Mandelbrot Image

This fourth part of the project implements the pong game background rendering. Where we previously used a pre-defined background picture, pre-loaded to the frame-buffer memory, we now generate a Mandelbrot fractal image on the FPGA. By selecting different regions and zoom-factors, you can generate a large number of cool images, but we limit the requirements to the full-scale image shown in Figure 1. Before doing any practical work for this lab, please create a new independent project and import the design files from Lab 7.

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.

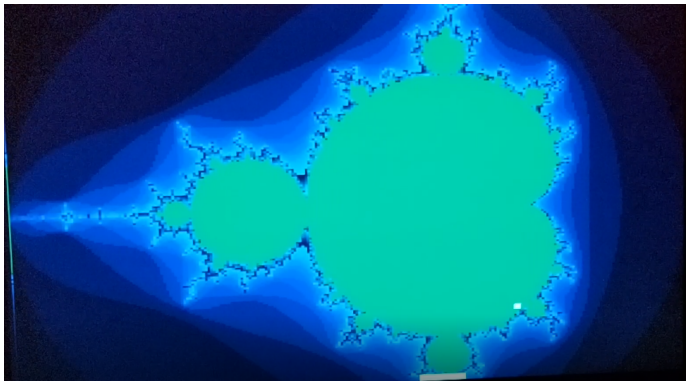

Figure 1: Image of the Pong Game with Mandelbrot background, generated on the FPGA.

Like the previous stages, this stage builds on the previous labs (labs 5, 6, and 7) in which you have already implemented the VGA controller, a background frame buffer, and the pong game control and graphics overlay. Figure 2 shows the context in which this lab corresponds to step 4. For the Mandelbrot generator itself, you can refer to Exercise 5 in which you have already worked with the corresponding datapath.

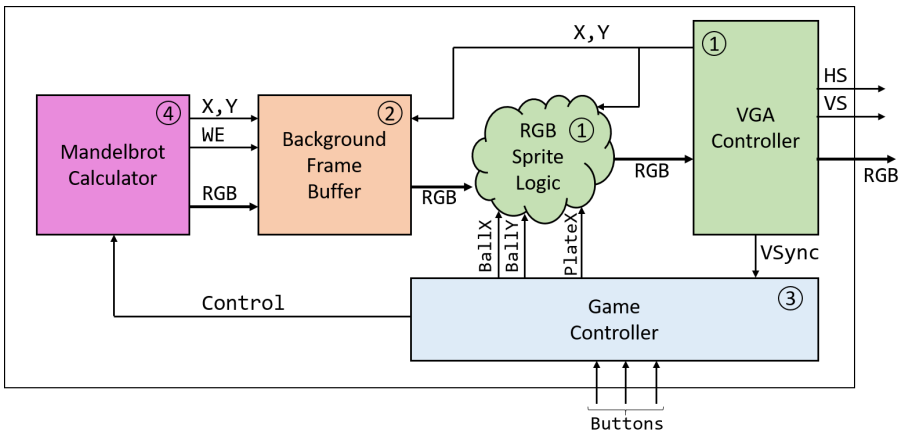

Figure 2: Full project schematic with Mandelbrot Controller to generate (and update) the background Mandelbrot Image.

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.

Remarks on Algorithm and Architecture

You have already encountered the Mandelbrot algorithm in the datapath in Exercise 6. For convenience, we first repeat the explanation below.

Algorithm

The Mandelbrot image is a visualization of the Mandelbrot set. A complex number c is a member of this set if the recurrence $z_{n+1} = z_n^2 + c$, with $z_0 = c$ remains bounded as $n \to \infty$. As an example, $c = -1$ is a member of the Mandelbrot set as the recurrence only returns 0 and -1. As c is a complex number, we can use the real- and imaginary-parts as image coordinates. Since no closed form expression is available to find out if the recursion for a given c converges, we simply iterate for a given c as long as $|z_n|^2 < 4$. Once $|z_n|^2 > 4$ we know that the series will diverge. The number of iterations until $|z_n|^2 > 4$ determines the color of the pixel associated with c in a Mandelbrot image.

In summary, the Mandelbrot image is obtained by computing (and testing the outcome of) the following core iteration, where we use real-valued instead of complex valued notation:

```
INPUTS: c_r, c_i, MAX_ITER;
OUTPUT: z_r, z_i;

n=0; z_r=c_r; z_i=c_i;
while((z_r*z_r + z_i*z_i)<2*2 && n<MAX_ITER) {
  z_r' = z_r*z_r - z_i*z_i + c_r;
  z_i  = 2*z_r*z_i + c_i;
  z_r  = z_r';
  n++;
}
```

The counter n determines the number of iterations for testing the divergence of c, that is, finding when $|z_n|^2 \geq 2^2$. The pictures are obtained by plotting n. If n reaches the maximum iteration number we color the pixel black and otherwise we use the lower bits for blue, middle bits for red and higher bits for green, which is why the colors are brighter closer to the mandelbrot set and dark blue further away from it.

