TÉCNICO LISBOA

Lab 4: Practice

Computer Architecture and Organization

Arquitetura e Organização de Computadores (AOC)

2025/2026

Overview

This work is intended to help you consolidate your knowledge and develop a better understanding of I/O using the Ripes[1] simulator. You will experiment with the I/O devices provided by the simulator, as well as use system calls to interact with the console.

Deadlines and Submission

NO SUBMISSIONS! This assignment can be seen as a tutorial and an exercise for you to practice! For this purpose, we also include some questions for you to think about and small tasks to resolve.

You can resolve it at our next lab session or at home. Surely, you can discuss your solutions and doubts with us at our next lab session. There will be NO EVALUATION of your solutions.

Consider this work as a preparation for the work to be developed in the class (which will be evaluated).

Environment Setup

For this assignment, we will use Ripes - a visual computer architecture simulator and assembly code editor built for the RISC-V instruction set architecture. The Ripes simulator has three virtual I/O devices, the LED matrix, the switches and the D-Pad. To enable them, go to the I/O tab and double-click on each of them in the Devices list, on the left, as seen in Figure 1. In fact, you can keep clicking to get more than one device of each, but for this tutorial, only use a single device of each type.

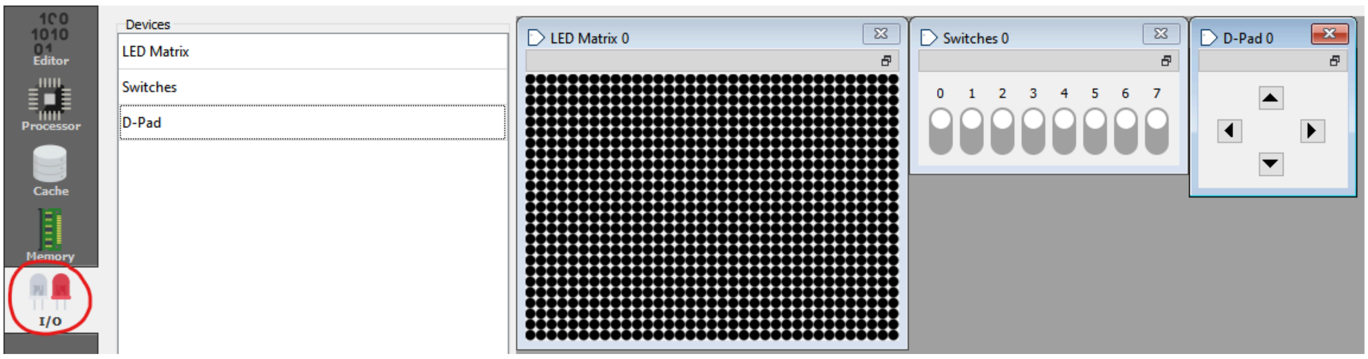
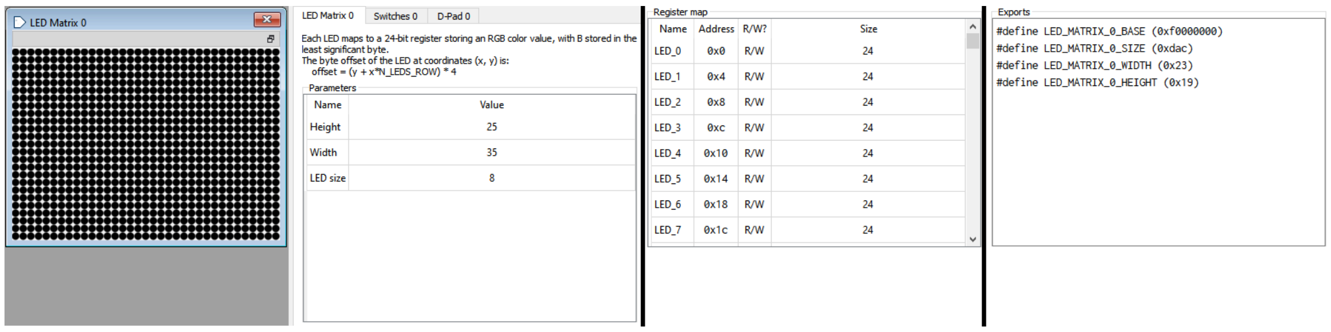


Figure 1 – The I/O tab and the three devices

If you click on the window corresponding to a device, its parameters and information will appear on the right menu. Figure 2 shows this menu for the LED matrix (split up in sections). The first menu shows the parameters, all three of which can be edited by double-clicking the parameter. The Register map menu describes how each element of the I/O device, in this case each pixel, is mapped to the memory. The address of each element is the offset from the LED matrix's base address, which is described in the Exports menu. The constants defined here are automatically exported to the code editor, and you can use them in your assembly to make the code easier to understand. When the code is assembled into a binary, these are automatically replaced with the corresponding values or addresses.

