Instrucciones de movimientos de datos

mov

La instrucción **mov (inmediato)** escribe un valor inmediato <const> en el registro destino <Rd>.

MOVS <Rd>, #<const>

El registro de flags (Registro CPSR) es actualizado según el resultado.

#<const> es un valor en el rango 0-255

Ejemplo

```
   movs  r0, #2      // Carga en el registro r0 la constante 2
```

MOV (registro)

Copia un registro fuente <Rm> al registro destino <Rd> En ciertas circunstancias, el ensamblador puede sustituir MOV por MVN, hay que tenerlo en cuenta cuando debuggeamos.

MOV{S} <Rd>, <Rm>

{S} Es un sufijo opcional. Si se especifica, se actualiza el registro de flags (Registro CPSR) de acuerdo al resultado de la operación.

<Rd> Es el registro destino. Puede ser PC ó SP. Si es PC entonces la instrucción causa un salto a la dirección contenida en <Rm>

<Rm> Es el registro fuente. Puede ser PC ó SP.

Ejemplos

```
   MOV   r0, r1      // Copia en el registro r0 el contenido del registro r1
   MOVS  r0, r3      // Copia en el registro r0 el contenido del registro r3
y actualiza cpsr
```

ldr y str

En la arquitectura ARM los accesos a memoria se hacen mediante instrucciones específicas ldr y str (luego veremos las variantes ldm, stm y las preprocesadas push y pop). El resto de instrucciones toman operandos desde registros o valores inmediatos, sin excepciones. En este caso la arquitectura nos fuerza a que trabajemos de un modo determinado: primero cargamos los registros desde memoria, luego procesamos el valor de estos registros con el amplio abanico de instrucciones del ARM, para finalmente volcar los resultados desde registros a memoria. Existen otras arquitecturas como la Intel x86, donde las instrucciones de procesado nos permiten leer o escribir directamente de memoria. Ningún método es mejor que otro, todo es cuestión de diseño. Normalmente se opta por direccionamiento a memoria en instrucciones de procesado en arquitecturas con un número reducido

de registros, donde se emplea la memoria como almacén temporal.

En nuestro caso disponemos de suficientes registros, por lo que podemos hacer el procesamiento sin necesidad de interactuar con la memoria, lo que por otro lado también es más rápido.

PUSH y POP

Push guarda/almacena uno ó una lista de registros <listreg> en el stack. El puntero a la pila, es el reg r13, por convencion lo llamamos SP (stack pointer). Este registro siempre apunta a la palabra del tope de la pila, última palabra ingresada en el stack.

PUSH <listreg>

Pop desapila uno ó una lista de registros <listreg> almacenados en el stack, para acceder a ello usa el registro sp.

POP <listreg>

Ejemplos:

```
push r1
push r2
push r4
/* código que modifica los registro r1, r2 y r4 */
pop  r4
pop  r2
pop  r1
```

Más info sobre el funcionamiento del stack, click acá

Observemos que el orden de recuperación de registros es inverso a como lo guardamos. En ARM es mucho más fácil que otros procesadores (por ejemplo 8086). Gracias a las instrucciones de carga/escritura múltiple podemos poner los registros en una lista, empleando una única instrucción.


```
push {r1, r2, r4}
/* código que modifica los registros r1, r2 y r4 */
pop {r1, r2, r4}
```

En este caso el orden no es relevante, el procesador siempre usa el orden ascendente para el push y el descendente para el pop, aunque nosotros por legibilidad siempre escribiremos los registros en orden ascendente.

Observación: Tenemos las instrucciones stm y ldm que son mucho más potentes que las instrucciones push y pop. El ensamblador de forma transparente traducirá a stm y ldm las instrucciones push y pop.

Aunque no usemos las instrucciones stm y ldm, les dejamos como es la sintaxis:

```
ldm{ modo_direc }{ cond } rn {!} , lista_reg
```

```
   stm{ modo_direc }{ cond } rn {!} , lista_reg
```

— *Mariano*

Volver

(5)