Please note that in the above algorithm, `z_r'` (`z_r` prime) in line 1 of the while loop is the new value of `z_r` in the next iteration of the loop, and is not used for computing the new `z_i` in line 2.

Mandelbrot Generator Architecture

The Mandelbrot algorithm is an iterative algorithm with an unknown number of iterations. Even though in practice we will set a maximum number of iterations (`MAX_ITER`), an isomorphic single-cycle datapath is not economic and will have an unreasonably long path. Hence, an iterative decomposed architecture appears to make sense and we propose to use a single iteration per clock cycle to simplify and reduce the control overhead. In this case, the number of cycles required to compute the Mandelbrot color value for each pixel varies from pixel-to-pixel and corresponds to the number of iterations required (up to the maximum).

We propose that you implement your Mandelbrot generator to generate one pixel after the other. The Mandelbrot generator outputs the `X,Y` coordinates of the computed pixel together with the corresponding color and a single-cycle pulse write-enable signal (`WExSO`) to indicate that the color output is valid (since it may be invalid while the iteration is still running).

Mandelbrot Generator Integration

As you can see from Figure 2, the Mandelbrot Generator accesses the memory through a dedicated port. This structure allows you to write to the memory in parallel to the continuous reading of the pixels for the VGA display. The corresponding write port has an address (`addrb`) and a write enable (`enb`) as well as a data input. You can derive the address from the `X,Y` coordinates provided by the Mandelbrot generator and the enable signal from the `WExSO` output. You do not need to worry about the generator writing to an address of the memory that is read at the same time from the display since in the worst case, this may only result in an invisible glitch.

Preparation

The Mandelbrot task builds on the previous tasks and extends your design. Nevertheless, we provide you again with a new set of files that you can use as a reference. These are intended mostly as a template. You can build on the project from the previous task (make a copy) and use only what is needed from the provided files or use them as a reference.

Download the provided `.zip` file from Moodle and create a new directory for Lab 8. The archive contains the following new or modified files (under the `src` directory):

- `mandelbrot.vhdl`: Entity declaration and template for the Mandelbrot generation. This block is instantiated in the provided top-level template.
- `mandelbrot_top.vhdl`: Top-level containing the component instantiations and declarations for the clock circuit generator, memory generator, VGA controller, the pong FSM, and the Mandelbrot generator.
- `mandelbrot_tb.vhdl`: VHDL testbench for the Mandelbrot generator that simulates the Mandelbrot generation stand-alone and writes a data file with the resulting image that can be read and checked visually in MATLAB. The image file is located in the simulation directory of the XILINX project.
- `dsd_prj_pkg.vhdl`: Updated version of the package from the previous lab with additional constants for the Mandelbrot generator.
- `ViewTBImgFile.m`: MATLAB script to load and check visually the Mandelbrot image generated by the VHDL testbench. Please read the documentation in the script to understand where the image is outputted after simulation.

Furthermore, we provide again the `.xdc` file and the board-files (when using the servers) as usual. You should then proceed as follows:

1. Familiarize yourself with the content of the files.
2. Copy your code from Lab 7 into the Lab 8 directory and modify your code from Lab 7 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`.

Task 1: Mandelbrot Generator Design

Start from the provided `Mandelbrot` entity and implement a Mandelbrot generator. You can visually test the generator with the provided testbench and the MATLAB script that displays the generated image. Note that the simulation to obtain the Mandelbrot image will take a few minutes. The final result should look similar to the image in Figure 3.

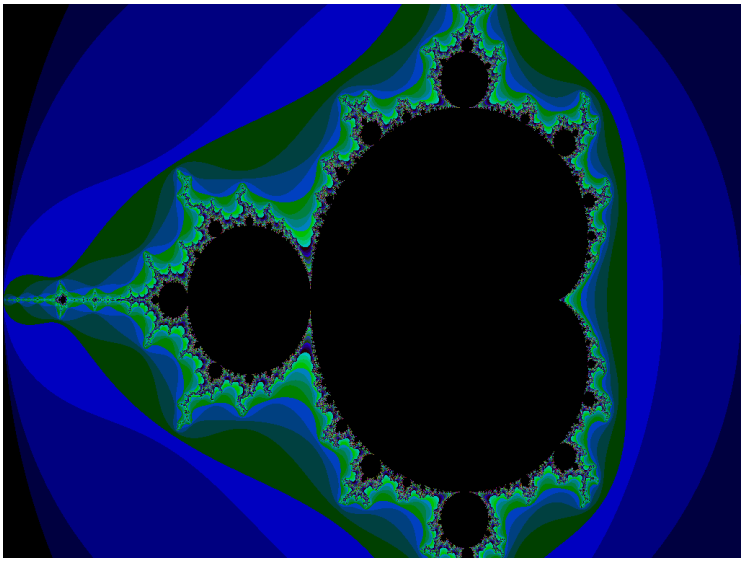

Figure 3: Image generated by `mandelbrot_tb.vhdl` and dispalyed using `ViewTBImgFile.m`.

