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

This course has already ended.

« 16. Exercise 13 (/comp.ce.240/spring-202… 17. Bonus exercises » (/comp.ce.240/spring…

COMP.CE.240 (/comp.ce.240/spring-2024/) / 16. Exercise 13 (/comp.ce.240/spring-2024/E13/) / 16.1 Exercise 13: Debugging exercise and system demonstration

Assignment description (/comp.ce.240/spring-2024/E13/debug_and_demo/)

My submissions (2/1000)

Exercise 13: Debugging exercise and system demonstration

Your task is to explore the operation of the synthesized system and fix the errors. You will also learn how to use Integrated Logic Analyzer (ILA), which allows you to debug the design inside the FPGA board.

Open the project

- Start Vivado 2017.2. You can find it from the start menu.
- Open the project that you created in exercise 10 (https://plus.tuni.fi/comp.ce.240/spring-2024/E10/synthesis/)

Programming the compiled project to FPGA

If your design worked properly before when using the reference design of the i2c block, you can now try programming the board using your own `i2c_config.vhd` design

- Go to the sources listing
- Proceed downwards in the hierarchy and find the `i2c_config.v` file (in the lowest hierarchy level you can go)

- Right click -> Replace file, and choose your own `i2c_config.vhd`

- After Vivado updates the hierarchy the new file may not show up in "Design sources" where it should be. Instead it can be found under "Non-module files", "Syntax error files" or something like that. This happens because the file was changed from Verilog to VHDL and Vivado doesn't register this automatically.
    - To fix this, change "Type" from Verilog to VHDL in Source File Properties (also called Source Node Properties)

Once the replacing is done, you can synthesize and program the FPGA normally as you did in the exercise 10 (https://plus.tuni.fi/comp.ce.240/spring-2024/E10/synthesis/). To make sure tri-state logic works correctly, it may be wise to regerenerate the output products as well before synthesis, just like you did before. Also make sure that Vivado hasn't changed your top-level block during the process.

If the sound output and leds etc. seems to be working after device has been programmed, you have a working design. Congratulations!

State machine recognization in Vivado

Many synthesis tools support state machine optimization and so does Vivado. The problem is that Vivado tends to be very strict when it comes to the design style of a state machine. So if the state machine is not designed in the code level as Vivado wants, it may not be recognized by Vivado, even if it might be recognized by other EDA tools such as Modelsim or Quartus. If Vivado doesn't infer the FSM, the same functionality will be synthesized as plain logic.

Some parts of this exercise needs your fsm to be inferred by Vivado. Don't worry though, you can still pass the exercise even if you can't get this to happen. You can still have perfectly working design. But if you managed to synthesize your design so that Vivado was able to infer state machines, you are in luck. It will make debugging easier.

- To verify this, you can look into synthesis reports and search for any hints about FSM inferring/encoding. There should also be a table of all the states and their bit encodings.
    - If you don't see anything like this, then it means that your FSM wasn't inferred by Vivado and instead it was implemented in plain logic. It's not a huge deal in our case, the design can work fine regardless of this. You can still probably guess all the state changes from the waveform later. The biggest problem is that the states between which the transitions go are not so obvious since you don't know the bit encoding. Also we'll most probably lose ability to use one-hot encoding.

- If Vivado did infer your FSM, you can check the encoding. If it looks like one-hot encoding you are fine.
    - The default fsm encoding setting is "auto", which lets Vivado decide itself the best encoding scheme. Depending on the optimization scheme Vivado is using, it may use e.g. some sort of binary encoding. As we know, the one-hot encoding is safer and easier to debug, so we might aswell tell Vivado to use that instead.

- Go to Synthesis part of the Project settings
- Look for option *-fsm_extraction* and change it to *one_hot*.
- Now if you run synthesis again and look into the reports you should see that Vivado now uses one-hot encoding scheme. You should take notes of the bit encodings of the states. You will need them later.

Using Integrated Logic Analyzer

Vivado Integrated Logic Analyzer (ILA) is a logic analyzer which stores values of selected signals on the rising edge of clock to the internal memory of the FPGA chip. The user defines a suitable trigger condition for when the values are stored. After that, the stored values can be viewed as a wave form inside Hardware manager.

