Práctica 13. Llamadas al sistema e instrucciones de transferencia de datos

Objetivos

- Conocer las llamadas al sistema.
- Conocer las instrucciones de transferencia de datos de RISC-V.

1. Introducción

En la práctica anterior vimos cómo inicializar los registros con la instrucción `li` pero lo normal es que esa información la introduzca el usuario. Igualmente, los resultados queremos mostrarlos por pantalla. Para todo ello se utilizan las **llamadas al sistema**.

Además, en esta práctica también abordamos las directivas del ensamblador que permiten indicar dónde ponemos las instrucciones (**segmento de texto**) y dónde van los datos (**segmento de datos**).

Para acceder a los datos que están en memoria (especificados en el segmento de datos) se utilizan las instrucciones de carga (leen datos de memoria y los guardan en un registro) y de almacenamiento (guardan datos de la memoria en un registro).

2. Instrucciones de carga y almacenamiento

Para hacer referencia al contenido del segmento de datos, se necesita un conjunto específico de instrucciones de acceso a memoria. RISC-V tiene una arquitectura de tipo *load/store* (carga/almacenamiento), que significa que sólo ciertas instrucciones específicas pueden acceder a la memoria: las de carga (cogen un dato de la memoria y lo transfieren a un registro) y almacenamiento

(guardan en memoria un dato contenido en un registro). El resto de las instrucciones del repertorio se limitan a operar con valores guardados en registros o bien con valores inmediatos.

2.1. Instrucciones de carga

- **Carga un byte en un registro sin signo:** `lbu rd, offset(rs1)` Carga en el registro `rd` el byte sin signo que está en la dirección indicada por el registro `rs1` más el `offset` con el signo extendido.
- **Carga un byte en un registro con signo:** `lb rd, offset(rs1)` Carga en el registro `rd` el byte con signo que está en la dirección indicada por el registro `rs1` más el `offset` con el signo extendido.
- **Carga un half en un registro sin signo:** `lhu rd, offset(rs1)` Carga en el registro `rd` el half sin signo que está en la dirección indicada por el registro `rs1` más el `offset` con el signo extendido.
- **Carga un half en un registro con signo:** `lh rd, offset(rs1)` Carga en el registro `rd` el half con signo que está en la dirección indicada por el registro `rs1` más el `offset` con el signo extendido.
- **Carga un word en un registro con signo:** `lw rd, offset(rs1)` Carga en el registro `rd` el word con signo que está en la dirección indicada por el registro `rs1` más el `offset` con el signo extendido.

2.2. Instrucciones de almacenamiento

- **Almacena un byte en memoria:** `sb rs2, offset(rs1)` Almacena el byte menos significativo del registro `rs2` en la memoria, en la dirección indicada por el registro `rs1` más el `offset` con el signo extendido.
- **Almacena un half en memoria:** `sh rs2, offset(rs1)` Almacena el half menos significativo del registro `rs2` en la memoria, en la dirección indicada por el registro `rs1` más el `offset` con el signo extendido.
- **Almacena un word en memoria:** `sw rs2, offset(rs1)` Almacena el word del registro `rs2` en la memoria, en la dirección indicada por el registro `rs1` más el `offset` con el signo extendido.

3. Directivas del ensamblador

El segmento de datos se define como la zona de la memoria de la máquina donde se almacenan los datos que necesita un programa, en contraposición al segmento de texto en donde están las instrucciones.

Las directivas de ensamblador (`.data`, `.globl main`, `.text`) no son instrucciones del microprocesador, sino que sirven para indicar ciertos parámetros de la ejecución del programa, definir el comienzo de los segmentos de datos y texto, o declarar datos en el segmento de datos, principalmente.

Utilizaremos las siguientes directivas:

3.1. Directivas de declaración de segmentos

`.text` Indica el comienzo del segmento de texto. En el segmento de texto se encontrarán únicamente instrucciones del microprocesador.

`.data` Indica el comienzo del segmento de datos, y por tanto todas las directivas que aparezcan a continuación, y hasta que se indique el comienzo de otro segmento, serán directivas de declaración de datos (`.byte`, `.half`, etc.).

3.2. Directivas de etiqueta global

`.globl` Declara una etiqueta global (accesible desde otros archivos). Ejemplo:

```
.globl main
```

3.3. Directivas de alineación de datos

`.align n` Alinea el siguiente dato a una dirección múltiplo de 2^n (0 byte, 1 half, 2 word, 3 double).

3.4. Directivas de declaración de datos (en el segmento de datos)

`.byte` Declara datos de tipo byte (8 bits). Ejemplo: `.byte 3, -2, 8, 'g'`

`.half` Declara datos de tipo halfword (16 bits). Ejemplo: `.half 3, -2, 8`

`.word` Declara datos de tipo word (32 bits). Ejemplo: `.word 3, -2, 8`

`.float` Declara datos de tipo float (32 bits). Ejemplo: `.float 3.3, -2.2`

.double Declara datos de tipo double (64 bits). Ejemplo: `.double 3.3, -2.2`

.string Declara una cadena de caracteres y añade el \0 al final. Ejemplo:
```
.string "Cadena de caracteres"
```

.zero n Reserva n bytes de memoria consecutivos en el segmento de datos con el valor 0. Esta directiva es útil cuando se quiere reservar espacio en la memoria donde almacenar, por ejemplo, resultados de operaciones, cadenas de texto introducidas por el usuario, etc.

