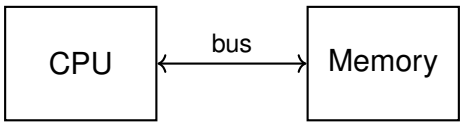
Chapter 6 – Memory-mapped Input/Output

1 Introduction

We learned in chapter 3 that a "von Neumann machine" consists in a "processor" connected to a "memory" by a "bus":

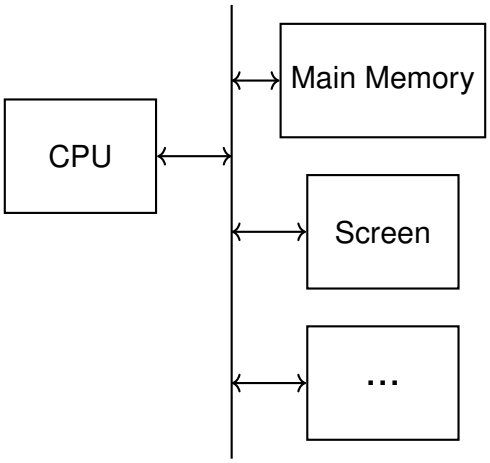


The processor reads program instructions from memory and executes them one by one. In chapter 5, we saw that **main memory** is also used to store all data structures used by the program.

Today, we are going to look at how a program can communicate with other components of the machine. Those are usually called **peripheral devices**, or just peripherals. Because some of these devices are physically connected to the outside world (e.g. keyboard, screen, network) they are sometimes referred to as as **Input/Output** devices.

Idea From the CPU's point of view, all peripheral devices behave as "memory": they can be read from (and written to) using ordinary load/store instructions. Each device is given a region of the **Memory Address Space**, and is responsible for responding to (read or write) requests in that region. This method is known as **"Memory-mapped Input/Output"** and is implemented in all modern systems.

As an example, the architecture of the SCAT machine is illustrated below. All components are connected to a shared bus.

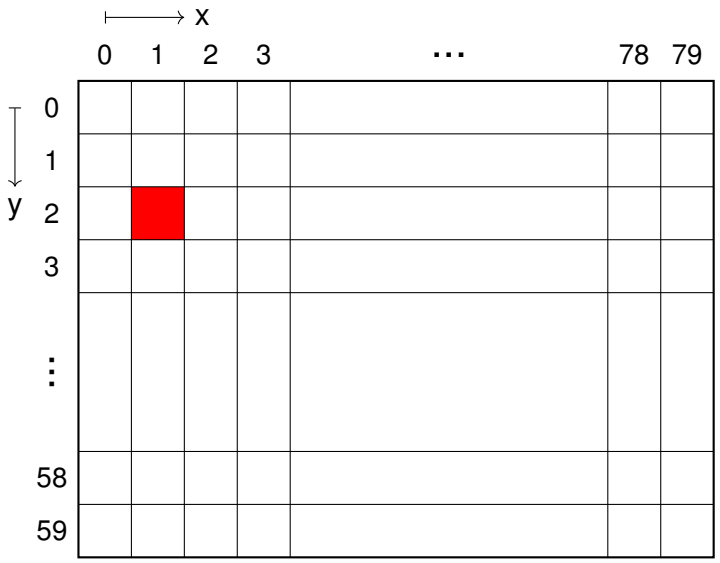


Our memory map includes the following regions:

- addresses from `0` to `0xAFFFFFFF` belong to main memory (256 Mbytes)
- addresses from `0xB0000000` to `0xB0004AFF` correspond to the screen controller (more details on the next page).

2 Graphics in SCAT

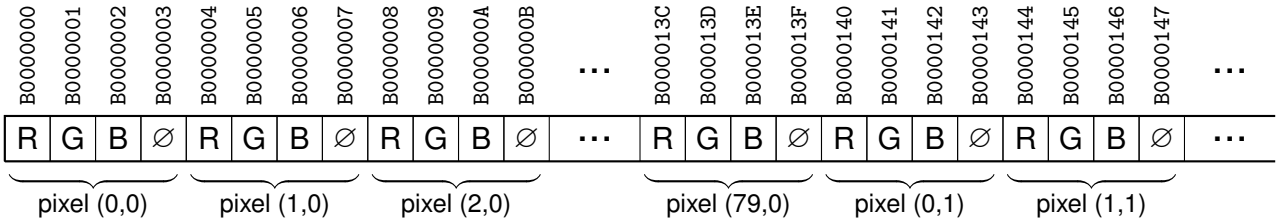
Type `screen` in the simulator to open the graphical display of our simulated machine. By default, the screen is filled with black. The color of each pixel can be controlled individually from the CPU. In the example below, the pixel with coordinates (1, 2) is set to red. The screen resolution is 80x60 pixels.



