Lab Session 2

SYSTEM CALLS

Goals

- To reinforce knowledge of ASCII encoding
- Making inventory of flow control instructions.
- To use flow control instructions to implement loops and conditions in assembly language.
- Understanding how instructions, data and, system calls are combined to make a program.
- 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

Material

- *pcspim-ES* Simulator
- Source codes:
  - For the lab session: *forever.s*, *ascii-console.s*, *ascii-keyboard.s*,
  - Homework or extension exercises: *readuint-dec.s*.

Introduction

ASCII Code

The current Unicode standard has been able to encode texts from a multitude of languages since 1991. It is the result of the evolution of the American standard ASCII (American Standard Code for Information Interchange) defined in 1963. The original ASCII standard evolved to better fit the needs of digital storage and communication systems.

Among the characteristics of the first versions of ASCII (Table 1), we can highlight:

- It encodes characters using 7 bits and adds an additional bit for parity. Then, the basic data for storage or transmission is a byte (8 bits)
- There are 128 different codes. The first 32 codes (from 0 to 31) and the last one (127) are reserved for control and they don´t represent any alphanumeric character.
- The rest 95 codes are used for representing Anglo-Saxon alphanumeric characters and punctuation marks. Therefore, in its origin it didn´t encode characters as ñ, ç or accent marks.
- Following an alphabetic order upper case characters have consecutive codes and the same occurs with lower case ones For example, ascii(‘B’) = ascii(‘A’) + 1; ascii(‘d’) = ascii(‘a’) + 3
- Numbers from ‘0’ to ‘9’ also are represented with consecutive codes. So, ascii(‘7’) = ascii(‘0’) + 7.

	_0	_1	_2	_3	_4	_5	_6	_7	_8	_9	_A	_B	_C	_D	_E	_F
0_	*NUL* 00 0	*SOH* 01 1	*STX* 02 2	*ETX* 03 3	*EOT* 04 4	*ENQ* 05 5	*ACK* 06 6	*BEL* 07 7	*BS* 08 8	*HT* 09 9	*LF* 0A 10	*VT* 0B 11	*FF* 0C 12	*CR* 0D 13	*SO* 0E 14	*SI* 0F 15
1_	*DLE* 10 16	*DC1* 11 17	*DC2* 12 18	*DC3* 13 19	*DC4* 14 20	*NAK* 15 21	*SYN* 16 22	*ETB* 17 23	*CAN* 18 24	*EM* 19 25	*SUB* 1A 26	*ESC* 1B 27	*FS* 1C 28	*GS* 1D 29	*RS* 1E 30	*US* 1F 31
2_	(SP) 20 32	! 21 33	" 22 34	# 23 35	$ 24 36	% 25 37	& 26 38	' 27 39	(28 40	) 29 41	* 2A 42	+ 2B 43	, 2C 44	- 2D 45	. 2E 46	/ 2F 47
3_	0 30 48	1 31 49	2 32 50	3 33 51	4 34 52	5 35 53	6 36 54	7 37 55	8 38 56	9 39 57	: 3A 58	; 3B 59	< 3C 60	= 3D 61	> 3E 62	? 3F 63
4_	@ 40 64	A 41 65	B 42 66	C 43 67	D 44 68	E 45 69	F 46 70	G 47 71	H 48 72	I 49 73	J 4A 74	K 4B 75	L 4C 76	M 4D 77	N 4E 78	O 4F 79
5_	P 50 80	Q 51 81	R 52 82	S 53 83	T 54 84	U 55 85	V 56 86	W 57 87	X 58 88	Y 59 89	Z 5A 90	[5B 91	\ 5C 92	] 5D 93	^ 5E 94	_ 5F 95
6_	` 60 96	a 61 97	b 62 98	c 63 99	d 64 100	e 65 101	f 66 102	g 67 103	h 68 104	i 69 105	j 6A 106	k 6B 107	l 6C 108	m 6D 109	n 6E 110	o 6F 111
7_	p 70 112	q 71 113	r 72 114	s 73 115	t 74 116	u 75 *117*	v 76 118	w 77 119	x 78 120	y 79 121	z 7A 122	{ 7B 123	| 7C 124	} 7D 125	~ 7E 126	*DEL* 7F 127

Table 1. The 7 bits ASCII Code. The 33 shaded cells correspond to non-printable control characters. (SP) is the space between words.

