Exercise 8:

a) Graphic Buffer Controller

The task of the *graphic_buffer_controller* module is to reload the two graphic buffers with data from the RAM and to select the currently active buffer. It also generates the read address for the RAM controller.

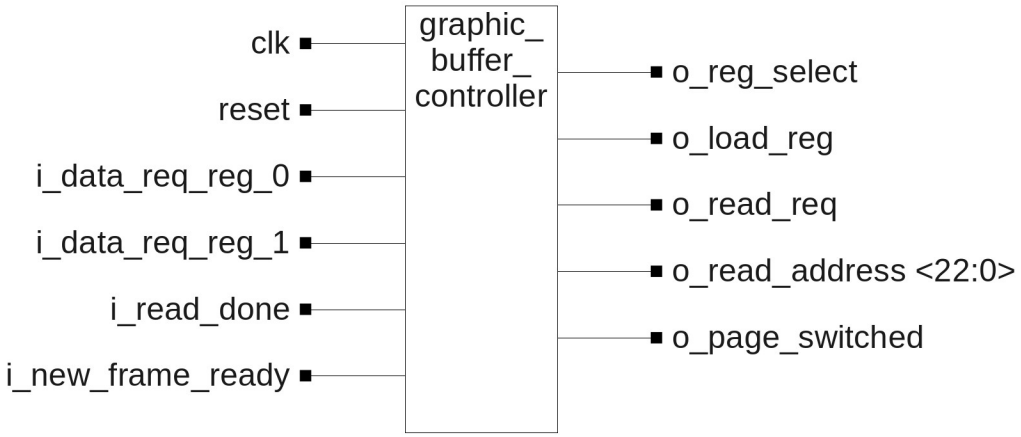

Figure 1: graphic_buffer_controller

Port declaration:

- *clk* : in std_logic
- *reset* : in std_logic
- *i_data_req_reg_0* : in std_logic
- *i_data_req_reg_1* : in std_logic
- *i_read_done* : in std_logic
- *i_new_frame_ready* : in std_logic
- *o_reg_select* : out std_logic
- *o_load_reg* : out std_logic
- *o_read_req* : out std_logic
- *o_read_address* : out std_logic_vector (22 downto 0)
- *o_page_switched* : out std_logic

Generic declaration:

- *G_H_RESOLUTION* : integer
- *G_V_RESOLUTION* : integer

On an incoming request for data through the inputs *i_data_req_reg_0* or *i_data_req_reg_1*, the controller immediately switches the active buffer to the buffer not requesting data on the next rising clock edge. The value of the output *o_reg_select* signifies the selected buffer (0 = buffer 0, 1 = buffer 1). If both buffers send a request simultaneously, one of them must have a higher priority. At the same time as changing the buffer, a read request is sent to the RAM controller by setting the output *o_read_req*. After the RAM controller completes the read operation (indicated by the input *i_read_done*), the unselected buffer is loaded with new data by setting the output *o_load_reg*.

After the next rising edge, the outputs *o_load_reg* and *o_read_req* must be disabled and the RAM address can be incremented. Now the controller is ready to process the next data request from one of the buffers.
The data is stored in the RAM in words of 16 bits (2 pixels). Since we are using a dynamic RAM, data has to be read in bigger bursts to mask the initial latency of opening an internal row. For this application, a burst size of 16 is chosen (32 pixels, 256 bits total). After each read operation, the address must be incremented by 16 to point to the next burst start point. After reading an entire frame (how many reads?), the address counter must be reset to the start address of the next frame.

To avoid forcing the renderer to change the image data of the currently displayed frame, a method called 'page flipping' is used. That means that the RAM is separated in two regions called pages. The start address of the first page is 0, while the second page starts at 262144 (0x40000).
While the first page is used to store the new data from the renderer, the frame that is currently displayed is read from the second page, and vice versa. When the renderer has finished a new frame (indicated by the signal *i_new_frame_ready*) the graphic output displays the rest of the current frame and then switches the page and displays the next frame. If the *i_new_frame_ready* signal is not set at the end of the frame, the old frame will be displayed again. On a reset, the first page (starting at address 0) should be selected.
The renderer itself waits until a successful page switch is signified through the output *o_page_switched* of the graphic buffer controller. The *o_page_switched* signal can be set back to 0 with the next RAM address increment after a page switch.

