⚠ You have previously submitted this assignment with David Rama Jimeno. Group can only change between different assignments.

« 13. Exercise 10 (/comp.ce.240/spring-202... 14. Exercise 11 » (/comp.ce.240/spring-202...

COMP.CE.240 (/comp.ce.240/spring-2024/) / 13. Exercise 10 (/comp.ce.240/spring-2024/E10/) / 13.1 Exercise 10: Synthesizing design with Vivado

Assignment description (/comp.ce.240/spring-2024/E10/synthesis/) My submissions (1/1000)

Exercise 10: Synthesizing design with Vivado

Your task is to synthesize the audio synthesizer design to Xilinx PYNQ-Z1 FPGA board with Xilinx Vivado software.

Before starting: The audio codec shield

Before starting, make sure you are on a computer which has the audio codec shield installed on top of the PYNQ board. The Codec board looks like **this (https://plus.tuni.fi/graderA/static/compce240-f2021/pynq_audio_plug.jpg)**. Although the boards in the class have more shields than in the picture (led matrix etc.) so you can't see the audio shield completely. You should, however, be able to see the headphone jacks.

All work stations in the class TC219 don't have it installed.

If there aren't any free PC:s that have the audio shields installed on the board, you can also do the exercise until you have generated the bitstream (chapters 1-8) and continue from **Program the device** chapter on different computer later.

Before Synthesis

Before starting, make sure you have simulated your design and that it works correctly. This is important because synthesis takes generally **VERY** long.

If you encounter any weird problems, check the Known issues -section at the end of this document and triple check that you have done everything exactly according to the instructions.

Contents

This course has already ended.

Obtain the premade I2C block

At this point you will get a ready made I2C-block so you can test your own blocks on FPGA.

- The I2C-block does the configuration of the DA-7212 audio codec (external part on top of the board)
    - You will implement the I2C-block on your own in later exercises (https://plus.tuni.fi/comp.ce.240/spring-2024/E11/i2c_config/)

- Download i2c_config.v (https://plus.tuni.fi/graderA/static/compce240-f2021/E10/i2c_config.v)
    - The file contains FPGA-specific netlist-description of the I2C-block. The netlist has been exported in Verilog format, which explains the .v extension.
        - The file has been made by running synthesis to Vivado project that contains only the course reference design of the i2c_config.vhd. After synthesis the netlist was exported using command *write_verilog*.
        - Since the file contains only a netlist (mainly LUTs and their connections) you can't really understand the functionality even if you look inside the file
        - You also can not change the functionality in any way e.g. by generic parameters
    - This is the file you will replace with your own implementation in exercise 13 (https://plus.tuni.fi/comp.ce.240/spring-2024/E13/debug_and_demo/)

- It's a good idea to put the file in your course directory, e.g. inside *<course_dir>/verilog/*, so it is close to your other source code.

Create a new Vivado project

First you need to setup a Vivado project.

- Start Vivado 2017.2. You can find it from the start menu.

- Create new project with name *synth_top*.
    - Good location for the project might be inside *<course_dir>/syn/* directory but you can put it anywhere you want. Make sure that it's located inside *P:* disk if you work in TC219.
    - It is a good idea to create project subdirectory in case something goes wrong and you need to start a new project from scratch.
    - Click "Next"

- Add sources
    - Add the code files made in the previous exercises to the project
        - *adder.vhd*
        - *audio_ctrl.vhd*
        - *multi_port_adder.vhd*
        - *synthesizer.vhd*
        - *wave_gen.vhd*
        - and additionally *i2c_config.v* which you downloaded
    - Adding sources later is also possible if you forget something
    - Languages:
        - Target language defines the language of the HDL wrapper (later). You can set this to VHDL.
        - Simulation language doesn't matter because you don't need Vivado simulator on this course
    - Click "Next"

- Add constraints
    - Constraints file (.xdc extension in Vivado) defines all the physical restrictions of the circuit and the board. It can be used for various things such as timing constraints (e.g. setup and hold times), routing issues etc. but we will use it only for mapping all the ports to physical pins on the FPGA.
    - If you already have a constraints file you can add it here
        - In this exercise you don't, so you can skip this part. We will make one and add it later.