Later, ASCII code was extended using an extra bit, total 8 bits. This is the standard ISO/IEC 8859. This new encoding includes 128 more codes, 32 of them are used for more control characters and the rest 96 codes represent new characters and punctuation marks for different languages. This standard defines different regional variants, therefore in

Western Europe the part IEC_8859-1 (also called *latin1*) is used. Appendix 1 shows the complete representation of characters using 8 bits ASCII. Currently, this encoding is included in the Unicode standard.

System Calls

Computers have an operating system that offers a catalog of processes also called system functions or function calls. With them, it is possible safely and efficiently access shared computer resources as the processor, main memory and peripherals. In the last block of this subject regarding input / output, we will study some implementation details.

The PC-Spim simulator has two text peripherals: the keyboard and the console, see Figure 1. They both use the ISO/IEC 8859-1 standard to encode the characters. The keyboard, in addition to alphanumeric codes, generates control codes by combining the ctrl key with the alphabetic keys. The simulator directly interprets the cursor keys and ctrl-C and therefore the simulated programs cannot read them.

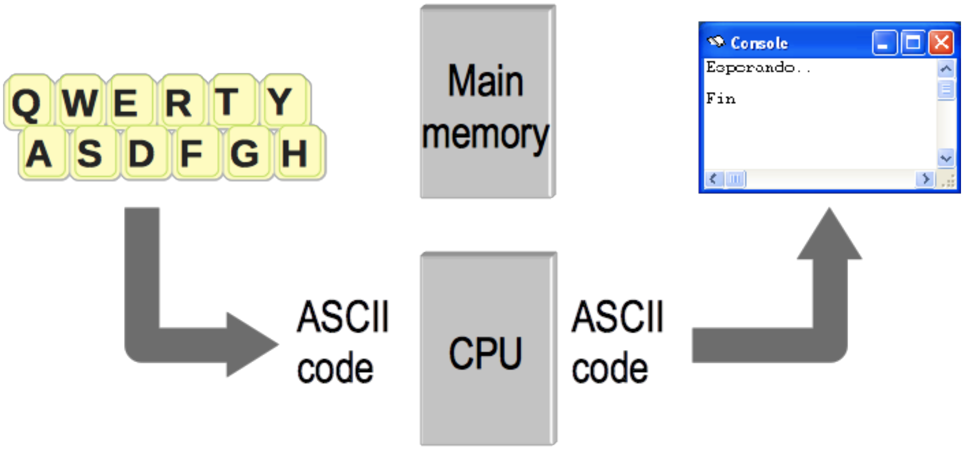


Figure 1. PC-Spim simulates a computer with a keyboard and screen that use 8 bits ASCII for encoding

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.

All the simulated system functions in PCSpim can be found in the Appendix 2 of this document. In this practice we are working only with the five functions referred to in Table 2. Note that, i) the index that identifies them must be always placed in the $v0 register, ii) some calls take the parameter contained in the $a0 register and, iii) the calls that return a result do this using 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	—
read_char	*12*	Reads a character	—	**$v0** = character

Table 2. System calls used in Lab Session 2.

