ESTRUCTURA DE COMPUTADORES
Grado en Ingeniería Informática

Sesión de laboratorio número 9

VARIABLES Y PASO DE PARÁMETROS

Objetivos

- Entender y usar las instrucciones y pseudoinstrucciones para la lectura y escritura en la memoria principal.
- Utilizar funciones del sistema que permiten entrada y salida de cadenas de caracteres.
- Manipular direcciones y recorrer vectores.
- Crear procedimientos con argumentos de tipo puntero.

Bibliografía

- D.A. Patterson y J. L. Hennessy, *Estructura y diseño de computadores*, Reverté, capítulo 2, 2011.

Introducción teórica

Variables estáticas y directivas relacionadas

El ensamblador del MIPS ofrece estos recursos para describir las variables estáticas de un programa en la memoria:

- El segmento `.data` donde ubicar los datos en la memoria.
- La directiva `.space` permite reservar memoria del segmento de datos. Útil para declarar variables sin inicializar.
- Las directivas `.byte`, `.half`, `.word`, `.ascii` y `.asciiz` permiten definir variables e inicializarlas.

Instrucciones y pseudoinstrucciones de acceso a la memoria de datos

El juego de instrucciones del MIPS para lectura y escritura de datos en la memoria comprende ocho instrucciones:

unidad	restricciones sobre la dirección	lectura con extensión de signo	lectura sin extensión de signo	Escritura
byte	ninguna	`lb`	`lbu`	`sb`
halfword	múltiplo de 2	`lh`	`lhu`	`sh`
word	múltiplo de 4	`lw`		`sw`

Tabla 1. Las instrucciones de lectura y escritura de enteros en la memoria principal

Todas ellas son del formato I y en ensamblador se escriben de la forma **`op rt,D(rs)`**. **`D`** es un **desplazamiento** de 16 bits (con signo) que se suma al contenido del registro **base** **`rs`** para formar la dirección de memoria donde se lee o escribe. Esta manera de especificar la dirección permite acceder a memoria con diversas intenciones que referimos a continuación.

El **direccionamiento absoluto** se realiza a una palabra fija en la memoria de la que se conoce la posición *A*: en principio sólo sería necesario considerar la constante *A* como desplazamiento y el registro **`$zero`** como base. Con este propósito conviene utilizar las pseudoinstrucciones de la forma **`op rs,A`**, que se descomponen en las instrucciones máquina correspondientes cuando *A* ocupa más de 16 bits.

Por ejemplo, la pseudoinstrucción **`lw $rt,A`** permite expresar la carga de un registro con el valor de una variable en memoria ubicada en la posición que se ha etiquetado como **`"A"`**. Esta línea puede traducirse en una o más instrucciones máquina, dependiendo del valor de la etiqueta:

- Si **`A`** es un número expresable con 16 bits, la traducción es **`lw $rt,A($0)`.**
- Si **`A`** es demasiado grande para eso, el ensamblador la descompone en la parte alta **`Ah`** y la parte baja **`Al`**. Una traducción puede ser:

```
lui $at,Ah
lw $rt,Al($at)
```

Direccionamiento indirecto, cuando la posición de la variable está en un registro. Es la visión del programador cuando debe acceder a una dirección calculada por el programa, o para seguir un puntero (hablaremos de eso más abajo) o para recorrer variables estructuradas.

Direccionamiento relativo a registro, cuando un registro contiene una dirección de referencia y el programador piensa en desplazamientos respecto de ella. Este direccionamiento también se utiliza para acceder a variables estructuradas: el registro contiene la dirección de la variable y el desplazamiento es el correspondiente al campo al que se accede.

Punteros

Se llama puntero a cualquier variable (en un registro o en la memoria) que contenga una dirección de la memoria principal. El contador de programa, por ejemplo, es el puntero que contiene la dirección de la próxima instrucción que se ha de leer y ejecutar —de hecho, hay procesadores donde el contador de programa recibe el nombre de *Instruction Pointer* (IP).

