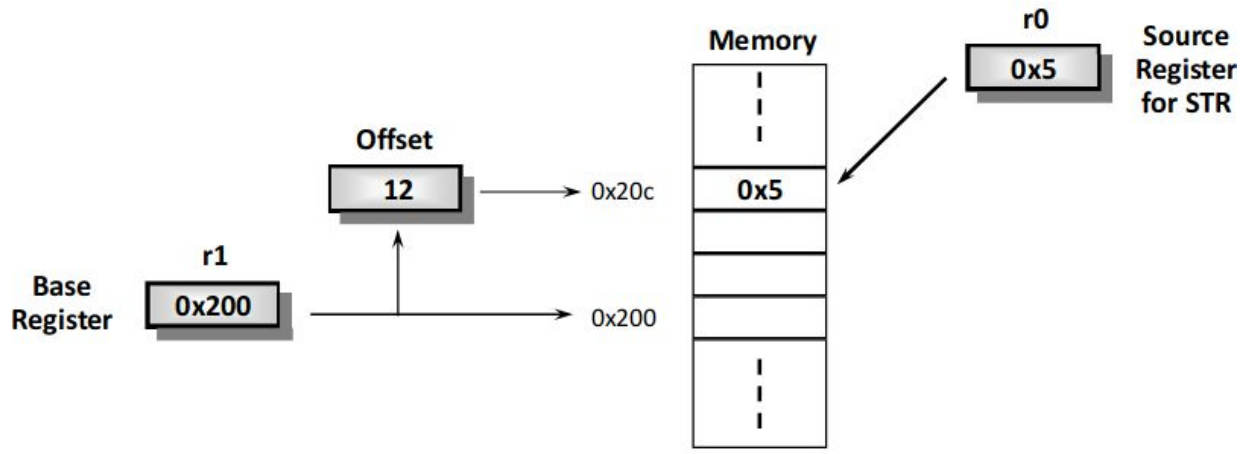
El offset puede ser sumado o restado del Registro Base prefijando el valor del offset o registro con `+` (por defecto) o `-`.

El offset puede aplicarse

- Antes de que se realice la transferencia: Direccionamiento Pre-indexado
    - Puede auto-incrementarse el Registro Base agregando `!` al final de la instrucción.
- Después de que se realice la transferencia: Direccionamiento Post-indexado
    - Causando que el Registro Base se vea auto-incrementado

