Lab Session 3

FUNCTIONS AND SYSTEM CALLS

Goals

- Understanding how instructions, data and system functions are combined to make a program.
- To write simple functions and call them from a program.
- Making inventory of flow control instructions.
- To know and make use of system calls by means of the **syscall** machine instruction.

References

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

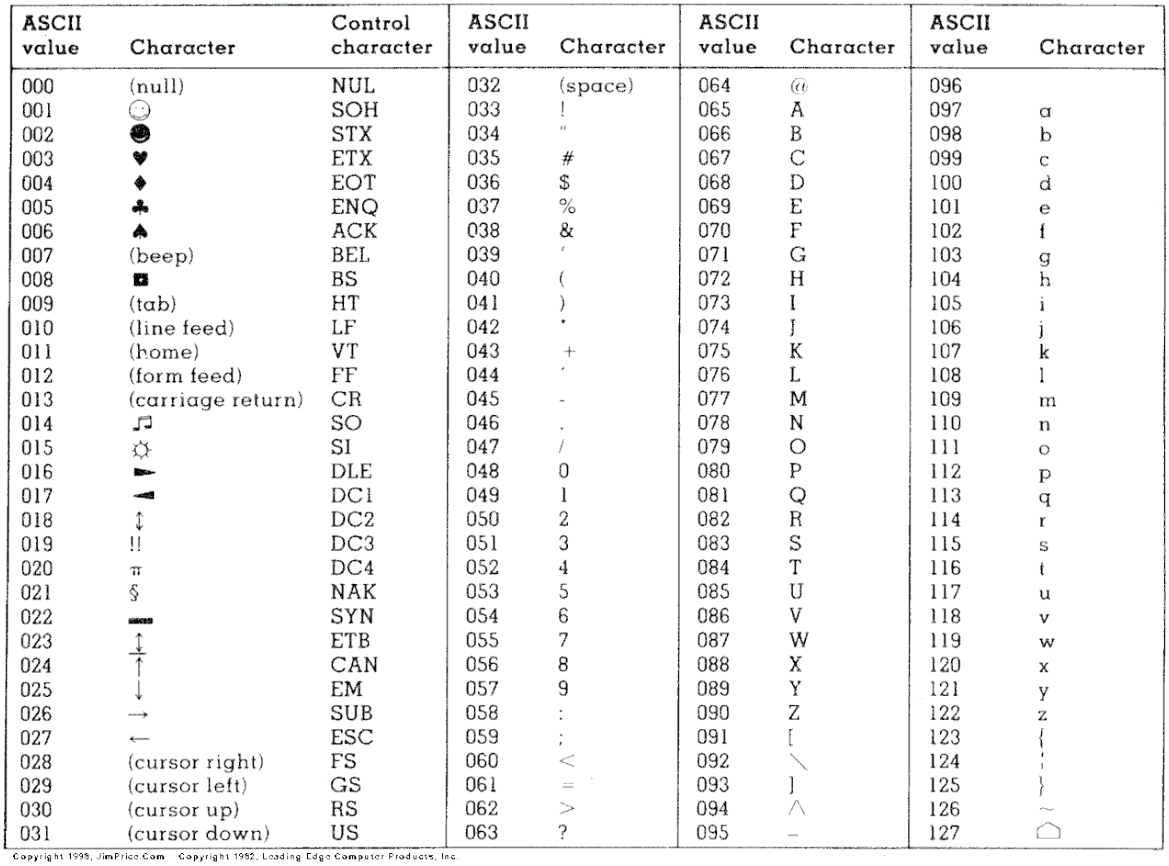
Introduction

Integers and Characters

We already know that in a computer the strings or words of bits have no a specific meaning by themselves. This means that, for example, the word 0x90324a00 can be interpreted, depending on the context, as a machine instruction, a memory address, an unsigned integer, a signed integer encoded 2-complement (in this case it would be negative), a real number coded according to IEEE 754 (in this case it would also be negative), a chain of 4 bytes in length, etc.

Although computers were originally designed to perform large numbers of arithmetic calculations, they were soon used to process text. A large number of current computers use 8-bit words to represent characters according to the American Standard Code for Information Exchange (ASCII) code. This code has some characteristics that we must always take into account: the codes for lower and upper case letters only differ by one bit and, numerically, the position of that bit makes the quantitative difference between the two codes of each letter (uppercase and lowercase) is 32. For example, the code for the letter "Q" is 81 and per the letter "q" is 113 (note that 81 + 32 = 113, and that the two codes in

binary are respectively 1010001 and 1110001). Another important value is zero, called null, which is used in C and Java to mark the end of a string.