Con respecto a las directivas de declaración de datos, caben múltiples posibilidades para llevar a cabo la misma tarea. Naturalmente, es decisión del programador hacer las cosas de uno u otro modo en función de su propio criterio. Así, si se desea almacenar una cadena en el segmento de datos se puede optar por emplear la directiva `.string`:

```
.string "La suma es \n"
```

Pero también se puede almacenar cada carácter individualmente como un byte:

```
.byte 'L','a',' ','s','u','m','a',' ','e','s','\n','\0'
```

Incluso se puede almacenar cada carácter como un byte pero utilizando su valor ASCII:

```
.byte 76, 97, 32, 115, 117, 109, 97, 32, 101, 115, 10, 0
```

4. Etiquetas

Las etiquetas son identificadores que se sitúan al principio de una línea y van seguidos de dos puntos (por ejemplo, `main:`). Sirven para hacer referencia a la posición o dirección de memoria del elemento definido en ésta, bien sea una instrucción del segmento de texto o un dato del segmento de datos. A lo largo del programa se puede hacer referencia a ellas en las instrucciones de acceso a memoria (para etiquetas definidas en el segmento de datos) o en las instrucciones de salto (para etiquetas definidas en el segmento de texto).

La pseudoinstrucción `la rd, direccion` (*load address*) carga en el registro `rd` la dirección especificada a continuación. Carga tan sólo la dirección, no su contenido. Una dirección de memoria responde al concepto ya conocido de puntero. Esta dirección puede venir especificada en múltiples formatos, siendo el más sencillo una etiqueta.

5. Llamadas al Sistema Operativo

El simulador CREATOR proporciona ciertos servicios a través de la emulación de llamadas al Sistema Operativo con la instrucción `ecall`. Para realizar una llamada, es necesario primero cargar un código de llamada (que indicará si se quiere imprimir un carácter, salir del programa, etc.) en el registro `a7`.

Si la llamada necesita un argumento, éste deberá estar en el registro `a0`. Si tiene dos argumentos, irán en los registros `a0` y `a1`. Las llamadas al sistema que necesiten algún argumento en coma flotante, lo recibirán en el registro `fa0`.

Algunas llamadas al sistema devuelven un resultado, y lo hacen en el registro `a0` o bien en `fa0` si el resultado es de coma flotante. Por ejemplo, la llamada que lee un número introducido por el usuario mediante el teclado almacena este número en el registro `a0`. Tras finalizar la llamada al sistema es responsabilidad del programador llevarse este número a otro registro o a la memoria para usos posteriores.

Nombre	Código (`a7`)	Entradas	Salidas	Descripción
print_int	1	`a0` entero		Imprime un entero
print_float	2	`fa0` real		Imprime un real de precisión simple
print_double	3	`fa0` real		Imprime un real de precisión doble
print_string	4	`a0` dirección de la cadena		Imprime una cadena
read_int	5	no tiene	`a0` tiene el entero leído	Lee un entero
read_float	6	no tiene	`fa0` tiene el real leído	Lee un real de precisión simple
read_double	7	no tiene	`fa0` tiene el real leído	Lee un real de precisión doble
read_string	8	`a0` dirección de la cadena leída `a1` máximo número de bytes a leer		Lee una cadena

Tabla 1: Llamadas al sistema de CREATOR

Estos servicios que se muestran en la tabla 1 tienen el siguiente significado:

- **print_int:** se pasa como argumento un entero (en el registro `a0`), el cual se imprime en la consola del simulador (panel inferior).
- **print_float:** se imprime un número real en la consola del simulador
- **print_double:** se imprime un número real con precisión doble en la consola del simulador.
- **print_string:** se imprime una cadena de caracteres residente en memoria y terminada en '\0'; para ello habrá que pasar la dirección donde comienza la cadena. Se debe tener en cuenta que el tratamiento de símbolos especiales (intro, tabulación, ...) sigue la notación de C ('\n', '\t', ...).
- **read_int**, **read_float**, **read_double**: lee de la consola una línea completa e ignora todos los caracteres posteriores al número.

- **read_string:** lee una cadena de n-1 caracteres (siendo n el valor de `a1`), los completa con '\0' y los guarda en la dirección de memoria indicada en `a0`.
- **sbrk:** devuelve un puntero a un bloque de memoria del *heap* que contiene n bytes (*malloc*).
- **exit:** finaliza el programa

6. Pseudoinstrucciones

Son instrucciones que podemos usar en el ensamblador pero que no se implementan, el ensamblador se encarga de traducirlas a instrucciones que sí entiende el procesador.