For most logic analyzers the trigger condition(s) can be changed without a new synthesis, which is nice. On the other hand, changing the signals under investigation requires a new synthesis. In Vivado terms only a new implementation is required. Synthesis results will keep up to date.

There is a limited amount of RAM in PYNQ-Z1 board, but it is quite sufficient for our use if a decent sampling frequency is used. For instance there is no point taking samples on every clock edge because we will have all the needed information with much fewer samples.
In theory we could just increase our sample depth, but all the RAM the board has is not available to us since we are using multiple clock regions.

Setting up the debug cores

Note: The steps in this chapter modify directly your constraints file (*synth.xdc* or similar). The changes should be appended in the end of the file.
So if you ever for some reason need to delete all the debug cores and redo this chapter, it's best to just manually clean all the debug related stuff from the constraints file prior to redoing this chapter. This will eliminate some problems that may occur otherwise.

- If the design is not synthesized (or synthesis isn't up to date), synthesize it

- Open synthesized design -> Set up debug
  - Choose all the signals that you want to debug. You also have to add the signals that you need as trigger conditions. Some interesting signals would be:
    - All the state registers (e.g. `curr_state_r`)
    - I2C-signals/buffers (e.g. `sclk_out`, `sdat_inout`, `sdat_inout~reg0`, `sdat_inout~en`)
    - `param_status_out` can also be useful, if you have any problems with the leds. It's also a good way to move the beginning of the sampling.
    - For trigger conditions, you should add your clock counter inside `i2c_config`. The name of the counter depends on your own naming (e.g. `clk_divider_r` , `ratio_counter_r` , `counter_r` or something totally different).

    - From the next page use sample depth 1024 and activate capture control

- After setting up it's wise to open the Debug-window in synthesized design and tweak the probes to your liking so they are easier to look in the waveform.
  You can open the window automatically in previous step, or in case it didn't open you can click "Window -> Debug" from top bar.
    - You should at least make sure that all the state registers are under one probe. It will make them better visible in the waveform.
      Same goes for your clock counter.
    - Remove all the unneeded probes and channels afterwards
    - If you know which probes you are using (only) for data and which (only) for triggers, you can set them so if you like. But keeping the default *Data and trigger* -setting for all shouldn't hurt (in fact it gives you more flexibilty). Setting them separately might save you some area and in turn save you some time in implementation (not actually measured).

- Save your design, run implementation and generate bitstream
    - Note that the implementation will take longer than the previous implementations without the debug hub and debug cores. That's because those are pretty huge. Reserve at least 10 to 15 minutes for this.
    - If interested, open the implemented design and go check the device window to see the space the implementation takes.
      It's also possible to see them and their connections in the Schematic window.

Drawing the waveform

There are two ways to get data from the board: Trigger mode and Capture mode. For the most part we will be using Capture mode. When using capture mode, you will define a trigger condition and when the condition is met, the logic analyzer will capture the state of every signal specified in memory. This will happen many times until we exceed the sampling depth.

Since the 50MHz clock is way too fast for our samples we will be using the clock counter for sampling. For instance the most significant bit of the clock counter should be oscillating with pretty good frequency for our case.

The Trigger mode can become handy e.g. if you want to specify a moment to start taking samples other than right in the beginning. Trigger mode can be used together with Capture mode. More about this later.

- Program the device. This time it will also ask for debug probes file, use the one suggested.

- You should see a waveform diagram. Add all the needed signals to the waveform from the +- button above
    - To see the states clearly, it's a good idea to use binary radix
    - Opening up the individual states is also a good idea

    - You can also use enumerations and/or rename the individual state signals to match the state name (more about this later)

- Check that "Capture mode" is set to "BASIC" in the settings on the bottom

- From the right, select Capture mode and add the clock counter probe to the list from the +-button ("Add probes" -button)
    - If you added the whole counter signal (every bit) when setting up the debug, you have a lot of options here. You can basically use any bit as trigger condition.
    - Trigger condition can be set on the right. To ease up the bit selection the radix can be set to binary. That way the value can be set bit by bit, with letters (and numbers) below
        - 0 (logical zero)
        - 1 (logical one)
        - X (don't care)
        - R (0-to-1 transition)
        - F (1-to-0 transition)
        - B (both transitions)
        - N (no transitions)

      It's easiest to have only one bit at a time active (rest of the bits in X state) although having more complicated trigger conditions is possible.
    - For starters you can use e.g. the most significant bit. At least for the course reference design, good bits tend to be around 8 to 10.
      You will get bigger time window to show when using higher bits, but you will also lose accuracy. If you need more accurate information, you can lower the bit.

