Lab Session 3

PROGRAM FUNCTIONS

Goals

- To write simple functions and call them from a program.

References

D. Patterson, J. Hennessy. **Computer organization and design. The hardware/software interface**. 4 th Edition. 2009. Elsevier

Introduction

Program Functions

Program Functions (called functions) are the translation of Java methods or functions in C language. The instruction pair **`jal eti`** (or function call) and **`jr $ ra`** (function return), linked to the register $ra ($31), give the basic support to the execution flow.

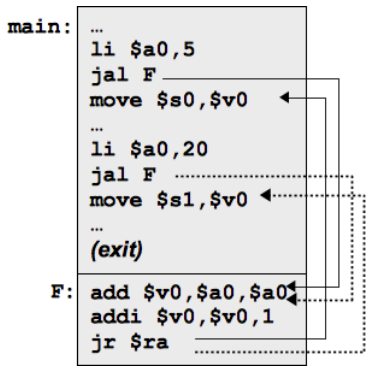

Figure 1. On the left, you can see the schema of a main () program that calls a function F from two different points in the code. At high level language this would be expressed as **`Int F(int a){return 2*a+1}`**. In both cases, instruction **`jal`** F stores the return address in **`$ra`**. Consequently, function **F** finishes with instruction **jr $ra**.

Arrows show the execution flow: the first call is shown in continuous line ⟶ and the second one in dotted ⋯▸.
Main program finishes executing the system call **exit**.

The agreement for the use of the registers also considers the separation between registers of the main program and registers of the function. Registers from $s0 to $s7 are

recommended to use for the global variables of the program, and registers from $t0 to $t9 for the local variables of the procedure or function. The agreement states:
If the main program uses a $ti register, you must expect that any calling function can change its contents.

- If a function needs to write in any register $si, you must preserve its content before and restore the value when finishing.
- If a function uses a $ti register, you must keep in mind that, between two executions of the same function, any other function could modify its content.

The agreement also considers the communication between the main program and the function, and regulates the use considering the number and type of data exchanged. For example, if the arguments are integer numbers and there are no more than four, they will pass in order in the registers $a0 to $a3. If the value returned by the function is an integer, it will be written in register $v0.

Summary: in the exercises of these lab sessions we recommend you to follow the rules of the following table when programming. These rules will be expanded later to allow program functions to call each other.

Registers	Use
$s0...$s7	Main Program
$a0...$a3	Parameters from the main program to functions
$t0...$t9	Function code
$v0	Function results

Table 1. Agreement for the use of registers

System Calls

The use of functions of the program is very similar to the system functions; the most notable difference is that the code of the system functions is hidden, is independent of the programs, is common to all of them and preserves the contents of the global and local registers. In summary, it is not necessary to know the address where the system functions are stored to use them.

The table below shows the system calls to be used in this practice.

Name	$v0	Description	Arguments	Result
print_int	*1*	Prints an integer value	**$a0** = integer to print	—
read_int	*5*	Reads an integer value	—	**$v0** = integer
exit	*10*	Ends the process	—	—
print_char	*11*	Prints a character	**$a0** = character to print	—

Table 2. System calls used in Lab Session 3.

Lab exercises

Exercise 1: System calls and program functions

Open and read the following code contained in the file "03_exer_01.s". Note that only the code segment (.text) is provided and there is no comment. You should add the comments once you understand how the program works.

```
          .globl __start
          .text 0x00400000
__start:  li $v0,5
          syscall
          move $a0,$v0
          li $v0,5
          syscall
          move $a1,$v0
          jal Mult
          move $a0,$v0
          li $v0,1
          syscall
          li $v0,10
          syscall
Mult:     li $v0, 0
          beqz $a1, MultRet
MultFor:  add $v0, $v0, $a0
          addi $a1, $a1, -1
          bne $a1, $zero, MultFor
MultRet:  jr $ra
```

First, you must detect which instructions belong to the main program and which ones belong to a function named Mult.

- Which are the last two instructions of the main program?
- Which is the last instruction of the function?
- Identify the four system calls used in the program. What function does perform each one of them?
- Find a loop within the function. How many times this loop is executed?
- What exactly does the function do?

Load the program and run it. Note that the input / output from the console is very poor.

- Can you run the entire program? When executing, note that the program prompts you to enter two numbers on the keyboard and then prints a result. However, there is no message indicating that a keyboard input is expected.

