B

INTRODUCCIÓN AL LENGUAJE ENSAMBLADOR DE SPARC

B.1. Introducción

En este apéndice se estudiarán las características específicas del lenguaje ensamblador para los procesadores con arquitectura SPARC.

Aunque este ensamblador se asemeja mucho a otros, tiene algunas características específicas; por ejemplo, en el ensamblador de SPARC las constantes inmediatas no requieren ningún símbolo específico y las referencias a memoria van entre corchetes, tanto si provienen de etiquetas como de registros.

B.2. Símbolos y expresiones

Se denominan **símbolos** a los nombres que representan a diversas entidades. Estos nombres se materializan en cadenas de caracteres con las limitaciones usuales en otros lenguajes de programación (no comenzar por un número, no contener ciertos caracteres especiales, etc.).

Según una primera clasificación, los símbolos pueden ser:

Permanentes: son símbolos que representan en todos los programas a las mismas entidades. De este tipo son los nombres de los registros y los operadores, estos símbolos se caracterizan por comenzar por el carácter %.

Definidos por el usuario: son símbolos que tienen una significación específica en cada programa: constantes, direcciones, nombres de procedimientos, etc.

Los símbolos definidos por el usuario toman valor cuando el programa se compila y en el momento de la ejecución quedan sustituidos por el valor al que representan que es inalterable.

La forma de asignar valores a los símbolos se efectúa de forma distinta en función de lo que representen:

1. Pueden ponerse como una etiqueta en algún punto del programa, en este caso el símbolo representa la dirección de la instrucción o dato que vaya a continuación. Las etiquetas se caracterizan por ir al principio de la línea y seguidas por dos puntos (:).

   Para asignar valor a este tipo de símbolos el ensamblador lleva una variable interna denominada *location counter* (LC) (contador de dirección). Este contador se inicializa a 0 al comenzar la compilación y se incrementa por el número de bytes necesarios para cada instrucción máquina traducida o cada área reservada para datos. Dicho de otra forma, el LC lleva un control de la dirección de memoria de cada dato o instrucción pero relativa al principio del programa. Cuando aparece una etiqueta el compilador le asigna el valor que tenga LC en ese momento. Este valor será una referencia, ya que el valor real de la dirección representada por las etiquetas será el valor del LC en el momento de compilar esa línea más la dirección de comienzo de programa. A los símbolos asignados de esta forma se les denomina **símbolos relocalizables**.

2. Pueden asignarse también mediante el signo =, en este caso el símbolo toma el valor de lo que haya a la derecha del signo = que debe ser constante. Después del signo = puede haber algo más que una simple constante numérica, es decir, puede aparecer una **expresión**.

Una **expresión** en una combinación de constantes y símbolos relacionados mediante operadores aritméticos entre los cuales los más usuales son: +, -, *, / (división entera). También existen, como operadores unitarios, el - y los siguientes:

`%uhi`: Extrae los bits 63 a 42 del argumento

`%ulo`: Extrae los bits 41 a 32 del argumento

`%hi`: Extrae los bits 31 a 10 del argumento

`%lo`: Extrae los bits 9 a 0 del argumento

Para estos operadores el argumento se pone entre paréntesis. Estos operandos son especialmente útiles para la instrucción `sethi` que sirve para cargar los 22 bits más altos de un registro.

Uno de los términos que pueden aparecer en las expresiones es el valor del LC en el momento de compilarse la expresión que se representa por un punto (.). Esto es muy habitual cuando se quiere tener en una constante la longitud de una cadena ya que se toma el valor del LC después de reservar su espacio y se le resta el símbolo que represente la dirección de comienzo de la cadena. La base en que se interpretan las constantes sigue las mismas convenciones que el lenguaje C.

Es necesario dejar bien claro que **las expresiones que dan valores a los símbolos se evalúan cuando el programa se compila** y por tanto todos los términos que en ellas aparecen deben estar previamente definidos. Por ello estas expresiones y las que aparecen en los lenguajes de alto nivel no tienen casi nada en común.

B.2.1. Tipos de símbolos

Abundando en lo anteriormente expuesto, los símbolos pueden clasificarse en función de su valor en dos clases:

Absolutos: Son símbolos cuyo valor es constante independientemente del lugar de memoria donde se sitúe el programa.

Relocalizables: Son símbolos cuyo valor depende de la posición del programa en memoria.

