Pong Game

This lab aims to implement the control and the top-level overlay graphics of the pong game. It builds on the previous two labs (Lab 5 and Lab 6) in which you have already implemented the VGA controller and a background frame buffer. Figure 1 shows the context in which this lab corresponds to step 3. You can reuse your design from Labs 5 and 6, with modifications as necessary. Before doing any practical work for this lab, please create a new independent project and import the design files from Lab 6.

Hand-in instructions: Please note you should not hand in any report on this lab, instead, this lab will form a part of the final project.

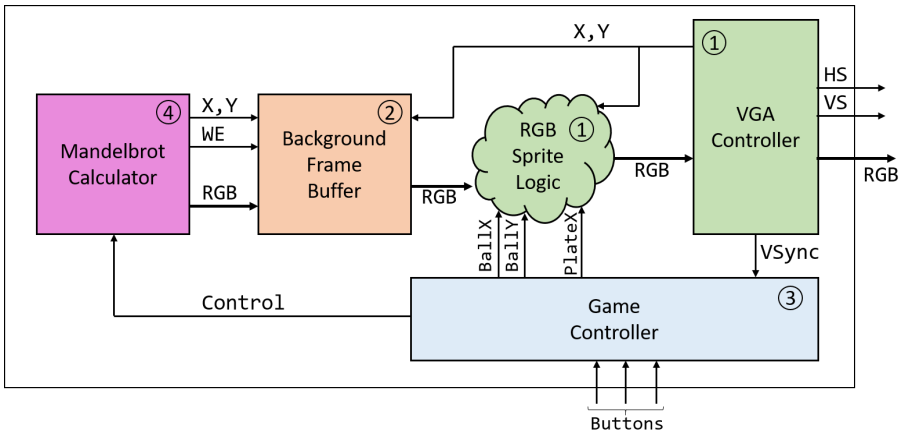

Figure 1: Full project schematic with pong game controller to generate and update the position of the ball and the plate and to display them as overlay to the background.

Important information ⚠

Please always check these things before you start a lab or when you have issues!

FPGA:

- Remember to use the reset button/switch on the FPGA to reset your design and to turn this off again if using a switch.
- Connect the FPGA board's Ethernet port to a computer, router, or any other device with an Ethernet port that can power the Ethernet chip on the FPGA board (no internet connection is needed). See the Lab 2 manual for an explanation.

VHDL:

- For combinational logic, use `process(all)`. Do not write your own sensitivity lists! Please remember to change files using `process(all)` to VHDL 2008. This is done by selecting the file in the `Source` tab. Look at the `Source File Properties` tab and change the type to `VHDL 2008` by clicking on the 3 dots in the `Type` field.
- For defining registers, use the clocked process style `process(CLKxCI, RSTxRI)`. Do not define combinational logic like `CNTxDP <= CNTxDP + AxDI` inside a clocked

process! See Task 4 in Exercise 3 and its solution for further explanation.

- Never write to the same signal in multiple concurrent statements! This means that if you assign to a signal in a process that signal can only be assigned to in that process and nowhere else. The only place you are allowed to assign multiple times to a signal is inside the (single) process where it is assigned to.

Virtual machines:

- EDA server users must start Vivado with `vivado -source load_board_files.tcl` as described in Lab 1. If you do not see the board files in the Vivado GUI, you have likely used the command with a spelling error or something similar.

Windows users:

- Avoid spaces and special characters in your filepaths! Vivado projects can become corrupted if you have spaces and special characters in your filepaths.
- You may have to disable your antivirus tool before running simulations in Vivado.

For common questions/hints to this lab, please see the last page of this document which contains various hints and best-practices.

Preparation

The pong game builds on the previous tasks and extends your design. Nevertheless, we provide you with a new set of files that you can use as a reference. However, as opposed to previous tasks, they are intended mostly as templates. You can build on the project from the previous task (make a copy) and use only what is needed from the provided files.

Download the provided `.zip` file from Moodle and create a new directory for Lab 7. The archive contains the following files (under the `src` directory):

- `vga_controller.vhdl`: Template for the VGA controller, as in Lab 6. Note that a signal `VSEdgexSO` has been added that you need to add to your controller when you re-use it (see Task 2).
- `pong_fsm.vhdl`: This file contains the entity declaration that comprises the core of this task. It controls the game and outputs the coordinates of the ball and the plate.
- `pong_top.vhdl`: Top-level containing the component instantiations and declarations for the clock circuit generator, memory generator, VGA controller, and the pong FSM.
- `dsd_prj_pkg.vhdl`: Package containing constants.

