Facultad de Informática
Universidad Complutense de Madrid

Fundamentos de Computadores II - Práctica 2
Trabajando con la memoria

Daniel Báscones (danibasc@ucm.es)

22 de febrero de 2023

1. Objetivos

Una vez familiarizados con el repertorio de instrucciones de RISC-V y con el entorno de desarrollo, en esta práctica se persiguen los siguientes objetivos:

- Aprender a realizar accesos a memoria.
- Comprender cómo utilizarlos para acceder a vectores, partiendo de la dirección base y el índice.
- Entender las estructuras de control anidadas, como bucles y condicionales.

2. Accesos a memoria en RISC-V

Los accesos a memoria son esenciales en cualquier arquitectura: permiten disponer de espacio práctiamente ilimitado, más allá de los escasos registros (32 para RISC-V) de los que disponemos. Su gestión se realiza a través de las instrucciones de *load* (carga) y *store* (guardado).

El funcionamiento de todas estas instrucciones es similar: para generar la dirección de memoria, se indican un registro *base* y un *desplazamiento* (que puede ser cero). La *dirección efectiva* es el resultado de sumar el desplazamiento al contenido del registro indicado (*dirección base*).

Una vez obtenida la dirección efectiva, una instrucción de *load* leerá el valor almacenado en dicha dirección y lo cargará en el registro *destino*. Una instrucción de *store* guardará en esta dirección de memoria el valor almacenado en el registro *fuente*. Por ejemplo:

```
lw s1, 16(a0) //base: a0  desplazamiento: 16  destino: s1
              //s1 = MEM[a0 + 16]
sw a0,  0(s3) //base: s3  desplazamiento: 0   fuente:  a0
              //MEM[s3 + 0] = a0
```

NOTA IMPORTANTE: Hay que tener en cuenta que la memoria se direcciona por bytes. Es decir, que cada dirección consecutiva: `0x10000`, `0x10001`, `0x10002`... contiene únicamente un byte. Trabajaremos con instrucciones que generalmente mueven 4 bytes a la vez. Cuando utilicemos estas instrucciones, únicamente utilizamos una dirección, por ejemplo `0x10000`. Si guardamos 4 bytes en dicha dirección (por ejemplo un registro completo), se guardarán de menos a más significativo en las direcciones `0x10000` hasta `0x10003`, ambas inclusive.

Existen varias instrucciones para los accesos a memoria:

- **Carga de palabra (*load word*)**: Esta instrucción carga una palabra completa de memoria (En nuestra arquitectura de 32 bits, se corresponde con 4 bytes):

  ```
  lw rd, imm(rs1)
  ```

  Con dicha instrucción, cargaremos en el registro destino `rd` el valor situado en la posición de memoria resultante de sumar `imm` al valor contenido en `rs1`.

  Ejemplo: Imaginemos que el contenido de `s1` es `0x10000`, y que en la dirección de memoria `0x10004` tenemos guardado el valor `0xbaca10ca`. Tras ejecutar la instrucción:

  ```
  lw a0, 4(s1)
  ```

  Tendremos el valor `0xbaca10ca` en el registro `a0`.

- **Guardado de palabra (*store word*)**: Esta instrucción guarda una palabra completa en memoria (de nuevo, 4 bytes):

  ```
  sw rs2, imm(rs1)
  ```

  Observamos una diferencia clave con la instrucción de *load*. En lugar de utilizar un registro como destino, utilizamos dos registros como fuentes (uno para el valor y otro para la dirección), ya que **no modificamos el banco de registros**. Con esta instrucción, guardaremos el valor situado en `rs2`, en la posición de memoria resultante de sumar `imm` con el valor contenido en `rs1`.

  Ejemplo: Imaginemos que el contenido de `s1` es `0x10000`, y que el registro `s3` contiene el valor `0xc0c010c0`. Tras ejecutar la instrucción:

  ```
  sw s3, 16(s1)
  ```

  Tendremos el valor `0xc0c010c0` guardado en la dirección de memoria `0x10010`. (Nótese que `0x10=16`).