[1] https://github.com/mortbopet/Ripes



LED Matrix 0

LED Matrix 0 | Switches 0 | D-Pad 0

Each LED maps to a 24-bit register storing an RGB color value, with B stored in the least significant byte.
The byte offset of the LED at coordinates (x, y) is:
offset = (y + x*N_LEDS_ROW) * 4

Parameters

Name	Value
Height	25
Width	35
LED size	8

Register map

Name	Address	R/W?	Size
LED_0	0x0	R/W	24
LED_1	0x4	R/W	24
LED_2	0x8	R/W	24
LED_3	0xc	R/W	24
LED_4	0x10	R/W	24
LED_5	0x14	R/W	24
LED_6	0x18	R/W	24
LED_7	0x1c	R/W	24

Exports

```
#define LED_MATRIX_0_BASE (0xf0000000)
#define LED_MATRIX_0_SIZE (0xdac)
#define LED_MATRIX_0_WIDTH (0x23)
#define LED_MATRIX_0_HEIGHT (0x19)
```

Figure 2 – The parameters, memory mapping and exports of the LED matrix

Problem 1: Drawing an image

This first problem will be your introduction to I/O in Ripes, in particular, to the LED matrix. Open the `draw_image.s` file in Ripes, and make sure you enable the LED matrix in the I/O tab of the simulator. The code has several constants already defined with the `.equ` directive. This directive works the same way as a `#define` in C, i.e., you can use the constants in your code and they will be replaced with the respective value when the code is assembled.

Your job is to draw an image on the LED matrix. An image is encoded in data section, where each byte corresponds to either a color of a pixel, or the end of a line, as shown in Figure 3. As indicated by the constants in the code (`IMG_WHITE`, `IMG_BLACK` etc.), a byte of `0x00` encodes a white pixel, `0x01` a black pixel, and `0xff` the end of a line (which is not drawn). Your goal is to decode this format, drawing the 30 (width) x 20 (height) image to the LED matrix.

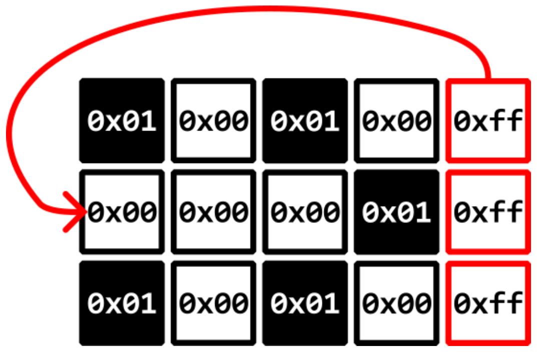


Figure 3 – Image encoding

Part of the loop structure is already in the code, and you should complete the rest. Pointers to the base of the LED matrix and to the start of the image data are already initialized, as well as the logic to load image data and loop until it ends. The code contains comments which should help guide you in what you need to do in each situation. Depending on the image byte read, you should either reposition the pointer to the LED matrix to the start of the next line or draw the appropriate pixel to the LED matrix and increment the pointer. Look to the comments for a more detailed guide.

1. Complete the code and make sure the image is displayed correctly on any LED matrix larger than the image. It should be a perfectly rectangular 30x20 image, with a white background. What does it say?

Problem 2: Snake

Next up is the game Snake, where you control a snake which, oddly enough, eats fruit to grow. It can move in 4 directions and wrap around the screen, dying if it crashes into itself. The file `snake.s` contains an incomplete implementation of the game for the Ripes simulator, which is explained below. Make sure to follow along in the code and relate it to the explanation provided.

Several constants are defined at the beginning of the file (see `.equ` statements). The `GRID_*` constants represent how the snake's possible directions (**U**`p`, **R**`ight`, **D**`own`, **L**`eft`) or the fruit (`FRUIT`) are encoded on the game grid (you'll learn more about this later). Further down are the `*_FLAGs`, which are used to communicate (or flag) game events happening, such as the snake dying (`DEATH_FLAG`) or eating fruit (`FRUIT_FLAG`). Finally, the timeout (`TIMEOUT_MS`) and frame times (`FRAME_TIME_MS`) are defined, in milliseconds. The timeout (2 seconds) is the time the game pauses after the initialization, before allowing the snake to move. The frame time represents the minimum amount of time that needs to pass between two updates of the game window (frame).

