Subrutinas/Funciones

Subrutinas

Un aspecto muy peculiar de la arquitectura ARM es que las llamadas a subrutinas se hacen mediante un sencillo añadido a la instrucción de salto B (Branch).

Sintaxis:

```
B{L} {<cond>} <dirección_destino>
```

Donde:

L setea en 1 el bit 24 (el bit L) en la instrucción. Con esta opción, la instrucción almacena una dirección de retorno en el registo de link LR (R14). Si no se pone el sufijo L, la instrucción simplemente salta sin almacenar ninguna dirección.

<cond> Es la condicion para que se ejecute la instrucción.

<dirección_destino> Especifica la dirección a donde hay que saltar. Puede ser una etiqueta, pero se codifica en 24 bits. La dirección de destino se calcula de la siguiente manera:

1. Extender a 30 bits el signo de los 24 bits en complemento A dos (signed_immed_24).
2. Hacer un shift a izquierda de dos bits para formar un valor de 32 bits (SignExtend_30).
3. El nuevo PC es: PC = PC + (SignExtend_30(signed_immed_24) « 2)

Si agregamos el sufijo L, la instrucción B es BL y se llama (Branch and Link), y se usa para llamar a una subrutina, mediante un salto a la subrutina y escribiendo en el registro LR (R14) la dirección de la siguiente instrucción.

```
  main :   mov r1, #1
           mov r2, #2
           bl subrut
           mov r4, #4      /* Siguiente instrucción */
           ...
  subrut : mov r3, #3
           bx lr
```

La instrucción BL se usa en combinación con la instruccion BX (Branch and Exchange).

Sintaxis:

```
BX{<cond>}  <Rm>
```

Donde:

<cond> Es la condicion para que se ejecute la instrucción.

<Rm> Registro que contiene el valor de la dirección de destino del salto

En particular, en el retorno de una subrutina se usa Rm=LR. El retorno se logra copiando el registro LR (R14) al PC (Program Counter).

Continuemos analizando el ejemplo de subrutina anterior

```
main :   mov r1, #1
         mov r2, #2
         bl subrut
         mov r4, #4      /* Siguiente instrucción */
         ...
subrut : mov r3, #3
         bx lr
```

Si seguimos el flujo del programa primero cargamos r1 a 1, luego r2 a 2 y lo siguiente que hay es una llamada a la subrutina. En dicha llamada el procesador carga en **lr** la dirección de la siguiente instrucción “mov r4, #4” y salta a la etiqueta subrut. Se ejecuta el “mov r3, #3” de la subrutina y después **“bx lr”** que vendría a ser la instrucción de retorno. Es decir, salimos de la subrutina retomando el flujo del programa principal, ejecutando “mov r4, #4”.

La convención AAPCS nos servirá para trabajar con las subrutinas de manera estandarizada:

Convención AAPCS

Podriamos seguir nuestras propias reglas, pero si queremos interactuar con las librerías del sistema, tanto para llamar a funciones como para crear nuestras propias funciones y que éstas sean invocadas desde un lenguaje de alto nivel, tenemos que seguir una serie de pautas, lo que denominamos AAPCS (Procedure Call Standard for the ARM Architecture).

1. **Parámetros input**: Podemos usar hasta cuatro registros (desde r0 hasta r3) para pasar parámetros y hasta dos (r0 y r1) para devolver el resultado.
2. **Parámetros output**: No estamos obligados a usarlos todos, si por ejemplo la subrutina sólo usa dos parámetros de tipo int con r0 y r1 nos basta. Lo mismo pasa con el resultado, podemos no devolver nada (tipo void), devolver sólo r0 (tipo int ó un puntero a una estructura más compleja), o bien devolver r1:r0 cuando necesitemos enteros de 64 bits (tipo long long).
3. **Alineación de la memoria**: Los valores están alineados a 32 bits (tamaño de un registro), salvo en el caso de que algún parámetro sea más grande, en cuyo caso alinearemos a 64 bits. La unidad mínima son 32 bits, por ejemplo si queremos pasar un char por valor, extendemos de byte a word rellenando con ceros los 3 bytes más significativos. Lo mismo ocurre con los enteros de 64 bits, pero en el momento en que haya un sólo parámetro de este tipo, todos los demás se alinean a 64 bits.
4. **Preservar registros**: Es muy importante preservar el resto de registros (de r4 en adelante incluyendo lr). La única excepción es el registro r12 que podemos cambiar a nuestro antojo. Normalmente se emplea la pila para almacenarlos al comienzo de la subrutina y restaurarlos a la salida de ésta. Podemos usar como registros temporales (no necesitan ser preservados) los registros desde r0 hasta r3 que no se hayan usado para pasar parámetros.
5. **Alineación del stack**: La pila debe estar alineada a 64 bits, esto quiere decir que de usarla

