Technische Universität Braunschweig

INSTITUT FÜR DATENTECHNIK UND KOMMUNIKATIONSNETZE

# Computer-aided Design Of Digital Circuits

# 1. Organizational Behavior

The goal of this lab is to develop and implement the well known game Pong on a FPGA evaluation board in a team of two students. To finish the lab successfully, the game must run properly with each of the modules you developed. Furthermore, the running game must be presented and you will be required to answer various questions about it. After that, you receive a certificate of completion and 5 or 6 credit points, depending on your field of study, without a grade.

To ensure the successful implementation of the game, the project is divided into several exercises. You will work on these exercises at the designated dates, which makes your participance necessary at every lab meeting. Usually, a lab session lasts for 3 hours. If you are unable to attend the lab, it is obligatory to inform Sean (whitty@ida.ing.tu-bs.de ) and Carsten (carsten.siemers@gmx.de) with a short email. In this case, or in the case you could not complete the exercise within one session, you are advised to work on your tasks on your own time until the next lab session. Other than one other lab, which also uses the same room, you have access to the laboratory Monday through Friday, from 08:00 until 18:00. Please note that the doors to the corridors close automatically at 18:00 and cannot be opened without a key.

# 2. Project Overview

## *2.1. Goal*

The game Pong must be implemented on a FPGA evaluation board, which provides all the necessary hardware for this project. This includes a PS/2 port for the keyboard used to control the game, a VGA port to connect a monitor, 1 MB of SRAM to store data and the FPGA itself (Xilinx Spartan 3 FPGA).
Because it would be too complex and time consuming to build the complete game from scratch, only selected parts are designed in this lab. The rest is provided to you. At the end of the lab, the pre-existing components and the newly designed ones must be combined to create a fully functional project.

## *2.2. Technical realization*

The game is controlled by an external keyboard, connected to the PS/2 port. The on-board buttons are only used for reseting the game. A monitor with an analog VGA input is used as the display device. All image data is buffered in the on-board SRAM before being displayed on the monitor.
All external hardware is directly connected to and controlled only by the FPGA.

## 2.3. Introduction to FPGAs

A Field Programmable Gate Array (FPGA) is an integrated circuit which is, contrary to Application Specific Integrated Circuits (ASICs), configurable after manufacturing. It mainly consists of programmable logic blocks (each manufacturer has a different name for them: Xilinx calls them “slices”), which are configured with a hardware description language like VHDL or Verilog. With a large number of these blocks, you can realize very complex digital systems comparable to ASICs.

The structure of a FPGA is shown in the figure below.

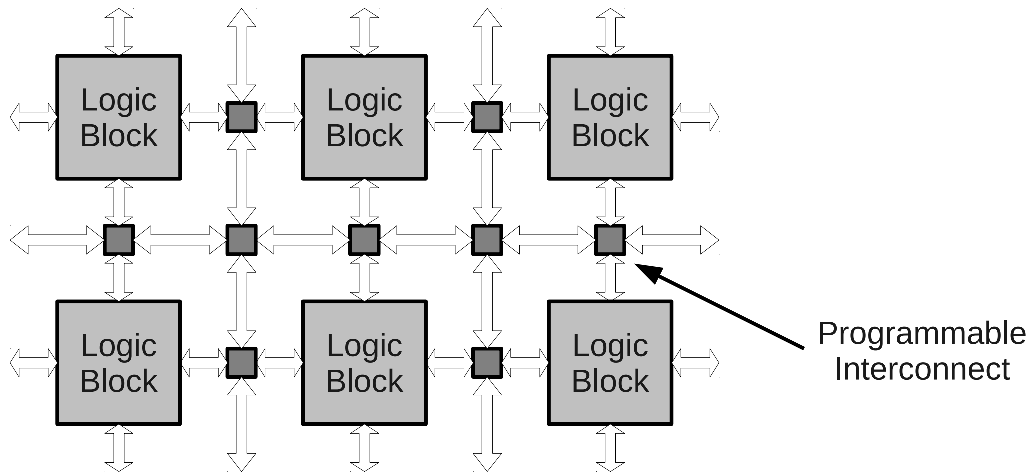


*Figure 1: FPGA structure*

The Programmable Logic Blocks are connected with programmable interconnects. The Logic Blocks and the interconnects are both usually based on SRAM. A simplified structure of a Programmable Logic Block is shown in the next figure.