- Choose the correct board type
    - As noted in TIE-50100 Vivado tutorials, this is very important to get right, otherwise the design won't work at all. By choosing the correct board type, the synthesizer knows the target technology in which the logic will be implemented.
    - Use the board type *xc7z020clg400-1*. Find it either by search or these filter options:

        - Family *Zynq-7000*,
        - Package *clg400*
        - Speed grade *-1*.
    - Click "Next"

- Verify that everything looks fine and click "Finish"

Create a top-level block diagram

A design needs a top-level block in which all the components are connected together. It can be done in different ways. We will use a graphical method.

- Click create block design
    - Name the block design as *system_top_level*. Click ok.
        - A blank design will show up. This will be the base of the diagram.

- If you didn't add all the source files yet, add them now

- Add modules to diagram
    - Right-click on the blank diagram and choose "Add module"
        - You should see a list of all the source files that are added in the project.
    - Choose *synthesizer.vhd* and click ok. A component should appear on the diagram.
    - Do the same for *i2c_config.v*.

Phase-locked loop (PLL)

We will use a ready made IP which implements a PLL from Xilinx. A PLL is a block that can generate multiple clocks with different frequencies by multiplying and dividing it's input clock. The clocks for synthesizer and i2c_config will be generated by PLL.

- Right click on the diagram and choose "Add IP"
    - IP that implements a PLL is called "Clocking Wizard".
    - Add it to block diagram either by dragging or double clicking the name.

- Double click on the Clocking Wizard block in the diagram in order to customize it
    - For every option that is not mentioned below, use default settings
        - Default settings can be restored by deleting the Clocking wizard and adding a new one.

- Select following options:
    - From the "Clocking options" page:
        - Primitive -> PLL
            - Clocking wizard can be used as MMCM (Mixed-mode clock manager) or PLL (Phase-locked loop). PLL is basically a subset of MMCM and it will be enough

in our case.

- Clocking Features -> Select Frequency synthesis, Safe clock startup. Uncheck everything else.
  - Phase alignment could be left checked but in our case it only generates an unnecessary buffer in PLL feedback loop, so it can be turned off.
  - Safe clock startup makes sure that all the generated clocks have been stabilized before "locked"-signal goes up. It means that none of the clocks can be used by other components too early, which is good.
- Input clock information -> Manual, 125.000 MHz
  - The PYNQ-Z1 board has one clock pin that gives a 125 MHz clock to the FPGA. We will use it as it is.
  - Don't use the secondary clock here

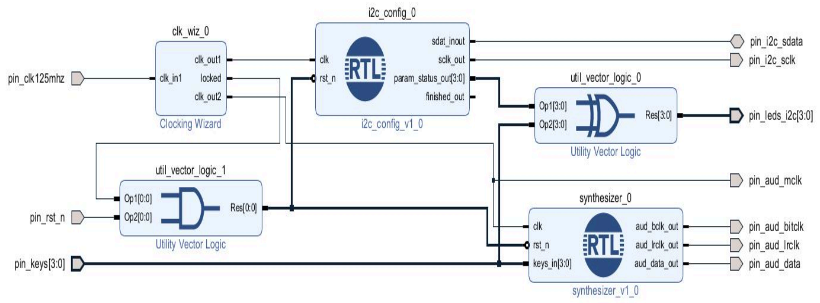
- From the "Output clocks" page:
  - Two output clocks:
    - clk_out1: Output freq (requested): 50.000 MHz
    - clk_out2: Output freq (requested): 12.288 MHz
      - The actual frequency might differ a bit from the requested for this one. It won't be a problem, it will be close enough.
    - Keep other settings for both clocks in defaults
  - Untick "Reset" from "Enable Optional Inputs" -section.

- Click "ok" once completed

Connect the blocks together

Now that all components are set up, it's time to connect them together.

The desired circuit is shown in this picture:



(https://plus.tuni.fi/graderA/static/compce240-f2021/E10/system_top_level_bd.jpg)

- Add I/O pins
  - To add I/O pins you can either
    - Right click on blank space in the diagram and click "Create port"
      or

        - Right click directly on the output of a component and choose "Make external". Since we want to rename our ports anyway, this is not really much faster than the first method.
    - Name the ports as in the picture.
    - Be careful with the direction of the port. Especially *inout*-ports.
    - Type can be left to *other* for all pins.
    - For ports wider than 1 bit, the width can be set by checking "Create Vector"

- The AND gate and XOR gate can be found from "Add IP" as "Utility Vector Logic"

- Make the connections between the components and the pins
    - Drawing the connections is pretty straightforward. Just click, hold and drag.
    - The bus widths will be automatically assigned. Make sure the widths between start and end points match each other though.