Video Memory (aka VRAM, aka the framebuffer) is a region of the Memory Address Space that corresponds to the video display. The contents of the framebuffer is used by the hardware to continually refresh the image displayed on the screen. The SCAT framebuffer goes from 0xB0000000 to 0xB0004AFF. As illustrated below, each pixel is **memory-mapped** to a 32-bits memory word, with a binary format that follows the RGB color model:

- the first byte encodes the Red component,
- the second byte encodes the Green component,
- the third byte encodes the Blue component,
- the fourth byte is ignored.

In video memory, pixels are mapped next to each other in row-major order:



Exercise 1 Write a program that paints the top left pixel in white.

Exercise 2 Write a program that paints the four corner pixels in yellow.

Exercise 3 Write a program that paints pixel (x,y) for any value of x and y. Your code will start by initializing two registers, as illustrated below:

```
leti r1, 40 ; x position
leti r2, 30 ; y position
...
```

Putting it all together: graphical programming

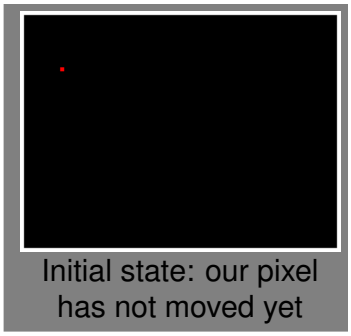
You can now choose between two “big” exercises: visualizing your bubble sort, or implementing a “bouncing ball” animation. Please read both sections below before making your choice.

Bouncing Ball

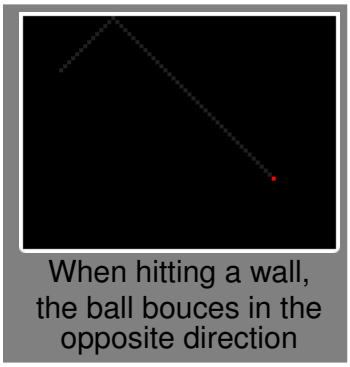
Exercise 4 Your goal is to implement a simple animation similar to the famous “DVD logo” animation, but where the moving ball is a single pixel. In other words: A moving “ball” (red pixel in the pictures below) moves diagonally until it hits the edge of the screen (drawn in white in our example). Each time it reaches such a wall, the ball “bounces” in the opposite direction and keeps on moving.

To make your work easier, you may want to go after simpler subgoals first, for instance:

- draw walls on all four borders of the screen (optional, but easy)
- implement an infinite loop to “advance” the ball by the current dx/dy (each -1 or +1)
- draw the ball in its new position and erase its previous position (in our example, we use dark gray instead of black; as a result, the ball leaves a “trail” along its way)
- detect when the ball hits a wall and change dx/dy accordingly



Initial state: our pixel has not moved yet



When hitting a wall, the ball bouces in the opposite direction

After a while, the trail looks quite nice

Bubble Sort Visualization

Exercise 5 The goal is to augment your bubble sort implementation from last week to visually show the effect of each sorting pass on the array.[1]

This idea is illustrated below, in a very simplistic implementation (25 lines of assembly). To simplify address calculations, we display each number as a horizontal line. In other words, if T[i]=j then we draw j white pixels on line i. The first picture below shows the initial state of the array (notice how the length of each white line represents one number) i.e. `13, 18, 5, 3, 10, 8, 20, 1, 14, 6`.

Initial state

Same picture, zoomed on the upper-left corner

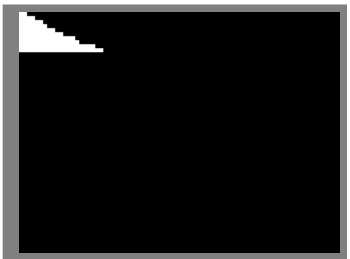
[1] If this description is not clear, ask us for help and/or search for “sorting algorithm visualization” on the web.

Then we run the same code after each sorting pass, and we slowly see the values moving into place:


After first pass


After second pass

...


Fully sorted

To make your work easier, you may want to go after simpler subgoals first, for example:

- clear the entire screen (you'll need that to erase your previous drawing)
- draw one line to represent one number
- loop on all values in the array
- note: you'll want to pause execution between two passes just to leave the user enough time to see what's happening. For that, the simplest strategy is to use a `breakpoint` in the simulator.

Obviously you can choose a different visualization, for instance with vertical lines, or different colors, or with rectangles, etc. For instance, a nice feature (though hard to implement) is to highlight each pair of numbers being swapped.