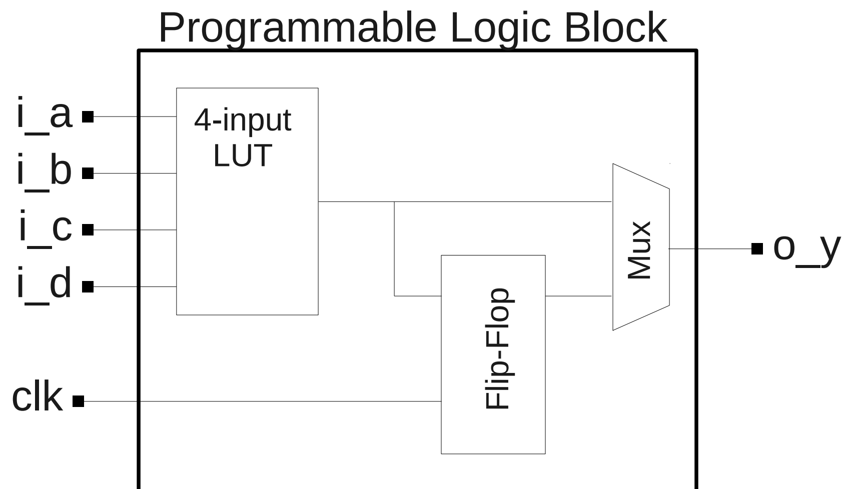


*Figure 2: Programmable Logic Block*

The logic function is realized with a 4-input look-up table (LUT). The output of the LUT is connected to a D-Flip-Flop and a Multiplexer. The behavior is selected with the Multiplexer, and the Block can act as simple logic or as a Flip-Flop.

The functionality of the LUT is shown in figure 3.