Esta clasificación es aplicable también a las expresiones.

Otra clasificación afecta a la **visibilidad** del símbolo, es decir el ámbito donde el símbolo es reconocido. Atendiendo a este criterio los símbolos se pueden clasificar en:

Locales: Sólo se reconocen en el fichero fuente actual que normalmente contendrá uno o varios procedimientos.

Externos o **referencias no resueltas**: Son símbolos que no están definidos en el programa, normalmente corresponden a procedimientos o constantes que radican en librerías. Estos símbolos son localizados por el montador que es quien al final da valores a estos símbolos.

B.3. Tipos de instrucciones

En el ensamblador de SPARC existen 3 tipos de instrucciones:

Directivos, que son órdenes dirigidas específicamente al compilador, en la nomenclatura de SPARC a los directivos también se les llama **pseudo-operaciones**. Se caracterizan porque todos ellos comienzan con un punto. Evidentemente, los directivos pueden cambiar según el compilador de ensamblador que se emplee, su versión, etc.

Instrucciones máquina, son las instrucciones que se traducen literalmente a código máquina. Estas instrucciones son la que aparecen en el juego de instrucciones de la máquina (ver apéndice A). Hay que tener en cuenta que el compilador de ensamblador no reordena las instrucciones para las bifurcaciones retardadas, ni para evitar las dependencias de datos; por ello, estas reordenaciones hay que hacerlas a mano. También hay que mencionar que, en el lenguaje ensamblador de SPARC, el operando destino siempre es el último.

Instrucciones sintéticas, son instrucciones similares a las de la máquina pero que no existen en realidad. Es el compilador el que traduce cada instrucción sintética por otra equivalente a partir del conjunto de instrucciones máquina. Estas instrucciones existen para facilitar la programación y la legibilidad de los programas. Evidentemente, las instrucciones sintéticas pueden depender del compilador de ensamblador que se utilice, sin embargo suele haber bastante consenso en su utilización. En la tabla B.1 se muestran las instrucciones sintéticas empleadas en el ensamblador para arquitecturas SPARC.

También, como en otros lenguajes, existe la posibilidad de incorporar **comentarios**, esto puede hacerse de dos formas:

- Para los comentarios que ocupan varias líneas, puede emplearse la sintaxis del lenguaje C, es decir, abriendo el comentario con `/*` y cerrándolo con `*/`.
- Si el comentario ocupa una línea, o sólo su parte final, basta con iniciarlo con el signo `!`.

Tabla B.1. Instrucciones sintéticas del lenguaje ensamblador de SPARC (continúa).

Instrucción sintética	Equivalencia	Comentarios
`mov` a, rd	`or %g0,` a, rd	$a \rightarrow rd$ [1]
`clr` rd	`or %g0, %g0,` rd	Pone el registro a 0
`clrb` [dirección]	`stb %g0,` [dirección]	Pone un byte de memoria a 0
`clrh` [dirección]	`sth %g0,` [dirección]	Pone media palabra de memoria a 0
`clr` [dirección]	`stw %g0,` [dirección]	Pone una palabra de memoria a 0
`clrx` [dirección]	`stx %g0,` [dirección]	Pone una doble palabra de memoria a 0
`clruw` $rs1$, rd	`srl` $rs1$`, %g0,` rd	Borra la palabra alta de $rs1$ y queda en rd
`clruw` rd	`srl` rd`, %g0,` rd	Borra la palabra alta
`inc` rd	`add` rd`, 1,` rd	Incrementa en 1
`inc` inmed., rd	`add` rd, inmed., rd	Incrementa en una constante
`inccc` rd	`addcc` rd`, 1,` rd	Incrementa en 1 cambiando los *flags*
`inccc` inmed, rd	`addcc` rd, inmed., rd	Incrementa en constante cambiando *flags*
`dec` rd	`sub` rd`, 1,` rd	Decrementa en 1
`dec` inmed., rd	`sub` rd, inmed., rd	Decrementa en una constante
`deccc` rd	`subcc` rd`, 1,` rd	Decrementa en 1 cambiando los *flags*
`deccc` inmed., rd	`subcc` rd, inmed., rd	Decrementa en constante cambiando *flags*
`not` $rs1$, rd	`xnor` $rs1$`, %g0,` rd	Complementa a 1 $rs1$ y queda en rd
`not` rd	`xnor` rd`, %g0,` rd	Complementa a 1
`neg` $rs1$, rd	`sub %g0,` $rs1$`,` rd	Complementa a 2 rs1 y queda en rd
`neg` rd	`sub %g0,` rd`,` rd	Complementa a 2
`signx` rd	`sra` rd`, %g0,` rd	Extiende el signo de 32 a 64 bits
`signx` $rs1$, rd	`sra` $rs1$`, %g0,` rd	Extiende signo de $rs1$ a rd (32 a 64 bits)
`btst` a, $rs1$	`andcc` $rs1$`,` a`, %g0`	Análisis de bit (*bit test*) [1]
`bset` a, rd	`or` rd`,` a`,` rd	Puesta a 1 de bit (*bit set*) [1]
`bclr` a, rd	`andn` rd`,` a`,` rd	Puesta a o de bit (*bit clear*) [1]
`btog` a, rd	`xor` rd`,` a`,` rd	Complemento de bit (*bit toggle*) [1]
`tst` $rs1$	`orcc %g0,` $rs1$`, %g0`	Prueba de un operando (cambia *flags*)
`cmp` $rs1$, a	`subcc` $rs1$`,` a`, %g0`	Comparación (cambia *flags*) [1]
`jmp` dirección	`jmpl` dirección`, %g0`	Salto ordinario
`call` dirección	`jmpl` dirección`, %o7`	(Difiere de la instrucción call en el modo de direccionamiento de la dirección del procedimiento)
`ret`	`jmpl %i7+8, %g0`	Retorno de subrutina
`retl`	`jmpl %o7+8, %g0`	Retorno de subrutina final
`restore`	`restore %g0, %g0, %g0`	Restore ordinario
`iprefetch` etiqueta	`bn,a,pt %xcc` etiqueta	Carga instrucción en caché

