ESTRUCTURA DE COMPUTADORES
Grado en Ingeniería Informática

Lab session 9

VARIABLES AND PARAMETER PASSING

Goals

- To understand the use of instructions and pseudoinstructions for reading and writing variables in main memory
- To use system functions for string input and output.
- To manipulate addresses and traverse vectors.
- To implement user functions with pointer type arguments.

Bibliography

- D.A. Patterson y J. L. Hennessy, *Computer Organization and Design*, Elsevier, chapter 2, 2014.

Introduction

Static variables and related directives

The MIPS assembler offers the following resources to declare the static variables of a program:

- Segment `.data` where data are allocated.
- Directive `.space` to reserve space for variables in the data segment, useful for declaring uninitialized variables.
- Directives `.byte`, `.half`, `.word`, `.ascii`, and `.asciiz` to declare and initialize variables.

Instructions and pseudoinstructions for data memory access

The MIPS instruction set includes eight instructions for reading and writing data in memory:

Unit	Address restriction	Read with sign extension	Read with zero extension	Write
byte	none	`lb`	`lbu`	`sb`
halfword	2's multiple	`lh`	`lhu`	`sh`
word	4's multiple	`lw`		`sw`

Table 1. Instructions for reading and writing data in main memory

These instructions use I format, and they take the form: **`op rt, D(rs)`**, where **`D`** is a **signed 16-bit offset** (AKA *displacement*) that is added to the content of **base** register (**`rs`**) to form the memory address where the reading or writing is performed. Using these two components, a register and an offset, memory can be accessed using one of several possible addressing modes:

Absolute addressing: This is possible when the address to access a static variable A is known: the constant value A can be used as the offset and **`$zero`** as the base register. You would use a pseudoinstruction with the form ***`op`*** **`rs, A`**, that will be expanded by the assembler into the required machine instructions when A takes more than 16 bits ($A \geq 2^{16}$).

For example, pseudoinstruction **`lw $rt,A`** loads a register with the value of the 32-bit variable allocated to address A. This translation of the pseudoinstruction by the assembler will depend on the particular value of A:

- If **`A`** can be represented with 16 bits, the translation is just: **`lw $rt,A($0)`**.
- Otherwise ($A \geq 2^{16}$), the pseudoinstruction needs be expanded to deal separately with the higher and lower halves of A (**`Ah`** and **`Al`**, respectively). For example:

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

Indirect addressing: when the position of a variable is in a register. It is a convenient tool when an address computed by the program is accessed, or to follow a pointer (more about that later), or to go through structured variables.

Relative addressing to a register: when a register contains a reference address and the programmer intends to do offsets around it. This addressing mode is also used to access structured variables: the register contains the structured variable base address and the offset corresponds to the variable field accessed.

Pointers

A pointer is any variable (in a register or in memory) that contains a main memory address. The program counter for instance is the pointer that contains the next instruction address to read and execute —in fact, there are processors that name the program counter as *Instruction Pointer* (IP).

Programmers use pointers as arrows that "point" to a memory location and say for instance "the PC points to the instruction that the processor will decode and will execute next" or "after executing instruction **`jal F`**, register **`$ra`** *points to* the instruction (the one that follows the call) where the execution flow will return when the execution of F ends. This happens when **`jr $ra`** is executed jumping to the instruction where **`$ra`** *points to*".

The pseudoinstruction **`la`** (load address) is similar to **`li`**, but its name indicates explicitly that it assigns an address to a register. It is, therefore, the pseudoinstruction that allows us to make a register point to a location represented by a label.

Pointers have their own **arithmetic**. Adding or subtracting a constant to the value of a pointer is seen as if that pointer moves to an address above or below in memory. For instance, in every processor instruction cycle, adding 4 to the PC content moves it to point to the next instruction, and if a branch is performed, adding a signed value to the PC content, it moves up or down as many words as are encoded in the instruction.