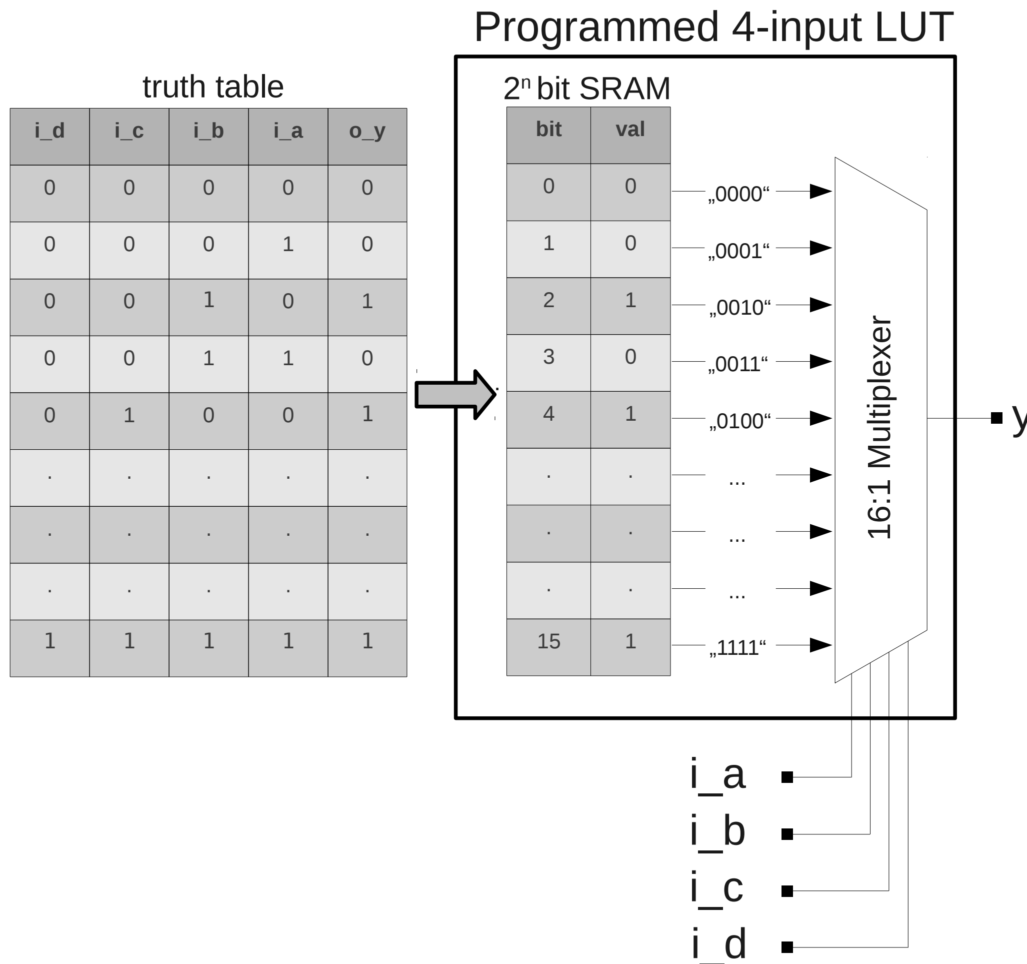


*Figure 3: Lookup Table*

For 4 inputs, we have $2^n = 16$ logical combinations. The output values of these 16 combinations are loaded into a $2^n = 16$ bit SRAM. These values feed a Multiplexer, which is controlled by the 4 inputs. Now, the LUT has the same function as an equivalent logic gate.

Modern high-end FPGAs offer several hundred thousands Logic Blocks with 4 to 6 inputs. Furthermore, many specialized blocks like DSP blocks, interfaces, Block RAMs or even complete processors are commonly integrated. For a more detailed view about FPGAs, you can find many information on the internet or in the literature.

**Web:**

www.xilinx.com
www.altera.com
www.actel.com

**Literature:**


Clive Maxfield, The Design Warrior's Guide To FPGAs, Elsevier 2004

### 2.4. Game rules

The goal of the game is to prevent the ball from leaving the playing field on your defensive side. The field is limited by a border on the top and bottom from which the ball bounces back. On the left and right side, there is no such border, and the player must use a moveable paddle to keep the ball from leaving the field. If the ball hits this paddle, it will be reflected just as if it had hit a border (angle of incidence = angle of reflection). The corners of the paddles have a slope, whose angle can be adjusted by altering the content of a ROM (see chapter about *game_engine*). If the ball hits these corners, it will be reflected differently than when hitting the paddle in the center region. If a player fails in keeping the ball on the field, the other player scores a point. The first player that reaches a predefined number of points (standard value is 15) wins the game. The right to serve changes after a given number of serves (standard value is 5).

# 3. Structure of the project

In order to obtain a better structure, the project is divided into several sub-modules by their function, which are then connected in the top-level entity *pong_top*.

