ECE4810J System-on-Chip (SoC) Design

Fall 2021

Lab #3 PYNQ Overlays

Due: 11:59pm Oct. 21st, 2021

Logistics:

- This lab is a team exercise.
- Please use the discussion board on Piazza for Q&A.
- All reports and code (if available) MUST be submitted to the assignment of Canvas.
- Internet usage is allowed and encouraged.
- No late submission is allowed for this lab.

1. Overview

In this lab, you will learn about PYNQ overlays. The goals of this lab are to:

- Load overlay, use overlay and create overlay

2. Loading an overlay

Overlays, or hardware libraries, are programmable/configurable FPGA designs that extend the user application from the Processing System of the Zynq into the Programmable Logic. Overlays can be used to accelerate a software application, or to customize the hardware platform for a particular application. For example, image processing is a typical application where the FPGAs can provide acceleration. A software programmer can use an overlay in a similar way to a software library to run some of the image processing functions (e.g. edge detect, thresholding etc.) on the FPGA fabric. Overlays can be loaded to the FPGA dynamically, as required, just like a software library. In this example, separate image processing functions could be implemented in different overlays and loaded from Python on demand.

PYNQ provides a Python interface to allow overlays in the PL to be controlled from Python running in the PS. FPGA design is a specialized task which requires hardware engineering knowledge and expertise. PYNQ overlays are created by hardware designers, and wrapped with this PYNQ Python API. Software developers can then use the Python interface to program and control specialized hardware overlays without needing to design an overlay themselves. This is analogous to software libraries created by expert developers which are then used by many other software developers working at the application level.

By default, an overlay (bitstream) called base is downloaded into the PL at boot time. The base overlay can be considered like a reference design for a board. New overlays can be installed or copied to the board and can be loaded into the PL as the system is running. An overlay usually includes:

- A bitstream to configure the FPGA fabric
- A Vivado design Tcl file to determine the available IP
- Python API that exposes the IPs as attributes

The PYNQ Overlay class can be used to load an overlay. An overlay is instantiated by specifying the name of the bitstream file. Instantiating the Overlay also downloads the bitstream by default and parses the Tcl file. Type: from pynq import Overlay overlay = Overlay("base.bit")

For the base overlay, we can use the existing BaseOverlay class; this class exposes the IPs available on the bitstream as attributes of this class.

[1]: from pynq.overlays.base import BaseOverlay base_overlay = BaseOverlay("base.bit")

Once an overlay has been instantiated, the help() method can be used to discover what is in an overlay about. The help information can be used to interact with the overlay. Note that if you try the following code on your own board, you may see different results depending on the version of PYNQ you are using, and which board you have.

[2]: help(base_overlay)

This will give a list of the IP and methods available as part of the overlay. From the help() print out above, it can be seen that in this case the overlay includes an leds instance, and from the report this is an AxiGPIO class:

""" leds : AxiGPIO

```
4-bit output GPIO for interacting with the green LEDs LD0-3
```

""" Running help() on the leds object will provide more information about the object including details of its API.

[3]: help(base_overlay.leds)

The API can be used to control the object. For example, the following cell will turn on LD0 on the board.

[4]: base_overlay.leds[0].toggle() Information about other IP can be found from the overlay instance in a similar way, as shown below.

[5]: help(base_overlay.video)

Play with the above steps and try to learn more about how to load an overlay. Please list out all the available overlays for the arty z-7 boards.

3. Partial Reconfiguration

From image v2.4, PYNQ supports partial bitstream reconfiguration. The partial bitstreams are managed by the *overlay* class. It is always recommended to use the *.hwh* file along with the *.bit* for the overlay class.

Preparing the Files

There are many ways to prepare the bitstreams. Users can choose to follow the project flow or the software flow to implement a partial reconfiguration Vivado project. For more information, please refer to the documentation page on partial reconfiguration.

After each reconfiguration, the PL status will update to reflect the changes on the bitstream, so that new drivers can be assigned to the new blocks available in the bitstream. To achieve this, users have to provide the metadata file (*.hwh* file) along with each full / partial bitstream. The *.hwh* file is typically located at: *<project_name>/<design_name>.srcs/sources_1/bd/<design_name>/hw_handoff/*.