Furthermore, we have the `.xdc` file and the board-files (when using the servers) as usual. Note that the `.xdc` has additional pins to connect to two buttons to control the game. Corresponding top-level ports have been added in `pong_top`. You also find pin constraints for the LEDs if you would like to use them.

You should then proceed as follows:

1. Familiarize yourself with the content of the files.
2. Copy your code from Lab 6 into the Lab 7 directory and modify your code from Lab 6 with the new files we have provided. For this, you only have to make small changes, like adding the new constants from `dsd_prj_pkg.vhdl`.
3. Create a new project based in your Lab 7 directory, ideally in the work directory as specificed in the project handout pdf, which is also in the lab handout `.zip` file.

Task 1: Game Controller Design

The basic Pong Game in its single-player version works as follows: the screen shows a ball (in our case a square) and another square (plate) at the bottom of the screen. The ball moves around and gets deflected when it hits any of the sides of the screen. The objective is to deflect the ball with the plate when it hits the bottom of the screen. If the ball hits the bottom of the screen in any other place than the plate, the game is over. The player can move the plate left and right by using two of the push-buttons on the FPGA board. To start the game, the two buttons for left and right need to be pressed together. In the initial version, of the game, the ball starts from a fixed position around the middle of the screen.

We start with the pen and paper design of the control of the game:

1. Develop a conceptual RTL diagram and the necessary state machines for the control of the game, the ball, and the plate.

You may consider the following hints, but you are free to structure the game controller in any way you like (following the hints is not a requirement for a good grade, there are many very valid approaches):

1. Consider a very simple state machine to control the state of the game.
2. Use counters to track the position of the ball and the plate.
3. The direction in which the ball is moving may be controlled as part of your FSM or as separate state variables (for X and Y).

In addition to the above, we strongly recommend you consider the following: The clock frequency of the FPGA is very fast for any update of the game state (e.g., ball position). To provide a better time base, the VGA Controller should be updated to provide a singe-cycle pulse (`VSEdgexSO`) before a new frame starts (you can detect the edge of the vertical sync signal). In this way, the state of the game is only updated on a frame-per-frame basis.

Note that the `pong_top.vhdl` contains the signals `VgaXxDI` and `VgaYxDI` to the `pong_fsm` entity. You can initially ignore these as they will be explained in Task 4.

Task 2: Implement the PONG Game Control in VHDL

In this task, we implement the basic game in VHDL. You can re-use and build on your previous project. The provided files serve mostly as examples. To this end, consider the following:

1. Add two top-level ports to the top-level as shown in `pong_top.vhdl` to access two additional push buttons. You also need to update your `.xdc` constraint file to include the button pin constraints. For simplicity, you can also simply import the new `.xdc` file provided with this lab.
2. Implement the pong FSM.
3. Write a small testbench for your pong FSM to provide the necessary input signals for a basic simulation and verify it in the waveform viewer. If you use the `VSEdgexSO` signal as a time base, you can generate it artificially in the testbench for the verification. The exact timing of `VSEdgexSO` is not important here and you can make it happen much faster than it would for the FPGA implementation to reduce simulation time.

IMPORTANT! The speed that you choose for the plate and the ball should allow for normal game play. It is not okay to set the speed of the plate and/or ball to be too fast/slow such that it is close to impossible to play the game or to lose the game.

Task 3: RGB Sprite Logic Update

In addition to the game play control, we need to implement the graphics of the ball and the plate. In this simple version of the game, we will show both the ball and the plate as simple white rectangles of fixed size. Corresponding constants are proposed in the provided package, but can be modified. The approach we follow here is to realize the ball and plate objects (sprites) by simply adding logic that checks the X and Y coordinates for which the VGA controller requests RGB values. If the X and Y coordinates are in the region of the ball or the plate, we show the corresponding white color instead of the background. You can later improve on this part.

