Lab Session 1

REGISTERS AND MAIN MEMORY

Goals

• To recall the basic operation of the PCSpim simulator.

• To know the basic aspects of the MIPS architecture: register file, ALU and main memory.

• To review the bases of assembler programming of the MIPS architecture.

• To recall the use of registers, pseudoinstructions and machine instructions, main memory addresses and integer data in the MIPS assembly language.

• To check the operation of the system call mechanism by printing an integer number.

References

- D. Patterson, J. Hennessy. **Computer organization and design. The hardware/software interface**. 4 th Edition. 2009. Elsevier, 2011. (Chapter 2)

Introduction

The general purpose register file

MIPS processor has several register files but in this session we will only focus on the general purpose integer registers. This set of registers allows arithmetic operations with memory addresses and integer data. These registers (with the exception of registers **$0** and **$31**) are identical and can be used for the same things. Register **$0** is always set to zero and register **$31** is used by instruction **`jal`.** For a better understanding of assembly code and to make easier the process of compiling a program, there is an agreement for the use of registers. This agreement will be explained during this session and next ones.

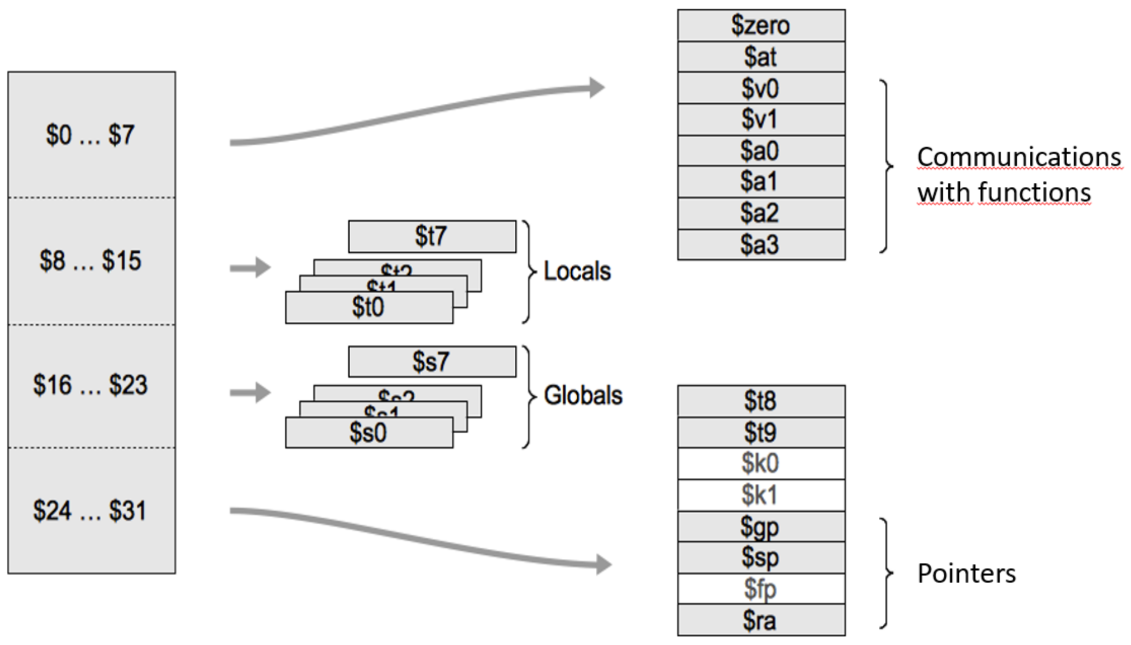


Figure 1. Agreement for register use. We can see four groups of 8 registers each: communications with functions ($0 a $7), local variables ($8 a $15), global variables ($16 a $23) and miscellaneous ($24 a $31). In the first group we can find registers $zero and $at.

Instructions and pseudo-instructions

MIPS processor executes a single instruction per CPU cycle. Many instructions perform a simple operation according their mnemonic code (arithmetic operation, read memory operation, write memory operation or branch). But sometimes a single action needs the very specific use of a single machine instruction or maybe the use of two or three consecutive machine instructions. For these cases we can use the pseudo-instructions. A pseudo-instruction permits to encode a set of basic actions in a single line of code using mnemonics and coding in a similar way that machine instructions do. The result is a more readable program. Let´s see some examples:

- The pseudo-instruction **move rs,rt** states a copy operation between two registers (*rs* = *rt*). This operation can be performed as particular case of a basic instruction, for example: **or rs,rt,$zero** or **addi rs,rt,0**.
- The pseudo-instruction **li** (*load immediate*) fixes a basic programming problem: allocates a constant **K** into a register. The programmer only needs to write a single line of code: **li $rt,K**. According to the value of **K**, we can find three different cases as Table 1 shows. Constant **K** can be up to 32 bits, **Kh** represents the most significant 16 bits of **K** and **KL** the less significant 16 bits:

Case	Example	Instruction machine
Kh=0	**K = 0x00000100** **KL = 0x0100**	`ori $rt,$0,Kl`
KL=0	**K = 0x00040000** **Kh = 0x0004**	`lui $rt,Kh`

Kh≠0 and KL≠0	**K = 0x00040010**
	KL = 0x0010
	Kh = 0x0004
		`lui $at,Kh`
		`ori $rs,$at,KL`

Table 1. Different translations for the pseudo-instruction **li $rt,K**, according to the value of constant **K**

- The pseudo-instruction **la** (*load address*) also solves a very important and frequent problem of programming. It copies the memory address associated to a label in a register. This is very useful for programmers because they don´t need to know the exact value of a memory address, they can simply use the name these address have along the program. For example, consider the following declaration:

```
      .data 0x2000B000
aux:   .word -1
cadena: .asciiz "Hola mundo"
```

The pseudo-instruction **la $t0,aux** copies in **$t0** the value **0x2000B000**, which is the memory address associated to the label **aux** (**aux** is an integer variable with value -1; **0xFFFFFFFF** in hexadecimal). The pseudo-instruction **la $t1,cadena** copies in **$t1** the value **0x2000B004**, which is the memory address where the ASCII code for character "**H**" in the string "**Hola mundo**" is stored.

Really this is a similar case to pseudo-instruction li, but in this case the value is a 32 bits data interpreted as a memory address. So, the translation into machine instructions will be performed by using instructions **lui** and **ori**. Consequently, the two previous pseudo-instruction **la** can be translated as follows:

```
lui $at,0x2000      # $at = 0x20000000
ori $t0,$at,0xB000  # $t0 = 0x2000B000
ori $t1,$at,0xB004  # $t1 = 0x2000B004
```

Notice that the pseudo-instructions can use the register **$1 ($at)** according to the agreement shown in Figure 1. In fact, **$at**, means *assembler temporary*. Programs cannot use explicitly this register.

Variables in main memory and assembly directives

MIPS assembly language offers the following resources to allocate the static variables of a program in main memory:

- Segment **.data** shows the start of data segment.
- Directive **.space** allows to reserve memory in data segment. This is used to declare non initialized variables.
- Directives **.byte**, **.half**, **.word**, **.ascii** y **.asciiz** allow to define and initialize variables.

Instructions and pseudo-instructions for memory data accessing

MIPS has eight instructions for access data in main memory:

data	restrictions on the address	reading with extension of sign	reading without extension of sign	Writing
byte	none	**lb**	**lbu**	**sb**
halfword	multiple of 2	**lh**	**lhu**	**sh**
word	multiple of 4	**lw**		**sw**

Table 2. Instructions for read and write memory operations with integer data

The format of these instructions is type I and they are coded as **op rt,D(rs)**. Where **D** is a 16 bits signed offset added to the content of the base register **rs** to generated the memory address where data is read or written. This encoding permits different ways to access memory data.

Absolute addressing mode. The memory address is a fixed and known position (**A**). Thus, to generate the address position (**A**) we can add **A** and register **$zero** as base. To this end, pseudo-instructions like **op rs,A** are very useful. The pseudo-instruction is translated in one or more machine instructions if A has more than 16 bits.

For example, pseudo-instruction **lw $rt,A** stores in register **$rt** the content of the memory address **A.** According to the number of bits of **A**, the translation can be different:

- If **A** is a number encoded in 16 bits, the translation is **lw $rt,A($0).**
- If **A** needs more than 16 bits to be encoded, the assembler splits A in high and low part, then a possible translation can be:

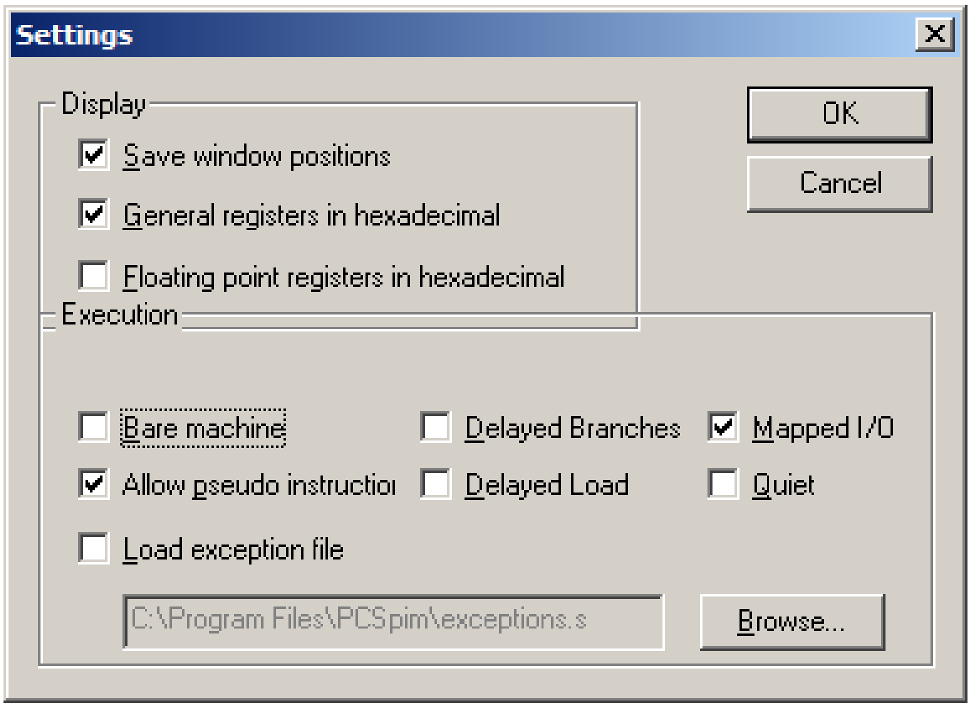
```
lui $at,Ah
lw $rt,Al($at)
```

Indirect addressing mode. The register contains the memory address of the variable. This is the programmer´s point of view when accessing to a memory address previously calculated by the program. It also permits to follow a pointer to access structured variables.

Displacement or based addressing mode. The register contains a base memory address and the programmers makes displacements about it. This addressing mode is also useful to access structured variables, the register contains de address of the variable and the offset means the displacement about the base pointer.

Lab exercices

Execute the PCSpim Simulator and check the settings to configure it as Figure 2 shows. Notice that you can select how to see the content of registers (decimal or hexadecimal).



Display
- [✓] Save window positions
- [✓] General registers in hexadecimal
- [ ] Floating point registers in hexadecimal

Execution
- [ ] Bare machine
- [ ] Delayed Branches
- [✓] Mapped I/O
- [✓] Allow pseudo instructio
- [ ] Delayed Load
- [ ] Quiet
- [ ] Load exception file

C:\Program Files\PCSpim\exceptions.s

Figura 2 . Simulator settings

Exercise 1: Variables in registers and pseudo-instructions

Assume the following program to calculate the perimeter of a rectangle known the length of its sides, 25 and 30.

```
        .globl __start
        .text 0x00400000
__start: li $t0,25
        li $t1,30
        add $s0,$t1,$t0
        add $s0,$s0,$s0
```

Start the simulator and load the program. As you can see in the code the perimeter is calculated using the expression **$s0 = 2·($t0+$t1)**. The lengths of the sides are stored in registers **$t0** and **$t1** and the result will be stored in **$s0**. Analyze how the program has been translated and assembled.

- How many machine instructions does the program include?
- Identify the machine instructions used to translate the pseudo-instructions contained in the program.
- Which is the memory address containing the instruction **add $s0,$s0,$s0**?
- What instruction of the program is encoded as 0x01288020?
- Write in hexadecimal the value of the perimeter calculated by the program.
- Modify the program to calculate the perimeter of a rectangle which lengths side are 75369 and 12976. Check the result. Identify now the machine instructions used to translate the pseudo-instructions. Why they are different?

Exercise 2: Variables in memory

Now, you work with a similar program but in this case the lengths of the rectangle sides are stored in main memory instead of registers.

```
        .globl __start
        .data 0x10000000
A:      .word 25
B:      .word 30
P:      .space 4
        .text 0x00400000
__start: la $t0,A
        la $t1,B
        la $t2,P
        lw $s0,0($t0)
        lw $s1,0($t1)
        add $s2,$s1,$s0
        add $s2,$s2,$s2
        sw $s2,0($t2)
```

Notice that labels A, B and P point to memory positions where the variables are stored. This labels can be used directly by the program. So, pseudo-instruction **la** (*load address*) permits to store the memory address (associated to the label) in a register. It is said that the register contains a pointer to the variable in main memory.

- How many bytes in main memory are occupied by the variables of the program?