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
```

Some details of the Input/Output System Calls

The functions *print_char* and *read_char* don´t change the format of the parameter. That is, *print_char* prints the code that receives in **$a0**, and *read_char* returns in **$v0** the code generated by the keyboard. However, *print_int* transforms the integer number received in **$a0** in the corresponding ASCII string of characters that can be read by humans. In a similar way, *read_int* transforms a string of characters typed by humans and calculates the corresponding integer number which is returned in **$v0**.

Another detail is *eco*. The function *read_int*, in addition to reading from the keyboard, writes the characters read on the console, so creating the illusion that the user is writing on the screen. The *read_char* function, in pcspim-ES, does not generate any echo.

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 ≥). This instruction is shown in Table 3. 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 2. Conditional branches in MIPS

This set of conditions can be extended using the arithmetic **`slt`** (set on less than) instruction. Thus, we obtain the other six pseudo instructions that can be seen in Table 4:

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 3. Conditional Branch Pseudoinstructions inMIPS

Table 5 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 5. Translations of pseudoinstructions **beqz** y **bgt** in MIPS machine instructions

Finally, the MIPS instruction set includes a pseudoinstruction for unconditional branch (*b*), see Table 6. This pseudoinstruction performs similar to the unconditional jump instruction *j*. The pseudoinstruction b is translated into a conditional branch instruction where the condition is always true, for instance: : **b eti** can be translated as **beq $0,$0,eti**.

`b A`
true

Table 6. Pseudoinstrucción for unconditional branch in MIPS

With these instructions any 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 have 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 Appendix 3 at the end of this document.

Lab exercises

Simulator pcspim-ES settings

When starting the simulator, first check that the settings are the same that the shown in Figure 2 (*Simulator->Settings...*).

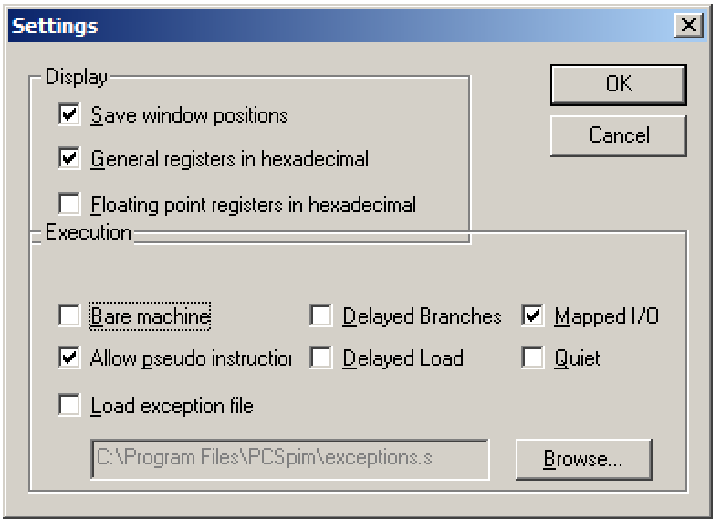


Figure 2. *pcspim-ES* settings for this session

Exercise 1: Infinite loop

Download the file *forever.s*. and read carefully the code of the program. Note that only the instructions segment is included (**.text**). Before start the simulation of this program, try to understand the code. The program just adds the integer numbers read from the console.

Look in the code for the four System Call instructions used and identify the action each one performs. Check: i) the use of $v0 to set the code of the selected System Call, ii) the use of $a0 to send the argument to be printed and, iii) the use of $v0 to get the input parameter (read number from console)

- Notice that the code includes a loop. ¿Which is the first instruction in the loop? ¿And the last one?
- ¿What do each iteration of the loop perform? ¿Could you explain the actions that each instruction and pseudoinstruction perform?

```
          .globl __start
          .text 0x00400000

__start:
          li $s0, 0
bucle:
          li $v0,5
          syscall
          addu $s0,$s0,$v0
          li $v0,1
          move $a0,$s0
          syscall
          li $v0,11
          li $a0,10
          syscall
          b bucle