Los programadores que utilizan punteros se forman la imagen mental de flechas que señalan o *apuntan a* (de ahí el nombre de puntero) un lugar de la memoria y dicen, por ejemplo, que el PC *apunta a* la instrucción que el procesador descodificará y ejecutará a continuación; o que, después de ejecutarse la instrucción `jal F`, el registro `$ra` *apunta a* la instrucción (la siguiente a la llamada) donde regresará el flujo de ejecución al final de la función `F`, al ejecutarse `jr $ra` que salta a la instrucción *donde apunta* `$ra`.

La pseudoinstrucción `la` (*load address*) es similar a `li`, pero su nombre indica explícitamente que **asigna** una dirección a un registro. Es, por tanto, la pseudoinstrucción que permite que un registro apunte a una posición de memoria representada por una etiqueta.

También los punteros tienen su propia **aritmética**. Sumar o restar constantes al valor de un puntero se visualiza como que el puntero *se desplaza* a una dirección más arriba o más abajo en la memoria. Por ejemplo, en cada ciclo de instrucción del procesador, al sumarle 4, el PC *avanza* para apuntar a la siguiente instrucción, y si se ejecuta una bifurcación, que suma un valor con signe al PC, éste *se desplaza* hacia arriba o hacia abajo tantas palabras como está codificado en la instrucción.

La aritmética de punteros se debe hacer sin signo ya que las direcciones se expresan en código binario natural (no tiene sentido hablar de signo ya que son implícitamente positivas); por tanto, la actualización de los punteros se realiza mediante las instrucciones `addu` y `addiu`.

Los punteros en **alto nivel**: mientras que en el lenguaje Java los punteros se ocultan, en C los punteros se declaran y manipulan explícitamente. A continuación presentamos algunas equivalencias entre código C y código ensamblador.

`int A = 4;`
`int * p;`
	`        .data`
	`A:      .word 4`
	`p:      .space 4`

`p = &A; /* p apunta a A */`	`        .text`
	`        la $s0,A`
	`        sw $s0,p`
	o bien:
	`        la $s0,A`
	`        la $s1,p`
	`        sw $s0,0($s1)`

`*p = *p + 1; /* incrementa el entero`
`                al que apunta p */`
	`        la $s0,p`
	`        lw $s1,0($s0)`
	`        addi $s1,$s1,1`
	`        sw $s1,0($s0)`

Parámetros por referencia

Los parámetros de las funciones pueden ser de dos clases:

- Los parámetros **por valor** son datos, como los que hemos visto hasta ahora. Para utilizar una función, el programa que la llama ha de asignar un valor al parámetro.
- Los parámetros **por referencia** son direcciones. Para utilizar una función, el programa que la llama ha de asignar una dirección al parámetro.

En los lenguajes de alto nivel también puede darse esta distinción. En C, los parámetros de tipo no estructurado son siempre por valor. Como el lenguaje permite manejar explícitamente los punteros, los programadores pueden pasar el puntero a una variable. Los parámetros estructurados (vectores, estructuras) se pasan siempre por referencia.

Las funciones del sistema que procesan cadenas toman siempre como uno de los argumentos de entrada la dirección de memoria donde se encuentra la cadena. A continuación referimos las llamadas al sistema del simulador PCSpim para leer e imprimir cadenas de caracteres.

$v0	Nombre	Descripción	Argumentos	Resultado
4	*print_string*	Imprime una cadena de caracteres acabada en nul ('\0')	`$a0` = puntero a la cadena	—
8	*read_string*	Lee una cadena de caracteres (de longitud limitada) hasta encontrar un '\n' y la deja en el buffer acabada en nul ('\0')	`$a0` = puntero al buffer de entrada `$a1` = número máximo de caracteres de la cadena	—

Tabla 2. Funciones del sistema para la entrada/salida de cadenas de caracteres

Variables estructuradas en ensamblador

Para declarar variables estructuradas, tenemos dos alternativas, dependiendo si la variable está inicializada o no.