- Buttons "Regenerate layout" and "Optimize routing" in the top bar can help in getting a clean looking diagram. Sometimes they might work in non-optimal way so you might consider moving things by hand.

Prepare the design for synthesis

After the connections have been made a few things needs to be done before synthesis.

- Generic parameters of RTL blocks can be changed from block diagram. Values set in the block diagram will override the default values set in the entity declaration of the component. Even though they should default to same values, it's good to be careful and check them.
    - Check the generics of the synthesizer block by double clicking it
      As a reminder:
        - clk_freq_g: 12 288 000 (Hz)
        - sample_rate_g: 48 000 (Hz)
        - data_width_g: 16 (bit)
        - n_keys_g: 4
    - This can't be done for presynthesized netlists such as i2c_config.v.
        - The generics used during the synthesis of this ready made block are the same that you will be using later in your own implementation.

- Finally, we need to make sure that the tri-state logic in I2C-block synthesizes correctly. Tri-state logic must exist only on the top-level and if you skip the next step, that might not be the case.
    - While in block design page, select "Sources"-tab in upper left corner under text "BLOCK DESIGN"
    - Expand "Design sources" and right click on the block design file (.bd extension)
    - Select "Generate output products"
    - From Synthesis options use "Global". Click "generate".

- Now that the block diagram is complete, it needs a HDL wrapper so that it can be synthesized.
    - The wrapper will be created in the language specified in the "Target language" -option which you set when creating the project. It can also be changed from *Tools* -> *Settings* -> *Project settings* -> *Target language*.
        - Either language is fine since we are not going to modify the wrapper ourselves. If you want to see the contents of it, VHDL will probably be more familiar to you.
    - Right click on the block design file and select "Create HDL Wrapper". Let Vivado manage everything and do its magic. Once done, the wrapper will show up in the sources tab.

- Right click the wrapper and set it as top from the menu.

Connect the design to FPGA's physical I/O-pins

The ports of the design have to be connected to physical pins on the board. Synthesis itself could actually be done before this, but this has to be done before implementation anyway, so you may aswell do it right now.

- As mentioned in the beginning you will make the connections using a constraints file. Other ways to do this exist but for designs about this size using a constraints file is by far the easiest option.

- Download the Pynq-Z1 XDC constraints file template (http://pynq.readthedocs.io/en/v2.1/overlay_design_methodology/board_settings.html) and use it as a base
    - You can copy the file to *<course_dir>/xdc/* directory. Rename it as *synth.xdc*.

- Map all the ports in corresponding package pins
    - The TCL syntax in the constraints file needs to be correct. It's best that you don't modify it other than the port names.
    - Use IOSTANDARD *LVCMOS33* for every pin connection
    - Uncomment the rows you need and change the port names accordingly to your design. The name of the port should go as an argument to *[get_ports { }]* statement (inside the braces {}).
    - Some connections are pretty straightforward, such as LEDs, buttons and I2C pins
    - Use the switch closest to the push buttons as your reset switch
    - To create a 125 MHz clock in the right pin (the clock pin on the FPGA is H16), use these two lines:

```
set_property -dict { PACKAGE_PIN H16   IOSTANDARD LVCMOS33 } [get_ports { p
create_clock -add -name sys_clk_pin -period 8.00 -waveform {0 4} [get_ports
```

- GPIO pins needed by the audio codec board can be found from its User guide (https://www.renesas.com/eu/en/document/mat/ard-audio-da7212-audio-shield-user-guide)
    - Table 2 can help here
    - Use the default pins, not the optional ones
    - Note that WCLK (Word Clock) = LRCLK (Left-Right Clock)
    - When it comes to data pins use only the one you need.
    - Note that pin numbers are obviously not the same in PYNQ board. You need to find out the matching ones.
        - **Hint**: In the shield all the J-connectors' pins are numbered with same standard:
          Pin number 1 is the one with a square, and the pin on the opposite side is number 2, pin next to pin 1 is number 3, opposite pin of that number 4 and so on. One half (the one with the squared pin) has all the odd numbers and the pins on the other side are even.
        - For finding the corresponding pins in the PYNQ board, the section about Arduino/chipKIT shield connector (= GPIO pins) in PYNQ-Z1 reference manual (https://reference.digilentinc.com/reference/programmable-logic/pynq-z1/reference-manual) can help a lot here
        - PYNQ-Z1 schematic (https://reference.digilentinc.com/_media/reference/programmable-logic/pynq-z1/pynq-z1_sch.pdf) can give you further information

- Save and add the constraints file to the design from *Add sources* -> *Add or create constraints*. The menu can be found either by right clicking the sources list or from under "File" menu.