          li $v0,10
          syscall
```

```
$s0 = 0;
do {
      $v0 = read_int();
      $s0 = $s0 + $v0;
      print_int($s0);
      print_char('\n');
} forever;
exit();
```

Source code and pseudo-code of *forever.s*

Notice, that the loop is infinite and the program never executes the last two instructions (**li**, **syscall**).

Now, load the program using the simulator. To answer the next questions you have to interpret the information provided by the simulator in different windows, as can be seen in Figure 3.

- How the line *b bucle* is translated?
- Do you know how to run the complete program? Do it using commad *Go* or F5 control key. Keep active the *Console* window while the program is running. Notice that the program is waiting for the inputs (integer numbers) but there is not any dialog that permits the user interaction. Enter integer signed numbers.

  Experimental technique: ***ctrl-C***. When the Console window is active, *ctrl-C* stops the program in the current instruction in execution, like it happens when using the Unix console. It has the same effect than *Simulator>Break*.

- Stop the program execution. The message *"Execution paused by the user at* <address> *Continue execution?"* will appear. Take note of this address and press the *No* button. Now, observe the main window of the Simulator (Figure 4) and look there the answer to the following questions:
  - Which is the last instruction executed? Use the noted address and look for the instruction in this instructions window.

- Which will be the next instruction to be executed at the moment of the break? Check the current Program Counter (PC) value and look for that instruction.
- Which is the content of the **$a0**, **$v0** y **$s0** registers in this moment? Look for the value in the processor window. Do you know how to change the decimal numbers to their hexadecimal encoding? Use the settings window Simulator>*Settings>Display*)

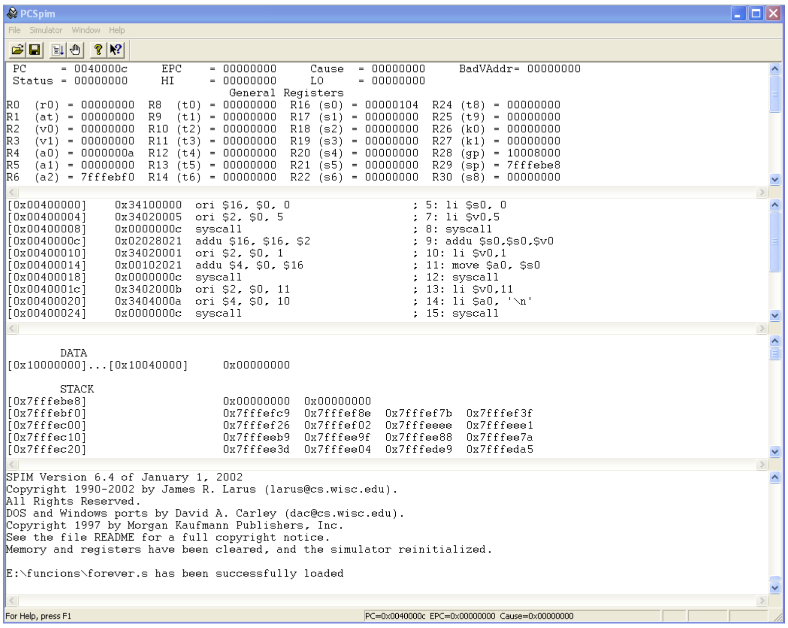


Figure 3. The simulator interface shows 4 sections. From top to bottom: (1) processor status, content of the most important registers; (2) instruction memory; (3) data memory and (4)

- Let´s continue running the program. You will see that the command *Go* (F5 key), proposes as *Starting Address* the address of the next instruction to be executed; i. e. the current value of the PC.
- If everything is clear, now you can move to Exercise 2.

Exercise 2: Loop control

Now, you have to modify the code to stop the loop when the user enters a 0. To do this you have to include a conditional branch to execute the instructions that implement the Syscall *exit()*. For this, use initially the code *forever* and store the modified code in the file named *"break.s"*:

```
$s0 = 0;
do {
        $v0 = read_int();
        if ($v0=0) break; ←
        $s0 = $s0 + $v0;
        print_int($s0);
        print_char('\n');
} forever;
exit();
```

break.s Pseudocode (exercise 2), the marked line with ← has been added to *forever.s* pseudocode (exercise 1).

- Add a label at the end of the loop for continuing the program in case *$v0 = 0*. Choose an appropriate name for this label (*end, exit*, etc). Do you know where you have to place it?
- Add now the appropriate conditional branch instruction.
- Check that the program works correctly and it stops when a 0 is enter by the keyboard.
- When the execution finishes, which are the values contained in **v0** and **$s0**? Which is PC value?

Exercise 3: Loop and counter

Let´s improve now the interface between the code *break.s* and the user. We want to count the number of enter numbers while the program is running and, print the final sum and the number of addends. The following dialogue on the screen must be reproduced:

```
$s0 = 0;
$s1 = 0
do {
        print_int($s1+1);
        print_char('>');
        $v0 = read_int();
        if ($v0=0) break;
        $s0 = $s0 + $v0;
        $s1 = $s1 + 1;
} forever;
print_char('=');
print_int($s0);
print_char('\n');
print_char('n');
print_char('=');
print_int($s1);
exit();
```

```
1>89
2>-230
3>67
4>0
=-74
n=3
```

Pseudocode and example of *counter.s*. This program reads integer numbers until a 0 is entered. Register $s1 contains the number of addends. The final sum is s printed when the loop finishes.

- Starting from the code of *break.s*, include the new instructions and store the new program as *counter.s*.
- Add instructions for initialize, increase and print the value of register **$s1**.
- Move out the loop the instructions for printing the content of **$s0** and complete the final code.
- Run it and check its correctness

Exercise 4. ASCII codes in the screen

Let´s consider the following loop written in high level language:

```
for ($s0=32; $s0<127; $s0=$s0+1) {…}
```

In MIPS assembly language and pseudocode it can be written as:

```
        li $s0,32
        li $s1,127
bucle:
        .......

        addi $s0,$s0,1
        blt $s0,$s1,bucle
```

```
$s0=32;
do {  …
      …
      $s0 = $s0+1;
}
while ($s0<127);
```

The loop *for* in exercise 4 uses a counter (**$s0**) that increases 1 in each iteration and a register (**$s1)** containing the limit value (**$s1=127**). In this case the loop condition (guarda) is easy and can be translate using a single conditional branch instruction.

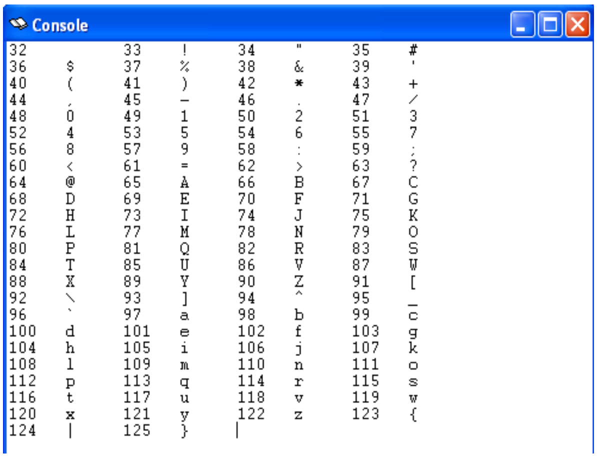
The code *ascii-console.s* prints the 7 bits ASCII code graphic characters, as shown in Figure 4. In this loop the codes from 0 to 31, the 127 code (**DEL**) and the extended characters from 128 to 255 are omitted.

```
        li $s0,32
        li $s1,127
bucle:
        li $v0,1
        move $a0,$s0
        syscall
        li $v0,11
        li $a0,9
        syscall
        li $v0,11
        move $a0,$s0
        syscall
        li $v0,11
        li $a0,10
        syscall