ASCII value	Character	Control character	ASCII value	Character	ASCII value	Character	ASCII value	Character
000	(null)	NUL	032	(space)	064	@	096	
001	☺	SOH	033	!	065	A	097	a
002	☻	STX	034	"	066	B	098	b
003	♥	ETX	035	#	067	C	099	c
004	♦	EOT	036	$	068	D	100	d
005	♣	ENQ	037	%	069	E	101	e
006	♠	ACK	038	&	070	F	102	f
007	(beep)	BEL	039	'	071	G	103	g
008	◘	BS	040	(	072	H	104	h
009	(tab)	HT	041	)	073	I	105	i
010	(line feed)	LF	042	*	074	J	106	j
011	(home)	VT	043	+	075	K	107	k
012	(form feed)	FF	044	,	076	L	108	l
013	(carriage return)	CR	045	-	077	M	109	m
014	♫	SO	046	.	078	N	110	n
015	☼	SI	047	/	079	O	111	o
016	►	DLE	048	0	080	P	112	p
017	◄	DC1	049	1	081	Q	113	q
018	↕	DC2	050	2	082	R	114	r
019	!!	DC3	051	3	083	S	115	s
020	¶	DC4	052	4	084	T	116	t
021	§	NAK	053	5	085	U	117	u
022	▬	SYN	054	6	086	V	118	v
023	↨	ETB	055	7	087	W	119	w
024	↑	CAN	056	8	088	X	120	x
025	↓	EM	057	9	089	Y	121	y
026	→	SUB	058	:	090	Z	122	z
027	←	ESC	059	;	091	[	123	{
028	(cursor right)	FS	060	<	092	\	124	|
029	(cursor left)	GS	061	=	093	]	125	}
030	(cursor up)	RS	062	>	094	^	126	~
031	(cursor down)	US	063	?	095	_	127	⌂

Figure 1. ASCII code for the representation of the first 128 characters.

At present, Unicode is the universal codification of the alphabets of human languages (Latin, Greek, Cyrillic, Bengali, Ethiopian, Thai, and so on). There are as many alphabets in Unicode as there are useful symbols in ASCII. The Java programming language uses Unicode to encode characters. By default, it uses 16 bits to represent a character. To learn more about Unicode, visit www.unicode.org.

We already know that the MIPS memory access instructions permit to access 32-bit (word) strings with **`lw`** and **`sw`**, 16 bit (half word) with **`lh`** and **`sh`** instruction, and 8 bit (byte) with **`lb`** and **`sb`**. It should be noted that the read instructions **`lh`** and **`lb`**, since they read less than 32 bits, complete the missing bits by extending the sign of the read word; that is, they implicitly interpret the read value as a two-character encoded integer.

But, what happens if what we read from memory is a character encoded in ASCII or Unicode? In this case, it makes no sense to interpret a sign bit since the value read should not be interpreted as an integer because it is a character. That's why the MIPS R2000 instruction set also includes unsigned lhu and lbu variants to read smaller 32-bit strings

where no sign extension should be made. These instructions set to zero the missing bits to fill in the target register. In fact, the lhu and lbu instructions are more popular than lh and lb.

Execution flow control in assembler

Jump instructions jointly with certain arithmetic instructions allow the construction of conditional and iterative structures. At low level, we can distinguish between:

• Unconditional jumps (follow in the address …); for example, instruction **`j eti.`**

• Conditional jumps or bifurcations. If condition is true then follow in the address where the instruction indicates. In the MIPS instructions set, we have six conditions for conditional jumps: note that three pairs of opposing conditions can be made (= y ≠, > y ≤, < y ≥).

The instruction set only allows comparisons = and ≠ between two registers and the comparisons >, ≤, < and ≥ between a register and zero:

beq rs,rt,A	**bgtz rs,A**	**bltz rs,A**
rs = rt	*rs > 0*	*rs < 0*
bne rs,rt,A	**blez rs,A**	**bgez rs,A**
rs ≠ rt	*rs ≤ 0*	*rs ≥ 0*

Table 1. Conditional branches in MIPS

This set of conditions can be extended using the arithmetic **`slt`** (set on less than) instruction. Thus we obtain these other six pseudo instructions:

beqz rs,A	**bgt rs,rt,A**	**blt rs,rt,A**
rs = 0	*rs > rt*	*rs < rt*
bnez rs,A	**ble rs,rt,A**	**bge rs,rt,A**
rs ≠ 0	*rs ≤ rt*	*rs ≥ rt*

Table 2. Conditional Branch Pseudoinstructions inMIPS

Table 3 shows, as an example, the translations of two branch pseudoinstructions in the appropriate machine instructions.