3. Pseudo-instrucción LA

Las instrucciones de acceso a memoria requieren una dirección. Cuando declaramos un símbolo en ensamblador (para nombrar por ejemplo una variable), lo definimos mediante una palabra:

```
var: .word 27
```

Será el compilador quien le asigne una dirección de memoria a la variable `var` de valor 27. Si queremos conocer esta dirección y guardarla en un registro, bastará con utilizar la instrucción:

```
la s1, var
```

Que cargará, en el registro `s1`, la dirección de memoria donde se almacena `var`. Si por ejemplo, queremos cargar el valor de `var`, haremos:

```
la s1, var
lw a0, 0(s1)
```

En caso de vectores, o estructuras más complejas, queda a cargo del programador el averiguar la dirección *efectiva* del elemento buscado. En el siguiente ejemplo leemos el elemento 2 del vector:

```
vec: .word 1, 6, 7, 9
la s1, vec    //en s1 tenemos la dirección base
lw a0, 8(s1)  //s1 + 8 nos da la dirección efectiva
//a0 contiene ahora un 7
```

¿Por qué se suma 8 si es el elemento 2? Tenemos que tener en cuenta que cada elemento ocupa 4 bytes. El elemento 0 del vector está en su dirección base. Al ocupar 4 bytes, el siguiente (elemento 1) del vector se sitúa 4 bytes por detrás. ¿El elemento 2? 8 bytes por detrás. En general, si quiero acceder a `vec[i]`, la dirección efectiva se calcula como:

$$dir(vec[i]) = dir(vec) + i * 4$$

Otra opción es incrementar la dirección base. Esto puede ser especialmente útil en accesos donde el desplazamiento no es conocido de antemano, sino dependiente del valor de un registro:

```
vec: .word 1, 6, 7, 9
la s1, vec     //en s1 tenemos la dirección base
li a0, 8       //a0 contiene el desplazamiento
add s1, s1, a0 //sumo desplazamiento a la dirección base
lw a0, 0(s1)   //s1 contiene la dirección efectiva
//a0 contiene ahora un 7
```

4. Desarrollo de la práctica

En esta práctica vamos a explorar el manejo de vectores. Para empezar, escribiremos un programa que sea capaz de encontrar el valor mínimo dentro de un vector de números enteros.

PISTA: Para calcular la dirección efectiva de `V[i]`, dispondremos de la dirección *base* y del índice *i*. La dirección *efectiva* se puede calcular como sigue:

```
//supongo que s1 contiene la dirección base de "V"
//supongo que t1 contiene el índice "i"
slli t2, t1, 2 //t2 contiene ahora (t1 << 2), igual a "t1*4"
add t3, s1, t2 //t3 contiene "s1 + t1*4", justo la dirección buscada
lw a0, 0(t3)   //cargo en a0 el valor de V[i]
```

a) Utilizando estas ideas, termina de codificar el siguiente código en ensamblador:

Código de alto nivel	Esquema de ensamblador

```
#define N 8
#define INT_MAX 65536
int V[N] = {-7,3,-9,8,15,-16,0,3};

int min = INT_MAX;
for (i = 0; i < N; i++) {
    if (V[i] < min)
        min = V[i];
}
```

```
.global main
.equ N, 8
.equ INT_MAX, 65536
.data
V:   .word -7,3,-9,8,15,-16,0,3

.bss
min: .space 4

.text
main:
    la s1, min
    li t0, INT_MAX
    sw t0, 0(s1)
//terminar de codificar
```

b) A continuación, vamos a aprovechar el código anteriormente escrito, para generar un algoritmo de ordenación que nos coloque los números de menor a mayor en un vector objetivo. Los números que se vayan colocando se borrarán del vector inicial, sustituyéndose por valores `INT_MAX`.

Código de alto nivel

```
#define N 8
#define INT_MAX 65536
int V[N] = {-7,3,-9,8,15,-16,0,3};
int W[N];

int min, index;

for (j = 0; j < N; j++) {
    min = INT_MAX;
    for (i = 0; i < N; i++) {
        if (V[i] < min) {
            min = V[i];
            index = i;
        }
    }
    W[j] = V[index];
    V[index] = INT_MAX;
}
```

Esquema de ensamblador

```
.global main
.equ N, 8
.equ INT_MAX, 65536
.data
V:   .word -7,3,-9,8,15,-16,0,3

.bss
W:   .space N*4
min: .space 4
ind: .space 4

.text
main:
    //codificar
```