        addi $s0,$s0,1
        blt $s0,$s1,bucle

        li $v0,10
        syscall
```

```
for ($s0=32; $s0<127; $s0=$s0+1){
        print_int($s0);
        print_char('\t'); // Tab
        print_char($s0);
        print_char('\n'); // line feed
        }
exit();
```

Figure 4. Code and pseudocode for de *ascii-console.s*

- Open the *ascii-console.s* file with any text editor and study the *for* structure contained. Use the simulator to check the program
- Notice the use of the control characters '\t' (code 9) and '\n' (code 10).
- Which is the representable set of Ascii characters in the screen? Change now the limits of the loop to check the codes from 0 to 255. Run the program and watch the result. When the console cannot print a character it shows ▮.
- Now, modify the *ascii-console.s* program to list the characters in inverse order, meaning from 126 to 32. Check the correctness of the new code using the simulator.
- Finally, modify the *ascii-console.s* code (and store as *ascii-console-tab.s*) to tabulate the codes from 32 to 126 in the screen as Figure 5 represents:




```
$s2=4;
for ($s0=32; $s0<127;
$s0=$s0+1){
        $s2 = $s2–1;
        print_int($s0);
        print_char('\t');
        print_char($s0);
        if ($s2==0)
                $s2 = 4;
                print_char('\n');
        else
                print_char('\t');
}
exit();
```

Figure 5. *ascii-console-tab.s* pseudocode and result

- Check the correctness of *ascii-console-tab.s* code

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 during the session, if you have enough time, or solve them at home.

Instructions and pseudoinstructions

1. Which output is produced by the following code?

```
        li $s0,'a'
        li $s1,10
bucle:
        li $v0,11
        move $a0,$s0
        syscall

        addi $s0,$s0,1
        addi $s1,$s1,-1
        bgtz $s1,bucle
```

   - What happens if the line **li $v0,11** is replaced by **li $v0,1**?
   - What happens if the line **addi $s0,$s0,1** is replaced by **addi $s0,$s0,-1**?
   - What happens if the line **addi $s1,$s1,-1** is replaced by **addi $s1,$s1,-2**?

2. Complete the following code to read characters from the keyboard and make the eco of the read characters only when they are numbers; the program ends when the “f” key is pressed.