Pseudoinstruction	Machine instruction
beqz rs,A	**beq rs,$zero,A**
bgt rs,rt,A	**slt $at,rt,rs** **bne $at,$zero,A**

Tabla 3. Translations of pseudoinstructions **beqz** y **bgt** in MIPS machine instructions

With these instructions conditional and iterative structures equivalent to those written at high level can be written. For example, if there is an instruction subset A1, A2 ... which are

only to be executed if the contents of a register $r is negative, a branching fork can be used if the opposite condition is given (**$r ≥ 0**):

```
            bgez $r,L
            A1
            A2
            …
L:
```

To iterate *n* times the A1, A2 ... instructions, you can use a register $r and write:

```
            li $r,n
bucle:      A1
            A2
            …
            addi $r,$r,-1
            bgtz $r,bucle
```

A table with the translation of different flow control structures can be found in the annex at the end of this document.

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.

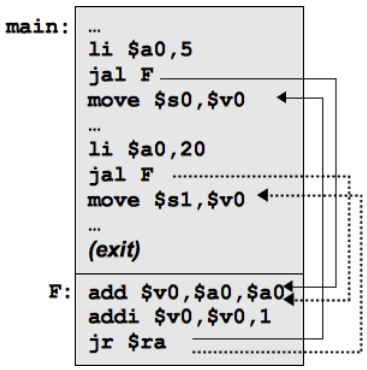


Figure 2. On the left, you can see the schema of a main () program that calls a function F from two different points in the code. At high level language this would be expressed as **`Int F(int a){return 2*a+1}`**. In both cases, instruction **`jal`** F stores the return address in **`$ra`**. Consequently,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

In 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 3. Agreement for the use of registers

System Calls

Write operations on the display or keyboard read operations access to parts of the computer that, for security and efficiency reasons, are not visible for current programs. Most computers, real or simulated, have input/output devices and the operating system offers a catalog of functions to manage them. Later, in terms of input / output, we will study the details.

In a MIPS, these system functions can be called using the **`syscall`** instruction. Each function is distinguished by a code that identifies it (called index), and it can accept a series of arguments and returns a possible result.

The calling mechanism is illustrated below with an example. The program reads an integer value from the keyboard and copies it to the memory address labeled with the name:

valor:

```
li $v0, 5          # Index for syscall read_int
syscall            # syscall for function read_int
sw $v0, valor      # Copying the integer value in memory
```

The list of available functions of the simulated system in PCSpim is shown in Table 5:

Service	Index Call	Arguments	Result
print_int	1	$a0 = integer	
print_float	2	$f12 = float	
print_double	3	$f12 = double	
print_string	4	$a0 = string	
read_int	5		integer (in $v0)
read_float	6		float (in $f0)
read_double	7		double (in $f0)
read_string	8	$a0 = buffer $a1 = length	
sbrk	9	$a0 = amount	address (in $v0)
exit	10		

Table 4. System calls for MIPS.

However, in this session we will work only with the four functions listed in Table 6. Note that the index that identifies the type of service is always indicated in register $v0, some calls read the parameter contained in the register $a0 and, if they return a result, always do it in $ v0:

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	—

Tabl6 5. System calls used in Lab Session3.

Therefore, the use of system functions is very similar to the functions of the program; 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.

Lab exercises

Exercise 1: System calls and program functions

Open and read the following code contained in the file "02_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?

Ejercise 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 Prompt 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 Prompt(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, Prompt 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 Prompt, 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);
Prompt('R',C);
Exit();
```

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

Figure 1. 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);
Prompt('R',R);
Exit();
```

Figure 2. 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.

Miscellaneous issues

These are questions for a pencil and paper, but in some cases you can check them using the simulator. You can solve them in the lab, if you have enough time, or solve them at home.

Instructions and pseudoinstructions

1. If a pseudoinstruction **`ca2 rt, rs`** was required to do the operation rt = complement_a_2 (rs), how would it be translated? Is there any standard MIPS pseudo-instruction equivalent to **`ca2`**?
2. With the help of the simulator, try loading a piece of code where the pseudo-instruction li $ 1.20 or li $ at, 20 appears. What happens?
3. How will a hypothetical pseudo-instruction such as **`beqi $ t0,4, eti`** (jump to eti if $ t0 = 4) be translated?
4. How will the pseudo-instruction b eti, (branch, unconditional jump to eti) be translated into conditional branching instructions of the I format, without using the j (jump) instruction?
5. Can you explain the difference between the calls **`print_char (100)`** and **`print_integer (100)`**?
6. Which is the difference between **`print_char ('A')`** and **`print_integer ('A'`**)?
7. In Table 3 you can see the translation of two of the six pseudo-instructions shown in Table 2. Which is the translation of the four missing?

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);
   Prompt('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 PromptM function, which is expressed in pseudocode as follows:

```
void PromptM(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 *$r*,*n*** **bucle:** **A** **addi *$r*,*$r*,-1** **bgtz *$r*,bucle** **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 values	**$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** = puntero a la cadena	—	**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		**$a0** = character read		**getc();**

NOTE. The asterisk (*) in Print * and Lee * shows that, in addition to the input / output operation, there is a binary to alphanumeric change representation or vice versa.