Keep in mind that each partial bitstream need a *.hwh* file.

Loading Full Bitstream

It is straightforward to download a full bitstream. By default, the bitstream will be automatically downloaded onto the PL when users instantiate an overlay object.

```
from pynq import Overlay
overlay = Overlay("full_bistream.bit')
```

To download the full bitstream again:

```
overlay.download()
```

Note that no argument is provided if a full bitstream is to be downloaded.

Another thing to note, is that if the Vivado project is configured as a partial reconfiguration project, the *.hwh* file for the full bitstream will not contain any information inside a partial region, even if the full bitstream always has a default *Reconfiguration Module* (RM) implemented. Instead, the *.hwh* file only provides the information on the interfaces connecting to the partial region. So for the full bitstream, don't be surprised if you see an empty partial region in the *.hwh* file. The complete information on the partial regions are revealed by the *.hwh* files of the partial bitstreams, where each *.hwh* file reveals one possible internal organization of the partial region.

Loading Partial Bitstream

Typically, the partial regions are hierarchies in the block design of the bitstream. In an *overlay* object, the hierarchical blocks are exposed as attributes of the object; users have to set the partial region for the overlay before they can reconfigure it. In the following example, let's assume there is a hierarchical block called *block_0* in the design.

```
overlay.set_partial_region('block_0')
```

After the partial region is set, users can use the *download()* method for partial bitstreams. Note that an argument is now needed if a partial bitstream is to be downloaded.

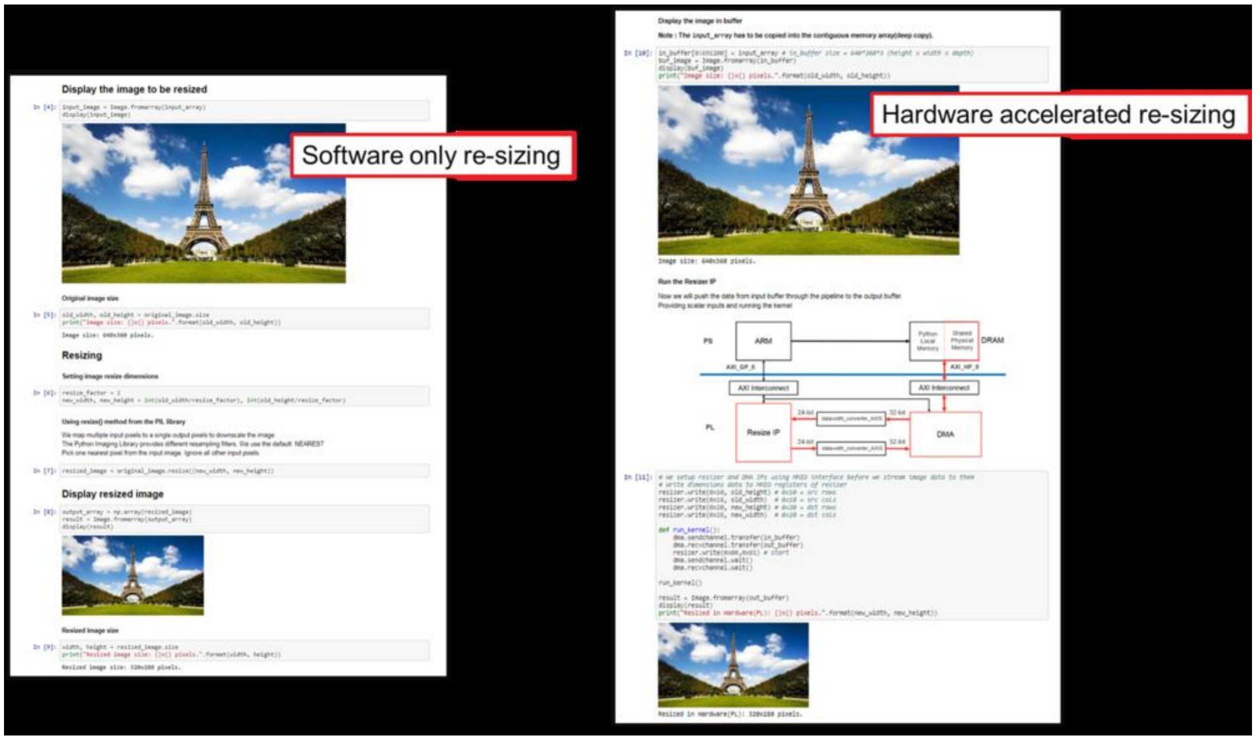
```
overlay.download('rm_0_partial.bit')
```

