INSTITUTE OF ENGINEERING AND TECHNOLOGY

DEVI AHILYA VISHWAVIDYALAYA, INDORE

Chip 2 Startup Project
Ministry of Electronics
& Information Technology
(MeitY)
Lab Manual
FPGA BOARDS- Pynq-Z2 & Urbana Board

Submitted by:

Akshay Pandey, ETC-A
Intern at
Chip2Startup Project
IET-DAVV,INDORE

Submitted to:

Dr. Vaibhav Neema Sir

TABLE OF CONTENTS

FPGA BOARDS

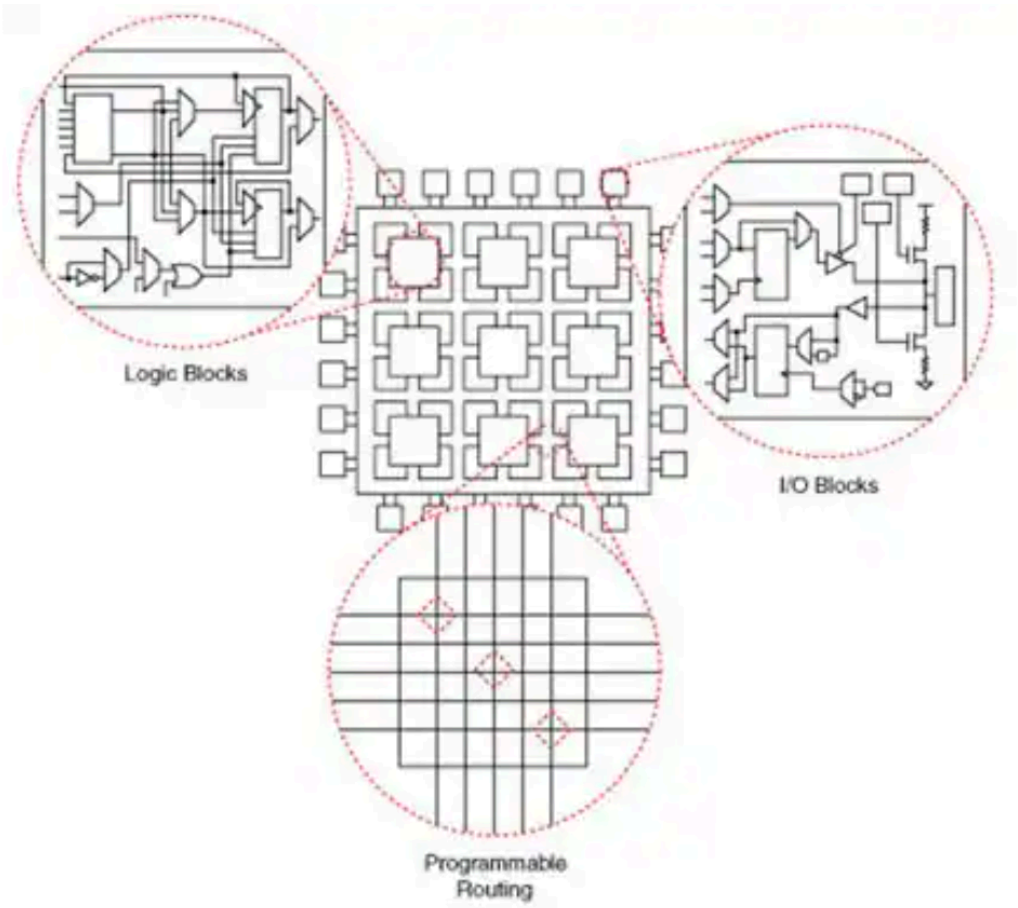
Programmable logic technologies, such as field-programmable gate arrays (FPGAs), are an essential component of any modern circuit designer's toolkit. With their expansive capabilities uniquely suited to a wide array of applications, FPGA development boards are ideal for solving many of the problems facing the rapidly evolving technology sector. The key benefits of programmable logic technologies include immense flexibility, cost savings over custom silicon, and increased performance through hardware parallelism.

Field Programmable Gate Array (FPGA) is an integrated circuit that consists of internal hardware blocks with user-programmable interconnects to customize operation for a specific application. The interconnects can readily be reprogrammed, allowing an FPGA to accommodate changes to a design or even support a new application during the lifetime of the part.

The FPGA has its roots in earlier devices such as programmable read-only memories (PROMs) and programmable logic devices (PLDs). These devices could be programmed either at the factory or in the field, but they used fuse technology (hence, the expression “burning a PROM”) and could not be changed once programmed. In contrast, FPGA stores its configuration information in a re-programmable medium such as static RAM (SRAM) or flash memory. FPGA manufacturers include **Intel**, **Lattice Semiconductor**, **Microchip Technology** and **Microsemi**.

FPGA architecture

A basic FPGA architecture (***Figure 1***) consists of thousands of fundamental elements called configurable logic blocks (CLBs) surrounded by a system of programmable interconnects, called a fabric, that routes signals between CLBs. Input/output (I/O) blocks interface between the FPGA and external devices.



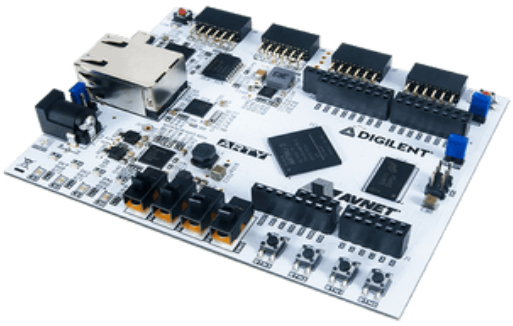
FPGA Applications

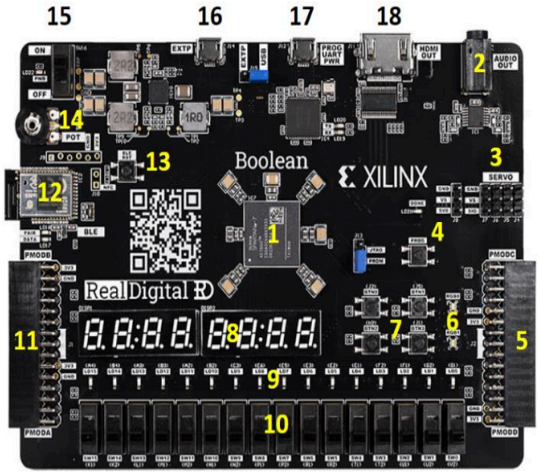
Due to their programmable nature, FPGAs are an ideal fit for many different markets. As the industry leader, AMD provides comprehensive solutions consisting of FPGA devices, advanced software, and configurable, ready-to-use IP cores for markets and applications such as:

- Aerospace & Defense - Radiation-tolerant FPGAs along with intellectual property for image processing, waveform generation, and partial reconfiguration for SDRs.
- ASIC Prototyping - ASIC prototyping with FPGAs enables fast and accurate SoC system modeling and verification of embedded software

- Automotive - Automotive silicon and IP solutions for gateway and driver assistance systems, comfort, convenience, and in-vehicle infotainment. - Learn how AMD FPGA's enable Automotive Systems
- Broadcast & Pro AV - Adapt to changing requirements faster and lengthen product life cycles with Broadcast Targeted Design Platforms and solutions for high-end professional broadcast systems.
- Consumer Electronics - Cost-effective solutions enabling next generation, full-featured consumer applications, such as converged handsets, digital flat panel displays, information appliances, home networking, and residential set top boxes.
- Data Center - Designed for high-bandwidth, low-latency servers, networking, and storage applications to bring higher value into cloud deployments.
- High Performance Computing and Data Storage - Solutions for Network Attached Storage (NAS), Storage Area Network (SAN), servers, and storage appliances.
- Industrial - AMD FPGAs and targeted design platforms for Industrial, Scientific and Medical (ISM) enable higher degrees of flexibility, faster time-to-market, and lower overall non-recurring engineering costs (NRE) for a wide range of applications such as industrial imaging and surveillance, industrial automation, and medical imaging equipment.
- Medical - For diagnostic, monitoring, and therapy applications, the Virtex FPGA and Spartan™ FPGA families can be used to meet a range of processing, display, and I/O interface requirements.
- Security - AMD offers solutions that meet the evolving needs of security applications, from access control to surveillance and safety systems.
- Video & Image Processing - AMD FPGAs and targeted design platforms enable higher degrees of flexibility, faster time-to-market, and lower overall non-recurring engineering costs (NRE) for a wide range of video and imaging applications.
-

- Wired Communications - End-to-end solutions for the Reprogrammable Networking Linecard Packet Processing, Framer/MAC, serial backplanes, and more
- Wireless Communications - RF, base band, connectivity, transport and networking solutions for wireless equipment, addressing standards such as WCDMA, HSDPA, WiMAX and others.