[1] En estas instrucciones a puede ser un registro o una constante inmediata.

Tabla B.1. Instrucciones sintéticas del lenguaje ensamblador de SPARC (conclusión).

Instrucción sintética	Equivalencia	Comentarios
`setuw` $valor, rd$ (o `set` $valor, rd$)	`sethi %hi(`$valor$`),` rd	$(valor \wedge$0x3FF$= 0)$
	`or %g0,` $valor, rd$	$(valor \in$[0, 4095]$)$
	`sethi %hi(`$valor$`),` rd; `or` rd`, %lo(`$valor$`),` rd	(otros casos)
`setsw` $valor, rd$	`sethi %hi(`$valor$`),` rd	$(valor \geq 0) \& (valor \wedge$ 0x3FF=0)
	`or %g0,` $valor, rd$	$(valor \in$[-4096, 4095])
	`sethi %hi(`$valor$`),` rd; `sra` rd`, %g0,` rd	$(valor < 0) \& (valor \wedge$ 0x3FF=0)
	`sethi %hi(`$valor$`),` rd; `or` rd`, %lo(`$valor$`),` rd	(otros casos con $valor \geq 0$)
	`sethi %hi(`$valor$`),` rd; `or` rd`, %lo(`$valor$`),` rd; `sra` rd`, %g0,` rd	(otros casos con $valor < 0$)
`setx` $valor, r, rd$	`sethi %uhi(`$valor$`),` r; `or` r`, %ulo(`$valor$`),` r; `sllx` r`, 32,` r; `sethi %hi(`valor`),` rd; `or` rd, r, rd; `or` rd`, %lo(`$valor$`),` rd	Transfiere una constante de 64 bits a un registro.

Nota:
Las instrucciones sintéticas que se sustituyen por varias instrucciones no deben situarse en un ciclo adelantado detrás de una instrucción de control de flujo.

B.4. Directivos del ensamblador

Los directivos del ensamblador de SPARC están bastante inspirados en los del MACRO-11 de DEC. Los clasificaremos según la función que realizan.

B.4.1. Principio del programa. Segmentos

Un programa en ensamblador de SPARC debe tener, al menos, dos segmentos: el de código y el de datos. El segmento de datos se identifica porque comienza por el directivo:

```
.data
```

Análogamente el segmento de código debe comenzar con el directivo:

```
.text
```

Si el programa se va a ejecutar directamente desde el sistema operativo (UNIX normalmente), deberá ponerse una etiqueta denominada `main` en el lugar donde se desee que comience la ejecución (dirección de transferencia). La dirección relocalizable representada por `main` es la que se cargará en el contador de programa cuando el sistema operativo le pase el control. Esta etiqueta tiene que declararse como global para que sea identificada desde fuera del programa. Esta declaración debe ponerse antes de la definición de la dirección. Esto se hace mediante el directivo:

```
.global main
```

Este directivo también debe usarse cuando se programan funciones o procedimientos a los que se pueda llamar desde fuera de nuestro programa, y en general, cuando un símbolo deba reconocerse desde el exterior del programa. En este caso, se pueden poner todos los símbolos afectados en un mismo directivo .global separándolos por comas.