Synthesis

The rest of the flow should be pretty automatic. If you haven't made any critical mistakes you should have a working design after a little while.

- Click "Run synthesis"
    - Vivado will turn your design to FPGA-specific netlist. This can take a while.

- Click "Run implementation"
    - This step is the same as "Place & Route". This will also take a while.
    - If you open the implemented design afterwards, you can see the area that your design takes on the board

- Once synthesized, generate a bitstream of your implementation

Program the device

Now you can transfer the design to the FPGA board

- Make sure the PYNQ board is connected to your PC with USB cable

- Make sure that the two jumpers in the left side of the board are in correct positions:
    - The one in upper corner should be in "SD" position
    - The one next to the power switch must be in "USB" position
        - If the board is connected to external power supply (other than usb), you can have this in the "REG" position

- Power on the PYNQ board if you haven't done already.
    - Red led LD13 should light up.
    - You should also see an orange led light up in the audio codec board
    - If there is an SD card connected to the board:
        - Green LD12 should also light up
        - The device might start blinking other leds after a minute or so before you have programmed it yourself. It just means that the embedded linux software inside the SD card is booting up. Don't worry about that, you can leave the SD card as it is. Proceed to programming your device normally.

- Open Hardware manager in Vivado
    - Open target -> Auto connect
    - Program device
    - The device is now programmed with your design

First test run

You can finally try if your synthesizer design works.

- Connect an audio output device to the codec board.
    - There should be speakers in the classroom that you can borrow.
    - Make sure you connect them to the audio codec on top of the board and not directly to the PYNQ board.
        - The audio jack in the PYNQ board can only be driven by PWM signal. It is also just a mono output.
        - We can't use that because our design depends on an external audio codec and we also get stereo output
    - There's two 3,5" audio jacks in the codec board - use the smaller one, which is also closer to the corner.
        - The other one is for line-in. We don't want to use that.
    - Here's (https://plus.tuni.fi/graderA/static/compce240-f2021/pynq_audio_plug.jpg) a picture of the correct setup.

- The rightmost switch is reset
    - Pull towards the outer edge (yourself) = reset
    - Push towards the center (away from you) = normal function

- After the reset, the I2C-controller configures the audio codec and after that switches on some leds. The leds indicate the amount of I2C configuration values sent to the codec in binary number. In our case it's 15 so all the leds are on.

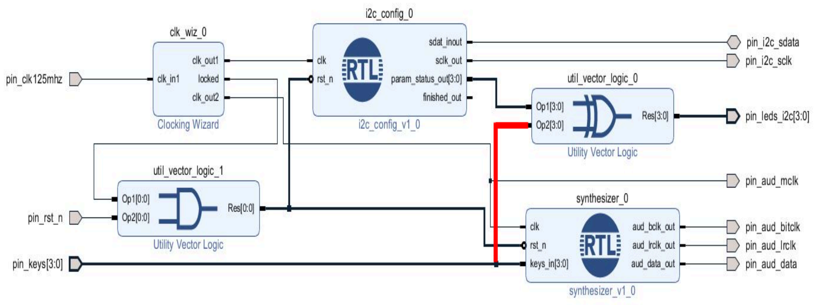
Press the buttons and if you are lucky, you can enjoy the euphonies. If everything seems to work, you can proceed to returning section.

If this does not happen:

- Do not worry too much! At the moment, the most important thing is to complete the synthesis without errors. We get back to chasing the system errors in exercise 13.

- Check the settings, especially the top level block diagram *system_top_level.bd* and the signal connections to I/O-pins (constraints file).

- For testing the settings, you can try the model design synthesizer.v (https://plus.tuni.fi/graderA/static/compce240-f2021/E10/synthesizer.v). Like i2c_config.v, it is a presynthesized netlist.
    - Replace the *synthesizer.vhd* with *synthesizer.v*
        - You can do this by finding synthesizer.vhd file from source hierarchy -> right click -> Replace file
        - The new file may show up first as non-module file, syntax-error file or something like that.
          To fix this click the file in the tree, and change the type from VHDL to Verilog. After the hierarchy updates, the file should be fine.
            - Sometimes you might still encounter some problems with the source file hierarchy. If you can't figure out how to fix the problem you can try rebooting Vivado/your pc, and maybe even creating a new project where you only add the verilog netlists as sources without adding the .vhd files.
              You can also always ask help from an assistant.
        - Delete the net between pin_keys input and the XOR component (util_vector_logic_0). Do not remove the net between the pin_keys input and the synthesizer_0 block. The net to delete is highlighted in red within the following figure:



(https://plus.tuni.fi/graderA/static/compce240-s2024/E10/system_top_level_bd_mod.png)

As you can see from the block diagram, this will result in the buttons no longer driving the LEDs. However, the buttons will still control the synthesizer and therefore should still control the generated audio.
This connection should be restored in the final design (future exercises) and only needs to be modified when performing synthesis with the example netlist.