INTRODUCTION TO PYNQ-Z2

Python Productivity for Zynq (PYNQ) is an open-source project by AMD that simplifies the use of Adaptive Computing platforms. It offers a Jupyter-based framework with Python APIs for AMD Xilinx Adaptive Computing platforms.

The PYNQ-Z2 is a Zynq development board tailored for use with PYNQ.

PYNQ-Z2 Board FPGA Highlights:

Xilinx Zynq SoC: The PYNQ-Z2 features a Xilinx Zynq-7000 SoC (System on Chip). This combines a programmable FPGA (Field-Programmable Gate Array) with a dual-core ARM Cortex-A9 processor. This allows for hardware customization using the FPGA while leveraging the processing power of the ARM cores for software applications.

PYNQ Framework Support: The board is designed to work seamlessly with the PYNQ (Python Productivity for Zynq) framework. PYNQ simplifies working with Zynq SoCs by enabling FPGA programming using Python, a popular and easy-to-learn language.

Easy to Learn and Use: PYNQ makes the PYNQ-Z2 a good option for beginners or those new to FPGAs. Python simplifies the development process compared to traditional hardware description languages (HDLs) used for FPGA programming.

Rich Connectivity: The PYNQ-Z2 offers various connectivity options, including Ethernet, HDMI input/output, microphone input, audio output, Arduino and Raspberry Pi interfaces, Pmod headers, general-purpose GPIO pins, and more. This allows for integration with various peripherals and sensors for project development.

Extensibility: The Pmod headers and general-purpose I/O pins enable the PYNQ-Z2 to connect with various Pmods (Plug-in Modules for Open-source Development) and other expansion boards, increasing functionality for different applications.

PYNQ-Z2 FEATURES

ZYNQ XC7Z020-1CLG400C

- 650MHz dual-core Cortex-A9 processor
- DDR3 memory controller with 8 DMA channels and 4 High Performance AXI3 Slave ports
- High-bandwidth peripheral controllers: 1G Ethernet, USB 2.0, SDIO
- Low-bandwidth peripheral controller: SPI, UART, CAN, I2C
- Programmable from JTAG, Quad-SPI flash, and MicroSD card
- **Programmable logic equivalent to Artix-7 FPGA**
  - 13,300 logic slices, each with four 6-input LUTs and 8 flip-flops
  - 630 KB of fast block RAM
  - 4 clock management tiles, each with a phase locked loop (PLL) and mixed-mode clock manager (MMCM)
  - 220 DSP slices
  - On-chip analog-to-digital converter (XADC)

Memory

- 512MB DDR3 with 16-bit bus @ 1050Mbps
- 16MB Quad-SPI Flash with factory programmed 48-bit globally unique EUI-48/64™ compatible identifier
- MicroSD slot

Power

- Powered from USB or 7V-15V external power source

USB and Ethernet

- Gigabit Ethernet PHY
- Micro USB-JTAG Programming circuitry
- Micro USB-UART bridge
- USB 2.0 OTG PHY (supports host only)

Audio and Video

- HDMI sink port (input)
- HDMI source port (output)
- I2S interface with 24bit DAC with 3.5mm TRRS jack
- Line-in with 3.5mm jack

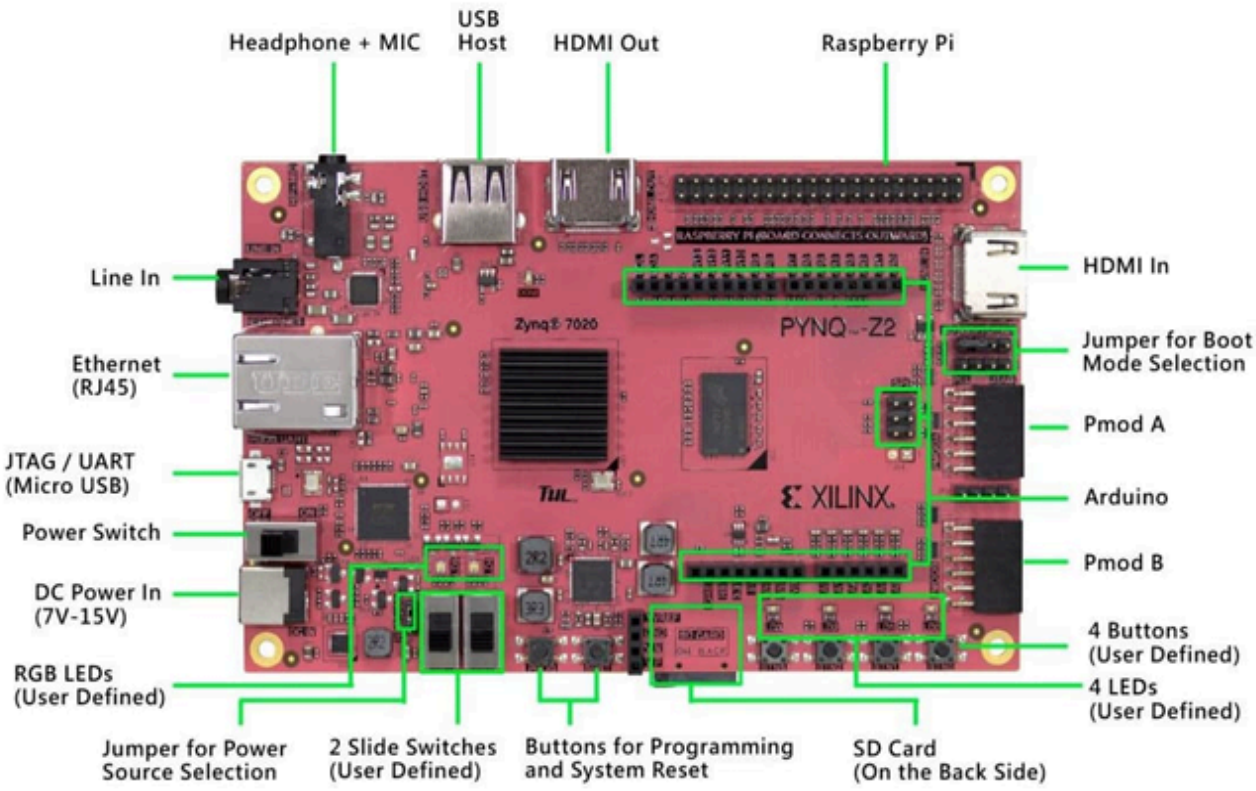
Switches, Push-buttons and LEDs

- 4 push-buttons
- 2 slide switches
- 4 LEDs
- 2 RGB LEDs

Expansion Connectors

- Two standard Pmod ports
  - 16 Total FPGA I/O (8 shared pins with Raspberry Pi connector)
- Arduino Shield connector
  - 24 Total FPGA I/O
  - 6 Single-ended 0-3.3V Analog inputs to XADC
- Raspberry Pi connector
  - 28 Total FPGA I/O (8 shared pins with Pmod A port)