- Aquí tenemos un vector de cuatro componentes de tipo entero inicializado:

```
          .text 0x10000000
vector:   .word 3, -9, 2, 7
```

- Un vector de la misma talla, pero no inicializado, podría declararse como sigue:

```
          .text 0x10000000
vector:   .space 16
```

Nótese que, en ambos casos, sólo se define una etiqueta, que indica la dirección donde comienza la variable. En el caso de un vector `v` cualquiera, eso corresponde al elemento `v[0]`.

Acceso a variables estructuradas en ensamblador

Para acceder a las variables estructuradas en ensamblador se utiliza una dirección base (puntero a la primera componente de la variable) y un desplazamiento para acceder a sus componentes. A modo de ejemplo, a continuación mostramos un programa en ensamblador

que recorre un vector de enteros e incrementa cada componente en una unidad. Nótese que la aritmética de punteros se hace mediante aritmética sin signo.

```
          .text 0x10000000
vector:   .word 3, -9, 2, 7
          .globl __start
          .text 0x00400000

__start:  la $s0, vector       # Puntero a vector[0]
          li $s1, 4            # Dimensión del vector
bucle:    lw $t0, 0($s0)       # Lee vector[i]
          addi $t0, $t0, 1     # Incrementa vector[i]
          sw $t0, 0($s0)       # Almacena vector[i]
          addi $s1, $s1, -1    # Decrementa contador
          addiu $s0, $s0, 4    # Actualiza puntero a vector[i+1]
          bgtz $s1, bucle
```

También podríamos haber empleado otra forma de acceder al vector donde se ve de manera más clara el acceso a partir de una dirección base y un desplazamiento; nótese que ahora, como no se manipula directamente la dirección de memoria, el desplazamiento se calcula con aritmética con signo ya que este puede ser positivo o negativo. Por otra parte, es importante darse cuenta de que la lectura de cada componente se realiza con la pseudoinstrucción **`lw $t0, vector($s0)`**, que traducida a diversas instrucciones máquina de acuerdo a la naturaleza del valor numérico que representa la etiqueta **`vector`**.

```
          .text 0x10000000
vector:   .word 3, -9, 2, 7
          .globl __start
          .text 0x00400000

__start:  li $s0, 0      # Desplazamiento inicial
          li $s1, 4            # Dimensión del vector
bucle:    lw $t0, vector($s0)  # Lee vector[i]
          addi $t0, $t0, 1     # Incrementa vector[i]
          sw $t0, vector($s0)  # Almacena vector[i]
          addi $s1, $s1, -1    # Decrementa contador
          addi $s0, $s0, 4     # Actualiza el desplazamiento
          bgtz $s1, bucle
```

¿Registros o memoria? Detalles a no perder de vista...

Muchas variables pueden estar en un registro o en la memoria principal.

Cuando una variable está en la memoria, su identidad es una dirección en lugar de un número de registro.

Los registros son limitados.

Con variables en la memoria no se puede hacer directamente ni aritmética ni control de flujo, hay que llevarlas al banco de registros para poder manipularlas.

Las variables en memoria pueden ser mayores de 32 bits.

Las variables en memoria pueden tener un valor inicial al cargar el programa; sin embargo, las variables en un registro necesitan una instrucción para inicializarlas.

Ejercicios de laboratorio

Ejercicio 1: Parámetros por referencia

En este primer ejercicio vamos a considerar el programa que tratamos en la práctica 3 que calculaba el producto de dos números enteros introducidos por el teclado, **`M`** y **`Q`**, calculaba su producto e imprimía el resultado **`R`**. Aquella práctica partía de un archivo fuente que contenía el programa principal y la función **`Mult`** y había que añadir dos nuevas funciones **`Input`** y **`Prompt`** para mejorar el diálogo con el usuario a través de la consola. El pseudocódigo de las tres funciones era el siguiente:

```
int Mult(int $a0,$a1) {
        $v0=0;
        while($a1≠0) {
                $v0=$v0+$a0;
                $a1=$a1−1;}
        return($v0); }
```

```
int Input(char $a0) {
        print_char($a0);
        print_char('=');
        $v0=read_int();
        return($v0); }
```

```
void Prompt(char $a0, int $a1) {
        print_char($a0);
        print_char('=');
        print_int($a1);
        print_char('\n');
        return; }
```