- When ready, set reset active from the switch on the board, run the trigger by pressing the "play button" and lift the reset
    - You should see a waveform of some kind. You can play with the trigger conditions (e.g. the bits) to change the accuracy of the waveform.

- If you seem to be having problems further on the waveform (e.g. beginning works fine but something goes wrong later), you might want to setup a trigger condition in Trigger mode alongside Capture mode.
    - Normally the Capture mode starts working right in the begining. If there is a trigger set up in Trigger mode side, the Capture mode starts working only after that trigger is met. Everything before that time moment will be ignored. That way we can get more accurate information of the part of the wave we are interested in at that moment.
    - Easy way to delay the beginning is for instance using some specific value of `param_status_out` as the trigger condition for Trigger mode

Using enumerations

The states are a bit hard to read from the waveform if they are shown only as bit vectors. There are at least two ways to make your life easier. For the waveform picture you will return, please use at least one of the below methods (preferably both if you can).

- Renaming the individual state signals
- Enumerations

Both methods work better if you have managed to make your FSM in a way that Vivado is able to infer it. Then you can just look at the synthesis reports and name/enumerate the states according to the state encoding shown in the report. The former method of the two is probably plain impossible if this is not the case.

- If you have a one-hot encoded state machine, you can just rename the individual state signals (each bit separately) from the right click menu
  - Only one signal should be high at a time, so you can just check which signal is high and look at the signal name to determine the current state

- Another way is to use enumerations for the whole bus. Basically you will be giving a bit vector value a human readable name
  - Right click the whole bus in the waveform and choose "Edit enumeration..."
  - Using the +-button, add as many entries as you have states
  - With binary radix it's easy to set all the values the same as the state encodings in the synthesis report. Give them corresponding names.
  - Once done, you can see all the states with given names in the waveform

Debugging using already compiled reference implementations

If there is a bug in your implementation that you are having some trouble locating, replacing a part of the system with previously compiled and verified reference blocks can help narrow down your search for the issue.
Note! If your design works perfectly, onnea! You don't need to perform these steps.
However, if you are still having some issues, reference files containing "node-level" netlist in Verilog format are provided. They are technology specific (Zynq-7000 family FPGA)

Files of the reference design (these should work):

- i2c_config.v (https://plus.tuni.fi/graderA/static/compce240-f2021/E10/i2c_config.v)
- synthesizer.v (https://plus.tuni.fi/graderA/static/compce240-f2021/E10/synthesizer.v)

The following lines explain how the files are added to the project:

- Look at the sources list and right click the file you want to replace. Click "Replace File".
- Replace VHDL files with corresponding .v files
- If the files show up in Non-module files etc. (e.g. not in Design sources), fix their file type. In case of .v files this should be *Verilog*.

- For safety, it might be a good idea to regenerate the output products
- The project is now ready for compilation

Return:

Obtain the files specified below and put them under E13 folder in your Git repository. Return them also by file return in the bottom of this page.

- Return the following files:
  - ILA waveform screen capture `ila.png`
    - Remeber to rename your state signals or use enumerations (or even better: both). Make sure that the use of them can be seen in the picture.
  - A list of found bugs in file `answer13.txt`. A brief description of each of them, how you fixed them, was it hard to find, and how the bug(s) could be avoided.

- If some of the system's VHDL files changed during the debugging, return the fixed files again
  - Otherwise you do not have to send any VHDL files

- **Additionally, every group demonstrates their own synthesizer implementation to an assistant!**

answer13.txt

Choose File No file chosen

ila.png

Choose File No file chosen

Submit with David Rama Jimeno Submit

Earned points

1 / 1

Exercise info

Assignment category
VHDL exercises

Your submissions
2 / 1000

Points required to pass
1

Deadline
Sunday, 28 April 2024, 23:59

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

Group size
1-2

Total number of submitters
49

« 16. Exercise 13 (/comp.ce.240/spring-202... 17. Bonus exercises » (/comp.ce.240/spring...