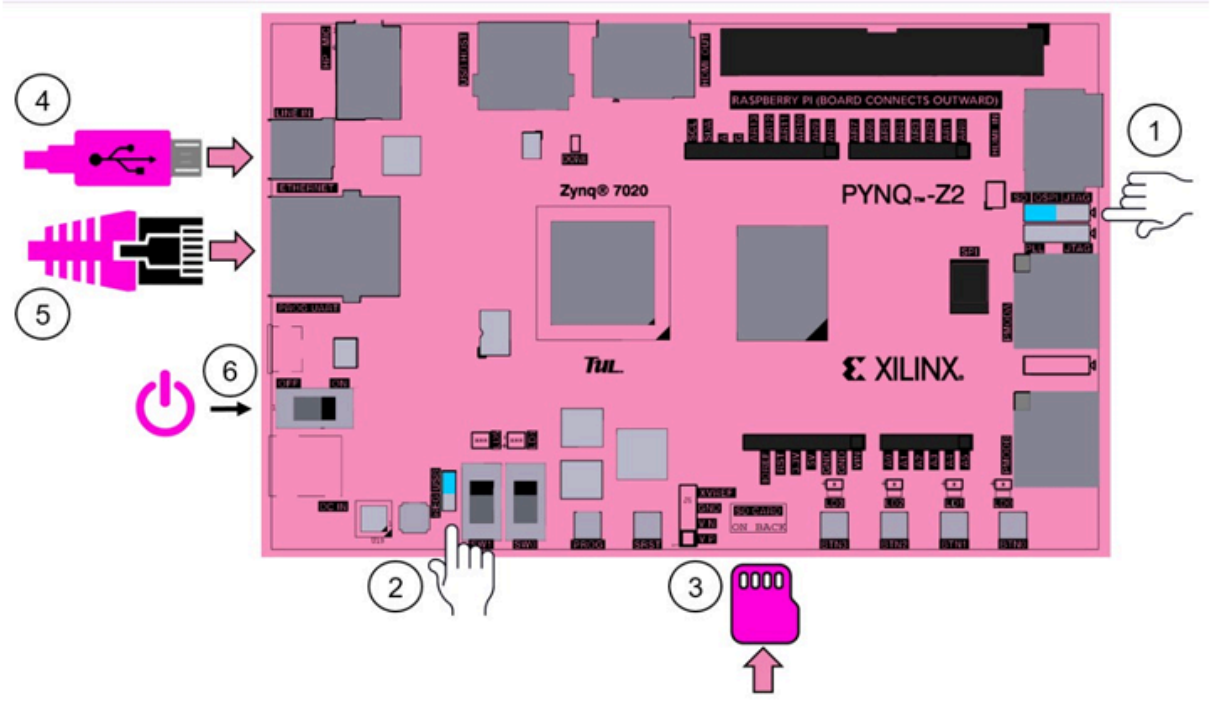
BOARD LAYOUT



PYNQ-Z2 SETUP GUIDE

Prerequisites

- PYNQ-Z2 board
- Computer with compatible browser (Chrome, Mozilla Firefox, Safari, Opera)
- Ethernet cable
- Micro USB cable
- Micro-SD card with preloaded image, or blank card (Minimum 8GB recommended)



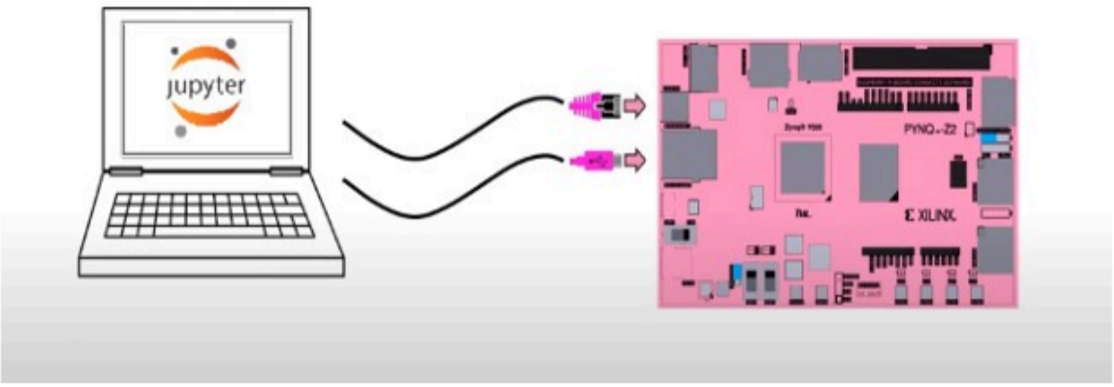
Once your board is setup, you need to connect to it to start using Jupyter notebook.

Ethernet

If available, you should connect your board to a network or router with Internet access. This will allow you to update your board and easily install new packages.

There are two ways to connect board to Network, use only one at a time:

1. Connect to a Computer (Advisable)



You will need to have an Ethernet port available on your computer, and you will need to have permissions to configure your network interface. With a direct connection, you will be able to use PYNQ, but unless you can bridge the Ethernet connection to the board to an Internet connection on your computer, your board will not have Internet access. You will be unable to update or load new packages without Internet access.

Connect directly to a computer (Static IP):

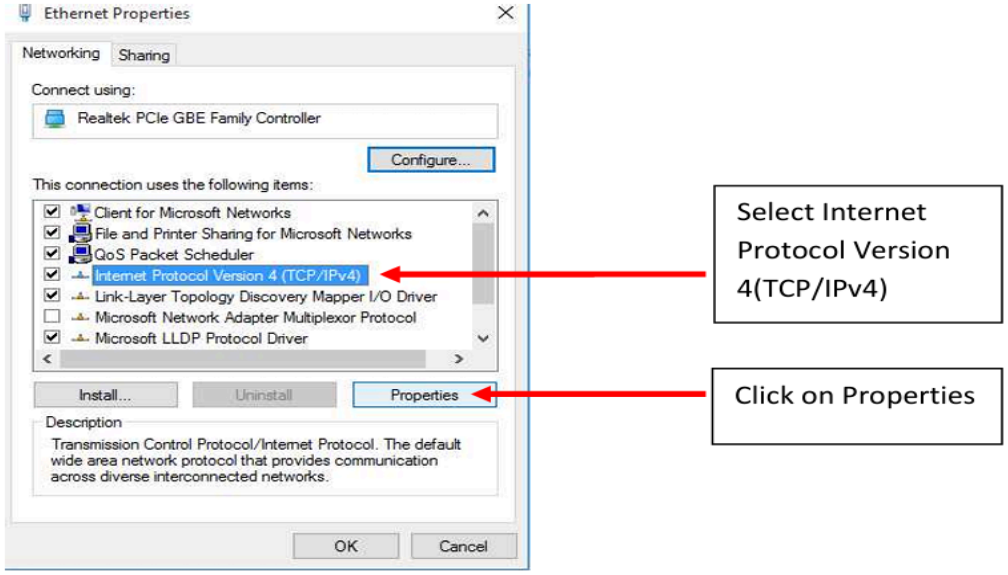
Step 1: Assign a static IP address

You need to set the IP address of your laptop/pc to be in the same range as the board. e.g. if the board is 192.168.2.99, the laptop/PC can be 192.168.2.x where x is 0-255 (excluding 99, as this is already taken by the board).