- Re-run synthesis and implementation, generate bitstream and reprogram the FPGA
- Make sure that you get the sound with provided netlist files
- When this works, you know that
    1. Settings of the tools are OK,
    2. Programmer-application works,
    3. USB-cable is correctly connected, and
    4. FPGA-board works

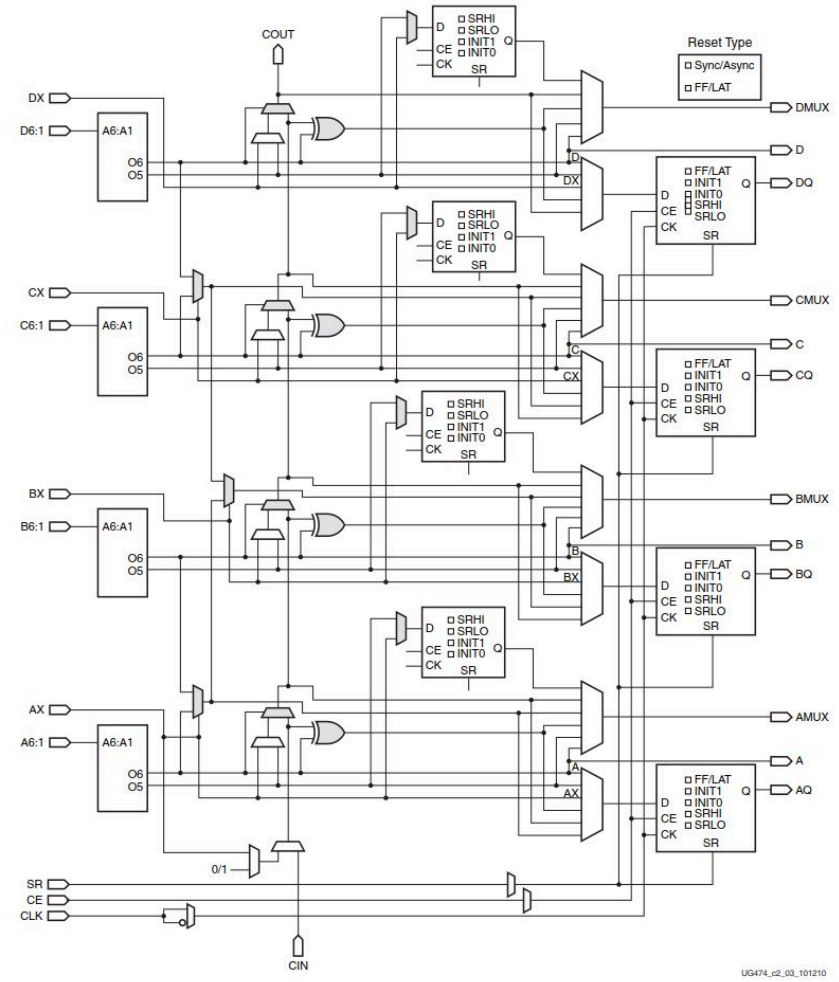
  Do not by any means understate these kind of "of course it shall work" type tests. If this did not work, most likely your own system would neither.

Looking at the reports

After the run you can have a look at your design and get some interesting values from the reports. Open the implemented design if not open already.

- "Project Summary" gives a brief overview of your design
    - The amount of warnings (both synthesis and implementation) might throw you off at first. It is, however, quite normal to have some warnings even in final working designs. Still this obviously doesn't mean that warnings shouldn't be given any attention.