Nuestro objetivo es reproducir de nuevo un diálogo como el siguiente: (en cursiva aparece el texto tecleado por el usuario):

```
M=215
Q=875
R=188125
```

En este caso, sin embargo, las variables **`M`**, **`Q`** y **`R`** van a ubicarse en el segmento de datos de la memoria; recuerde que en la práctica 3 estas variables estaban almacenadas en registros. Así pues, vamos a construir tres funciones nuevas, llamadas **`InputV`**, **`PromptV`** y **`MultV`**. Las dos primeras sirven, como ya sabemos, para introducir los valores por el teclado e imprimir el resultado en pantalla, respectivamente; la tercera se encarga de calcular el producto de los dos valores introducidos por el teclado. A diferencia de las funciones ya diseñadas **`Input`**, **`Prompt`** y **`Mult`** que recibían los parámetros por valor a través de registros, las nuevas reciben los parámetros también a través de los registros pero ahora lo hacen por referencia, es decir, en el registro se encuentra el *puntero* o dirección de memoria de la variable. Como ayuda, puede partir del código que que escribió para las funciones **`InputV`**, **`PromptV`** y **`MultV`** y modificar su diseño de acuerdo a los nuevos requisitos.

Vea un ejemplo en pseudocódigo de **`main()`** y de la función **`void InputV(char lletra, int *var)`** que definen el comportamiento de la función **`InputV`**: ésta recibe el carácter con el que rotularemos la entrada del teclado y la dirección de memoria donde se almacenará el valor leído.

```
main() {                          void InputV(char $a0, int *$a1) {
    int M;                            print_char($a0);
    $a0 = 'M';                        print_char('=');
    $a1 = &M;                         *$a1 = read_int();
    InputV($a0, $a1);                 return; }
    exit; }
```

Como punto de partida de este ejercicio proporcionaremos en el fichero "09_exer_01.s" el código ensamblador siguiente equivalente al pseudocódigo anterior que ilustra la declaración de una variable **`M`** localizada en memoria y su inicialización desde el programa principal con ayuda de la función **`InputV`**:

```
          .globl __start
          .data 0x10000000
M:        .space 4

          .text 0x00400000
__start:  li $a0,'M'
          la $a1, M
          jal InputV
          li $v0,10
          syscall

InputV:   li $v0, 11
          syscall
          li $v0, 11
          li $a0, '='
          syscall
          li $v0, 5
          syscall
          sw $v0, 0($a1)
          jr $ra
```

Después de comprender el programa de partida, cárguelo y ejecútelo con el simulador. No dude en formatearlo y añadir comentarios para hacer más claro su propósito.

- ¿Dónde está el valor de la variable que ha leído? **Técnica experimental:** interprete la ventana *data segment* del simulador.
- Si en el programa principal quisiera sumarle 1 a la variable **`M`** que acaba de leer con **`InputV`**, ¿qué opciones de las siguientes serían correctas?

a)

```
…          no
jal InputV
addi $a1,$a1,1
```

b)

```
…          no
jal InputV
lw $s0,M
addi $s0,$s0,1
```

c)

```
…
jal InputV
lw $s0,M
addi $s0,$s0,1
sw $s0,M
```

si

d)

```
…
jal InputV
lw $s0,0($a1)
addi $s0,$s0,1
sw $s0,0($a1)
```

si

e)

```
…
jal InputV
addi $v0,$v0,1
```

no

f)

```
…
jal InputV
li $s0,M
addi $s0,$s0,1
```

no

Ahora estamos en condiciones de reescribir las funciones **`PromptV`** y **`MultV`**, el pseudocódigo de las cuales mostramos a continuación. Nótese que la función **`MultV`** ya trata el caso en que Q<0.

```
void PromptV(char $a0, int *$a1) {
        print_char($a0);
        print_char('=');
        print_int(*$a1);
        return; }
```

```
void MultV(int *$a0, int *$a1, int *$a2) {
        $t0 = *$a0;
        $t1 = *$a1;
        if ($t1<0) {$t0 = –$t0; $t1 = –$t1;}
        $t2 = 0;
        iterar $t1 veces
                $t2=$t2+$t0;
        *$a2=$t2;
        return; }
```