Direccionamiento Pre-indexado

Ejemplo: `STR r0, [r1, #12]`



Para almacenar en la dirección `0x1f4`

```
STR r0, [r1, #-12]
```

Para auto-incrementar el Registro Base a `0x20c`

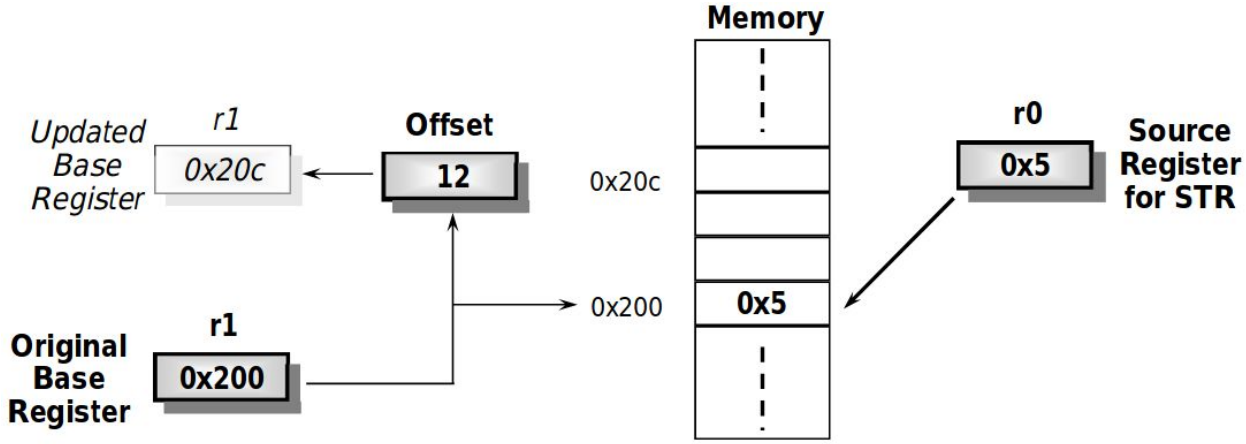
```
STR r0, [r1, #12]!
```

Si `(r2)=3` puede accederse a `0x20c` multiplicándolo por 4

```
STR r0, [r1, r2, LSL #2]
```

Direccionamiento Post-indexado

Ejemplo: `STR r0, [r1], #12`



Para auto-incrementar el Registro Base a la dirección 0x1f4

```
STR r0, [r1], #-12
```

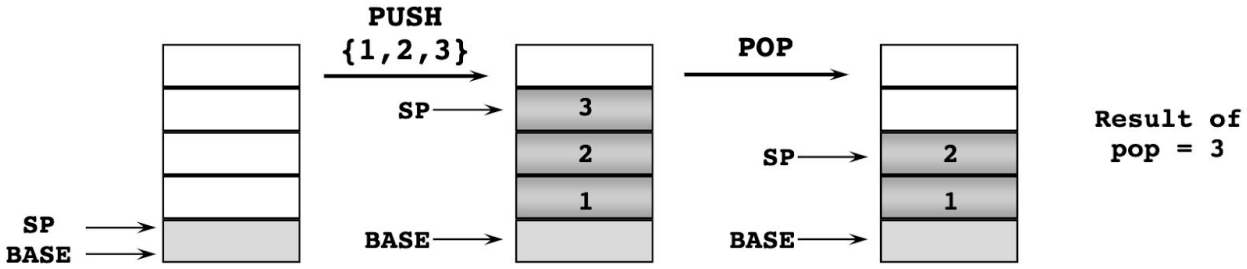
Si `(r2)=3` puede auto-incrementarse a 0x20c multiplicándolo por 4

```
STR r0, [r1], r2, LSL #2
```

Pila (Stack)

Una pila es un área de la memoria que crece a medida que nuevos datos se insertan (push) en la parte superior de la misma, y se reduce a medida que los datos se remueven (pop) desde la parte superior. Dos punteros definen los límites actuales de la pila:

- Un puntero de base: apunta a la parte inferior de la pila (la primera posición).
- Un puntero a la pila: apunta a la parte superior actual de la pila.



Pilas y subrutinas

Uno de los usos de las pilas es crear un área de memoria temporal para los registros para las subrutinas. Cualquier registro que sea necesario puede ser agregado (push) a la pila al principio de la subrutina y tomado (pop) desde la pila para recuperar el valor antes de volver de la subrutina.

```
stmfd sp!,{r0-r12, lr} @ apilar los registros y dir. de retorno

ldmfd sp!,{r0-r12, pc} @ desapilar los registros y retornar
```

Interrupción de Software (SWI)

Una interrupción de software es un tipo de interrupción causada por una instrucción especial en el set de instrucciones. El software invoca una interrupción de software, a diferencia de una interrupción de hardware, y se considera una de las formas de invocar llamadas al sistema.

SWIs en ARSim#

En ARMSim# se utilizan interrupciones de software para operaciones comunes de entrada/salida.