- Mover contenido de registros: `mv rd, rs` Copia el contenido de `rs` en `rd` (manteniendo el valor original en `rs`).
- Dar un valor a un registro: `li rd, inmediate` Guarda el valor inmediate en el registro `rd`.
- Crear puntero: `la rd, direccion` (*load address*) carga en el registro `rd` la dirección especificada a continuación. Carga tan sólo la dirección, no su contenido.

7. Ejemplos

7.1. Ejemplo de uso de instrucciones de carga/almacenamiento

Un ejemplo de uso de instrucciones de carga/almacenamiento es el siguiente:

```
1       .data # Zona de memoria de la máquina donde se almacenan los
      datos
2   datos:     .word 34, -23
3              .half 8
4       .globl main # Directiva de etiqueta global
5       .text # Inicio del segmento de texto (instrucciones)
6   main: #etiqueta main
7     la t0, datos
8     lw t1, (t0)   #t1 vale 34
9     lw t2, 4(t0)  #t2 vale -23
```

```
10      lh t3, 8(t0)  #t3 vale 8
11
12      add t3, t3, t2  #t3 = t3+t2
13
14      sw t3, 4(t0)  #Guarda el resultado donde antes estaba -23
15
16      li a7,10 # Llamada al sistema de tipo "exit"
17      ecall
```

1. Abre el simulador.
2. Escribe el programa anterior.
3. Ensambla el programa. Para ello, hay que utilizar el botón *Compile/Linked*.
4. Clicando en el botón *Simulator* veremos cómo se han cargado en memoria las instrucciones de nuestro programa (la primera está en la dirección `0x0`). Como las instrucciones ocupan 4 bytes, ocupan las posiciones `0x0`, `0x4`, etc. Además, en la columna de la derecha (`Loaded instructions`) vemos la instrucción en lenguaje ensamblador y una columna más a la izquierda (`User Instruction`) vemos la instrucción de nuestro programa. En el caso de que se trate de una pseudoinstrucción veremos que son diferentes. Si visualizamos el segmento de texto de la memoria, nos aparece la instrucción en lenguaje máquina (en hexadecimal).
5. Ejecuta el programa mediante el botón *Run* o el botón *Instr.*.
6. Comprueba que al ejecutar paso a paso los registros y la memoria toman los valores correspondientes.

7.2. Ejemplo de uso de llamadas al sistema

El siguiente programa pide por la consola un número y a continuación lo muestra elevado al cuadrado:

```
1       .data # comienzo del segmento de datos
2
3   cad1: .string "Introduzca un número" #Forma de declarar una
      cadena
4   cad2: .string "El resultado es: " #Forma de declarar una cadena
5
6       .globl main # la etiqueta main es global
7
8       .text # comienzo del segmento de texto (instrucciones)
9
10  main: # Comienza la función main
```

```
11
12      li a7, 4 # código de la llamada print_string
13      la a0, cad1 #imprime la cadena que está en la dirección a0
14      ecall
15
16      li a7, 5 # código de la llamada read_int
17      ecall # ahora se introduce un número en la consola
18
19      mv t0, a0 # se guarda el contenido de a0 (el número) en t0
20
21      mul t1,t0,t0 # se eleva al cuadrado
22
23      li a7, 4 # código de la llamada print_string
24      la a0, cad2 #imprime la cadena que está en la dirección $a0
25      ecall
26
27      li a7,1 # código de llamada print_int
28      mv a0, t1 # a0 = número que se quiere imprimir
29      ecall # se realiza la llamada
30
31      li a7, 10 # código de llamada exit
32      ecall # llamada al sistema
```

1. Abre el simulador.
2. Escribe el archivo.
3. Ensambla el programa.
4. Comprueba cómo se han cargado en memoria las instrucciones de nuestro programa.
5. Ejecuta el programa.
6. Comprueba que al finalizar la ejecución del programa el valor del registro `t1` es el cuadrado del número y en `t0` tienes el número que leíste.

8. Ejercicio

8.1. Ejercicio

Utilizando como guía la descripción de las instrucciones anteriores, se proponen los siguientes ejercicios. Hay que tener siempre presente que con el simulador se puede utilizar el modo de ejecución paso a paso y emplear *breakpoints* para analizar detalladamente la ejecución de un programa, permitiendo detectar errores en la programación.

Se recuerda que la estructura general de un programa presenta un aspecto como el siguiente:

```
.data # Zona de memoria de la máquina donde se almacenan los
 datos
#####################################################
# Todos los datos necesarios en el programa
#####################################################
.globl main # Directiva de etiqueta global
.text # Inicio del segmento de texto (instrucciones)
main: #etiqueta main
#####################################################
#Zona de memoria de la máquina donde se almacenan las
#instrucciones de nuestro programa.
# Aquí va el código de nuestros ejercicios.
#####################################################
li a7,10 # Llamada al sistema de tipo "exit"
ecall
```

Realice un programa que declare dos datos de tipo *word* en el segmento de datos con valores 3 y 7. Además tiene que leer cuatro enteros A, B, C y D por pantalla y que muestre lo siguiente:

- A+B
- C-D
- A+B+C+D
- Mostrar A al cubo
- A $<<$ 2
- Leerá los dos enteros del segmento de datos e imprimirá el resto de dividir 7 entre 3.