El programa principal deberá llamar a las funciones diseñadas de la manera indicada en el pseudocódigo siguiente (el símbolo "&" representa la dirección de la variable que le sigue):

```
main() {
        int M, Q, R;
        InputV('M', &M);
        InputV('Q', &Q);
        MultV(&M, &Q, &R);
        PromptV('R', &R);
        exit; }
```

Una vez compruebe que el programa funciona adecuadamente, conteste a las siguientes preguntas:

- ¿En qué dirección de memoria se encuentra almacenada la variable **`R`**? 0x10000008
- Ejecute el programa con los valores **`M=5`** y **`Q=-5`**. Consulte el segmento de datos del programa e indique cuáles son los valores de les variables **`M`**, **`Q`** y **`R`** almacenados en la memoria.

Ejercicio 2. Direccionamiento de cadenas de caracteres

Una cadena no es más que un vector donde cada componente almacena un carácter. Si se utiliza la codificación ASCII, entonces cada carácter ocupará un byte.

En este ejercicio vamos a trabajar con cadenas de caracteres. Partimos del programa siguiente almacenado en el fichero "09_exer_02.s", el funcionamiento del cual deberá averiguar.

```
          .globl __start
          .data 0x10000000
demana:   .asciiz "Escriba alguna cosa: "
cadena:   .space 80

          .text 0x00400000
__start:  la $a0, demana
          la $a1, cadena
          li $a2, 80
          jal InputS
          li $v0,10
          syscall

InputS:   li $v0, 4
          syscall
          li $v0, 8
          move $a0, $a1
          move $a1, $a2
          syscall
          jr $ra
```

En particular, estudie la función llamada **InputS** y diga qué hace el programa completo. Nótese que el perfil de la función es **void InputS(char *$a0, char *$a1, int $a2)**. Compile el programa y ejecútelo.

- ¿Dónde está la cadena que ha tecleado? Búsquela en la ventana *data segment* del simulador.

Ahora queremos completar el programa anterior para que imprima la cadena que hemos introducido por teclado, reproduciendo el comportamiento reflejado a continuación:

```
main() {
      char[] t1 = "Escriba alguna cosa: "
      char[] t2 = "Ha escrito: "
      char[80] cadena;

      InputS(&t1, &cadena, 80);
      PromptS(&t2, &cadena);
      exit;
}
```

```
Escriba alguna cosa: Estoy completamente quemado
Ha escrito: Estoy completamente quemado
```

Diálogo que ha de conseguir con InputS y PromptS. En la parte de abajo, tiene un ejemplo de ejecución, donde la parte tecleada por el usuario está en cursiva y la parte escrita por el programa en negrita

Con esta finalidad implemente la función **void PromptS(char *$a0, char *$a1)**, que imprime seguidas en la consola las dos cadenas a las que apuntan **$a0** y **$a1**.

Ejercicio 3. Recorrido de cadenas de caracteres

Como complemento del ejercicio anterior vamos a escribir una nueva función que calcule la longitud de una cadena de caracteres que se le pasará por referencia. La declaración de la función es **`int StrLength(char *c)`** y devuelve el número de caracteres de la cadena. Supondremos que la cadena acaba con el carácter NUL (valor cero del código ASCII). Por otro lado, tenga en cuenta que, mientras no se llene completamente el buffer, la llamada al sistema **`read_string`** introduce el carácter LF (*line feed*, valor 10 del código ASCII) antes del carácter NUL.