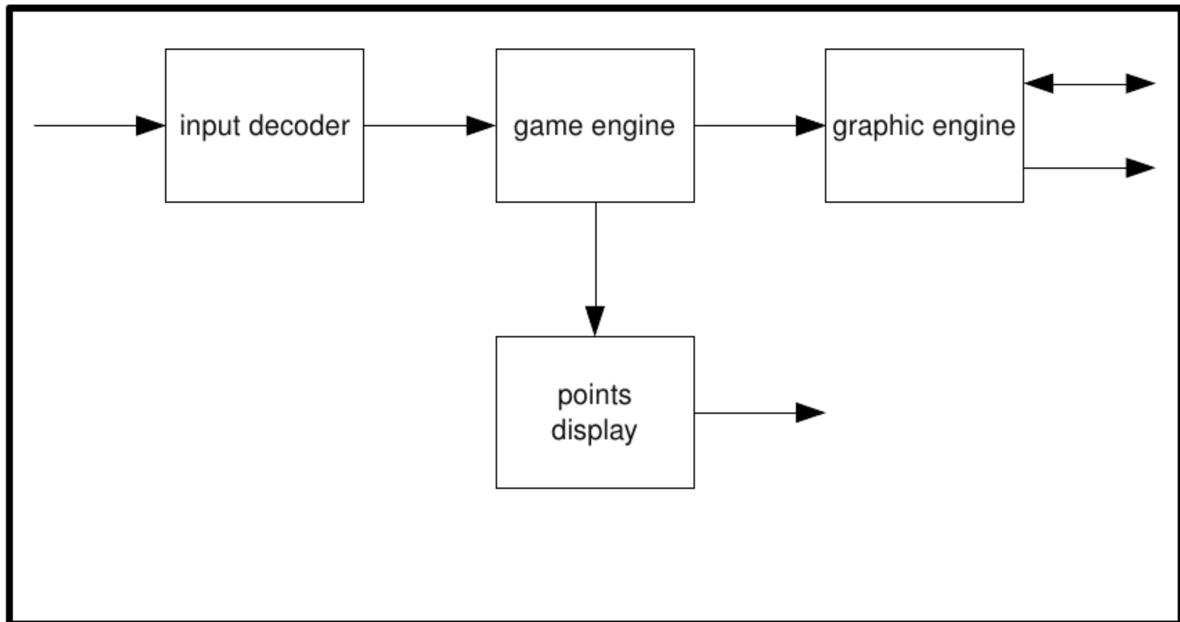


*Figure 4: The entity pong_top*

List of components of the entity *pong_top*:

- *reset circuit*
- *input_decoder*
- *game_engine*
- *points_display*
- *graphic_engine*

## *3.1.* Entity *reset_circuit*

The reset circuit resets the entire circuit after power-up and ensures the correct function of the reset button on the FPGA board.
(Note: the reset circuit is not visible in the figure of the entity *pong_top*)

## *3.2.* Entity *input_decoder*

This module scans the serial data stream from the PS/2 port and converts the received keyboard commands into the control signals necessary to control the game.
It provides basic communication from the keyboard to the host device (in this case the FPGA board). Communication from the host to the keyboard is not implemented (and not necessary for this application).

## *3.3.* Entity *game_engine*

While all other components are only needed to communicate with external hardware, the entity *game_engine* is responsible for the pong game itself.

The components *paddle_movement* and *ball_movement* control the movement of the ball and the paddles and store their positions.
The *event_trigger* is basically a clock divider. It ensures that the ball and paddle movement occur not at every clock cycle. (Note: it does not generate a clock signal for the movement blocks, but a kind of enable signal).
The *random_number_generator* delivers random movement data for the ball during serve, while the *rom* is used to store a physically correct movement model for ball hits at the corners of the paddles.
A state machine in the entity *game_states* stores the current state of the game, including the points for each player and the current serving status.

