Félix García Carballeira, Alejandro Calderón Mateos, José Daniel García Sánchez,
María Soledad Escolar Díaz, Luis Miguel Sánchez García, Rubén Saá Álvarez, María-Cristina Marinescu

Lab Work 2. MIPS assembly and introduction to PCSpim

The goal of this work is for the student to become familiar with the data types and the programming in assembly (MIPS32). To realize this lab work you will be using the SPIM simulator at:

http://pages.cs.wisc.edu/~larus/spim.html

SPIM is a self-contained simulator which executes programs written in MIPS32. It provides a debugger and a minimal number of OS services.

1. SPIM

The Windows SPIM interface looks similar to the capture below:

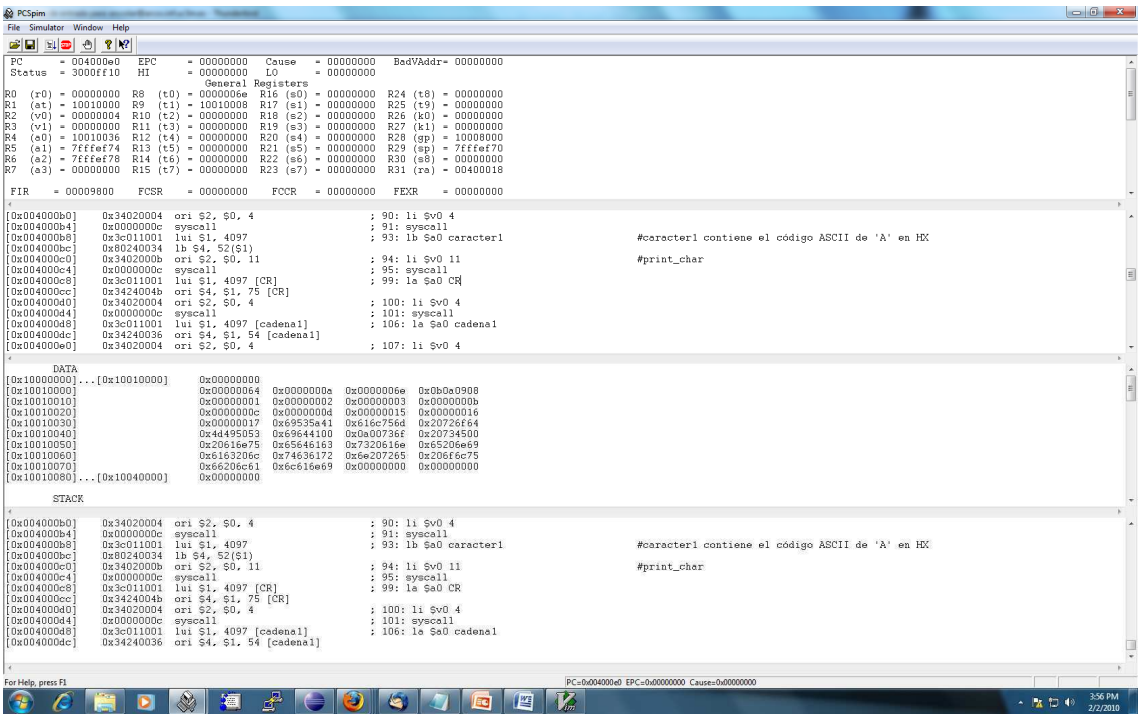

Figura 1 Interfaz del simulador SPIM

There are four horizontal panels as follows:

1. MIPS registers: the contents of the register file. The PC is shown on the first line. There are also several other special purpose registers. Following are the 32 general purpose integer registers R0 to R31. At the very end there are the 32 floating point registers (for simple as well as double precision).

```
 PC       = 00400020   EPC      = 00000000   Cause    = 00000000   BadVAddr= 00000000
 Status   = 3000ff10   HI       = 00000001   LO       = 0000004e
                                 General Registers
R0  (r0) = 00000000  R8  (t0) = 00000009  R16 (s0) = 00000000  R24 (t8) = 00000000
R1  (at) = 00000000  R9  (t1) = 00000024  R17 (s1) = 00000000  R25 (t9) = 00000000
R2  (v0) = 0000000a  R10 (t2) = 00000009  R18 (s2) = 00000000  R26 (k0) = 00000000
R3  (v1) = 00000000  R11 (t3) = 0000009d  R19 (s3) = 00000000  R27 (k1) = 00000000
R4  (a0) = 00000006  R12 (t4) = 00000002  R20 (s4) = 00000000  R28 (gp) = 10008000
R5  (a1) = 7ffff6dc  R13 (t5) = 00000001  R21 (s5) = 00000000  R29 (sp) = 7ffff6d8
R6  (a2) = 7ffff6ec  R14 (t6) = 00000000  R22 (s6) = 00000000  R30 (s8) = 00000000
R7  (a3) = 00000000  R15 (t7) = 00000000  R23 (s7) = 00000000  R31 (ra) = 00400018

 FIR     = 00009800    FCSR     = 00000000    FCCR     = 00000000    FEXR     = 00000000
```