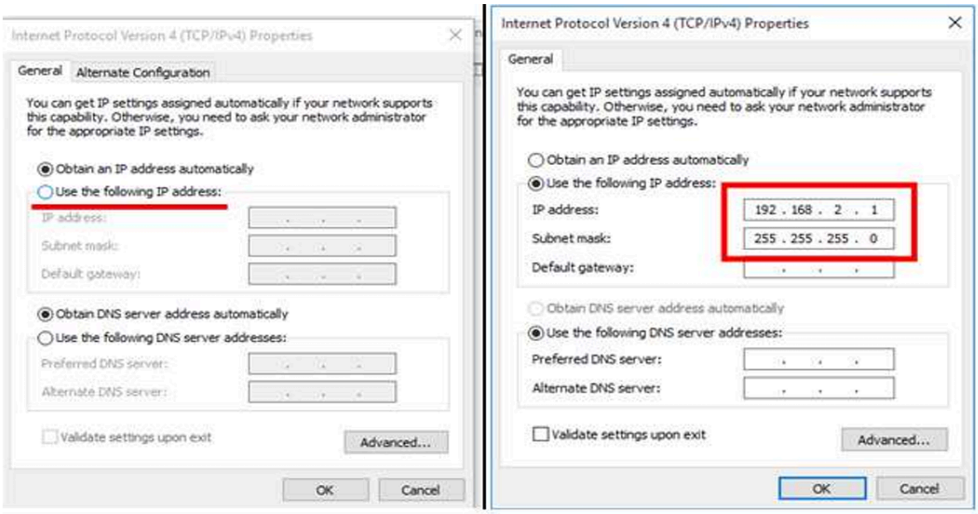
You should record your original settings, in case you need to revert to them when finished using PYNQ.

Go to Control Panel -> Network and Internet -> Network Connections
Double click on the network interface to open it, and click on Properties.

Select Internet Protocol Version 4(TCP/IPv4) and click Properties



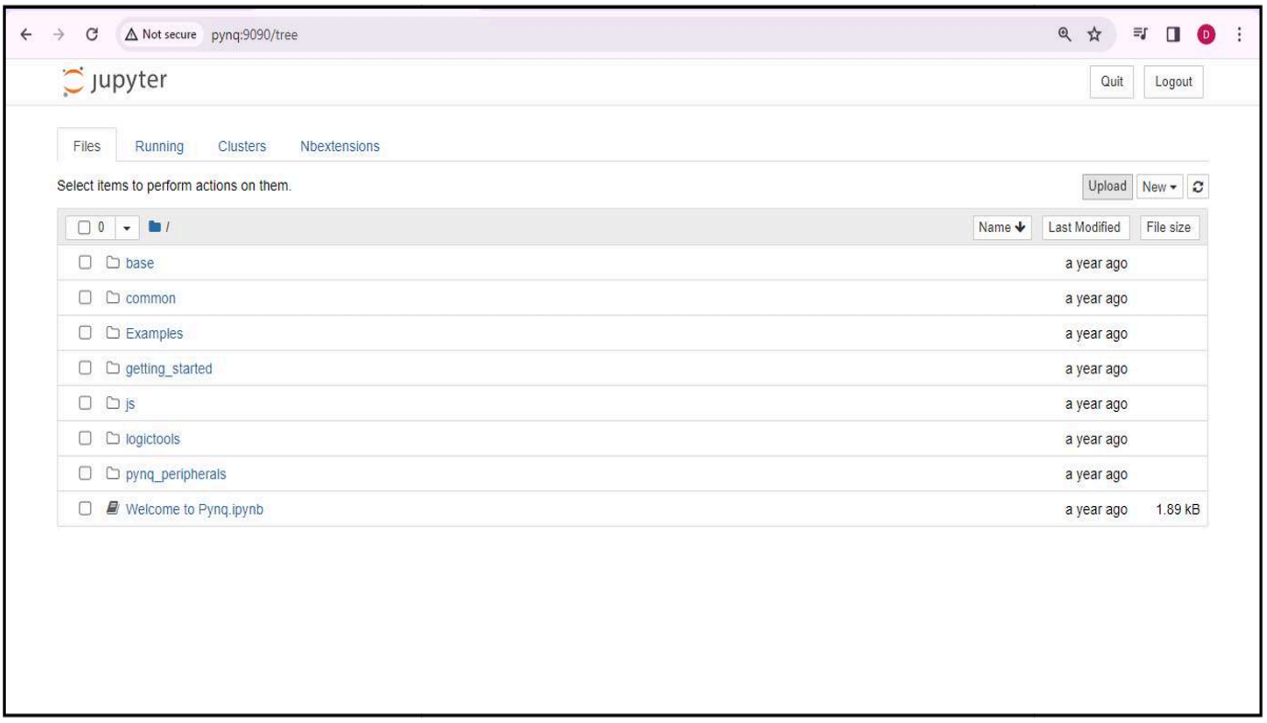
Select use the following IP address.
Set the IP address to 192.168.2.1 (or any other address in the same range as the board)
Set the subnet mask to 255.255.255.0 and click **OK**



Now the connect the board to the ethernet port of your laptop

Browse to http://192.168.2.99:9090/

A new tab will get open:



Jupyter Notebook

Both hardware and online setup is done.

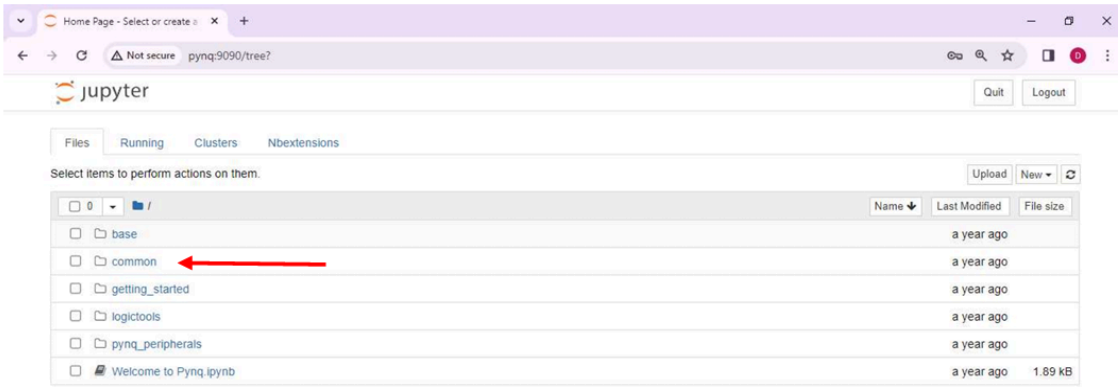
EXPERIMENT-1

OBJECTIVE:-
To connect USB webcam to the board and do some operations on the picture captured.

HARDWARE REQUIRED:-
USB camera

THEORY:-
USB webcams are devices used for capturing video data and transmitting it to a host device via USB interface. When connected to a PYNQ-Z2 board, which features an FPGA and an ARM processor, USB webcams can be utilized for advanced image processing and computer vision tasks. The FPGA enables real-time processing of video data, while the ARM processor handles higher-level tasks. By leveraging the PYNQ framework and Python programming, developers can easily interface USB webcams with the PYNQ-Z2 board, opening up possibilities for applications such as real-time object detection, surveillance systems, multimedia applications, and educational projects.

PROCEDURE:-
1. Set up the board as instructed and launch Jupyter Notebook.
2. Connect USB webcam to the USB port of the board and place the camera somewhere.
3. Now, Go to common > usb_webcam.ipynb.
4. A notebook will open. Read all the steps and run each cell one by one.
5. Observe the output from each cell.


Home Page - Select or create
Not secure
pynq:9090/tree?
jupyter
Quit
Logout
Files
Running
Clusters
Nbextensions
Select items to perform actions on them.
Upload
New
Name
Last Modified
File size
base
a year ago
common
a year ago
getting_started
a year ago
logictools
a year ago
pynq_peripherals
a year ago
Welcome to Pynq.ipynb
a year ago
1.89 kB


Not secure
pynq:9090/tree/common
jupyter
Quit
Logout
Files
Running
Clusters
Nbextensions
Select items to perform actions on them.
Upload
New
/ common
Name
Last Modified
File size
seconds ago
data
a year ago
overlay_download.ipynb
a year ago
14.7 kB
programming_pybind11.ipynb
a year ago
8.04 kB
python_random.ipynb
a year ago
1.39 MB
python_retrieving_shell.ipynb
a year ago
4.65 kB
usb_webcam.ipynb
a year ago
519 kB
wifi.ipynb
a year ago
5.34 kB
zynq_clocks.ipynb
a year ago
5.39 kB

RESULT:-

EXPERIMENT-2

OBJECTIVE:-
FACE DETECTION using Pynq-z2 .