para preservar registros, debemos reservar un número par de ellos. Si sólo necesitamos preservar un número impar de ellos, añadimos un registro más a la lista dentro del push, aunque no necesite ser preservado. Además de pasar parámetros y preservar registros, también podemos usar la pila para almacenar variables locales, siempre y cuando cumplamos la regla de alinear a 64 bits y equilibremos la pila antes de salir de la función.

Cuando programamos no es necesario seguir estas reglas. Es más, podemos escribir una función sin seguir la norma incluso si trabajamos bajo Linux, pero no es recomendable ya que no podríamos

Para poder reusar nuestras funciones en otros proyectos es necesario seguir estas reglas. Aunque cuando programamos en el emulador, ó en Bare Metal (Programas sin el sistema operativo como intermediario) podemos no seguir algunas reglas como la alineación del stack.

Lo mejor para entender estas reglas es con una serie de ejemplos:

Subrutinas en ensamblador llamadas desde ensamblador

En este primer ejemplo crearemos nuestras propias funciones con pasaje de parámetros o argumentos

```
   /* Organizacion del Computador UNGS: Programa en ensamblador ARM:
ejer03funsuma.s
      EJEMPLO SIMPLE DE LLAMADO A FUNCION/SUBRUTINA CON PASAJE DE PARAMETROS
      RESPETANDO LA CONVENCION AAPCS
      En este ejemplo la funcion queda arriba del main
     .fnstart - .fnend, esto se usa si la funcion  es llamada desde C
      Link Register ó Registro de Enlace. Almacena la dirección de
     retorno cuando una instrucción BL ó BLX ejecuta una llamada a una
rutina.
   */
   .data
   /* Definicion de datos */
   @
   @
   .text                   @ Defincion de codigo del programa
   @ ------------ Código de la función
   mifuncion:
      .fnstart
      add r0,#1              @ lo que hace  la funcion mifuncion
      bx lr                  @ salimos de la funcion mifuncion
      .fnend
   @ ------------- Código del main
     .global main            @ global, visible en todo el programa
     main:
          mov r0, #0xB            @ R0 <-- 11
                                  @ solo paso un parametro r0
          bl mifuncion            @ Llamamos a la funcion
          mov r1, #0xA            @ R1 <-- 10
          mov r2, #0x7            @ R2 <-- 7
          @
```

```
        mov r7, #1      // Salida al sistema
        swi 0           // Salida al sistema operativo
```

Subrutinas anidadas

En el ejemplo anterior vimos un sencillo esquema que vale para un sólo nivel de subrutinas, es decir, dentro de subrut no podemos llamar a otra subrutina porque sobreescribimos el valor del registro **lr**. La solución para extender a cualquier número de niveles es almacenar el registro **lr** en pila con las instrucciones **push** y **pop**.

```
    main :   mov r1, #1
             mov r2, #2
             bl nivel1       @ No es necesario guardar lr
             mov r5, #5      /* Siguiente instrucción */
             ...
             /* ---- Subrutinas ---- */
    nivel1 : push {lr}       @ como esta funcion llama a otra
             mov r3, #3      @ se guarda lr, pq sino se pierde
             bl nivel2       @ llama a la funcion anidada
             pop {lr}        @ restauramos el lr de esta funcion
             bx lr
             /* ------------ */
    nivel2 : mov r4, #4      @ no es necesario guardar lr pq
             bx lr           @ es la ultima funcion
```

Vemos en el último nivel (nivel2) podemos ahorrarnos el tener que almacenar y recuperar lr en la pila.

Las instrucciones de salto en la arquitectura ARM abarcan una zona muy extensa, hasta 64 Mb (32 Mb hacia adelante y otros 32 Mb hacia atrás). Este rango esta determinado por los 24 bits para codificar el destino del salto. En caso de necesitar un salto mayor recurrimos a la misma solución de la carga de inmediatos del mov, solo que el registro a cargar es el pc.

```
   ldr pc, =etiqueta
```

Volver