To load a different RM:

```
overlay.download('rm_1_partial.bit')
```

4. PYNQ-Helloworld

This assignment is based on the "Hello World" introduction application to the Xilinx PYNQ framework. https://github.com/Xilinx/PYNQ-HelloWorld



Display the image to be resized

```
In [4]: input_image = Image.fromarray(input_array)
        display(input_image)
```

Original image size

```
In [5]: old_width, old_height = original_image.size
        print("Image size: {}x{} pixels.".format(old_width, old_height))
Image size: 640x360 pixels.
```

Resizing

Setting image resize dimensions

```
In [6]: resize_factor = 2
        new_width, new_height = int(old_width/resize_factor), int(old_height/resize_factor)
```

Using resize() method from the PIL library

We map multiple input pixels to a single output pixels to downscale the image
The Python Imaging Library provides different resampling filters. We use the default: NEAREST
Pick one nearest pixel from the input image. Ignore all other input pixels.

```
In [7]: resized_image = original_image.resize((new_width, new_height))
```

Display resized image

```
In [8]: output_array = np.array(resized_image)
        result = Image.fromarray(output_array)
        display(result)
```

Resized image size

```
In [9]: width, height = resized_image.size
        print("Resized image size: {}x{} pixels.".format(width, height))
Resized image size: 320x180 pixels.
```

Display the image in buffer

Note : The input_array has to be copied into the contiguous memory array(deep copy).

```
In [10]: in_buffer[0:691200] = input_array # in_buffer size = 640*360*3 (height x width x depth)
         buf_image = Image.fromarray(in_buffer)
         display(buf_image)
         print("Image size: {}x{} pixels.".format(old_width, old_height))
Image size: 640x360 pixels.
```

Run the Resizer IP

Now we will push the data from input buffer through the pipeline to the output buffer.
Providing scalar inputs and running the kernel

```
In [11]: # We setup resizer and DMA IPs using MMIO interface before we stream image data to them
         # write dimensions data to MMIO registers of resizer
         resizer.write(0x10, old_height) # 0x10 = src rows
         resizer.write(0x18, old_width)  # 0x18 = src cols
         resizer.write(0x20, new_height) # 0x20 = dst rows
         resizer.write(0x28, new_width)  # 0x28 = dst cols

         def run_kernel():
             dma.sendchannel.transfer(in_buffer)
             dma.recvchannel.transfer(out_buffer)
             resizer.write(0x00,0x81) # start
             dma.sendchannel.wait()
             dma.recvchannel.wait()

         run_kernel()

         result = Image.fromarray(out_buffer)
         display(result)
         print("Resized in Hardware(PL): {}x{} pixels.".format(new_width, new_height))
Resized in Hardware(PL): 320x180 pixels.
```

Quick Start:

Open a terminal on your PYNQ board and run:

```
sudo pip3 install pynq-helloworld
```

Currently this repository is compatible with pynq package v2.5.1.

Go to your jupyter home folder (on edge boards, this is /home/xilinx/jupyter_notebooks), and run the following to deliver the notebooks:

```
pynq get-notebooks pynq-helloworld -p .
```

The -p option specifies the target folder location. Then you should be able to try the notebooks!