```
__start:
      li $s0,'0'
      li $s1,'9'
      li $s2,'f'
```

```
bucle:
      li $v0,12
      syscall
      bif $v0,$s2,fin
      bif $v0,$s0,bucle
      bif $v0,$s1,bucle
      move $a0,$v0
      li $v0,11
      syscall
      b bucle
fin:
      li $v0,10
      syscall
```

3. If a pseudoinstruction **`ca2 rt, rs`** is required in order to do the operation rt = complement_a_2 (rs), how would it be translated? Is there any standard MIPS pseudo-instruction equivalent to **`ca2`**?
4. 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?
5. How will a hypothetical pseudo-instruction such as **`beqi $ t0,4, eti`** (jump to eti if $ t0 = 4) be translated?
6. Could you explain the difference between the calls **`print_char (100)`** and **`print_integer (100)`**?
7. Which is the difference between **`print_char ('A')`** and **`print_integer ('A')`**?
8. 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?

Appendix 1

ASCII (ISO/IEC 8859-1) (Used in PC-Spim)

<table>
<tr><th></th><th>_0</th><th>_1</th><th>_2</th><th>_3</th><th>_4</th><th>_5</th><th>_6</th><th>_7</th><th>_8</th><th>_9</th><th>_A</th><th>_B</th><th>_C</th><th>_D</th><th>_E</th><th>_F</th></tr>
<tr><td>0_</td><td>NUL
00
0</td><td>SOH
01
1</td><td>STX
02
2</td><td>ETX
03
3</td><td>EOT
04
4</td><td>ENQ
05
5</td><td>ACK
06
6</td><td>BEL
07
7</td><td>BS
08
8</td><td>HT
09
9</td><td>LF
0A
10</td><td>VT
0B
11</td><td>FF
0C
12</td><td>CR
0D
13</td><td>SO
0E
14</td><td>SI
0F
15</td></tr>
<tr><td>1_</td><td>DLE
10
16</td><td>DC1
11
17</td><td>DC2
12
18</td><td>DC3
13
19</td><td>DC4
14
20</td><td>NAK
15
21</td><td>SYN
16
22</td><td>ETB
17
23</td><td>CAN
18
24</td><td>EM
19
25</td><td>SUB
1A
26</td><td>ESC
1B
27</td><td>FS
1C
28</td><td>GS
1D
29</td><td>RS
1E
30</td><td>US
1F
31</td></tr>
<tr><td>2_</td><td>(SP)
20
32</td><td>!
21
33</td><td>"
22
34</td><td>#
23
35</td><td>$
24
36</td><td>%
25
37</td><td>&
26
38</td><td>'
27
39</td><td>(
28
40</td><td>)
29
41</td><td>*
2A
42</td><td>+
2B
43</td><td>,
2C
44</td><td>-
2D
45</td><td>.
2E
46</td><td>/
2F
47</td></tr>
<tr><td>3_</td><td>0
30
48</td><td>1
31
49</td><td>2
32
50</td><td>3
33
51</td><td>4
34
52</td><td>5
35
53</td><td>6
36
54</td><td>7
37
55</td><td>8
38
56</td><td>9
39
57</td><td>:
3A
58</td><td>;
3B
59</td><td><
3C
60</td><td>=
3D
61</td><td>>
3E
62</td><td>?
3F
63</td></tr>
<tr><td>4_</td><td>@
40
64</td><td>A
41
65</td><td>B
42
66</td><td>C
43
67</td><td>D
44
68</td><td>E
45
69</td><td>F
46
70</td><td>G
47
71</td><td>H
48
72</td><td>I
49
73</td><td>J
4A
74</td><td>K
4B
75</td><td>L
4C
76</td><td>M
4D
77</td><td>N
4E
78</td><td>O
4F
79</td></tr>
<tr><td>5_</td><td>P
50
80</td><td>Q
51
81</td><td>R
52
82</td><td>S
53
83</td><td>T
54
84</td><td>U
55
85</td><td>V
56
86</td><td>W
57
87</td><td>X
58
88</td><td>Y
59
89</td><td>Z
5A
90</td><td>[
5B
91</td><td>\
5C
92</td><td>]
5D
93</td><td>^
5E
94</td><td>_
5F
95</td></tr>
<tr><td>6_</td><td>`
60
96</td><td>a
61
97</td><td>b
62
98</td><td>c
63
99</td><td>d
64
100</td><td>e
65
101</td><td>f
66
102</td><td>g
67
103</td><td>h
68
104</td><td>i
69
105</td><td>j
6A
106</td><td>k
6B
107</td><td>l
6C
108</td><td>m
6D
109</td><td>n
6E
110</td><td>o
6F
111</td></tr>
<tr><td>7_</td><td>p