- Utilization report gives information about the amount of used logic elements
    - For Zynq-7000 boards such as PYNQ-Z1 these are called "Logic Slices"
        - One configurable logic block (CLB) contains two slices
        - One slice contains 4 LUTs, 8 flip-flops, some multiplexers and carry logic. There are two types of slices: SLICEL and SLICEM. The difference between them is that SLICEM can also be used as RAM while SLICEL is more simple and thus it can not. Schematic of SLICEL-type slice in the picture below



(https://plus.tuni.fi/graderA/static/compce240-f2021/E10/slicel.jpg)

- Further information about logic slices for interested: 7 Series FPGAs Configurable Logic Block User Guide (https://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf)
- By exploring the device-tab in opened implementation you can see which slices and which parts of them are in use in your implementation

- To get the utilization information you can either
  - Open the implemented design and choose "Report Utilization"
    or
  - Run tcl command *report_utilization* in Tcl console. To get information in lower hierarchy you can add the switch -*hierarchical* to the command.
  - By default the report shows utilization with plain numbers. That's nice but sometimes you might be interested in the utilization measured in percentages instead.
    To get the percentages
    - For the graphical method, you can click the %-sign
    - For the tcl command the percentages are shown in normal mode. In hierarchical mode the percentages are not shown by default but the switch -*hierarchical_percentages* can be used together with -*hierarchical* in order to get them.

- From the timing report you can see all the timing related things such as possible timing violations.
  You can also calculate e.g. the maximum clock frequency in which your design may (in theory) work.
  - To see the timing report you can either
    - Open the implemented design and choose "Report Timing Summary"
      or
    - Run tcl command *report_timing_summary* in Tcl console
  - If there are no violations, you shouldn't need to do anything crucial here
  - To calculate the maximum theoretical clock frequency (for every clock) of your design:
    - Find the requirement clock period of the clock (with the current frequency)
    - Find the Worst Negative Slack for the Max Delay -path. The value should be positive if there are no violations. In the gui the value you want may be called "WNS".
    - Subtract the slack from the required clock period
    - This is the minimum clock period. Maximum frequency is the multiplicative inverse of that.
    - Note that this should be done for every clock separately

- You should have been viewing the reports as post-implementation reports. The reports can also be viewed post-synthesis. You might notice some differences between them since Vivado does some further optimizations in the implementation phase.

- The RTL schematic of your design can be viewed from "RTL analysis -> Open Elaborated Design -> Schematic"

Return

Obtain the files specified below and put them under E10 folder in your Git repository. Return them also by file return in the bottom of this page.

- Files to return:
  - `answer10.txt`
  - `wave_gen0.png`
  - `synth.xdc`

- In your answer10.txt:
  - (For utilization and timing, use the post-implementation reports)
  - Utilization of the board
    - The amounts and the percentages of (for the whole board):
      1. LUTs
      2. FFs
      3. IOs

    4. PLLs
  - The division of resource usage block by block
    Use the form "Slice LUTs" / "Slice Registers" / "Slice"
    1. i2c bus controller
    2. all wave generators
    3. multiport adder
    4. audio ctrl
- Timing concerning both clocks "clk_out1" and "clk_out2".
  - The requirement period (ns) and slack (ns) of both clocks' max delay paths
  - From the values you got, calculate the minimum clock period (ns) and the maximum clock frequency (MHz)
- Did you manage to do the test run succesfully with your own synthesizer? If not, what happened and what did not happen?

- For wave_gen0.png:
  - Open the RTL schematic of your design
    - Find the wave generator's instance 0 and open it
    - Attach a screenshot of the internal structure of the block. (Try to deduce how the figure matches to your VHDL-code.)

- Return also your own `synth.xdc` constraints file that you used

Known issues

Here are some issues that might occur and some possible fixes to them. Proceed in up to down order.
If the things listed here do not work, you can always ask an assistant for help.

- If Vivado for some reason crashes in creating/opening a project
  - Reboot Vivado
  - Reboot your PC
    - If the crash happened when creating an entirely new project you might also want to delete the project subdirectory that Vivado created and start over just in case. It may have ended up in corrupt state.
      You wont lose much work anyway since the project was just created.

answer10.txt

Choose File No file chosen

wave_gen0.png

Choose File No file chosen

synth.xdc

Choose File No file chosen

Submit with David Rama Jimeno Submit

Earned points

1 / 1

Exercise info

Assignment category
VHDL exercises

Your submissions
1 / 1000

Points required to pass
1

Deadline
Sunday, 7 April 2024, 23:59

Late submission deadline
Friday, 31 May 2024, 23:59

Group size
1-2

Total number of submitters
55

« 13. Exercise 10 (/comp.ce.240/spring-202... 14. Exercise 11 » (/comp.ce.240/spring-202...