The following guide-lines may be useful for your design:

- Your design needs to generate the image pixel-by-pixel (in the same order as the raster-scan) to be compatible with the testbench. It also needs to generate an image with a resolution of 1024 pixels (0-to-1023) in X-direction and 768 pixels (0-to-767) in Y-direction.
- To generate the image, your design needs to track the X- and Y-coordinates as well as the corresponding values (real- and imaginary-part) of c. Instead of computing c from X and Y, it is easier to simply have separate counters for X and Y for the real- and imaginary-parts of c. Therefore, you have separate counters inside the Mandelbrot generator for the current pixel X- and Y-coordinates on the screen and counters for the coordinates of c in the complex plane.
- While the counters for X and Y count pixel/line-by-pixel/line, the counters for the components of c increment according to the Zoom factor to span the desired range for c for the image.
- The provided package already contains constants for the starting-point for c (`C_RE_0` and `C_IM_0`) in the top-left image corner and a suitable increment for the real- and imaginary-parts of c (`C_RE_INC` and `C_IM_INC`). The provided values show the entire Mandelbrot set. By modifying the top-left corner coordinates you can change the focus point. By changing the increments to smaller values you can zoom in.
- The starting-point for c (`C_RE_0` and `C_IM_0`) is at (-2, -1) on the complex plane, but since the Mandelbrot set is symmetric around the real-axis as shown in Figure 4, it makes no difference for plotting it. That is, starting at (-2, -1) and (-2, 1) can be considered equivalent, but starting at (-2, -1) means that we only need to add the increments to the starting point, instead of subtracting the increment for the imaginary axis as we would have to do if we start at (-2, 1).

- Note that the provided package also provides a constant for the maximum number of iterations. You can reduce this number to speed up your simulations (and increase it later in the FPGA implementation).

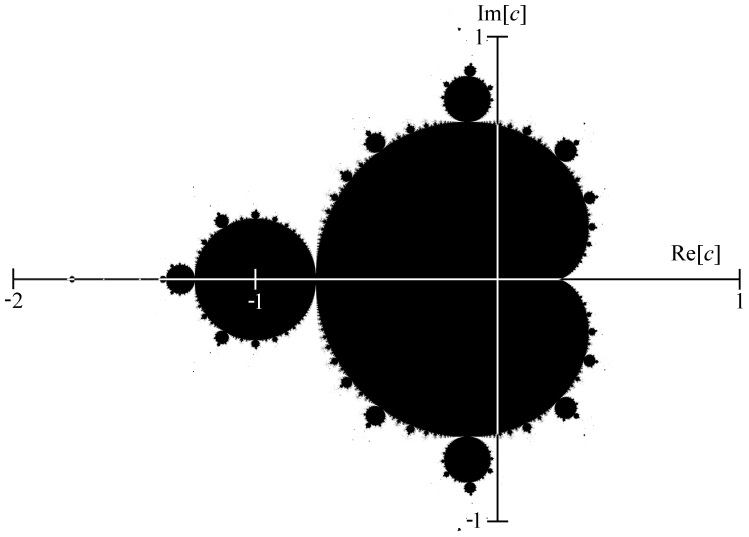

Figure 4: Mandelbrot with real- and imaginary-axis. Note the symmetry along the real-axis.

Task 2: Mandelbrot Generator Integration

Integrate now the Mandelbrot Generator into the top-level. Connect it to the memory and pay attention to the fact that the memory only has space for 256x192 pixels. Hence, similar to the VGA controller, you need to generate the address appropriately from the X and Y coordinates provided by the Generator. Test the Mandelbrot generator together with the rest of your design on the FPGA.

Task 3: Improvements to the Mandelbrot Generator

You can now play with your baseline architectures and experiment as well as make improvements. For example:

- Try different starting points for your Mandelbrot image and different Zoom values.
- You may notice that due to the numerical precision limitation you can not zoom in too far. You can try to increase the numerical precision. However, if you do that, pay attention to the timing report of the XILINX tool. If the design can no longer meet the timing, you will need to add a slower clock to the clock generator for the Mandelbrot generator. Pay attention to use this slower clock only for the Mandelbrot generator AND the corresponding side of the memory. You thus need to use a BRAM with two different clock ports.
- You may have noticed that the Mandelbrot generator only generates a single pre-defined picture. To make this more interesting, you can define multiple starting points and zoom factors. To this end, you can use an array type for the corresponding constants and then select from them. The selection could be controlled by the state machine of the game (for example to change the background for each game) or in any way you like.

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;
```