B.4.2. Reserva de espacio en memoria

El ensamblador de SPARC permite reservar espacio por bloques con el directivo:

`.skip` n

donde n es el número de bytes que se quieren reservar. Este directivo causa que el LC se incremente en n.

Para reservar espacio inicializado de variables escalares utilizaremos los directivos `.byte`, `.half`, `.word`, `.xword`, `.single`, `.double` o `.quad` en función del tipo de dato que queramos inicializar: byte, media palabra (16 bits), palabra (32 bits), palabra extendida (64 bits), números en punto flotante en simple, doble o cuádruple precisión, respectivamente.

La sintaxis de estos directivos es la siguiente:

$$\left\{\begin{array}{l}\texttt{.byte}\\ \texttt{.half}\\ \texttt{.word}\\ \texttt{.xword}\\ \texttt{.single}\\ \texttt{.double}\\ \texttt{.quad}\end{array}\right\} v_1[, v_2, ...]$$

aquí v_1, v_2, etc. representan los valores a los que se inicializa la unidad de almacenamiento especificada. Se pueden especificar uno o más valores para la inicialización, reservándose tantas unidades del tipo especificado como valores se indiquen.

Las máquinas con arquitectura SPARC exigen alineación para todos los tipos de datos, lo que supone que antes de la mayoría de los directivos anteriores el LC debe estar alineado. Para conseguir la alineación basta incluir el directivo

`.align` n

que fuerza a que el LC sea múltiplo de n, dejando algunos bytes a 0.

La alineación del código también es necesaria, sin embargo, no hace falta emplear el directivo `.align` ya que `.text` alinea automáticamente la primera instrucción, y por tanto todas las demás, a 4.

Para reservar espacio para cadenas de caracteres podría emplearse el directivo .byte con los códigos ASCII de los caracteres de la cadena pero son más útiles los directivos `.ascii` y `.asciz` cuya sintaxis es la siguiente:

$$\left\{\begin{array}{l}\texttt{.ascii}\\ \texttt{.asciz}\end{array}\right\}\ \text{"cadena de caracteres 1" [,"cadena de caracteres 2"...]}$$

Ambos directivos guardan en bytes sucesivos los códigos ASCII de los caracteres que se indican. La diferencia entre ambos es que `.ascii` reserva espacio estrictamente para las cadenas de caracteres que se indican y `.asciz` hace lo mismo, pero terminando cada una de las cadenas con un byte nulo.

B.5. Operaciones de entrada y salida

Las operaciones de entrada y salida en ensamblador se pueden efectuar de forma sencilla mediante las funciones *putchar* y *getchar* (las mismas que se emplean en los programas escritos en lenguaje C).

Una llamada a *getchar* devuelve, en el byte más bajo del registro $o0$, el código ASCII del último carácter pulsado en el terminal *stdin*, aunque hay que pulsar la tecla de retorno de carro al final de la cadena que se introduzca.

Para escribir un carácter en la pantalla del terminal *stdout*, basta depositar, en el byte de orden más bajo del registro $o0$, el código ASCII del carácter que se quiera imprimir y llamar a la función *putchar*.

A partir de estas funciones se pueden construir otras que lean o que escriban cadenas completas, que lean o escriban números (para lo que habrá que hacer las conversiones pertinentes), etc.