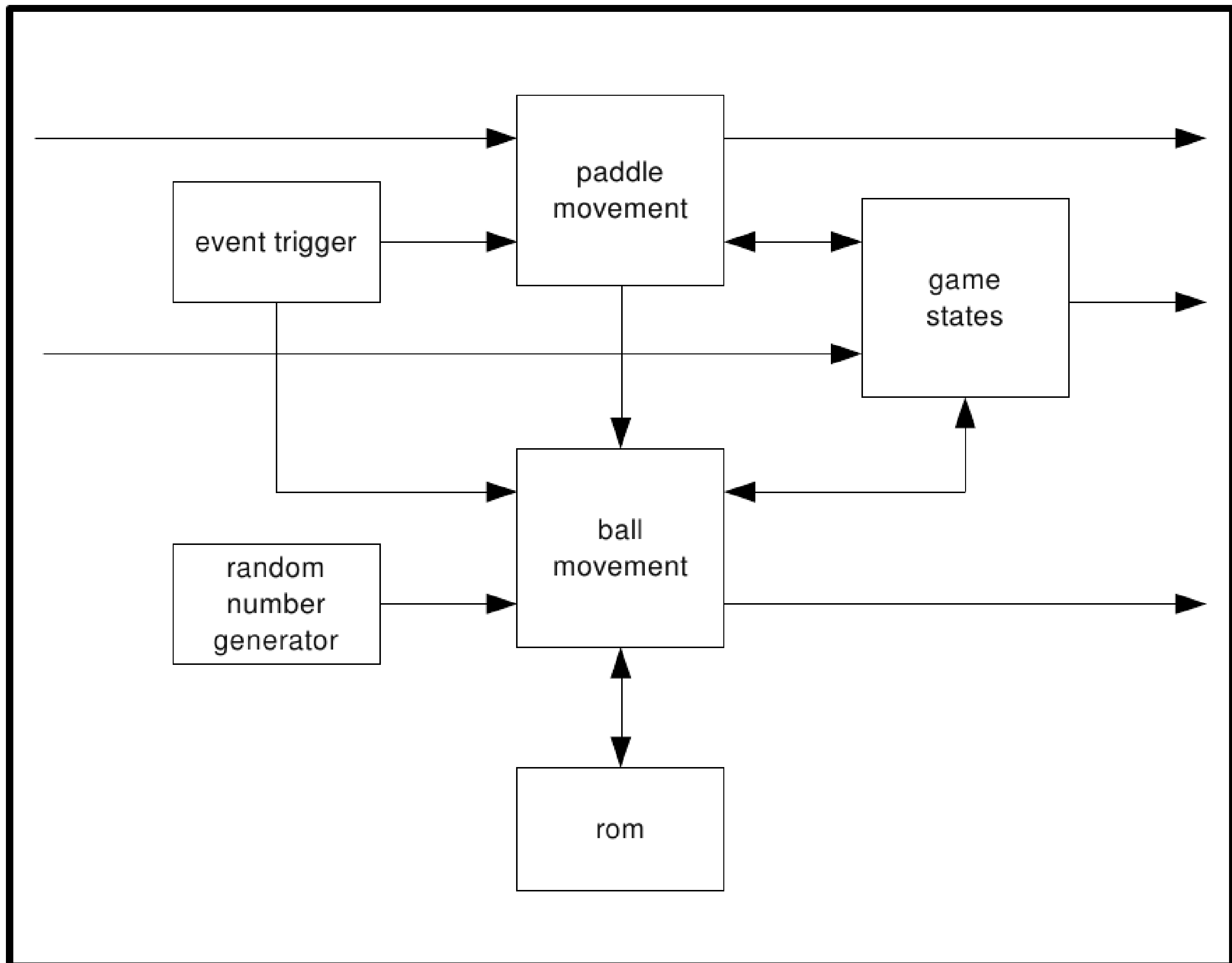


*Figure 5: The entity game_engine*

## 3.4. Entity *points_display*

This entity converts the binary value of both players' points into BCDs (binary coded digits) and displays the points on the on-board seven-segment LED displays.