Pointer arithmetic should be unsigned since addresses are expressed in natural binary code (it does not make sense to consider them signed since they are implicitly positive). Therefore, the updating of pointers is done using **`addu`** and **`addiu`** instructions.

High-level languages treat pointers in a different way depending on the language considered. Whereas in Java pointers are hidden, in C, pointers can be declared and explicitly handled. The following table shows the equivalence of pointer operations between C and the MIPS assembly language.

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

`p = &A; /* p points to A */`	`.text`
	`la $s0,A`
	`sw $s0,p`
	or:
	`la $s0,A`
	`la $s1,p`
	`sw $s0,0($s1)`

`*p = *p + 1; /* increments the integer pointed by p */`	`la $s0,p`
	`lw $s1,0($s0)`
	`lw $s2,0($s1)`
	`addi $s2,$s2,1`
	`sw $s2,0($s1)`

Parameter passing by reference

Function parameters can be of one of two kinds:

- **Parameters by value** are data values passed directly in a register. The function caller must **assign a value** to this type of parameters before making the call. The parameter is found in the register.
- **Parameters by reference** are addresses. The function caller must **assign an address** to this type of parameters before making the call. The function must load the actual value of the parameter from memory, using that address as a pointer.

In C there is also this distinction. In C non-structured parameters (int, char, etc) are always passed by value. As the language allows the programmer to explicitly handle pointers, they can also pass the pointer to a variable. Structured parameters (vectors, structures) are always passed by reference.

The system functions for string processing take always the string starting memory address as one of the input parameters. The system calls available on the PCSpim Simulator to read and print strings are:

$v0	Name	Description	Arguments	Result
4	*print_string*	Prints a string ended by nul ('\0')	`$a0` = pointer to string	—
8	*read_string*	Reads a string (of limited legth) until finding '\n' and leaves it on the buffer ended by nul ('\0')	`$a0` = pointer to the input buffer `$a1` = maximum number of string characters	—

Table 2. System functions for string input/output

Structured variables in assembly language

There are two ways to declare structured variables, depending on if they are initialized or not.

- A vector of 4 integers initialized:

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

- A vector of the same size but not initialized:

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

Notice that, in both cases, only one label is defined, which indicates the variable starting address. In the case of a vector `V`, that corresponds to element `V[0]`.

Access to structured variables in assembly language

To access structured variables in assembly language a base address is used (pointer to the first component of the structured variable) and an offset to access its components. As an example, what follows is an assembly language program that traverses a vector of integers, adding 1 to

every vector component. Note that the pointer update at the end of the loop conveniently uses unsigned addition.

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

__start:  la $s0, vector       # Pointer to vector[0]
          li $s1, 4            # Vector dimension
loop:     lw $t0, 0($s0)       # Reads vector[i]
          addi $t0, $t0, 1     # Increments vector[i]
          sw $t0, 0($s0)       # Stores vector[i]
          addi $s1, $s1, -1    # Decrements counter
          addiu $s0, $s0, 4    # Updates pointer to vector[i+1]
          bgtz $s1, loop
```

Another way to access to the vector is possible which more clearly shows the access from a base address and an offset; notice that now, as the memory address is not directly handled, the offset is computed using signed arithmetic, since now it can be positive or negative. On the other hand, it is important to realize that every component is read with pseudoinstruction **`lw $t0, vector($s0)`**, that is translated into different machine instructions according to the numeric value that represents the label **`vector`**.

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

__start:  li $s0, 0            # Initial offset
          li $s1, 4            # Vector dimension
loop:     lw $t0, vector($s0)  # Reads vector[i]
          addi $t0, $t0, 1     # Increments vector[i]
          sw $t0, vector($s0)  # Stores vector[i]
          addi $s1, $s1, -1    # Decrements counter
          addi $s0, $s0, 4     # Updates offset
          bgtz $s1, loop
```

Registers or memory? Details to bear in mind...

Variables may reside in registers or in main memory.

When a variable is allocated to main memory, its location is given by a memory address, instead of a register name.