Después de implementarla puede utilizarla con el programa del ejercicio anterior para calcular la longitud de la cadena introducida por el teclado y mostrar la longitud en la consola. Por ejemplo, un posible diálogo del programa tendrá ahora este aspecto:

```
Escriba alguna cosa: Estoy completamente quemado
Ha escrito: Estoy completamente quemado
La longitud es: 27
```

Cuestiones diversas

1. Indique en cuál o cuáles instrucciones máquina se podría traducir la pseudoinstrucción ***`lw $t0,var`*** si la dirección de la variable **`var`** (o sea, el valor de la etiqueta **`var`**) es:
   - **`0x1000`**
   - **`0x100000`**
   - **`0x101000`**

2. Suponga que la dirección de la variable **`A`** es **`0x10000000`**. Compare estos dos fragmentos de código equivalentes:

```
lw $t0, A
addi $t0, $t0, 1
sw $t0, A
```

```
la $t0, A
lw $t1, 0($t0)
addi $t1, $t1, 1
sw $t1, 0($t0)
```

   ¿Cuál de los dos códigos máquina resultantes es el más corto?

3. Considere el fragmento de código siguiente:

```
alpha:    .asciiz "á"
          lb $t0, alpha
```

   ¿Qué valor tendrá el registro **`$t0`** después de su ejecución? ¿Qué valor contendría si en lugar de **`lb`** se hubiera utilizado **`lbu`**? ¿Cuál de las dos instrucciones es más correcto usar en este caso?

4. Haga esta prueba con el simulador: añada la instrucción **`addi $ra,$ra,-4`** al final del cuerpo de la función **`InputS`**, justamente antes de la instrucción **`jr $ra`**, y haga que un programa la llame. ¿Qué pasa? Explique el comportamiento.

Ejercicios adicionales con el simulador

Ejercicio 4: Más recorrido de cadenas

Escriba el código para la función **`char StrChar(char *c, int n)`**, que devuelve el carácter n-ésimo de la cadena **`*c`**. Por no complicar mucho el código, suponga que *n* nunca será mayor que la longitud de la cadena.

Ejercicio 5: Vectores de enteros

Queremos diseñar un programa que calcule la suma de dos vectores de enteros A y B y deje el resultado en un vector C. La dimensión de los vectores y sus valores serán introducidos por teclado. Un ejemplo de diálogo del programa para vectores de dimensión 4 es el siguiente:

```
D=4
A[0]=100
…
A[3]=130
B[0]=200
…
B[3]=230
C[0]=300
…
C[3]=360
```

El programa principal utilizará las funciones que referimos a continuación. Note que no parte de cero, ya que puede inspirarse en algunas de las funciones ya diseñadas en programas anteriores.

- **`void InputVector(char L, int D, word *V)`**
- **`void PromptVector(char L, int D, word *V)`**
- **`void AddVector(word dim, word *V1,word *V2,word *V3)`**

Si tuviera que escribir una nueva versión de estas tres mismas funciones para vectores de halfword (palabras de 16 bits) o de tipo byte (palabras de 8 bits), ¿qué habría que cambiar en el perfil? ¿Y en el cuerpo?

Apéndice

Funciones del sistema

Nombre	$v0	Descripción	Argumentos	Resultado
print_int	1	Imprime el valor de un entero	`$a0` = entero a imprimir	—
print_float	2	Imprime el valor de un *float*	`$a0` = *float* a imprimir	—
print_double	3	Imprime el valor de un *double*	`$a0` = *double* a imprimir	—
print_string	4	Imprime una cadena de caracteres	`$a0` = dirección en memoria de la cadena	—
read_int	5	Lee el valor de un entero	—	`$v0` = entero leído
read_float	6	Lee el valor de un *float*	—	`$a0` = *float* leído
read_double	7	Lee el valor de un *double*	—	`$a0` = *double* leído
read_string	8	Lee una cadena de caracteres	`$a0` = dirección en memoria de la cadena	—
exit	10	Acaba el proceso	—	—
print_char	11	Imprime un carácter	`$a0` = carácter a imprimir	—