B.6. Pasos para la ejecución de un programa

Para conseguir la ejecución de un programa escrito en lenguaje ensamblador de SPARC, después de editarlo, con la extensión "`.s`", hay que compilarlo con el siguiente comando:

```
gcc -g -Wa,-xarch=v8plus fichero_fuente -o fichero_ejecutable
```

Después de eso el fichero ya puede ejecutarse desde la línea de comandos.

```
LF=10                                   ! ASCII fin de línea (line feed)=10
        .data
mensaje:.ascii "Introduzca una cadena: "
L=.- mensaje
cadena: .skip 80
        .text
        .global main
main:
        save %sp,-64,%sp
/* Impresión del mensaje */
        sethi %hi(mensaje),%o0          ! o0 contiene la dirección de
        or %o0,%lo(mensaje),%o0         ! comienzo del mensaje
        or %g0, L,%o1                   ! o1 contiene su longitud
        call imprime_cadena
/* Lectura de la cadena */
        or %g0,%g0,%l1                  ! CICLO ADELANTADO A call
                                        ! Inicialización del índice
        sethi %hi(cadena),%l0           ! l0 contiene la dirección de
        or %l0,%lo(cadena),%l0          ! comienzo de la cadena
bucle1:
        call getchar                    ! Lee carácter y lo deposita en o0
        nop                             ! CICLO ADELANTADO
        stb %o0,[%l0+%l1]               ! Almacena carácter
        subcc %o0,LF,%g0                ! Comparación con LF (fin de cadena)
        bne bucle1                      ! Condición de final
        add %l1, 1,%l1                  ! CICLO ADELANTADO
/* Impresión de la cadena */
        or %g0,%l0,%o0                  ! o0 contiene la dirección de
                                        ! comienzo de la cadena
        or %g0,%l1,%o1                  ! o1 contiene su longitud
        call imprime_cadena
        nop                             ! CICLO ADELANTADO
        jmpl %i7+8,%g0                  ! Devuelve el control al S.O. (ret)
        restore %g0,%g0,%g0

imprime_cadena:                         ! en i0 tenemos la dirección de la cadena
                                        ! y en i1 su longitud
        save %sp, -64,%sp
        or %g0,%g0,%l1                  ! l1 es el índice
bucle:
        ldub [%i0 +%l1],%o0             ! Extrae el carácter de memoria
        call putchar                    ! Imprime el carácter contenido en o0
        add %l1, 1,%l1                  ! CICLO ADELANTADO
        subcc %i1, 1,%i1                ! Actualizan los índices
        bne bucle                       ! Condición de final del bucle
        nop                             ! CICLO ADELANTADO
        jmpl %i7+8,%g0                  ! Retorno
        restore %g0,%g0,%g0
```

Fig. B.1. Programa en lenguaje ensamblador de SPARC para leer una cadena de caracteres y escribirla.

```
LF=10                                   ! ASCII fin de línea (line feed)=10
        .data
mensaje:.ascii "Introduzca una cadena: "
L=.- mensaje
cadena: .skip 80
        .text
        .global main
main:
        save %sp,-64,%sp
/* Impresión del mensaje */
        set mensaje,%o0                 ! o0 contiene la dirección de
                                        ! comienzo del mensaje
        mov L,%o1                       ! o1 contiene su longitud
        call imprime_cadena
/* Lectura de la cadena */
        clr %l1                         ! CICLO ADELANTADO a call
                                        ! inicialización del índice
        set cadena,%l0                  ! l0 contiene la dirección de
                                        ! comienzo de la cadena
bucle1:
        call getchar                    ! lee carácter y lo deposita en o0
        nop                             ! CICLO ADELANTADO
        stb %o0,[%l0+%l1]               ! Almacena carácter
        cmp %o0,LF                      ! LF=fin de cadena
        bne bucle1                      ! Condición de final
        inc %l1                         ! CICLO ADELANTADO
/* Impresión de la cadena */
        mov %l0,%o0                     ! o0 contiene la dirección de
                                        ! comienzo de la cadena
        mov %l1,%o1                     ! o1 contiene su longitud
        call imprime_cadena
        nop                             ! CICLO ADELANTADO
        ret                             ! Devuelve el control al S.O.
        restore

imprime_cadena:                         ! en i0 tenemos la dirección de la cadena
                                        ! y en i1 su longitud
        save %sp, -64,%sp
        clr %l1                         ! l1 es el índice
bucle:
        ldub [%i0 +%l1],%o0             ! Extrae el carácter de memoria
        call putchar                    ! Imprime el carácter contenido en o0
        inc %l1                         ! CICLO ADELANTADO
        deccc %i1                       ! Actualizan los índices
        bne bucle                       ! Condición de final del bucle
        nop                             ! CICLO ADELANTADO
        ret
        restore
```

Fig. B.2. Programa en lenguaje ensamblador de SPARC para leer una cadena de caracteres y escribirla. Este programa hace uso de instrucciones sintéticas

B.7. Ejemplo de programación

En esta sección veremos un ejemplo de programa completo. El ejemplo que proponemos pide una cadena por teclado y luego la imprime por pantalla. El programa está autodocumentado para facilitar su comprensión. En una primera versión no emplearemos las instrucciones sintéticas (figura B.1). En la versión más evolucionada de la figura B.2 se muestra el mismo programa pero utilizando instrucciones sintéticas. Como puede verse, esta última versión es muchísimo más legible que la anterior.