5. Creating Overlays

Please go through this tutorial and create an overlay on your own choice, use it and test that it works from jupyter notebook.

https://pynq.readthedocs.io/en/v1.3/10_creating_overlays.html#creating-overlays

References:

1. Arty Z7 Reference Manual https://digilent.com/reference/programmable-logic/arty-z7/reference-manual
2. https://xilinx-wiki.atlassian.net/wiki/spaces/A/overview
3. https://pynq.readthedocs.io/en/v2.4/pynq_overlays/partial_reconfiguration.html
4. https://www.hackster.io/90432/programming-python-on-zynq-fpga-ec4712
5. https://github.com/Xilinx/PYNQ-HelloWorld
6.

Acknowledgement:

- Sergiu Masanu, Mircea Stan (U of Virginia)
- Andreas Gerstlauer (U of Austin)
- Digilent
- Xilinx PYNQ project
- Xilinx University Program

Deliverables:

- Group Deliverables (Compile everything as a single pdf report besides the code):
  - Based on **part 2**, please submit
    - Please list out all the available overlays for the arty z-7 boards.
  - Based on **part 3**, please submit the following
    - Screenshot that shows full bit stream is loaded
    - Screenshot that shows partial bit stream is loaded
  - Based on **part 4**, please submit the following
    - A screen shot of the software vs. hardware accelerated resizing similar to what is shown in part 4, but with a different image that you choose
  - Based on **part 5**, please submit the following
    - The whole overlay package that you created
    - Screenshot that shows your overlay works
  - Answer the following questions (based on your understanding, feel free to use the internet):
    - Please list out any advantages of using overlay.
    - In general, when you choose software approach vs. hardware acceleration approach, what are the considerations? Any tradeoffs?
- Individual Deliverables
  - Complete the peer-evaluation form appended in the end of this document

Grading Policy

Factor	**Percentage**
Part 2	10%
Part 3	20%
Part 4	20%
Part 5	40%
Questions	10%

ECE4810J System-on-Chip (SoC) Design

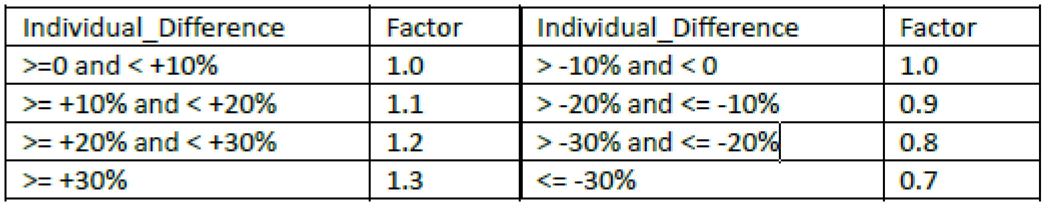
Fall 2021

Lab #3 Peer Evaluation Form

Each team member is required to provide a peer evaluation for the team effort of the lab. The score of the peer evaluation should be integers ranging between 0 to 5, inclusively, with 5 indicating the biggest contribution. A score should be given to each team member including yourself according the team member's contribution based on your observation. A brief description of specific contribution of each team member should also be provided. Note this form needs to be finished without discussing with your teammate, if all forms are found the same, then you have to justify how each member makes exactly the same contribution (through demonstration to the instructor).

Name	Level of contributions (0 – 5)	Description of contributions
(yourself)		
Team member 1		
Team member 2		
Team member 3		

Your lab grade is calculated based on the following:
Individual_Average = (sum of all team member's marks) / (size of the student team)
Group_Average = sum of Individual_Average of all team members / size of student team
Individual_Difference = Individual_Average / Group_Average – 1.0
Using the calculated Individual_Difference, we find a factor from the following lookup table.



Individual_Difference	Factor	Individual_Difference	Factor
>=0 and < +10%	1.0	> -10% and < 0	1.0
>= +10% and < +20%	1.1	> -20% and <= -10%	0.9
>= +20% and < +30%	1.2	> -30% and <= -20%	0.8
>= +30%	1.3	<= -30%	0.7

Your final lab grade is calculated by :
Final grade of lab = points for team effort * factor

Note that the final grade of the lab will not exceed the maximum points assigned to the project. If the peer-evaluation form is not submitted, then your individual_difference will be based on the available data (without yours) only, and your final score will be:

Final grade of lab = points for team effort * factor * 95%