2. Text segment: includes the assembly code. Every pseudo-instruction occupies one line and is identified by the memory address on the very left end of the line. The first memory address for the .text segment is 0x00400000.

```
[0x00400000]    0x8fa40000  lw $4, 0($29)                ; 183: lw $a0 0($sp)              # argc
[0x00400004]    0x27a50004  addiu $5, $29, 4             ; 184: addiu $a1 $sp 4            # argv
[0x00400008]    0x24a60004  addiu $6, $5, 4              ; 185: addiu $a2 $a1 4            # envp
[0x0040000c]    0x00041080  sll $2, $4, 2                ; 186: sll $v0 $a0 2
[0x00400010]    0x00c23021  addu $6, $6, $2              ; 187: addu $a2 $a2 $v0
[0x00400014]    0x0c100009  jal 0x00400024 [main]        ; 188: jal main
[0x00400018]    0x00000000  nop                          ; 189: nop
[0x0040001c]    0x3402000a  ori $2, $0, 10               ; 191: li $v0 10
[0x00400020]    0x0000000c  syscall                      ; 192: syscall                    # syscall 10 (exit)
[0x00400024]    0x34080000  ori $8, $0, 0                ; 7: li $t0, 0
[0x00400028]    0x34090000  ori $9, $0, 0                ; 8: li $t1, 0
[0x0040002c]    0x3c011001  lui $1, 4097                 ; 9: lw $t2, n
[0x00400030]    0x8c2a0024  lw $10, 36($1)
```

3. Data segment and stack: consists of the data declared in the .data segment. These are allocated linearly starting from the memory address 0x10000000. Following is the content of the stack which starts at address 0x7fffefff.

```
        DATA
[0x10000000]...[0x10010000]     0x00000000
[0x10010000]                    0x0000000c  0x00000019  0x00000024  0x00000010
[0x10010010]                    0x0000001c  0x00000025  0x0000007c  0x0000009c
[0x10010020]                    0x0000009d  0x00000009  0x00000000  0x00000000
[0x10010030]...[0x10040000]     0x00000000

        STACK
[0x7ffff6d8]                    0x00000003  0x7ffff7b9
[0x7ffff6e0]                    0x7ffff7b3  0x7ffff79a  0x00000000  0x7fffffe1
[0x7ffff6f0]                    0x7fffffb9  0x7fffffa0  0x7fffff5f  0x7fffff28
[0x7ffff700]                    0x7ffffeec  0x7ffffebb  0x7ffffea4  0x7ffffe80
[0x7ffff710]                    0x7ffffe6c  0x7ffffe5f  0x7ffffe48  0x7ffffe1d
```

4. Debugging window: here you can visualize control and error messages.

```
See the file README for a full copyright notice.
Loaded: C:\Program Files (x86)\PCSpim\exceptions.s
C:\Users\Carlos\Documents\inversor.s successfully loaded
Memory and registers cleared and the simulator reinitialized.

SPIM Version 9.0.1 of January 2, 2011
Copyright 1990-2010, James R. Larus.
All Rights Reserved.
SPIM is distributed under a BSD license.
See the file README for a full copyright notice.
Loaded: C:\Program Files (x86)\PCSpim\exceptions.s
C:\Users\Carlos\Documents\contador.s successfully loaded
```

2. Problems

Problem 1: (to be submitted in class)

You have to use the *exercise1.s* located at the end of this document in the annex 1. Load it into the SPIM simulator, execute it, and answer the following questions:

a. What is the address in memory where the main starts in *exercise1.s*?

b. What value of the memory address does register $ra contain after executing the program? What instruction resides at this address in memory?

c. Memory address 0x10010046 contains a character from the string that is allocated in the .data segment. What value does this address contain and what is the corresponding ASCII character?

d. At what address can you find the first “p” character from the string that is allocated in the data segment?

e. Which memory address corresponds to **hueco+4?**

f. Which memory address corresponds to **cadena+7?**

g. Write up a short program to print the third byte of the string on the screen.