- How many instructions for memory access (read/write) does the program contain?
- Which is the memory address where the perimeter value is written?
- Why the pseudo-instruction **la $t0, A** was translated into just one machine instruction and **la $t1,B** was translated in two?
- Justify the value (4) that appears in the directive **.space 4**
- If instead of the directive **.space 4** we made use of the directive .word 0, would it affect the final value of P?
- Which is the value contained in register **$t1** after the execution of the instruction **lw $s1,0($t1)**?

Exercise 3: Printing the perimeter

For user interaction the program needs to execute specific functions implemented in the operating system. To do this, the program code has to include *system calls*. Among others, these function permit to read data from the keyboard and display the results in the screen. The machine instruction **syscall** allow the execution of different system functions according to a code number previously stored in register **$v0**.
In this session we just focus on a particular system call to display a value (the perimeter, in our case) into the screen. In next sessions we will emphasize in the general mechanism for system calls.
Add the following chunk of code to the previous program and check if the result is shown in the computer screen.

```
move $a0,$s2   # copies perimeter in $a0
li $v0,1       # code for print_int
syscall        # system call
```

This code has three instructions. The first one, copies the value of the perimeter in register **$a0**. The second one sets "1" in register **$v0.** 1 is the code number for a function of the operating system that can display the data stored in **$a0** (an integer value) in the computer screen. The operating system interprets the content of **$a0** as an integer in two complements. Finally, the third instruction performs the call to the operating system that stats all actions needed to display the value in the screen.

Summarizing, a system call is a combination of three elements, a set of parameters stored in registers **$ a0,** and maybe also in **$ a1**; a code number (written in **$v0**) to identify the type of function requested; and the instruction **syscall** that triggers the system call.

- What is the encoding for the machine instruction **syscall**?
- In which instruction/s has been translated the pseudo-instruction **move $a0,$a1**
- Now replace the instruction **sw $s2,0($t2)** with **sw $s2,2($t2)**. What happens when you try to run the program? Reasoning the answer.

Exercise 4: Questions

Some of the following questions can be answered using the PCSpim simulator.

1. Which of these instructions is an incorrect translation for the pseudo-instruction **move $t0,$t1** (it copies the content of **$t1** in **$t0?**
   - **add $t0, $t1, $zero**
   - **addi $t0, $t1, 0**
   - **sub $t0, $t1, $zero**
   - **and $t0, $t1, $zero**
   - **or $t0, $t1, $zero**
   - **andi $t0, $t1, 0xFFFF**

2. Which of these instructions is a good translation for the pseudo-instruction **li $t0,100** (I stores the decimal value 100 in **$t0**)?
   - **ori $t0, $zero, 0x64**
   - **andi $t0, $zero, 0x64**
   - **addi $t0, $zero, 0x64**
   - **ori $t0, $zero, 100**
   - **addi $t0, 0x64, $zero**
   - **xori $t0, $zero, 100**
   - **andi $t0, $zero, 100**
   - **addi $t0, $zero, 100**

3. The following chunk of code causes runtime error. Explain why.

```
    li $t0, 0x10003000
    lw $t1, 2($t0)
```

4. Which is the difference between these two instructions **lui** and **lw** (and also with **lh** and **lb**)?

5. Assume the following declaration for the variable N:

```
    .data 0x10000000
N:    .space 4
```

Write the appropriate instructions to assign it the following values:

- **N = 0**
- **N = -1**
- **N = 0x100000**
- **N = 0x100040**
- **N = 200000** (in decimal)

6. How the pseudo-instruction **li $t0,-1** can be translated into machine instructions?

7. Assume the following variable declaration:

```
        .data 0x10000000
X:      .space 4
```

For each of the proposed cases, write the final value contained in memory address **X:**

```
la $t0,X
sw $zero,0($t0)
```

```
la $t0,X
sh $zero,0($t0)
sh $zero,2($t0)
```

```
la $t0,X
sb $zero,0($t0)
sb $zero,1($t0)
sb $zero,2($t0)
sb $zero,3($t0)
```

```
la $t0, X
lui $t1, 0x0001
sw $t1,0($t0)
```

```
la $t0,X
lui $t1,0xFFFF
ori $t1,$t1,0xFFFF
sw $t1,0($t0)
```

```
lui $t0,0x1000
andi $t1,$t1,0x0000
sw $t1,0($t0)
```

```
la $t0,X
lw $t1,0($t0)
sw $t1,0($t0)
```

```
li $t1,50
sw $t1,X
```

```
li $t0,0x50
sw $t0,X
```

```
li $t0,0x10000000
li $t1,0xFFFFFFFF
sw $t1,0($t0)
```