The `.data` section contains important game variables, such as the time when the `last_frame` happened, and the current `timeout`. The direction the snake is moving is also stored here, both in the same encoding as the `GRID_*` flags (`direction`), and in vector form (`direction_x, _y`). Hint: If this vector form sounds weird, you would like to revisit the slides from our theoretical lectures. The game grid is an array of bytes with the same size/shape as the LED matrix, and it stores the snake's segments, their direction, and which cell contains the fruit. Later we will see how this grid is used to move the snake and detect collisions. The x and y positions of the head and tail of the snake are next (`head_x/y, tail_x/y`), followed by the state of the random number generator (RNG), i.e., `rng_state`, which is used to spawn the fruit in random locations.

Reaching the `.text` segment, you see a jump to `main`, which calls two functions. First, the game state is initialized with the `init` subroutine, then the game's `main_loop` is invoked. The `init` subroutine makes calls to the following five (5) functions:

- `init_saved_regs` to initialize four saved registers (s6 through s9), which keep their value throughout the entire execution. Frequently used addresses are stored in these registers (namely: *i)* `s6` for the `game_grid` base address, *ii)* `s7` for the `LED_MATRIX` base, and *iii)* s8 for the `D_PAD` base address), while the `s9` register is used to store the game event flags that were previously mentioned;
- the next two calls will reset the LED matrix (`reset_led_matrix`) and the game grid (`reset_game_grid`). The structure of both subroutines is very similar, iterating over the respective matrix and filling it with the background color / zeros;
- the snake is then initialized (`init_snake`), setting the position of the head and tail (which are the same at the start of the game), and updating the grid and LED matrix accordingly;

- finally, the first fruit is spawned (`spawn_fruit`), making use of a simple RNG algorithm called xorshift32[2]. After generating a valid random position, the fruit is placed there, updating the grid and the LED matrix.

Now that the game is initialized, `main` calls to the `main_loop`. A simplified flowchart of the main loop's logic is shown in Figure 4. We begin by checking if the game is timed out, which will skip the snake movement if so. Otherwise, every frame we move the snake, we check if it died (to reset the game) and wait for the next frame while "polling" the player's D-Pad.

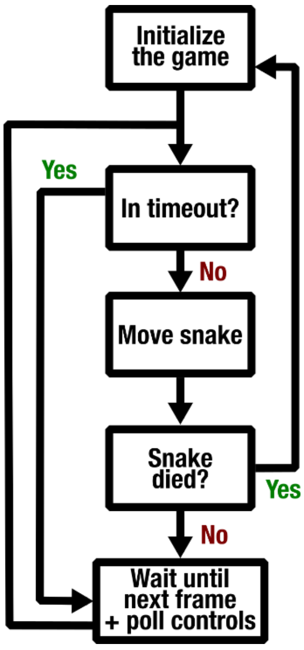


Figure 4 – Simplified flowchart of the game loop

Let's analyze `move_snake` to see how the snake moves, starting with an overview of how it is represented in memory. In short, there is no need to know where every segment of the snake is, only the head and the tail. The rest of the snake is encoded in the game grid, which also tells the tail where to go next. Figure 5 illustrates an example of how the snake's movement works.

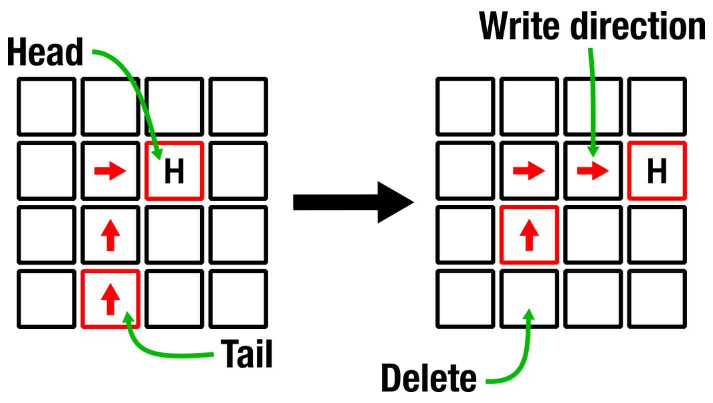


Figure 5 – Example of how the snake moves, and how it is encoded on the grid

[2] https://en.wikipedia.org/wiki/Xorshift#

In this example, the snake has 4 segments. Imagine the player's last input was right, so the snake will move the head right. The head and tail move at the same time every frame, but it's easier to understand if we start with the tail: using the tail's position, we access its cell on the `game_grid`, which contains a direction (`GRID_U`, `GRID_R`, etc., which you've seen in the constants). This direction tells the tail where to go next, in this case, up. As such, we clear the cell, storing a `0` to indicate there is no snake there anymore, and move the tail position up (stored in the `tail_x, tail_y` game variables, in memory). To move the head, we have to write the direction we just moved in in the old head position, so that the tail knows where to go when it reaches that cell. We write `GRID_R` to that cell and move the head (stored in the `head_x, head_y` variables) to the right. No matter how large the snake is, we only need to move the head and the tail each frame! The other segments are all stored in the grid, and do not change.