Bibliografía y referencias

PAUL, R.P. 2000. *SPARC Arquitecture, Assembly Language Programming & C*. 2 edn. Prentice-Hall.

WEAVER, D.L., & GERMOND, T. (EDITORES). 1994. *The SPARC Architecture Manual, version 9*. Prentice-Hall International.

CUESTIONES Y PROBLEMAS

B.1 Escribir una versión del programa de ejemplo citado en el texto que efectúe la lectura de la cadena por teclado mediante una función. Esta función debe depositar la cadena leída a partir de la dirección de memoria especificada en su registro $i0$ ($o0$ para los programas que la llamen) y devolver la longitud de la cadena leída en su registro $i1$ ($o1$ para los programas que la llamen).

B.2 Escribir una nueva versión de la función citada en el problema anterior que almacene en memoria la cadena terminada con un carácter nulo. El segundo parámetro ($i1$) debe tener una doble función: actuando como parámetro de entrada, recibirá el número máximo de caracteres que se espera en la cadena, y, como parámetro de salida, devolverá el número de caracteres realmente leído.

B.3 Escribir una función en lenguaje ensamblador de SPARC que escriba en la pantalla del terminal la cadena que se encuentra en memoria a partir de la dirección especificada en su registro $i0$ ($o0$ para los programas que la llamen). La función debe escribir caracteres hasta que se encuentre un carácter nulo.

B.4 Construir una versión de la función de escritura de cadena, mencionada en el problema anterior, en que la escritura de la cadena termine, bien cuando se encuentre un carácter nulo, o bien cuando se complete el número de caracteres especificado en el registro $i1$ (lo que ocurra antes).

B.5 Haciendo uso de algunas de las funciones escritas en los problemas anteriores, escribir un programa, en lenguaje ensamblador de SPARC, que lea una cadena de caracteres del teclado del terminal, la almacene en memoria y luego la escriba por pantalla pasada a mayúsculas.

B.6 Escribir una función, en lenguaje ensamblador de SPARC, que tome una cadena cuya dirección se especifique en el registro $i0$ ($o0$ para los programas que la llamen), la interprete como un número en hexadecimal y deposite ese número en el registro $i1$ ($o1$ para los programas que la llamen).

B.7 Escribir una función, en lenguaje ensamblador de SPARC, que escriba a partir de la dirección dada por el registro $i1$, una cadena que represente el valor del contenido de su registro $i0$ ($o0$ para los programas que la llamen) en hexadecimal.

B.8 Utilizando las funciones de los problemas anteriores, escribir un programa en lenguaje ensamblador de SPARC que pida por teclado un número hexadecimal y escriba por la pantalla, también en hexadecimal, el doble de ese número.

B.9 Escribir una función en ensamblador de SPARC con dos parámetros. La función debe extraer de la cadena que comienza en la dirección especificada por el primer parámetro un número, interpretarlo en decimal y devolver el valor obtenido en el segundo parámetro. Se puede suponer que la cadena especificada por el primer parámetro sólo contiene caracteres numéricos.

B.10 Escribir una función en ensamblador de SPARC con dos parámetros. La función debe devolver, en la dirección especificada por el primer parámetro, una cadena que contenga el valor del segundo parámetro escrito en decimal.

B.11 Empleando algunas de las funciones anteriores escribir un programa en lenguaje ensamblador de SPARC que pida por teclado un número en hexadecimal y lo escriba por pantalla en decimal.

B.12 Empleando algunas de las funciones anteriores escribir un programa en lenguaje ensamblador de SPARC que pida por teclado un número en decimal y lo escriba por pantalla en hexadecimal.

B.13 Escribir un programa, en lenguaje ensamblador de SPARC, que pida por teclado un número en hexadecimal, lo almacene en un registro, y escriba por pantalla en decimal el valor del contenido de su byte de orden más bajo.