TÉCNICO
LISBOA

Homework 1
Response Sheet

Group Nº 1

Francisco Esteves	**IST ID:** 110299
Sofia Duarte	**IST ID:** 109528

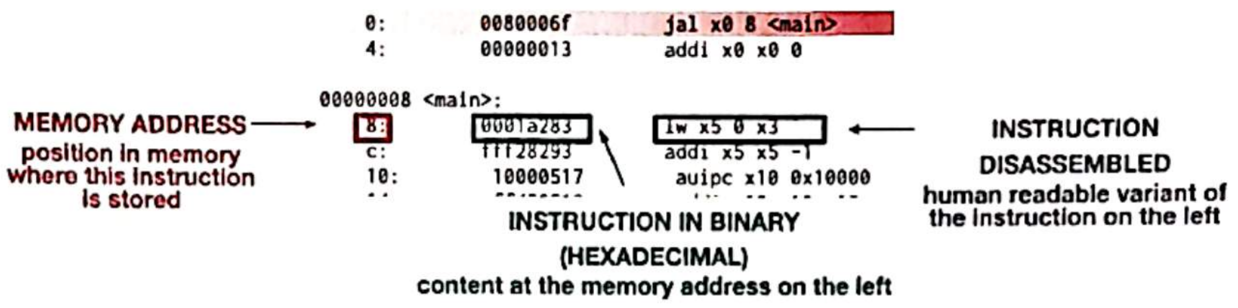
Computer Architecture and Organization

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!)

It is verified that the instructions which use RISC-V pseudo-instructions li disassembled into the RISC-V instructions addi. This happens because whenever li is used, the immediate value fits in a 12-bit immediate.
Regarding the "la, a0, Output" pseudo-instruction, since Output is the value of the address of the variable ~~[illegible]~~ with that name, which does not fit in a 12-bit immediate, to load the address into a0 (x10), two instructions are performed (as seen in the disassembled: "auipc x10, 0x10000" and
"addi x10, x10, -12 # first time la # is used
"addi x10, x10, -40 # second time la # is used"
Auipc loads the 0x10000 immediate, shifts it left by 12-bits, and adds it to the program Counter (PC) value, storing the result in x10. Addi takes the -12 (-40, when the instruction is executed for the second time) immediate, which fits a 12 bit immediate and stores it in x10, effectively canceling the PC value added with the auipc instruction, thus storing the address of the output variable in a0.
In addition to that, where on the original code, RISC-V calling conventions were used for the registers a0 ~~[illegible]~~ and a7, the disassembled ~~using the~~ code ~~[illegible]~~ uses the corresponding registers x10 and x17 respectively.

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
0x0000 0004	0x00b0 0893	addi x17, x0, 11
0x0000 0020	0x00b0 0893	addi x17, x0, 11

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!

The original instruction, "li a7, 11" corresponds to the disassembled "addi x17, x0, x11" instruction. The change from li to addi occurs because li is a RISC-V pseudo-instruction. The li is converted to a addi, without using a lui instruction before because 11 fits in a 12-bit immediate.
~~The~~ Instead of using a7, x17 is used, because a7 is just a symbolic name from RISC-V calling conventions which maps to x17.

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!

yes, it is observed a choge in the value of a7, because lui adds the value "11" directly to the upper portion of the register, appending 12 zeros to the low end of the register. Thus the value of a7 would be 0x0000 b000.

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

The first time "LI a7,11" instruction is executed, the program counter (PC), which holds the memory address of the instruction being executed, holds the value 0x0000 0004. The second time, the PC holds the value 0x0000 0020.

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!

The expected result is 0x0000 0038, as the auipc instruction adds to the value of the program counter, when running the instruction (which is 0x0000 0038), the immediate value 0, which is placed in the upper portion of the register and shifted by 12 bits, to the left (x5 ← PC + (0<<12)).

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

It is expected that the program to execute from the instruction auipc x5, 0 (which corresponds for the value 0x0000 0038 for the program counter, PC.
However x0 is hardwired to 0. So the instruction jalr, which should have stored in x0, the "link" (the address of the next instruction (PC+4)), could not store the "link".

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 (>).

It is verified that set of instructions "auipc x5, 0
jalr x0, 0(x5)" run in loop.
Because "auipc x5, 0" places the value of the program counter, PC, of the instruction itself in the register x5, then "jalr x0, 0(x5)" jumps back to the instruction corresponding to that PC, thus creating a loop

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!

The size of the Output array is 8 bytes, because .zero is a directive indicating N bytes initialized to 0.
The size of Keys array is 7 bytes, because .string allocates a null terminated string and Keys has 6 characters, each one ~~[illegible]~~ occupying 1 byte (each number from 0 to 127 corresponds to a character in the ASCII table).

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!

There are several ways to perform this initialization/declaration in C programming language. The snippet of the code would be:
~~[illegible]~~ char Output[8] = {0};
char Keys[] = "!AHILO"

c. Fill in the table below:

	Memory Address (in hexadecimal)	Memory Content (in hexadecimal)	Memory Content (in ASCII)
Output[0]	0x1000 0000	0x00	NA
Output[3]	0x1000 0003	0x00	NA
Output[7]	0x1000 0007	0x00	NA
Keys[0]	0x1000 0008	0x21	!
Keys[1]	0x1000 0009	0x41	A
Keys[2]	0x1000 000A	0x48	H
Keys[3]	0x1000 000B	0x49	I
Keys[4]	0x1000 000C	0x4C	L
Keys[5]	0x1000 000D	0x4f	O

d. How would the content 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?

The content itself won't change, only the address ~~of the [illegible]~~ where the array Keys starts would change.
In RISC-V, the memory allocated for the declared and initialized variables, follows the order by which they are declared and initialized
Thus if we define an additional array temp:.word, with a size of 32 words, between Output and Keys, the starting address of the Keys array would change to 0x1000 0088.

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.
   Note: The provided system call (`ecall`) is already configured to print to the console the content of the `Output` array. You should **not** 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)
   The logic behind the implemented code is explcined in the comments we present on the code.

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 not use any pseudo-instructions!
   Note: The provided system call (`ecall`) is already configured to print to the console the content of the `Output` array. You should **not** 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)
   As in question 6, the logic behind the implemented code is explaind in the comments we present on the code.

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