HARDWARE REQUIRED:-
USB camera

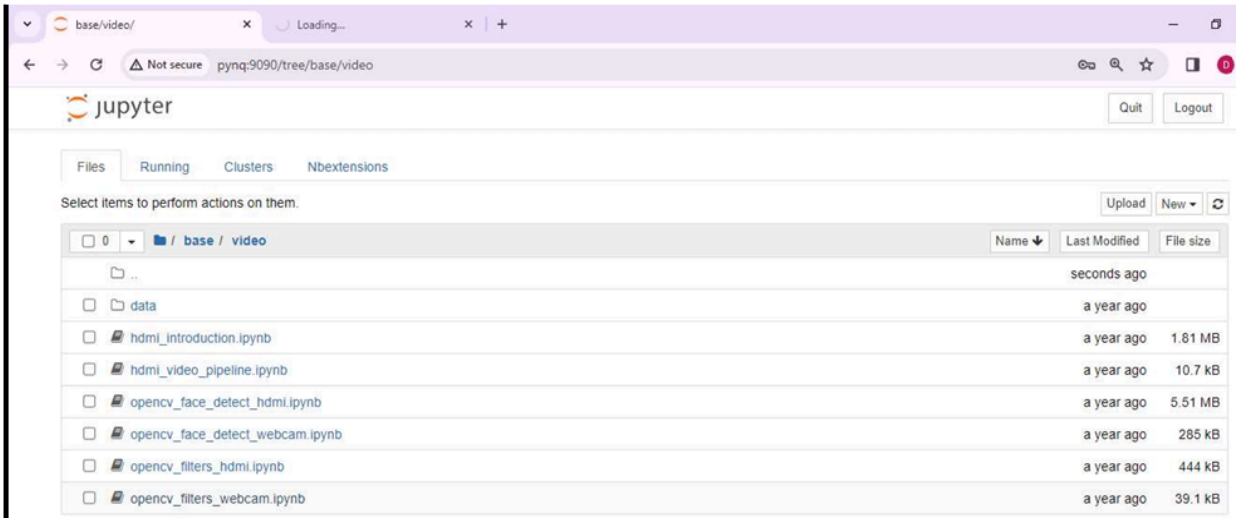
EXPERIMENT OVERVIEW:-
In this experiment, a webcamera is connected to the USB port of the board. The webcam captures the runtime images to which open cv face detection will be applied. The example detects and locates face and eyes from the image.

THEORY:-
Face recognition involves several key steps:
Face Detection: Identifying and locating faces within images or video frames using techniques like Haar cascades or deep learning models.
1. Preprocessing: Preparing detected faces by converting them to grayscale, aligning them, and normalizing pixel intensities for better processing.
2. Feature Extraction: Extracting distinctive features from faces, such as Eigenfaces, Fisherfaces, or Local Binary Patterns Histograms (LBPH), to represent them as feature vectors.
3. Recognition: Comparing the extracted features of detected faces with known individuals' features stored in a database or training set to determine identity.
4. Classification: Using algorithms like Nearest Neighbor, Support Vector Machines (SVM), or Deep Learning to classify detected faces based on similarity or distance metrics
5. Real-Time Application: Implementing optimized algo from cameras or webcams for real time face recognition applications.

PROCEDURE:
1. Set up the board as instructed and launch Jupyter Notebook.
2. Connect USB webcam to the USB port of the board and place the camera somewhere.
3. Now, Go to base > video > opencv_face_detect_webcam.ipynb.
4. Run each cells line by line.

5. Webcam will take input as your face and it fill further detect your face.
6. Observe the output (detects and locates face and eyes from the image).

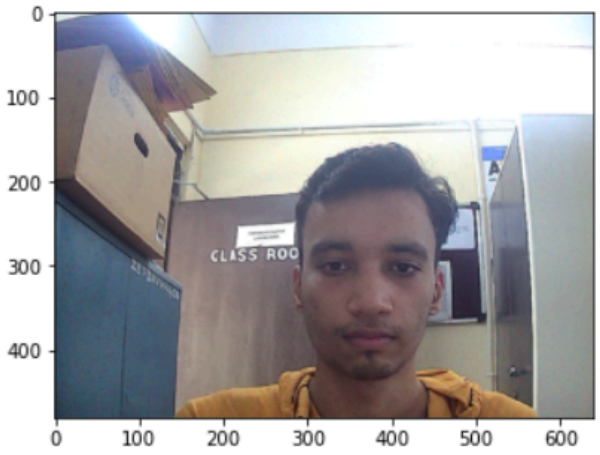
Click on opencv_face_detect_webcam.ipynb



INPUT:-

Step 4: Now use matplotlib to show image inside notebook

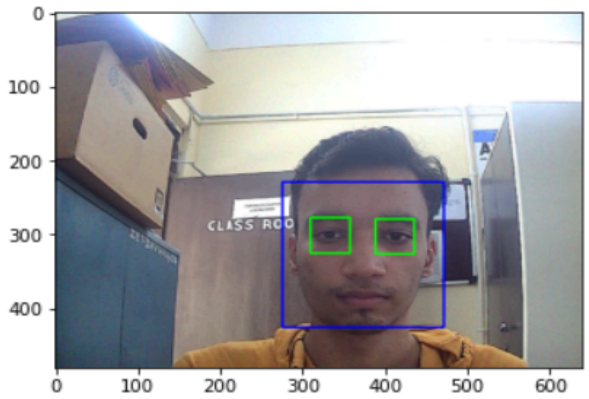
In [16]:
```
# Output webcam image as JPEG
%matplotlib inline
from matplotlib import pyplot as plt
import numpy as np
plt.imshow(frame_vga[:,:,[2,1,0]])
plt.show()
```



OUTPUT:-

Step 7: Now use matplotlib to show image inside notebook

In [19]:
```
# Output OpenCV results via matplotlib
%matplotlib inline
from matplotlib import pyplot as plt
import numpy as np
plt.imshow(np_frame[:,:,[2,1,0]])
plt.show()
```



EXPERIMENT-3

OBJECTIVE:-
EDGE DETECTION .

HARDWARE REQUIRED:-
USB camera

EXPERIMENT OVERVIEW:-
In this project, the USB camera is connected with the PYNQ USB Port. Computer Vision-based edge detection algorithm like Canny Edge algorithm is applied to the real-time video using Python Programming. The result will be displayed on the monitor with edge detection applied.