The number of registers is scarce, compared to typical memory size.

Memory variables cannot be used directly as operands or results of arithmetic, logic or branch instructions. They need be copied to registers first.

Memory variables can be larger than 32 bits.

Memory variables may have a given initial value when the program is loaded; on the contrary, register variables have to be explicitly initialized with appropriate instructions.

Lab exercises

The following exercises require the PCSpim Simulator for Windows available in Poliformat at: *Resources* > *Lab* > *Tools* > *pcspim.exe*. It is already installed in the lab.

Exercise 1: By-reference parameters

In this first exercise we will consider a program that we deal with in lab 3 that computes the product of two integers, *M* and *Q*, entered by the keyboard and then prints the result *R*. In that lab session, you started from a source file containing the main program and the *Mult* function and added two new functions, *Input* and *Output*, to improve the console dialogue. As a reminder, the pseudocode of those three functions was roughly:

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

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

```
int Mult (int $a0, $a1)
{
  $v0 = $a0 x $a1;
  return($v0);
}
```

Our goal is for the program to produce again a dialogue such as (user input in *italics*):

M=*215*
Q=*875*
R=188125

In this lab, however, variables *M*, *Q*, and *R*, are memory variables declared in the data segment; they were register variables in lab 3, hence passed by value. Our goal now is to build three new functions, **InputV**, **OutputV** and **MultV**. These new functions also receive arguments in the registers, but now M, Q and R will be passed by reference. This implies that argument registers will contain the addresses of these variables, not their values.

Remember the purpose of these functions is: **InputV** and **OutputV** serve to enter values from the keyboard and to print a value on the screen, respectively; **MultV** must calculate the product of the two values entered from the keyboard.

See the following pseudocode example of a main progam **main()** on one side, and the input function **void InputV(char charact, int *var)** on the other, that takes a *prompt* character (passed by value) and the address where the function will store the character typed from the keyboard (i.e., the result is passed by reference).

```
main() {
  int M;
  $a0 = 'M';
  $a1 = &M;
  InputV($a0, $a1);
  exit;
}
```

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

As a starting point for this exercise, open the file *09_exer_01.s* with the following assembly code, that is equivalent to the above pseudocode. It shows the declaration of variable **M** in memory and its initialization from the main program, using function **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
          ...
```

Once you have inspected and understood this code, load and run it with the PCSpim simulator. Feel free to format and add comments to the code to make its purpose clearer.

- After execution of this code, where is the value of the variable read from the keyboard? **Experimental technique:** inspect the *data segment* window on the simulator.
- Assume the main program intends to add 1 to the variable **M**, after reading it with **InputV**; which of the following options are correct for this purpose?

a)

```
…
jal InputV
addi $a1,$a1,1
```

b)

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

c)

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

d)

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

e)

```
…
jal InputV
addi $v0,$v0,1
```

f)

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

You are ready now to write functions **OutputV** and **MultV**, whose pseudocode is shown next. Note that function **MultV** deals with the case Q<0.

```
void OutputV(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
  $t0 = *$a0 x *$a1;
  *$a2 = $t0;
  return;
}
```

The main program must call the designed functions as shown in the following pseudocode (the symbol "&" represents the address of the variable that follows):

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

Once you have verified that the program works as it should, answer the following questions:

- What is the address of variable *R*?
- Run the program with values **M=5** and **Q=-5**. Then inspect data segment window in the simulator and find the values of *M*, *Q* and *R* in main memory.

Exercise 2. String parameters

A string is a vector where each component stores a character. If encoded in ASCII, then each character will take one byte. Since a string of just a few characters would easily be larger than the processor's word size, string parameters (and vectors in general) are passed by reference to user functions and also to system functions (see e.g., the system function **print_string**).

In this exercise, we will work with strings. We start with the following program, included in file *09_exer_02.s*. You have to find out what it does. For that purpose, you need to analyze function **InputS** and infer what the whole program is trying to do. This program function would be declared: **void InputS(char *$a0, char *$a1, int $a2)**.

