ESTRUCTURA DE COMPUTADORES
Grado en Ingeniería Informática

Lab session 9

VARIABLES AND PARAMETER PASSING

Goals

- Understanding and using instructions and pseudoinstructions for reading and writing in main memory
- Using system calls to perform string input and output.
- Handling addresses and vectors.
- Creating procedures 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 these resources in order to describe the static variables of a program in memory:

- Segment **`.data`** where data are allocated.
- Directive **`.space`** carries out memory reservation on the data segment, useful for declaring uninitialized variables.
- Directives **`.byte`**, **`.half`**, **`.word`**, **`.ascii`** and **`.asciiz`** allow variable definition and initialization.

Instructions and pseudoinstruction for data memory access

The MIPS instruction set includes 8 instructions for reading and writing data in memory:

Unit	Address restriction	Reading with sign extension	Reading without sign extension	Writing
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 integers in main memory

These instructions use I format and in assembly language they are written as **`op rt,D(rs)`**. **`D`** is a **signed 16-bit offset** that is added to the content of **base `rs`** register to form the memory address where the reading or writing is performed. This way of memory addressing allows several memory access modes, as explained next:

Absolute addressing: done on a fixed memory word whose position A is known: constant A is as offset and register **`$zero`** as base. With this purpose it is convenient to use pseudoinstructions in the form of **`op rs,A`**, that are decomposed into the required machine instructions when A has more than 16 bits.

For instance, pseudoinstruction **`lw $rt,A`** allows the expression of the load of a register with the value of a memory variable allocated in an address tagged as A. This line can be translated into one or more machine instructions, depending on the tag value:

- If **`A`** is a 16-bit number, the translation is **`lw $rt,A($0).`**
- Otherwise, the assembler decompose it into the higher part **`Ah`** and the lower part **`Al`**. For instance:

```
     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 that a register pointing to a location to be represented by a label.

Also 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. While 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 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 two types:

- **Parameters by value** are data values, so in order to pass them to a function, the program that calls the function has to **assign a value** to the parameter.
- **Parameters by reference** are addresses, so in order to pass them to a function, the program that calls the function has to **assign an address** to the parameter.

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 goes through a vector of integers and adds one to every vector component. Notice that pointer arithmetic relies on unsigned arithmetic.

```
          .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 take into account...

Variables can be allocated on a register or in main memory.

When allocating a variable in main memory, its identity is an address instead of a register number.

The number of registers is very limited.

With memory allocated variables neither direct computation or flow control can be performed, they have to be first allocated into registers.

Memory allocated variables can be bigger than 32 bits.

Variables in memory can have an initial value when the program is loaded; however, variables on register have to be explicitly initialized with appropriate instructions.

Lab exercises

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

Exercise 1: Parameter passing by reference

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 a source file containing the main program and the *Mult* function was used and it was asked to add two new functions, *Input* and *Prompt* to improve the user interaction through the console. The pseudocodes of the three functions are the following:

```
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;
}
```

Our goal is to produce again a dialogue like the following one (italics show the text typed by the user):

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

In this case, however, variables M, Q, and R are allocated into the data memory segment; remember that in lab 3 these variables were stored on registers. So, we are going to build three new functions, called **`InputV`**, **`PromptV`** and **`MultV`**. The first two functions are used, as we know, to enter values by the keyboard and to print the result on the screen, respectively; the third one is responsible for calculating the product of the two values entered by the keyboard. Unlike the functions already implemented: **`Input`**, **`Prompt`** and **`Mult`,** which received their parameters by value through registers, the new functions also receive parameters through the registers but now they are passed by reference, which means that on the register is the pointer or memory address of the variable. It will help to start from the former code and then customize the implementation according to the new requirements.

Look at the following pseudocode example of **`main()`** and function **`void InputV(char charact, int *var)`** that define the behavior of function **`InputV`** that receives the character sent from the keyboard and the memory address that will store that character.

main() {	void InputV(char $a0, int *$a1) {

```
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;
	}
	```

```
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, we provide file "09_exer_01.s" file with the following assembly code that is equivalent to the above pseudocode, which illustrates the declaration of variable `M` in memory and its initialization from the main program, with the help of **`InputV`** function:

```
          .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
```

After understanding the proposed program, load it and run it with the Simulator. Feel free to format it and add comments to make its purpose clearer.

- Where is the value of the read variable? **Experimental technique:** look at the *data segment* window on the simulator.
- If the main program intends to add 1 to variable `M`, just read with **`InputV`**; which of the following options are correct?

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
```

Now we are ready to rewrite functions **`PromptV`** and **`MultV`**, their pseudocode is shown next. Notice that function **`MultV`** deals with the case 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;`
	`  iterate $t1 times`
	`    $t2 = $t2 + $t0;`
	`  *$a2 = $t2;`
	`  return;`
	`}`

The main program has to 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);
  PromptV('R', &R);
  exit;
}
```

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

- Which memory address stores variable R?
- Run the program with values **`M=5`** and **`Q=-5`**, then look at the program data segment and get the values of variables **`M`**, **`Q`** and **`R`** stored in main memory.

Exercise 2. String addressing

A string is a vector where each component stores a character. If ASCII encoding is used, then each character will occupy one byte

In this exercise we will work with strings. We start from the following program included on file "09_exer_02.s". You have to find out how it works.

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

          .text 0x00400000
__start:  la $a0, askfor
          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
```

In particular, analyze function **`InputS`** and indicate what does the whole program. Notice that the function declaration is **`void InputS(char *$a0, char *$a1, int $a2)`**. Compile the program and run it.

- Where is the string typed on the keyboard? Look for it on the simulator's *data segment* window.

Now we want to complete the previous program for printing the string that we have introduced via the keyboard, reproducing the behavior indicated below:

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

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

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

Dialogue that you should get through InputS and PromptS. At the bottom, you have an example of execution, where the text typed by the user is in italics and the text written by the program in bold.

To achieve this implement function **`void PromptS(char *$a0, char *$a1)`**, that prints one after another the two strings pointed by **`$a0`** y **`$a1`**.

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: 27
```

Question pool

1. Indicate with what machine instructions will the pseudoinstruction be translated ***`lw $t0,var`*** if the address of variable **`var`** (that is 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 will be shorter?

3. Consider the following code fragment:

```
alpha:    .asciiz "α"

          lb $t0, alpha
```

   What value will have register **`$t0`** after its execution? What value will have it if instead of **`lb`** we use **`lbu`**? Which one of the two instructions is more correct in this case?

4. Do the test on the simulator: add 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`**. In order to simplify the 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 let the result on vector C. With the keyboard we introduce the vectors dimension and their values. 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 functions which we refer below. Note that you don't start from scratch, since you can rely on functions already implemented.

- `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)`

If you had to write a new version of these three functions for vectors of halfword (16-bit words) or of type byte (8-bit words) components, what would have to be changed in the function declaration? And in the implementation?