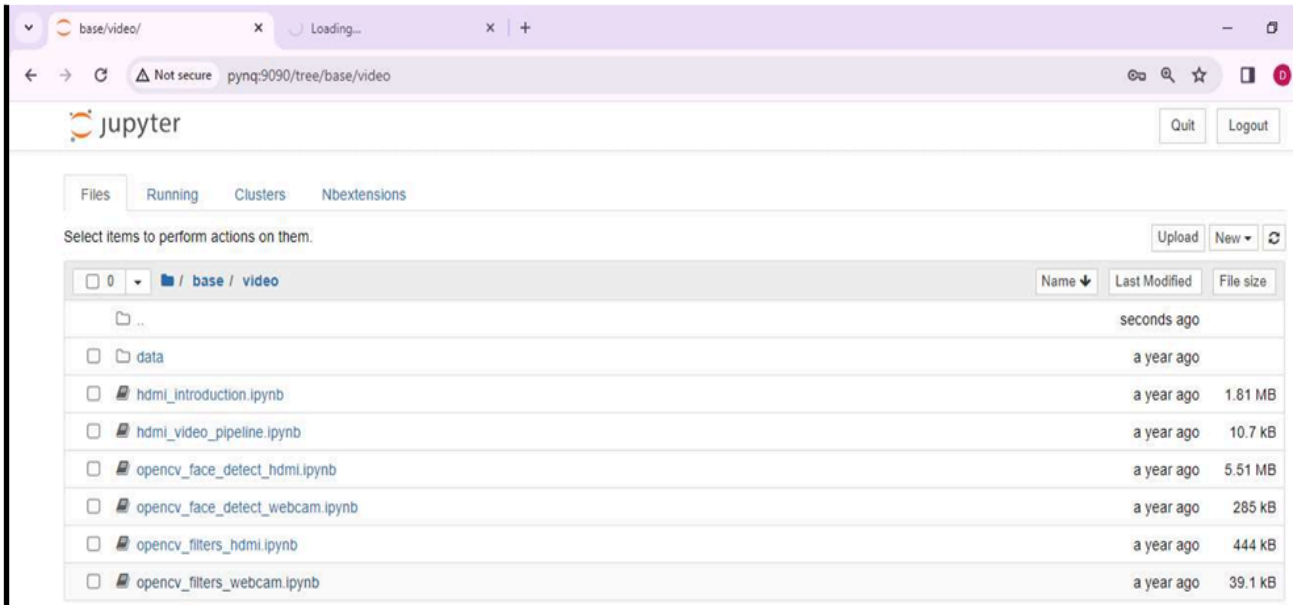
THEORY:
The PYNQ-Z2 board facilitates real-time image processing using the OpenCV library. By interfacing with a USB webcam, the board captures live video input, which is then processed using OpenCV filters. These filters include edge detection algorithms like Laplacian

Edge Detection: Edge detection is a fundamental technique in image processing aimed at identifying boundaries within images. Edges typically represent significant changes in intensity or color within an image and can correspond to object boundaries or other important features. Edge detection algorithms aim to locate these boundaries by identifying regions of rapid intensity change.

PROCEDURE: 1. Set up the board as instructed and launch Jupyter Notebook.
2. Connect USB webcam to the USB port of the board and place the camera somewhere.
3. Now, Go to base > video > opencv_filters_webcam.ipynb
4. Run each cells line by line.
5. Webcam will take input as your face and fill further detect your face.
6. Observe the output (EDGE DETECTION).


Home Page - Select or create
Not secure
pynq:9090/tree?
jupyter
Quit
Logout
Files
Running
Clusters
Nbextensions
Select items to perform actions on them.
Upload
New
/
Name
Last Modified
File size
base
a year ago
common
a year ago
getting_started
a year ago
logictools
a year ago
pynq_peripherals
a year ago
Welcome to Pynq.ipynb
a year ago
1.89 kB


jupyter
Quit
Logout
Files
Running
Clusters
Nbextensions
Select items to perform actions on them.
Upload
New
/ base
Name
Last Modified
File size
..
seconds ago
arduino
a year ago
audio
a year ago
board
a year ago
microblaze
a year ago
pmod
a year ago
rpi
a year ago
trace
a year ago
video
a year ago


base/video/
Loading...
Not secure
pynq:9090/tree/base/video
jupyter
Quit
Logout
Files
Running
Clusters
Nbextensions
Select items to perform actions on them.
Upload
New
/ base / video
Name
Last Modified
File size
..
seconds ago
data
a year ago
hdmi_introduction.ipynb
a year ago
1.81 MB
hdmi_video_pipeline.ipynb
a year ago
10.7 kB
opencv_face_detect_hdmi.ipynb
a year ago
5.51 MB
opencv_face_detect_webcam.ipynb
a year ago
285 kB
opencv_filters_hdmi.ipynb
a year ago
444 kB
opencv_filters_webcam.ipynb
a year ago
39.1 kB

RESULT:-

Step 6: Show results

Now use matplotlib to show filtered webcam input inside notebook.

In [8]:
```
%matplotlib inline
from matplotlib import pyplot as plt
import numpy as np

frame_canny = cv2.Canny(frame_webcam, 100, 110)
plt.figure(1, figsize=(10, 10))
frame_vga = np.zeros((480,640,3)).astype(np.uint8)
frame_vga[:,:,0] = frame_canny
frame_vga[:,:,1] = frame_canny
frame_vga[:,:,2] = frame_canny
plt.imshow(frame_vga)
plt.show()
```

EXPERIMENT-4

OBJECTIVE:-

ROAD SIGN RECOGNITION USING BNN with Pynq board.

HARDWARE REQUIRED:-

USB camera

EXPERIMENT OVERVIEW:-

This example recognizes input image with a binarized neural network. It uses a German road-sign dataset and can classify classes of road signs. The example has successfully identified the road signs from the given input images.

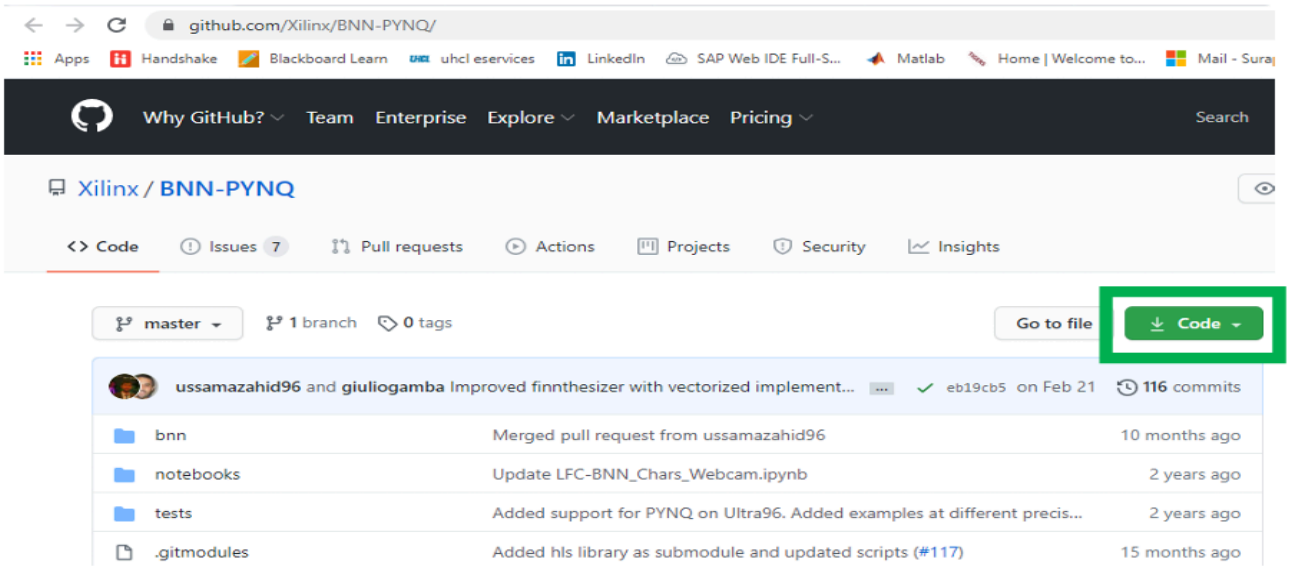
THEORY:-

The experiment focuses on the application of Bayesian Neural Networks (BNNs) for road sign recognition, deployed on a PYNQ board. This integration leverages the strengths of BNNs in providing probabilistic outputs and the flexibility of the PYNQ board, a platform based on the Xilinx Zynq System-on-Chip (SoC), to achieve efficient and robust image classification.