## 3.5. Entity *graphic_engine*

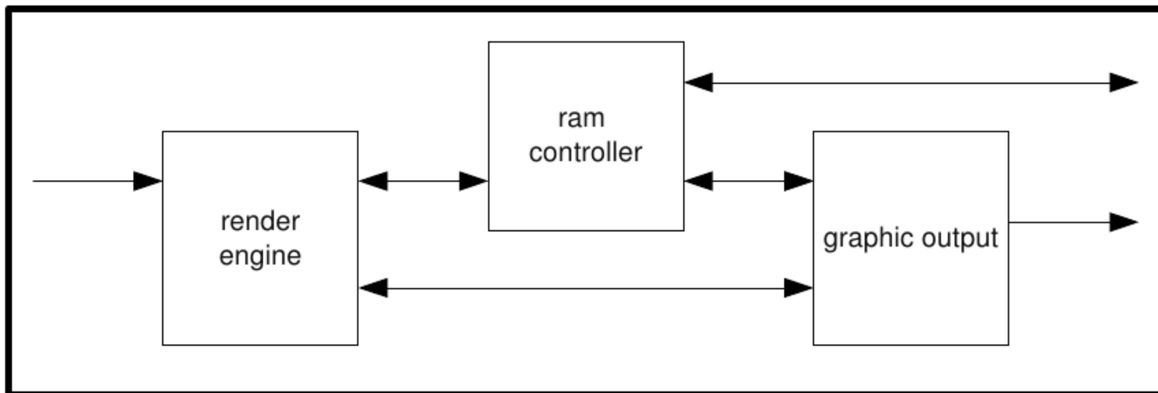


*Figure 6: The entity graphic_engine*

The entity *graphic_engine* connects all components responsible for the processing of the image data. The structure of the graphic engine is divided into two parts: one for the generation of the images (*render_engine*), and one for driving the display and displaying a previously created images on it (*graphic_output*). To ensure that generation and displaying of the images are totally independent of each other, the rendered image is stored in one part of the on-board SRAM before being displayed, while the graphic output displays the previously rendered image that was stored at different part of the RAM.

### 3.5.1. Entity *pixel_clk_generator*

This module generates the pixel clock needed by all components of the entity *graphic_output*. The clock cycle length is equal to the time it takes to display one pixel on the screen. In the case of a 640x480 pixels resolution at 60 Hz refresh rate, the cycle length is 40 ns (25 MHz) and the pixel clock can be derived from the system clock (which runs at 50 MHz) by a division by two.
(Note: the pixel clock generator is not visible in the figure of the entity *graphic_engine*)