La sintaxis para hacer un llamada es: `swi <swi code>`

SWI Codes

Code	Description and Action	Inputs	Outputs	EQU

0x00	Mostrar caracter por consola	R0: el caracter		SWI_PrChr
0x02	Mostrar string por consola	R0: dirección de un string terminado en null		
0x11	Detener la ejecución			SWI_Exit
0x12	Asignar Bloque de Memoria	R0: tamaño del Bloque en bytes	R0: dirección del Bloque	SWI_MeAlloc
0x13	Desasignar todo Bloque de Memoria			SWI_DAlloc
0x66	Abrir Archivo (Modo 0: Input / Modo 1: Output / Modo 2: Append)	R0: dirección de un string terminado en null con el nombre del archivo R1: Modo	R0: manejador de archivo (-1 si el archivo no abre)	SWI_Open
0x68	Cerrar Archivo	R0: manejador de archivo		SWI_Close
0x69	Escribir string	R0: manejador de archivo o Stdout (1) R1: dirección de un string terminado en null		SWI_PrStr
0x6a	Leer string desde un Archivo	R0: manejador de archivo R1: dirección destino R2: max bytes a almacenar	R0: número de bytes almacenados	SWI_RdStr
0x6b	Escribir entero	R0: manejador de archivo o Stdout (1) R1: entero		SWI_PrInt
0x6c	Leer entero desde un Archivo	R0: manejador de archivo	R0: entero	SWI_RdInt
0x6d	Obtener el tiempo actual (ticks)		R0: número de ticks (milisegundos)	SWI_Timer

Modos de direccionamiento

Nombre	Nombre Alternativo	Ejemplos
Registro a registro	Registro directo	`mov r0, r1`
Absoluto	Directo	`ldr r0, mem`
Literal	Inmediato	`mov r0, #15` `add r1, r2, #12`
Indexado, base	Registro indirecto	`ldr r0, [r1]`
Pre-Indexado, base con desplazamiento	Registro indirecto con offset	`ldr r0, [r1, #4]`
Pre-indexado, autoindexado	Registro indirecto con pre-incremento	`ldr r0, [r1, #4]!`
Post-indexado, autoindexado	Registro indirecto con post-incremento	`ldr r0, [r1], #4`
Doble registro indirecto	Registro indirecto indexado	`ldr r0, [r1, r2]`
Doble registro indirecto escalado	Registro indirecto indexado escalado	`ldr r0, [r1, r2, lsl #2]`
Relativo al PC		`ldr r0, [PC, #offset]`

Modo pre-indexado

La dirección efectiva del operando es la suma de los contenidos del registro base Rn y un offset.

`[Rn, #offset]`	`DE=[Rn]+offset`
`[Rn, ±Rm, shift]`	`DE=[Rn]±[Rm] shifteado`

Modo pre-indexado con reescritura

La dirección efectiva del operando se genera de la misma manera que en el modo Pre-indexado pero la dirección efectiva se escribe también en Rn.

`[Rn, #offset]!`	`DE=[Rn]+offset` `Rn⬅[Rn]+offset`
`[Rn, ±Rm, shift]`	`DE=[Rn]±[Rm] shifteado` `Rn⬅[Rn]±[Rm] shifteado`

Modo post-indexado

La dirección efectiva del operando es el contenido de Rn. El desplazamiento se agrega a esta dirección y el resultado se escribe de nuevo en Rn.

`[Rn], #offset`	`DE=[Rn]` `Rn⬅[Rn]+offset`
`[Rn], ±Rm, shift`	`DE=[Rn]` `Rn⬅[Rn]±[Rm] shifteado`

Anexo

Sumario de Set de Instrucciones

Fuente: ***[ARM instruction summary]***

Operation		**Assembly syntax**
Move	Move	MOV{cond}{S} Rd, <Oprnd2>
	Move NOT	MVN{cond}{S} Rd, <Oprnd2>
	Move SPSR to register	MRS{cond} Rd, SPSR
	Move CPSR to register	MRS{cond} Rd, CPSR
	Move register to SPSR	MSR{cond} SPSR{field}, Rm
	Move register to CPSR	MSR{cond} CPSR{field}, Rm
	Move immediate to SPSR flags	MSR{cond} SPSR_f, #32bit_Imm
	Move immediate to CPSR flags	MSR{cond} CPSR_f, #32bit_Imm
Arithmetic	Add	ADD{cond}{S} Rd, Rn, <Oprnd2>
	Add with carry	ADC{cond}{S} Rd, Rn, <Oprnd2>
	Subtract	SUB{cond}{S} Rd, Rn, <Oprnd2>
	Subtract with carry	SBC{cond}{S} Rd, Rn, <Oprnd2>
	Subtract reverse subtract	RSB{cond}{S} Rd, Rn, <Oprnd2>
	Subtract reverse subtract with carry	RSC{cond}{S} Rd, Rn, <Oprnd2>
	Multiply	MUL{cond}{S} Rd, Rm, Rs
	Multiply accumulate	MLA{cond}{S} Rd, Rm, Rs, Rn
	Multiply unsigned long	UMULL{cond}{S} RdLo, RdHi, Rm, Rs