```
          .globl __start
          .data 0x10000000
prompt:   .asciiz "Write something: "
string:   .space 80

          .text 0x00400000
__start:  la $a0, prompt
          la $a1, string
          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
```

Compile the program and run it.

- Where is the string that was typed on the keyboard stored? Locate it in the simulator *data segment* window.

Now we want to extend the previous program so that it also prints the user-typed string. The new behavior we are after is the following:

```
main() {
  char[] t1 = "Write something: "
  char[] t2 = "You have written: "
  char[80] cadena;


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

```
Write something: I am completely burned out
You have written: I am completely burned out
```

Pseudo-code of the required program. After calling InputS, the program calls function OutputS to print the read string, preceded by string t2. Below the code, an example of execution, where the user input is typed in italics and the program outputs in boldface.

Complete the program and implement the function **void OutputS(char *$a0, char *$a1)**, which just prints the two strings pointed to by **$a0** y **$a1**, one after another.

Exercise 3. Accessing strings

As a complement to the previous exercise, we will write a new function that calculates the length of a string that will be passed by reference. The function declaration is **`int StrLength(char *c)`** and it returns the number of characters on the string. We will assume that the string ends with character NUL (zero value ASCII code). On the other hand, please note that, while the buffer is not filled completely, the system call **`read_string`** introduces character LF (line feed, value 10 ASCII code) before the NUL character.

After implementing the function you can use it on the previous program to calculate the length of the string entered by the keyboard and display the corresponding length on the console. For instance, a possible dialogue with the program would be:

```
Write someting: I am completely burned out
You have written: I am completely burned out
String length is: 26
```

Question pool

1. Expand the pseudo-instruction ***`lw $t0,var`*** when the address of variable **`var`** (i.e., the value of label **`var`**) is:
   - **`0x1000`**
   - **`0x100000`**
   - **`0x101000`**

2. Suppose that the address of variable **`A`** is **`0x10000000`**. Compare the following two code fragments that are equivalent:

```
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)
```

   Which one of the corresponding machine codes is shorter?

3. Consider the following code fragment:

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

   What will be the value in **`$t0`** after its execution? What would it be if we used **`lbu`** instead of **`lb`**? Which one of the two instructions is more appropriate in this case?

4. Try this test on the simulator: include the instruction **`addi $ra,$ra,-4`** at the end of function **`InputS`**, just before instruction **`jr $ra`**, and make a program to call it, what happens? Explain the observed behavior.

Additional exercises with the simulator

Exercise 4: More on string access

Write the code for function **`char StrChar(char *c, int n)`**, that returns the n-th character on string **`*c`**. To simplify your code, suppose that *n* will never be bigger than the string length.

Exercise 5: Vectors of integers

We want to design a program that calculates the addition of two vectors of integers A and B and store the result in vector C. The user must be required to type the vectors' dimension and then the values of their components. An example of the program dialogue with vectors of dimension 4 is the following:

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

The main program will use the functions referred below. Note that you don't start from scratch, since you can rely on functions you have already implemented in this lab.

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

If you had to write a new version of these three functions for vectors of half words (16-bit) or bytes (8-bit), what should be changed in their declaration and implementation?

Appendix: System functions in PCSpim

Name	Index $v0	Description	Arguments	Result
print_int	1	Print integer	`$a0` = integer to print	—
print_float	2	Print float	`$a0` = float to print	—
print_double	3	Print double-precision	`$a0` = double to print	—
print_string	4	Print null-terminated string	`$a0` = string address	—
read_int	5	Read integer	—	integer in `$v0`
read_float	6	Read float	—	float in `$f0`
read_double	7	Read double	—	double in `$f0`
read_string	8	Read string (null-terminated result)	`$a0` = string address `$a1` = max string length	*By reference, `$a0`*
exit	10	Terminate process	—	—
print_char	11	Print character	`$a0` = character to print	—