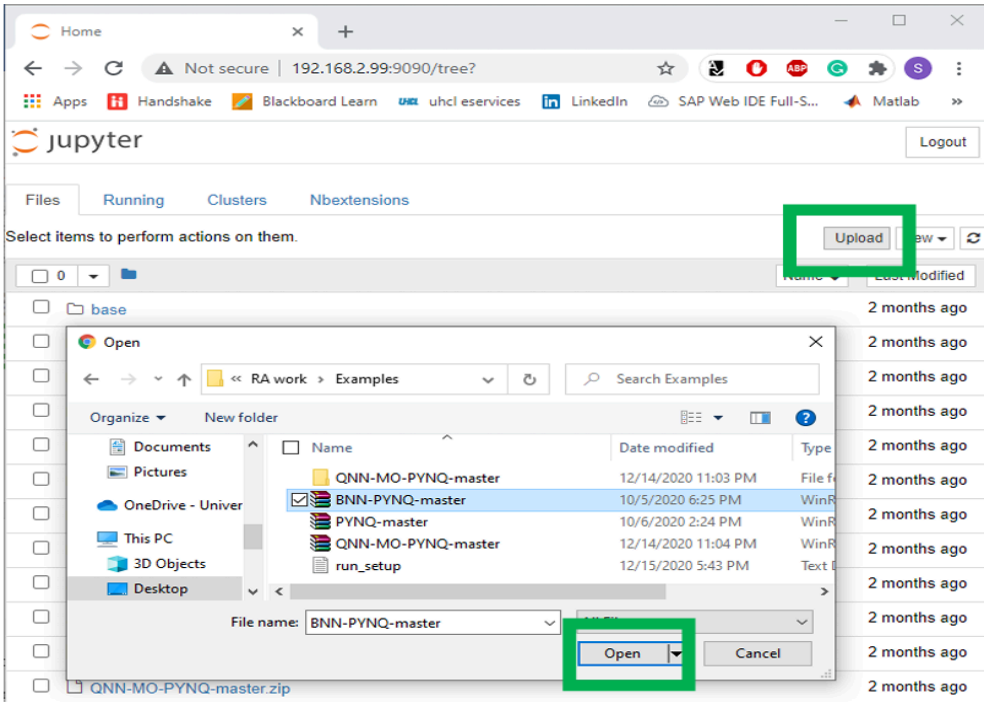
Road sign recognition is a critical component in autonomous driving and advanced driver-assistance systems (ADAS). The objective is to accurately identify traffic signs in real-time, ensuring safe and reliable vehicle operation. This task requires high accuracy and low latency to process video frames quickly and make prompt decisions.

PROCEDURE:-

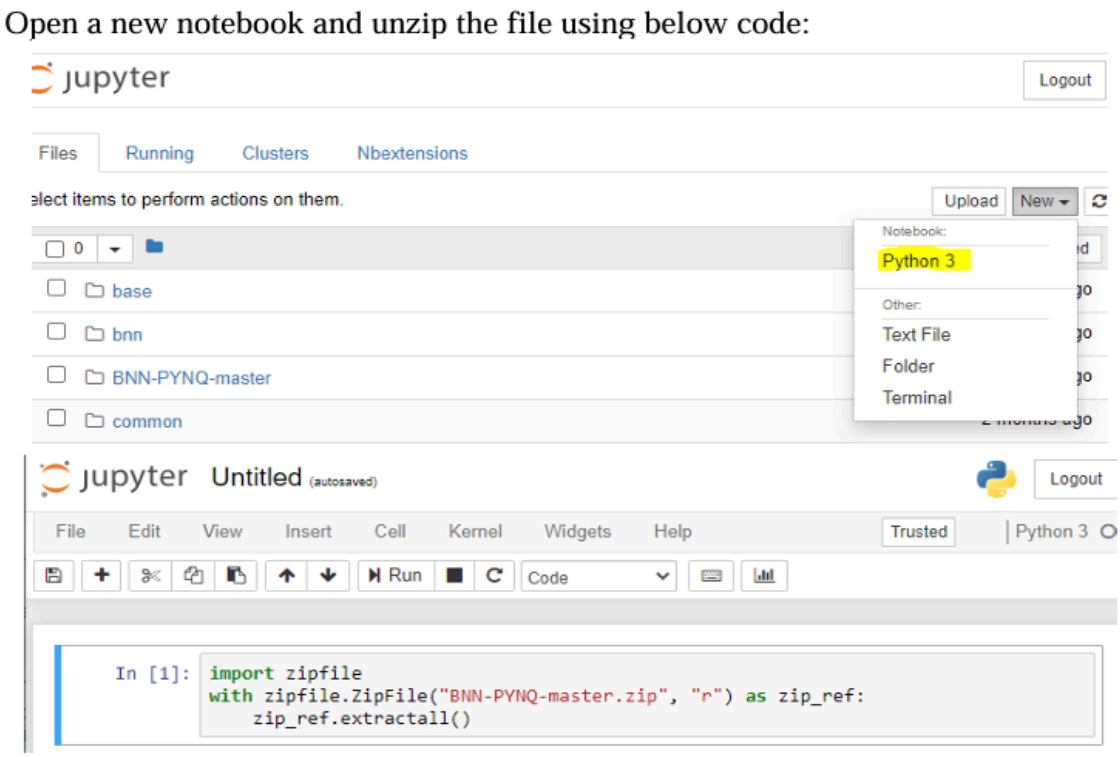
1.For this experiment, we need to download XILINX/BNN-PYNQ SOURCE CODE FROM GITHUB.



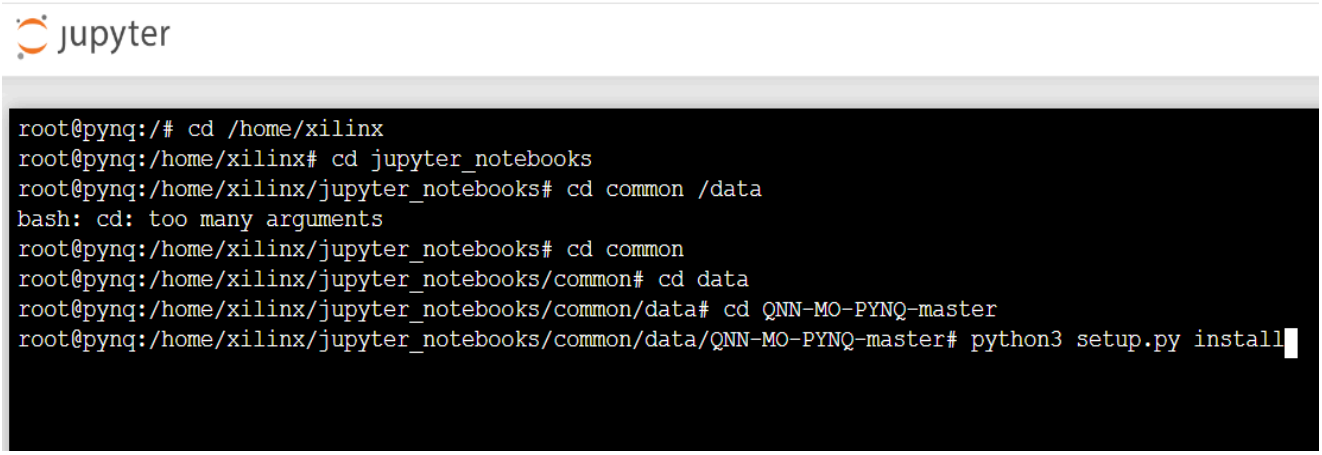
2. Upload the code on jupyter notebook by selecting the downloaded zip folder.