70
112</td><td>q
71
113</td><td>r
72
114</td><td>s
73
115</td><td>t
74
116</td><td>u
75
117</td><td>v
76
118</td><td>w
77
119</td><td>x
78
120</td><td>y
79
121</td><td>z
7A
122</td><td>{
7B
123</td><td>|
7C
124</td><td>}
7D
125</td><td>~
7E
126</td><td>DEL
7F
127</td></tr>
<tr><td>8_</td><td>PAD
80
128</td><td>HOP
81
129</td><td>BPH
82
130</td><td>NBH
83
131</td><td>IND
84
132</td><td>NEL
85
133</td><td>SSA
86
134</td><td>ESA
87
135</td><td>HTS
88
136</td><td>HTJ
89
137</td><td>VTS
8A
138</td><td>PLD
8B
139</td><td>PLU
8C
140</td><td>RI
8D
141</td><td>SS2
8E
142</td><td>SS3
8F
143</td></tr>
<tr><td>9_</td><td>DCS
90
144</td><td>PU1
91
145</td><td>PU2
92
146</td><td>STS
93
147</td><td>CCH
94
148</td><td>MW
95
149</td><td>SPA
96
150</td><td>EPA
97
151</td><td>SOS
98
152</td><td>SGCI
99
153</td><td>SCI
9A
154</td><td>CSI
9B
155</td><td>ST
9C
156</td><td>OSC
9D
157</td><td>PM
9E
158</td><td>APC
9F
159</td></tr>
<tr><td>A_</td><td>(NBSP)
A0
160</td><td>¡
A1
161</td><td>¢
A2
162</td><td>£
A3
163</td><td>¤
A4
164</td><td>¥
A5
165</td><td>¦
A6
166</td><td>§
A7
167</td><td>¨
A8
168</td><td>©
A9
169</td><td>ª
AA
170</td><td>«
AB
171</td><td>¬
AC
172</td><td>(SHY)
AD
173</td><td>®
AE
174</td><td>¯
AF
175</td></tr>
<tr><td>B_</td><td>°
B0
176</td><td>±
B1
177</td><td>²
B2
178</td><td>³
B3
179</td><td>´
B4
180</td><td>µ
B5
181</td><td>¶
B6
182</td><td>·
B7
183</td><td>¸
B8
184</td><td>¹
B9
185</td><td>º
BA
186</td><td>»
BB
187</td><td>¼
BC
188</td><td>½
BD
189</td><td>¾
BE
190</td><td>¿
BF
191</td></tr>
<tr><td>C_</td><td>À
C0
192</td><td>Á
C1
193</td><td>Â
C2
194</td><td>Ã
C3
195</td><td>Ä
C4
196</td><td>Å
C5
197</td><td>Æ
C6
198</td><td>Ç
C7
199</td><td>È
C8
200</td><td>É
C9
201</td><td>Ê
CA
202</td><td>Ë
CB
203</td><td>Ì
CC
204</td><td>Í
CD
255</td><td>Î
CE
206</td><td>Ï
CF
207</td></tr>
<tr><td>D_</td><td>Ð
D0
208</td><td>Ñ
D1
209</td><td>Ò
D2
210</td><td>Ó
D3
211</td><td>Ô
D4
212</td><td>Õ
D5
213</td><td>Ö
D6
214</td><td>×
D7
215</td><td>Ø
D8
216</td><td>Ù
D9
217</td><td>Ú
DA
218</td><td>Û
DB
219</td><td>Ü
DC
220</td><td>Ý
DD
221</td><td>Þ
DE
222</td><td>ß
DF
223</td></tr>
<tr><td>E_</td><td>à
E0
224</td><td>á
E1
225</td><td>â
E2
226</td><td>ã
E3
227</td><td>ä
E4
228</td><td>å
E5
229</td><td>æ
E6
230</td><td>ç
E7
231</td><td>è
E8
232</td><td>é
E9
233</td><td>ê
EA
234</td><td>ë
EB
235</td><td>ì
EC
236</td><td>í
ED
237</td><td>î
EE
238</td><td>ï
EF
239</td></tr>
<tr><td>F_</td><td>ð
F0
240</td><td>ñ
F1
241</td><td>ò
F2
242</td><td>ó
F3
243</td><td>ô
F4
244</td><td>õ
F5
245</td><td>ö
F6
246</td><td>÷
F7
247</td><td>ø
F8
248</td><td>ù
F9
249</td><td>ú
FA
250</td><td>û
FB
251</td><td>ü
FC
252</td><td>ý
FD
253</td><td>þ
FE
254</td><td>ÿ
FF
255</td></tr>
<tr><td></td><td>_0</td><td>_1</td><td>_2</td><td>_3</td><td>_4</td><td>_5</td><td>_6</td><td>_7</td><td>_8</td><td>_9</td><td>_A</td><td>_B</td><td>_C</td><td>_D</td><td>_E</td><td>_F</td></tr>
</table>

Appendix 2

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.

(**) En *pcspim-ES*, la función 12 lee un carácter del teclado sin producir un eco en la consola. En otras versiones del simulador sí escribe el eco.

Appendix 3

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**