• Can you do a step-by-step execution?

Experimental technique: Using breakpoints. It is very useful to stop the program at a point where it is convenient to examine the registers or the memory without having to go step by step from the beginning. Simply, you have to program in the simulator the address of the instruction where the execution is to be stopped. Use this technique to stop the execution within Mult and observe the value of the return address contained in register $ra. It should indicate as breakpoint of the execution flow the address of the instruction **jr $ra**.

- Which is the value of the return address?
- Which is the instruction pointed by the return address?

Exercise 2: Programming functions

Let´s improve the dialogue of the previous program through the console. This improvement consists in associating the symbol of a letter to each value that is read or written. Thus, you can name the multiplying as 'M', the multiplier as 'Q' and the product as 'R'. You must program two functions that will be added to the program of the previous section:

- To enter values by the keyboard. The Input function has as an argument the symbol of the letter that we are going to write to the console. The function should write in the console this symbol followed by the character '=' and then read an integer (multiplier or multiplicand). The function should return this value read.
- For printing the result. The Output function has two arguments: the letter and the result (integer) to be printed. The function must write the letter, the character "=", the value of the result and the end-of-line character LF (line feed, value 10 of the ASCII code).

For clarity, we will express these two functions in pseudocode:

```
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; }
```

Note that the arguments received by the functions are stored in registers. For example, the Input function receives the character to print in register $a0; Similarly, Output receives the two arguments (one character and one integer) in registers $a0 and $a1. This detail is very important: this way of passing the parameters to the functions is called by value.

When you have made the encoding of the two functions Input and Output, you must completely rewrite the body of the main program to label the multiplying with the letter "M", the multiplier with the "Q" and the result of the product with "R" . The resulting dialog should appear in the console as Figure 5 shows:

```
A=Input('M');
B=Input('Q');
C=Mult(A,B);
Output('R',C);
Exit();
```

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

Figure 2. On the left, the pseudocode of the main program to write. On the right, a resulting dialog example. In bold, we show the text written by the program. In italics, the text typed by the user.

Exercise 3: Conditional Instructions

In the given program the Mult function only works correctly if the multiplier Q is positive. Try running the program with Q = -5: the function loop will lengthen and you should stop the program. To do this you can press the key combination CTRL + C or press the menu icon labeled with the word *Stop*.

In this section you are asked to slightly modify the main program so that if Q <0, instead of calculating *R = Mult (M, Q)* calculate *R = Mult (-M, -Q*), i.e. change the sign of both arguments before calling the function in order to maintain the correct result. This action expressed in pseudocode is shown in Figure 6.

```
M=Input('M');
Q=Input('Q');
If (Q<0)
      M=–M;
      Q=–Q;
R=Mult(M,Q);
Output('R',R);
Exit();
```

Figure 3. A way to solve the Mult limitation and permit to operate with negative multipliers

The crucial point here is to discover how to change the sign of an integer.

Additional exercises using the simulator

You can do them in the lab, if you have enough time, or finish them at home.

Exercise 4: Iterations

1. Make the necessary changes in the main program so that the M × Q calculation is repeated until one of the two operands entered by the keyboard is zero, that is, the multiplication will be repeated while the two operands are different from zero. The same expressed in pseudocode:

```
repeat
  M=Input('M');
  Q=Input('Q');
  R=Mult(M,Q);
  Output('R',R);
while ((M≠0) && (Q≠0));
Exit();
```

2. Design a program that asks for a number n and write the multiplication table of n, from n*1 to n*10. To make the programming simpler you can use a OutputM function, which is expressed in pseudocode as follows:

```
void OutputM(int x, int y, int r) {
  print_int(x);
  print_char('x');
  print_int(y);
  print_char('=');
  print_int(r);
  print_char('\n');
}
```

Exercise 5: Selector

Encode the void **`PrintChar(char c)`** function, which prints in console a character following the C style: in quotation marks and showing the special cases '\ n' (ASCII character number 10) and '\ 0' (ASCII character number 0).

```
void PrintChar(int x) {
    putchar('''');  /* comilla */
    switch (x){
        case 0:  print_char('\'); print_char('0'); break;
        case 10:    print_char('\'); print_char('n'); break;
        default:print_char(x);
    }
    putchar(''''); /* comilla */
}
```

Annex

Examples of flow control

In the next Table:

- Symbols *cond, cond1*, etc., refer to the six simple conditions (= and ≠,> and ≤, <and ≥) that relate two values contained in registers. The asterisk (*) indicates the opposite condition; for example, *if cond = ">"* the opposite is *cond * = "≤"*.