### 3.5.2. Entity *render_engine*

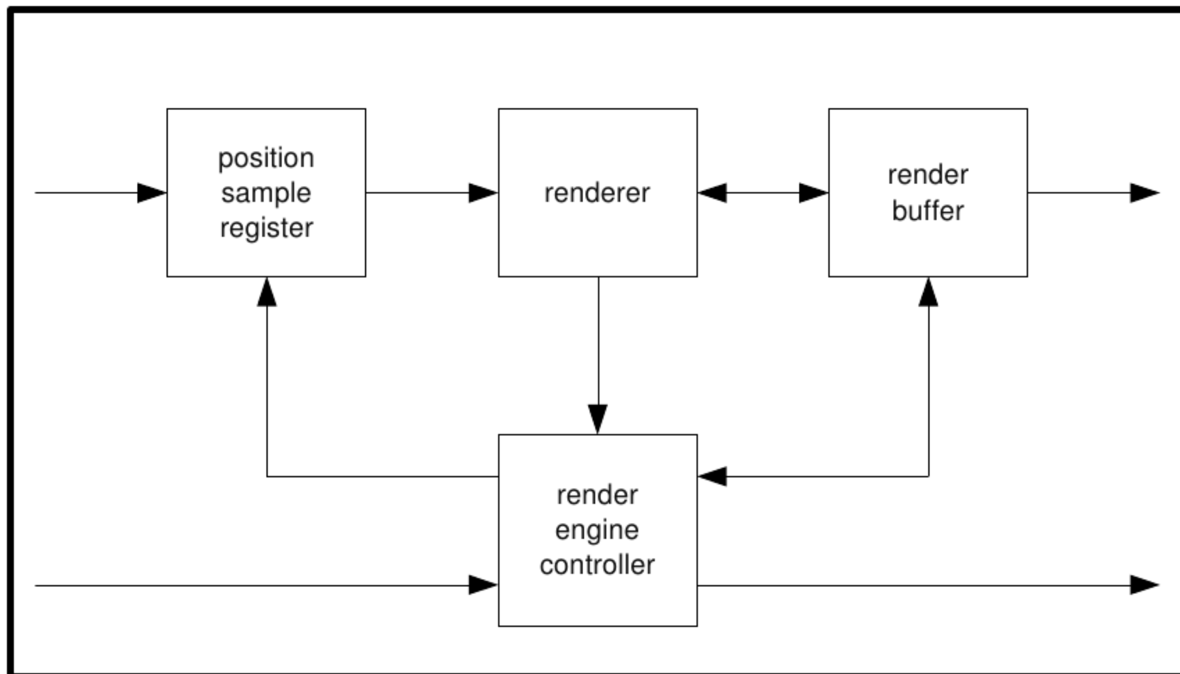


*Figure 7: The entity render_engine*

The entity *render_engine* groups and connects all components used to render new frames.
In the first step of the rendering process, the *position_sample_register* samples the ball and paddle positions. Then the *renderer* generates image data for eight pixels and stores them into the *render_buffer*, from which they are then written to the RAM. Afterwards, the *renderer* generates the next eight pixels, and so on.
The whole rendering process is controlled by the *render_engine_controller*.

### 3.5.3. Entity *ram_controller*

The RAM controller is the interface between the SRAM and the FPGA. On the FPGA, it is situated between the modules *render_engine* and *graphic_output* and stores all data coming from the *render_engine* in the RAM and reads the data requested by the *graphic_output*.