	Multiply unsigned accumulate long	UMLAL{cond}{S} RdLo, RdHi, Rm, Rs
	Multiply signed long	SMULL{cond}{S} RdLo, RdHi, Rm, Rs
	Multiply signed accumulate long	SMLAL{cond}{S} RdLo, RdHi, Rm, Rs
	Compare	CMP{cond} Rd, <Oprnd2>
	Compare negative	CMN{cond} Rd, <Oprnd2>
Logical	Test	TST{cond} Rn, <Oprnd2>
	Test equivalence	TEQ{cond} Rn, <Oprnd2>
	AND	AND{cond}{S} Rd, Rn, <Oprnd2>
	EOR	EOR{cond}{S} Rd, Rn, <Oprnd2>
	ORR	ORR{cond}{S} Rd, Rn, <Oprnd2>
	Bit clear	BIC{cond}{S} Rd, Rn, <Oprnd2>
Branch	Branch	B{cond} label
	Branch with link	BL{cond} label
	Branch and exchange instruction set	BX{cond} Rn
Load	Word	LDR{cond} Rd, <a_mode2>
	Word with user-mode privilege	LDR{cond}T Rd, <a_mode2P>
	Byte	LDR{cond}B Rd, <a_mode2>
	Byte with user-mode privilege	LDR{cond}BT Rd, <a_mode2P>
	Byte signed	LDR{cond}SB Rd, <a_mode3>
	Halfword	LDR{cond}H Rd, <a_mode3>
	Halfword signed	LDR{cond}SH Rd, <a_mode3>

	Multiple block data operations	-
	Increment before	LDM{cond}IB Rd{!}, <reglist>{^}
	Increment after	LDM{cond}IA Rd{!}, <reglist>{^}
	Decrement before	LDM{cond}DB Rd{!}, <reglist>{^}
	Decrement after	LDM{cond}DA Rd{!}, <reglist>{^}
	Stack operation	LDM{cond}<a_mode4L> Rd{!}, <reglist>
	Stack operation, and restore CPSR	LDM{cond}<a_mode4L> Rd{!}, <reglist+pc>^
	Stack operation with user registers	LDM{cond}<a_mode4L> Rd{!}, <reglist>^
Store	Word	STR{cond} Rd, <a_mode2>
	Word with user-mode privilege	STR{cond}T Rd, <a_mode2P>
	Byte	STR{cond}B Rd, <a_mode2>
	Byte with user-mode privilege	STR{cond}BT Rd, <a_mode2P>
	Halfword	STR{cond}H Rd, <a_mode3>
	Multiple block data operations	-

	Increment before	STM{cond}IB Rd{!}, <reglist>{^}
	Increment after	STM{cond}IA Rd{!}, <reglist>{^}
	Decrement before	STM{cond}DB Rd{!}, <reglist>{^}
	Decrement after	STM{cond}DA Rd{!}, <reglist>{^}
	Stack operation	STM{cond}<a_mode4S> Rd{!}, <reglist>
	Stack operation with user registers	STM{cond}<a_mode4S> Rd{!}, <reglist>^
Swap	Word	SWP{cond} Rd, Rm, [Rn]
	Byte	SWP{cond}B Rd, Rm, [Rn]
Coprocessors	Data operation	CDP{cond} p<cpnum>, <op1>, CRd, CRn, CRm, <op2>
	Move to ARM register from coprocessor	MRC{cond} p<cpnum>, <op1>, Rd, CRn, CRm, <op2>
	Move to coprocessor from ARM register	MCR{cond} p<cpnum>, <op1>, Rd, CRn, CRm, <op2>
	Load	LDC{cond} p<cpnum>, CRd, <a_mode5>
	Store	STC{cond} p<cpnum>, CRd, <a_mode5>
Software interrupt		SWI 24bit_Imm