- In the high-level column, symbols A, B, etc. indicate simple or compound instructions; in the assembler column, the symbols A, B, etc. represent equivalent assembly blocks.

Conditionals

<table>
<tr><th>High Level</th><th>Assembler</th></tr>
<tr><td>if (cond1)
A;
else if (cond2)
B;
else
C;
D;</td><td>
if: bif (cond1*) elseif
A
j endif
elseif: bif (cond2*) else
B
j endif
else: C
endif: D

if: bif (cond1) then
bif (cond2) elseif
j else
then: A
j endif
elseif: B
j endif
else: C
endif: D
</td></tr>
<tr><td>if (cond1 && cond2)
A;
B;</td><td>
if: bif (cond1*) endif
bif (cond2*) endif
A
endif: B
</td></tr>
</table>

if (*cond1*||*cond2*) A; B;	**if:** ***bif (cond1)*** **then** ***bif (cond2*)*** **endif** **then:** **A** **endif:** **B** **if:** ***bif (cond1*)*** **endif** ***bif (cond2*)*** **endif** **A** **endif:** **B**

Selectors

High Level	Assambler

switch (exp){
case X :
A;
break;
case Y :
case Z :
B;
break;
default:
C;
}
D;
	bif (exp != X) caseY
	caseX: **A**
	j endSwitch
	caseY: **bif (exp != Y) default**
	caseZ: **bif (exp != Z) default**
	B
	j endSwitch
	default: **C**
	endSwitch: **D**
	
	bif (exp == X) caseX
	bif (exp == Y) caseY
	bif (exp == Z) caseZ
	j default
	caseX: **A**
	j endSwitch
	caseY:
	caseZ: **B**
	j endSwitch
	default: **C**
	endSwitch: **D**

Iterations

High level	Assambler
while (*cond*) A; B;	**while:** ***bif (cond*)*** **endwhile** **A** **j while** **endwhile** **B**
do A; while (*cond*) B;	**do:** **A** ***bif (cond)*** **do** **B**

do
A;
if(*cond1*) continue;
B;
if(*cond2*) break;
C;
while (*cond3*)
D;
	do: **A**
	bif (cond1) **while**
	B
	bif (cond2) **enddo**
	C
	while: ***bif (cond3)*** **do**
	enddo: **D**

iterar *n* veces /* *n*>0 */ A; B;	**li *$r*,*n*** **loop:** **A** **addi *$r*,*$r*,-1** **bgtz *$r*,loop** **B**

System Calls in PCSpim simulator

$v0	Name	Description	Argument	Result	Equivalent Java	Equivalent C
1	*print_integer*	Prints an integer value	**$a0** = integer to print	—	**System.out.print(int $a0)**	**printf("%d",$a0)**
2	*print_float*	Prints a float point value	**$f12** = float to print	—	**System.out.print(float $f0)**	**printf("%f",$f0)**
3	*print_double*	Prints a double precision float point value	**$f12** = double to print	—	**System.out.print(double $f0)**	**printf("%Lf",$f0)**
4	*print_string*	Prints a string of characters ended with nul ('\0')	**$a0** = pointer to string	—	**System.out.print(int $a0)**	**printf("%s",$a0)**
5	*read_integer*	Reads an integer value	—	**$v0** = integer read		
6	*read_float*	Reads a float point value	—	**$f0** = *float* read		
7	*read_double*	Reads a float point value (double precision)	—	**$f0** = *double* read		
8	*read_string*	Reads a string of characters (of limited length) until it finds a '\ n' and stores it in a buffer ending in nul ('\ 0')	**$a0** = pointer to input buffer **$a1** = max number of characters in the string			
9	*sbrk*	Reserves a heap memory block	**$a0** = block length in bytes	**$v0** = pointer to the memory block		**malloc(integer n);**
10	*exit*		—	—		**exit(0);**
11	*print_character*	Prints a character	**$a0** = character to print			**putc(char c);**
12	*read_character*	Reads a character		**$v0** = character read		**getc();**