For the basic implementation, consider the following:

1. Update your VGA controller to provide an additional output `VSEdgexSO` which goes active-high for a single cycle in each frame. This provides the time reference to the pong game controller. You can refer to the updated VGA Controller entity provided with this lab as an example for the interface.
2. Instantiate the `pong_fsm` in your top level. You can refer to the files provided with this lab as a proposal for the entity of the `pong_fsm` and its instantiation on the top level (see `pong_top` for an example).
3. The pong game controller outputs the coordinates of the ball and the plate. Extend your top-level to display the ball and the plate as squares by modifying the input RGB values to the VGA controller depending on the current X and Y coordinates from the VGA controller. The ball and plate should be overlaid on your background. This step corresponds to an update of the "RGB Sprite Logic" shown in the block diagram in Figure 1.
4. Implement and test everything on the FPGA.

Task 4: Extensions of the Pong Game

In this class, we avoid to make the game-play too complex (implementing complex control in VHDL is tedious and more a job for a processor). However, you can consider some small changes to make the game more appealing. For example, you can choose the starting point and movement direction of the ball randomly when the game restarts. To this end, you can use the X and Y coordinates provided by the VGA controller as a source for random numbers. These are called `VgaXxDI` and `VgaYxDI` in `pong_top.vhdl`.

Common Questions

Common questions/remarks for this lab are:

- **How do I create a VHDL file?** After creating a project in Vivado you can click `File` → `Add Sources` → `Add or create design sources`. Alternatively, just use your normal code editor and create new files with the `.vhdl` (recommended) or `.vhd` extensions.
- **How do I resolve the warning 'The PS7 cell must be used in this Zynq design ...?** This warning can be safely ignored as it's unrelated to what we do on the FPGA.
- **How do I resolve the error 'Unconstrained Logical Port'?** While VHDL itself is case-insensitive, **`.xdc` constraints are case sensitive** and your port names should match those in the `.xdc` file in case as well.
- **Outlook does not allow `.vhd` files:** You cannot send `.vhd` files in Outlook, try renaming to `.vhdl` as the `.vhd` extension is also used for **V**irtual **H**ard **D**isk on Windows.

- **Remember that order matters in processes!** Since the order of assignments are done sequentially in a process, meaning that in the example below `DxSO` is only assigned `AxSI and BxSI` and never `AxSI or BxSI`.

  Listing 1: This implementation ignores the line `AxSI or BxSI` as it is always overwritten by the final assignment to `DxSO`.

```
process(all)
begin
  if (CxSO = '0') then
    DxSO <= AxSI or BxSI;
  end if;

  CxSO <= not AxSI;
  DxSO <= AxSI and BxSI;
end process;
```

- **Remember to separate the description of the flip-flops from the combinational logic!** Use a single process for updating the flip-flops and a separate process or concurrent assignments for updating the adder as shown below. This is really important! We also discuss this in Exercise 3.

  Listing 2: VHDL code to show how to define flip-flops for a counter.

```
CNTxDN <= CNTxDP + 1; -- Increment outside clock-process

process(CLKxCI, RSTxRI)
begin
  if (RSTxRI = '1') then
    CNTxDP <= (others => '0');
  elsif CLKxCI'event and CLKxCI = '1' then
    CNTxDP <= CNTxDN;
  end if;
end process;
```

- **Remember to never write to the same signal in multiple concurrent statements!** When assigning to a signal in a process, you can only assign to that signal in that (single) process. The code below in Listing 3 shows the signal `CNTxDN` being assigned to in two different concurrent statements, which is not allowed. With this code, you will see an 'X' for `CNTxDN` in the waveform viewer. The solution is to put the default assignment in a process like shown in Listing 4.

  Listing 3: VHDL code which shows how not to assign to a signal!

```
CNTxDN <= CNTxDP;

process(all)
begin
  if (In0xSI = '1') then
    CNTxDN <= CNTxDP + 1;
  end if;
end process;
```

- **Use default values in your process!** To avoid introducing errors in your code from missing assignments to signals, you should always use a default value for all signals

assigned to in a `process(all)` when describing combinational logic as shown in Listing 4. This is done as the first thing in a process.

Listing 4: VHDL code which shows the assignment of a default value.

```
process(all)
begin
  -- Default values
  CNTxDN <= CNTxDP;
  AxD    <= (others => '0');
  BxD    <= (others => '0');

  -- Actual logic after default values
  if (In0xSI = '1') then
    CNTxDN <= CNTxDP + 1;
    AxD    <= In1xSI;
  elsif (In1xSI = '1') then
    CNTxDN <= CNTxDP - 1;
    BxD    <= In1xSI;
  end if;
end process;
```