Introducción a la Familia de Arquitecturas ARM

Versión	Familia
ARMv1	ARM1
ARMv2	ARM2, ARM3
ARMv3	ARM6, ARM7
ARMv4	Strong ARM, ARM7TDMI, ARM9TDMI
ARMv5	ARM7EJ, ARM9E, ARM10XE
ARMv6	ARM11
ARMv7	Cortex

Ejemplo en C

C

```
y = a*(b+c);
```

ARM

```
adr r4,b        @ obtener dirección de b
ldr r0,[r4]     @ obtener valor de b
adr r4,c        @ obtener dirección de c
ldr r1,[r4]     @ obtener valor de c
add r2,r0,r1    @ calcular resultado parcial
adr r4,a        @ obtener dirección de a
ldr r0,[r4]     @ obtener valor de a
mul r2,r2,r0    @ calcular valor final de y
adr r4,y        @ obtener dirección de y
str r2,[r4]     @ almacenar y
```

¿Por qué enseñar lenguaje assembler y arquitectura de computadoras?

Considerando que la ciencia de la computación está relacionada principalmente con el uso de la computadora puede argumentarse que el lenguaje ensamblador es irrelevante. Ahora, ¿el cirujano estudió metalurgia para comprender cómo funciona un bisturí? ¿estudia termodinámica el piloto para comprender cómo funciona un motor de reacción? ¿una persona que lee noticias estudia electrónica para entender cómo funciona una cámara de fotos? La respuesta a todas estas preguntas es *no*. Entonces, ¿por qué enseñar lenguaje de ensamblador y arquitectura de computadoras al estudiante? En primer lugar, porque la educación no es lo mismo que la

formación. El estudiante de ciencias de la computación no está solo capacitado para usar varias herramientas de las computadoras. Un curso universitario que lleve a un título de grado también debe cubrir la historia y las bases teóricas de la materia. Sin un conocimiento de la arquitectura informática, el científico informático no puede comprender cómo se han desarrollado las computadoras y de qué son capaces.

Se puede presentar un caso sólido para la enseñanza continua del lenguaje ensamblador dentro del plan de estudios de informática. Sin embargo, un lenguaje ensamblador no puede enseñarse como si fuera otro lenguaje de programación de propósito general. Tal vez, más que cualquier otro componente del plan de estudios de ciencias de la computación, la enseñanza de un lenguaje ensamblador admite una amplia gama de temas en el corazón de la ciencia de la computación. Un lenguaje ensamblador no debe usarse sólo para ilustrar algoritmos, sino para demostrar lo que realmente está sucediendo dentro de la computadora.

¿Por qué aprender ensamblador?

Dado el avance de los lenguajes de alto nivel, ¿por qué es necesario aprender programación en lenguaje ensamblador? Las razones son:

1. La mayoría de los usuarios industriales de microcomputadoras programan en lenguaje ensamblador.
2. Muchos usuarios de microcomputadoras continuarán programando en lenguaje ensamblador ya que necesitan el control que estos proporcionan.
3. Ningún lenguaje de alto nivel adecuado ha sido ampliamente disponible o estandarizado.
4. Muchas aplicaciones requieren la eficiencia del lenguaje ensamblador.
5. La comprensión del lenguaje ensamblador puede ayudar a evaluar los lenguajes de alto nivel.
6. Casi todos los programadores de microcomputadoras finalmente encuentran que necesitan algún conocimiento de lenguaje ensamblador, generalmente para depurar programas, escribir rutinas de E/S, acelerar o acortar las secciones críticas de los programas escritos en lenguajes de alto nivel, utilizar o modificar funciones del sistema, y entender los programas de otras personas.