Problem 2:

You have to use the *exercise2.s* located at the end of this document in the annex 2. This file consists of an algorithm written in MIPS32. Load it into the SPIM simulator, execute it, and analyze the result. In memory.pdf answer the following questions:

a. What is the purpose of the algorithm? What is the meaning of the result which is displayed on the screen?

b. What are registers $t0 and $t1 used for? What’s the difference between them? Why are both of them used?

c. What does the following instruction do and why is it used in the implementation of the algorithm: mfhi $t5

d. Given the following table, fill in the values of the registers before executing the instruction bge $t0, $t2, etiqueta3

REGISTERS	Iteration 1	Iteration 2	...	Iteration n
R8 ($t0)				
R9 ($t1)				
R11 ($t3)				
R13 ($t5)				
PC				

Problem 3:

Develop the MIPS32 assembly code for the following functionality and submit it in a file with the name *exercise3.s*:

a. Print on the screen the last string character.

b. Print on the screen the number of characters in the string (without the end-of-string character).

c. Replace the spaces in the string with "-" and print the result on the screen.

The strings which you use for testing your program must be declared in the .data segment to be of .asciiz type as follows:

```
.data
     testString1:          .asciiz "this is a test"
```

For this string the result would be:

```
t                    (last character of the string)
14                   (number of characters)
this-is-a-test       (string with spaces replaced by "-")
```

Note: the program must work correctly independent of the contents of the string.

Problem 4:

Develop the MIPS32 assembly code for the following functionality and submit it in a file with the name *ejercicio4.s*: Save into a destination array the n elements of a source array which contains integers on 32 bits (.word). The elements in the destination array must be in reverse order from those in the source array. Print on the screen all the n elements of the destination array separated by a space " ".

Important: You must use the appropriate system call to dynamically reserve the space for the destination array.

The source array must be declared in .data as follows:

```
.data
     sourceArray:    .word 10, 20, 30, 40, 50
     n:              .word 5
```

where sourceArray is the source array containing the n integer elements. The result for the example above is:

```
50 40 30 20 10
```

Note: the program must work correctly independent of the number of elements in the source array. n must always correspond to the number of integers in the source array.

ANEX 1. exercise1.s

```
.data
      var1:       .word 258
      hueco:            .space 16               bytes
      var2:       .word 0x000F0102
      var3:       .double 130.125
      var4:       .float 525.130
      var5:       .word 10
      cadena:           .asciiz "My first program"
      caracter1:  .byte 17, 18, 19
                        .align 1
      caracter2:  .byte 20, 21, 22
                        .align 2
      caracter3:  .byte 23

.text
.globl main
      main:
            lw $t0 var1
            lw $t1 var2
            l.d $f8 var3
            l.s $f16 var4
            lw $t2 var5

            add $t0,$t0,$t2
            sw $t0, hueco

            add.d $f8, $f8, $f8
            s.d $f8, hueco+4
            l.d $f10 hueco+4

            add.s $f16, $f16, $f16
            li $t3 12
            s.s $f16, hueco($t3)
            l.s $f17, hueco($t3)

            la $a0 cadena+1
            li $v0, 4
            syscall

            #reserv dynamic memory
                  li $v0, 9
                  li $a0, 4
                  syscall


            move $t4, $v0
            sw $t2, ($t4)

            jr $ra
```

ANEX 2. exercise2.s

```
.data
      a:    .word 12, 25, 36, 16, 28, 37, 124, 156, 157
      n:    .word 9
.text
.globl main
      main:
            li $t0, 0
            li $t1, 0
            lw $t2, n
            li $t4, 2
            li $a0, 0

      etiqueta1:
            bge $t0, $t2, etiqueta3
            lw $t3, a($t1)
            divu $t3, $t4
            mfhi $t5
            beqz $t5, etiqueta2
            addi $t0, $t0, 1
            addi $t1, $t1, 4
            b etiqueta1

      etiqueta2:
            addi $a0, $a0, 1
            addi $t0, $t0, 1
            addi $t1, $t1, 4
            b etiqueta1

      etiqueta3:
            li $v0, 1
            syscall
            jr $ra
```