Growing the snake when it eats some fruit is also very simple: when the head moves to a cell containing a fruit (`GRID_FRUIT`), we skip moving the tail for one frame. If the head moves forward but the tail does not, the snake effectively gains one segment. Detecting a collision is a matter of checking the grid value too. The values that encode the snake's direction (`GRID_U`, etc.) also encode its presence. If the value is not zero, then it means a snake segment is there.

Now that you understand the concepts, look at the `move_snake` code again. We begin by updating the cell the head was in with the direction it will move to (right, in our example). The `get_grid_offset` function is used to translate between an (x, y) position and the offset on the `game_grid`. After setting the direction, we can move the head by using the `get_next_grid_pos` function. This function adds a position (the head's old position) and a direction (the movement direction), while also handling screen wrapping. For example, if the head moves off the right edge of the screen, it will appear from the left edge and keep moving right. Back to `move_snake`, the next step is collision detection. We check if the cell we moved to contains a fruit, setting the fruit flag, or a snake segment, setting the death flag. The head is then drawn in its new position, and we can begin moving the tail.

We first check the `s9` event register to see if it contains the fruit flag, as the tail will not move if a fruit was eaten. Its movement is similar, except the direction comes from the grid. The grid value for the tail's current position is loaded (up, in our example), and we use the `get_vec_from_grid` function to translate between the grid directions (`GRID_U`, etc.) and the corresponding vector. Now that we have the vector, we can delete the old tail on both the grid and the LED matrix. With the vector, we call `get_next_grid_pos` once again, which will tell us the new tail position, handling any eventual screen-wrapping. After we store the new tail position, we are done!

Finally, back in the main loop, the `wait_and_poll` subroutine is responsible for waiting between each frame, reducing the timeout, and poll the D-Pad repeatedly while waiting. By using the `GetTime` syscall, we determine the time passed since the `last_frame`, repeatedly looping until `FRAME_TIME_MS` milliseconds have passed. The `poll_dpad` subroutine handles reading the D-Pad, converting the button presses into the three directions variables in memory.

What you need to do

1. The snake was too hungry and ate the code responsible for reading the right button on the D-Pad. Navigate to the `poll_dpad` subroutine and complete it, storing the appropriate values in the direction addresses. *Hint*: Look at how the other D-Pad buttons are implemented as a guide.

2. As mentioned before, the controls are polled repeatedly while waiting for the next frame. Can you think of an advantage this has over polling them once per frame? If you can't think of anything, temporarily move the call to `poll_dpad` from `wait_and_poll` to the main loop and try playing the game. Do you notice any problems with the controls?

3. Implement a pause button using <u>switch number 2</u>.
   a. Add your code to the start of the game's `main_loop` (before checking the timeout, right after the `1:` label)
   b. Either insert the code in-place or create a new subroutine that handles reading the switches and pausing. Which one will make for more organized code?
   c. Read the switches by loading from the appropriate address. Remember to go into the I/O tab to figure out the name for the address.
   d. Mask the value to get only the value of switch 2.
   e. If the switch is on, loop to read it again until it is turned off, thereby pausing the game.

4. Whenever the snake dies, the `s9` register is set to the `DEATH_FLAG`. Use this to print "`ouch!`" to the console whenever the snake dies.
   a. Declare the string you'll need in the data section, using `<name>: .string "ouch!"`.
   b. In the Ripes simulator, go to *`Help > System calls`*. Identify the system call which can print the string, and determine the inputs it needs.
   c. Modify the `main_loop` to print the string if the snake dies.

5. Notice that, if the snake has more than one segment, if you perform a 180-degree turn (for example, turn left while moving right), the snake will die. This happens because the snake crashes with the segment immediately behind the head. To prevent accidentally dying this way, change the `poll_dpad` subroutine to ignore inputs if they would result in a 180-degree turn.
   a. To achieve this, use `direction_x` and `direction_y` to skip reading buttons on the same axis you are moving. For example, if the x direction is non-zero, you can ignore the left and right buttons since 1) you want to prevent 180-degree turns and 2) you are already moving in the other direction.