### 3.5.4. Entity *graphic_output*

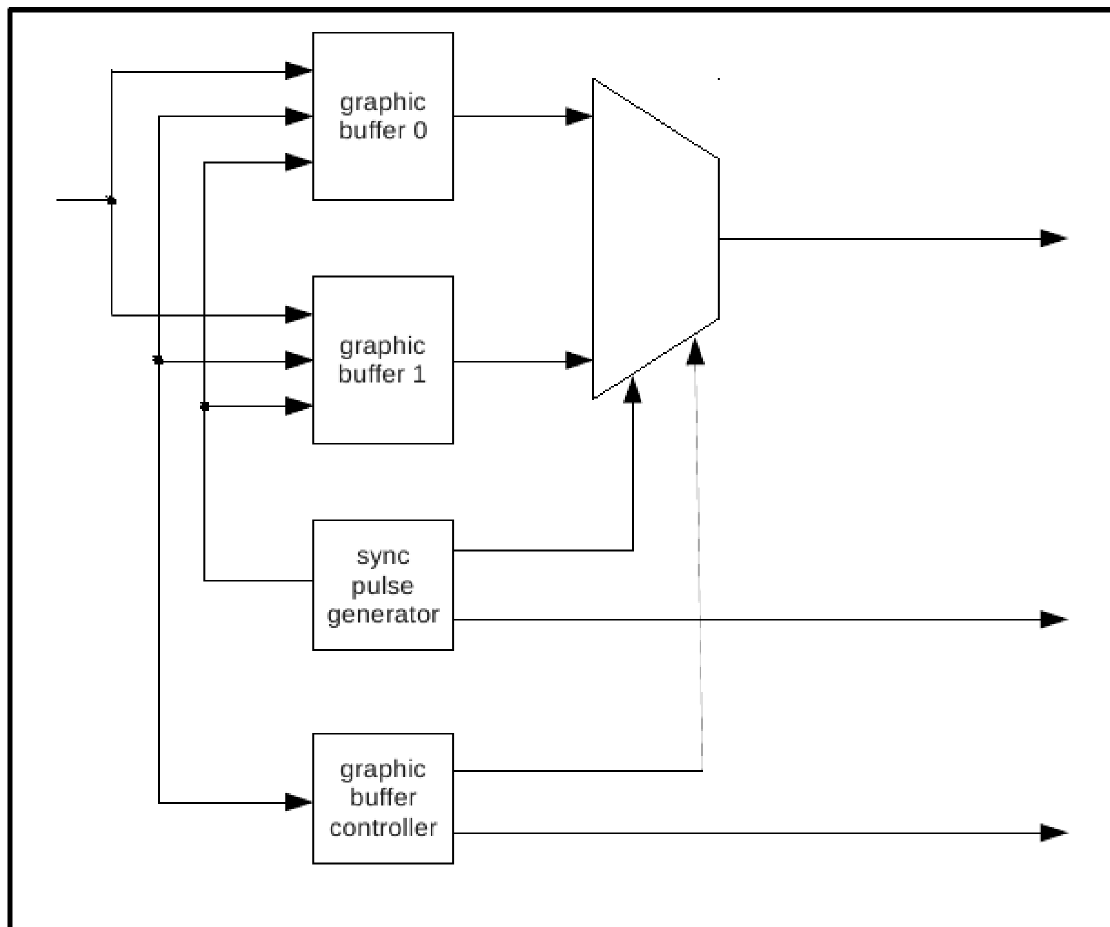


*Figure 8: The entity graphic_output*

The module *graphic_output* is responsible for driving the VGA port of the board with all signals necessary to display an image on the connected monitor.
The VGA signal consists of two components. The image data itself is read from the SRAM by the *graphic_buffer_controller* and is then stored in two buffers (*buffer_0* and *buffer_1*) for a parallel-to-serial conversion.
The second part of the VGA signal is the synchronization impulses. They define the resolution and refresh rate of the monitor. These signals are generated in the module *sync_pulse_generator*.
Besides the four sub-modules above, the *graphic_output* includes a multiplexer that selects the image data from the currently active buffer.

# 4. Coding rules

For a better structure of the created files and readability of the VHDL code, a set of coding rules was used during the creation of this project and must be adhered to during the lab.

## 4.1. Case

Although VHDL is not case sensitive, the following rules apply:

- port names, variables and signals are written in lower case
- constants, generics and states are written in capital letters

## 4.2. Signal naming

- all generics start with *G_*
- all constants start with *C_*
- type names start with *t_*
- signals used as shift registers start with *sr_*
- signals used as counters start with *cnt_*
- signals used for states start with *state_*
- inputs start with *i_*
- inouts start with *io_*
- outputs start with *o_*

## 4.3. File naming

- testbench file names have the same name as the tested module except that the ending is *_tb.vhd*
- a package file has the ending *_package.vhd*
- files with simulation data (e.g. stimuli for inputs) start with the name of the tested unit and have the ending *.vec* (for vector)

## 4.4. Entity, architecture and instance names

- entity names should be the same as the file name in which they were written
- architectures are named *rtl* for synthezisable code and *beh* (behaviour) for test benches or other non-synthezisable code
- instance names end with *_inst*, except in testbenches. There the tested component's instance name is *uut_* (unit under test), followed by the entity name of the component

## 4.5. Initial values

All variables, signals and ports are declared with initial values. This is important, because later the Xilinx tools will use these values to initialize the content of registers during the power-up of the circuit.

# 5. Lab Schedule

The lab follows the schedule shown in table 1.

| Lab Session | Exercise | Module |
|---|---|---|
| | | |
| 1 | Exercise 1 | Combinational Logic |
| | | Developing and Testing a Simple Register |
| 2 | Exercise 2 | Component Declarations |
| | | Debugging - Counter |
| 3 | Exercise 3 | Differences between Variables and Signals |
| | | Serial To Parallel Conversion |
| 4 | Exercise 4 | Finite State Machines – Moore/Mealy |
| | | Testbenches |
| 5 | Exercise 5 | 4x4 Multiplier |
| 6 | Exercise 6 | Sync Pulse Generator |
| 7 | Exercise 7 | Graphic Buffer |
| 8 | Exercise 8 | Graphic Buffer Controller |
| 9 | Exercise 9 | Graphic Output |
| 10 | | Graphic Output Synthesis |
| 11 | Exercise 10 | Ram Controller |
| 12 | Exercise 11 | Pseudo Random Number Generator |
| | | Event Trigger |
| | | Input Decoder |
| 13 | Exercise 12 | PONG Synthesis |

*Table 1: Lab schedule*

Note that only the highlighted modules are necessary for the pong game.