To implement the Graphic Buffer Controller, you must first design a state machine. You can use the template of the state chart given on the next page.

Hints: If you want to increment the RAM-address, you can simply set a flag in your state-machine and increment the address in another process depending on that flag. To calculate the RAM-address, you should first examine the binary representations of the start addresses! Which bits do you have to actually change to increment the address or flip the page?

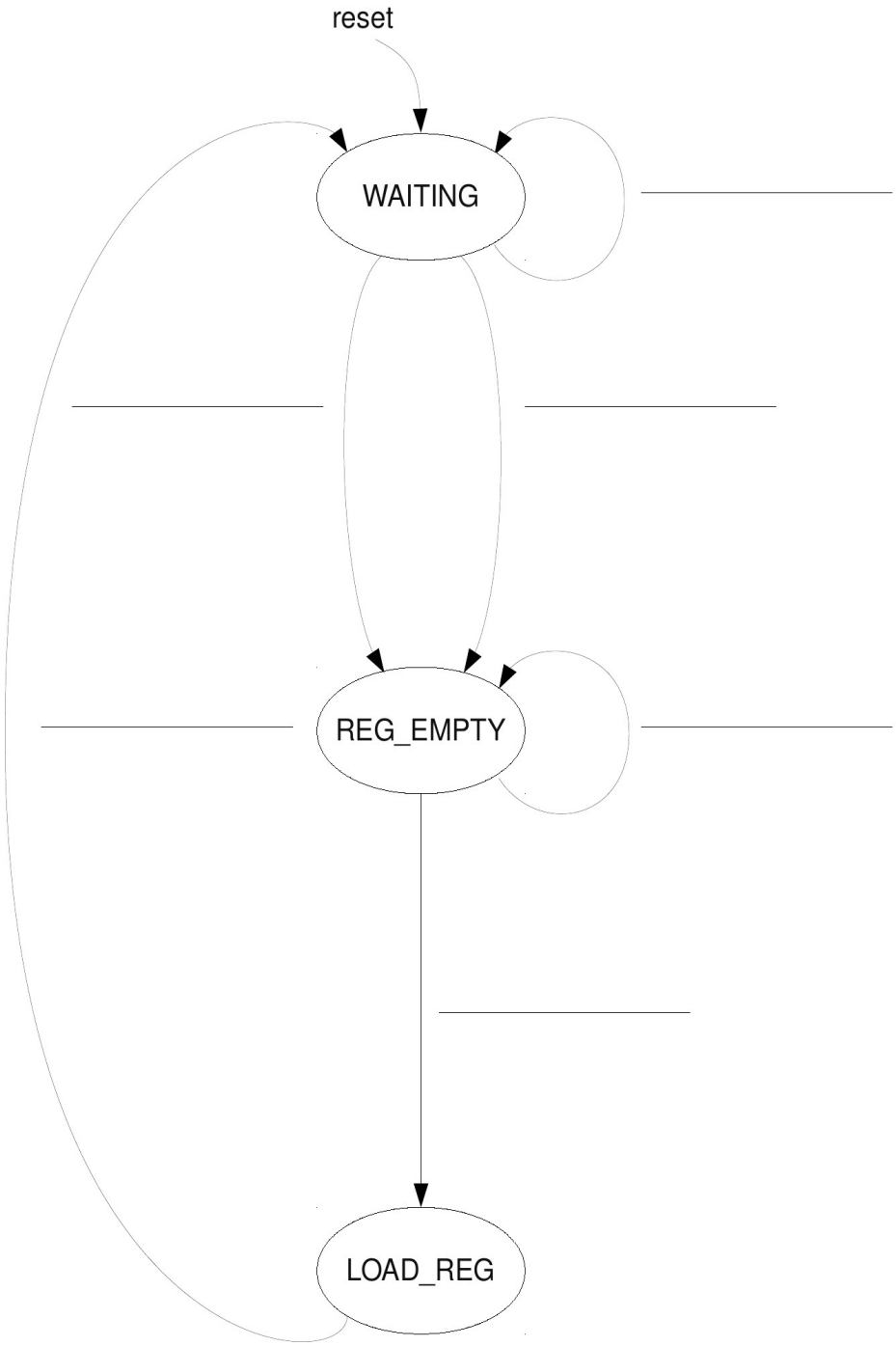

Figure 2: State chart of the Mealy machine for the graphic_buffer_controller