TÉCNICO LISBOA

Homework 1
Response Sheet

Group Nº _______

______________________________	IST ID: _______
______________________________	IST ID: _______

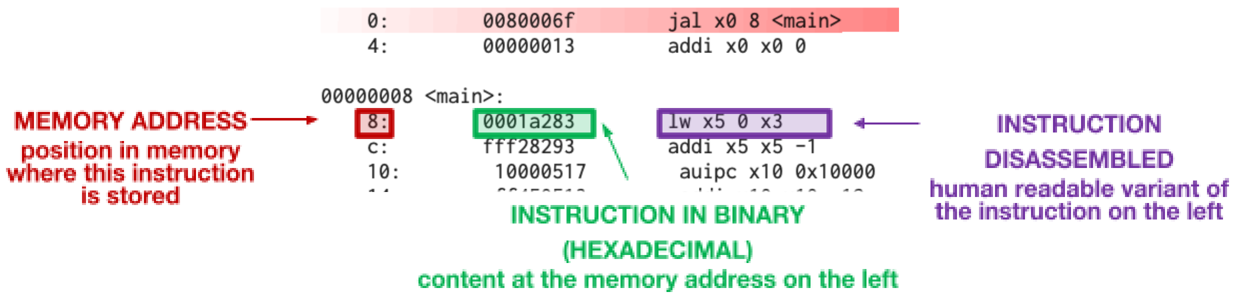
Computer Architecture and Organization

Arquitetura e Organização de Computadores (AOC)

2025/2026

RISC-V ISA, Fundamentals …

1. Configure the `Ripes` simulator to use the single-cycle processor. For this purpose, click on the "*Select Processor*" icon, and select the "*Single-cycle processor*" option.
   Load the `homework_1.s` program in the Ripes simulator and observe the content presented on the left-hand and the right-hand side of the *Editor* tab (i.e., the *program view* with the disassembled code). Although we discussed this several times in the theoretical classes, you can find in the figure below a reminder of how to interpret the information presented on the right-hand side of the Ripes' *Editor* tab.



```
0:        0080006f        jal x0 8 <main>
4:        00000013        addi x0 x0 0

00000008 <main>:
8:        0001a283        lw x5 0 x3
c:        fff28293        addi x5 x5 -1
10:        10000517        auipc x10 0x10000
```

MEMORY ADDRESS
position in memory where this instruction is stored
INSTRUCTION IN BINARY (HEXADECIMAL)
content at the memory address on the left
INSTRUCTION DISASSEMBLED
human readable variant of the instruction on the left

2. Try to relate the instructions in the `homework_1` code on the left-hand side of the *Editor* tab with the disassembled ones on the right-hand side. Typically, an instruction on the left should match the corresponding instruction on the right-hand side (i.e., one-to-one matching is expected). Do you observe any deviations from this pattern? Why?
   (Note: You are not expected to analyze the `ecall` instruction!)

3. For the `li a7, 11` instruction, indicate below the (instruction) memory address and content (both in hexadecimal), as well as the disassembled instruction representation.
(Note: Fill as many lines as you think it is necessary!)

Memory Address (in hexadecimal)	Memory Content (in hexadecimal)	Disassembled instruction

a. Do you observe any differences between the “original” instruction (on the left-hand side) and the disassembled one? Which? Why do they occur? Justify your answers!

b. If we substitute the `li a7, 11` instruction with `lui a7, 11` do you expect to observe any changes to the value of a7? Why or why not? Justify your answer!

c. What is the value of the Program Counter that you see when running this instruction?

4. Consider the RISC-V assembly instructions from the starter code, as presented below:

```
auipc x5, 0
jalr x0, 0(x5) # i.e., jalr x0, x5, 0 in Ripes jargon
```

a. What is the operation performed by the `auipc x5, 0` instruction? What is the result that you expect to see? Why? Explain!

b. What does the `jalr x0, 0(x5)` do? What do you expect to see? Justify your answer!

c. When these two instructions are executed in a sequence (one after another), what is the behavior that you observe? Why? Justify your answer!
Hint: You may like to place a breakpoint at the `auipc` instruction, run the code until that point (>>), and then see what happens when you go instruction by instruction (>).

5. Focus on the `.data` segment of the provided program and fill the table below:

```
.data
   Output: .zero 8
   Keys:   .string "!AHILO"
```

a. What is the size of `Output` and `Keys` arrays in bytes? Justify your answer!

b. How would you perform this initialization/declaration in the C programming language? Provide your code below. Don't forget to indicate the data types!

c. Fill in the table below:

	Memory Address (in hexadecimal)	**Memory Content (in hexadecimal)**	**Memory Content (in ASCII)**
`Output[0]`			*NA*
`Output[3]`			*NA*
`Output[7]`			*NA*
`Keys[0]`			**!**
`Keys[1]`			
`Keys[2]`			
`Keys[3]`			
`Keys[4]`			
`Keys[5]`			

d. How would the <u>content</u> of `Output` and `Keys` arrays change if we allocate an additional array of 32 words between them (`tmp: .word …`)? What do you expect to observe?

6. Develop the code to print the message "**HI!OLA!**" (no space) to the console. For this purpose, you should place the "**HI!OLA!**" message in the `Output` array by strictly using the characters stored in the `Keys` array. This means that you will need to load the characters that you need from the `Keys` array and store them in their corresponding places in the `Output` array (i.e., `Output[0]="H", Output[1]="I", Output[2]="!"`, etc.). You can use registers x5 to x9 for this development.
   <u>Note</u>: The provided system call (`ecall`) is already configured to print to the console the content of the `Output` array. You should **<u>not</u>** delete that part of the code or alter its content anyhow!
   (If you think it is needed, in the space below, you can provide some small notes regarding your rationale, development, and optimization effort, if any)

7. Develop the code to change the letters in the `Output` message from the UPPERCASE to the lowercase, i.e., your final message should look like this "**hi!ola!**". To achieve this functionality, you should add the decimal value of 32 to every ASCII letter in the `Output` array by resembling the functionality expressed in the C code excerpt provided below! You can use registers x5 to x9 for this development and you should <u>not</u> use any pseudo-instructions!
   <u>Note</u>: The provided system call (`ecall`) is already configured to print to the console the content of the `Output` array. You should **<u>not</u>** delete that part of the code or alter its content anyhow!
   (If you think it is needed, in the space below, you can provide some small notes regarding your rationale, development, and optimization effort, if any)

```
int num_chars = 7;
for (int i=0; i < num_chars; i++)
   if (Output[i] != '!')
       Output[i] + = 32;
```