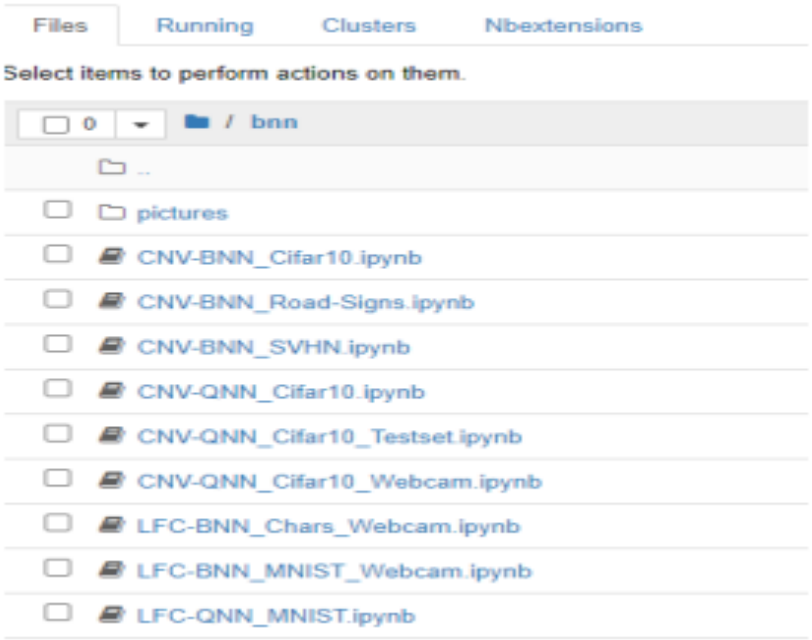
3. Open a new notebook and unzip the the file using following code:-



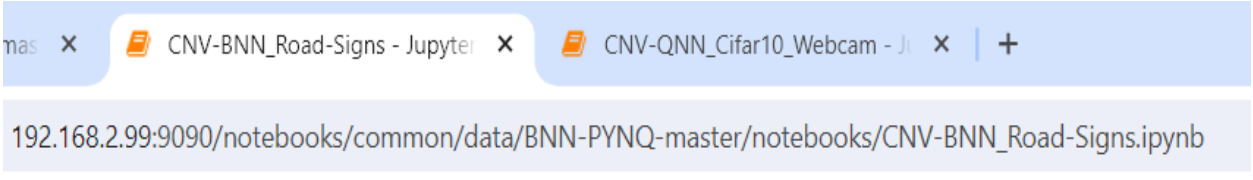
4. Open new terminal in notebook and run the setup.py file using “python3 setup.py install” command. Replace QNN with BNN in below code



5. This will install the project package on board and create a directory in the jupyter home area



6. Now go open CNV- BNN Road-signs python notebook by using path:-



7. The code will open like below:-

BNN on Pynq

This notebook covers how to use Binary Neural Networks on Pynq. It shows an example of image recognition with a binarized neural network inspired at VGG-16, featuring 6 convolutional layers, 3 max pool layers and 3 fully connected layers

1. Instantiate a Classifier

Creating a classifier will automatically download the correct bitstream onto the device and load the weights trained on the specified dataset. By default there are three sets of weights available for the BNN version of the CNV network using 1 bit weights and 1 activation (W1A1) - this example uses the German Road Sign dataset.

In [1]:
```
import bnn
print(bnn.available_params(bnn.NETWORK_CNVW1A1))

classifier = bnn.CnvClassifier(bnn.NETWORK_CNVW1A1, 'road-signs', bnn.RUNTIME_HW)
```

```
['cifar10', 'road-signs', 'streetview']
Setting network weights and thresholds in accelerator...
```

8. To input images into the code we use:-

3. Open images to be classified

The images that we want to classify are loaded and shown to the user

```
from PIL import Image
import numpy as np
from os import listdir
from os.path import isfile, join
from IPython.display import display

imgList = [f for f in listdir("/home/xilinx/jupyter_notebooks/bnn/pictures/road_signs/") if isfile(join("/h

images = []

for imgFile in imgList:
    img = Image.open("/home/xilinx/jupyter_notebooks/bnn/pictures/road_signs/" + imgFile)
    images.append(img)
    img.thumbnail((64, 64), Image.ANTIALIAS)
    display(img)
```

Here the images used are:-

9. Now to locate signs we will use:-

6. Locate objects within a scene

This example is going to create an array of images from a single input image, tiling the image to try and locate an object. This image shows a road intersection and we're aiming at finding the stop sign.

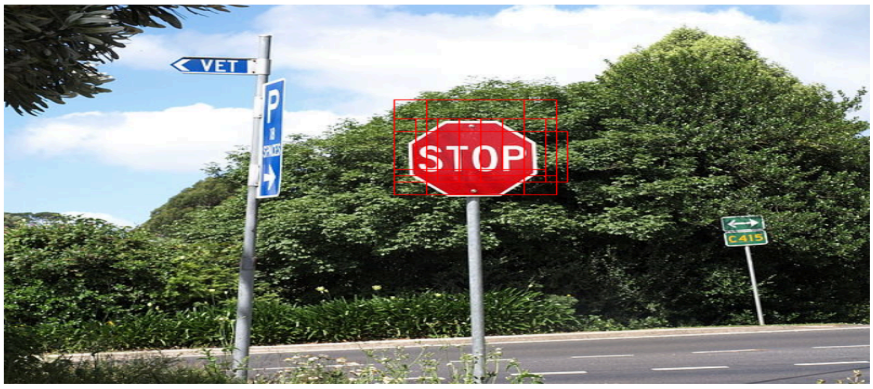
In [6]:
```
from PIL import Image
image_file = "/home/xilinx/jupyter_notebooks/bnn/pictures/street_with_stop.JPG"
im = Image.open(image_file)
im
```

Out[6]:

10. Now the image in which the BNN identified the stop signal is shown:-

```
results = classifier.classify_images(images)
end = results == 14
indicies = []
indicies = end.nonzero()[0]
from PIL import ImageDraw
im2 = Image.open(image_file)
draw2 = ImageDraw.Draw(im2)
for i in indicies:
    draw2.rectangle(bounds[i], outline='red')

im2
```

RESULT:-



EXPERIMENT-5

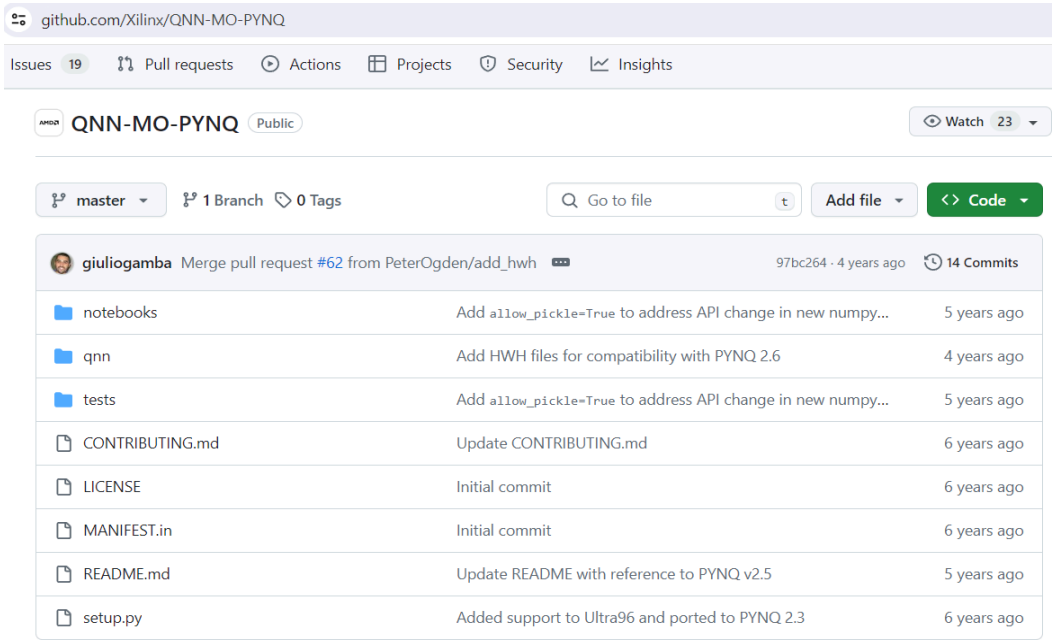
Objective:-
Image detection using QNN with Pynq board.

HARDWARE REQUIRED:-
USB camera
Experiment Overview:-
We will detect images of following objects by putting it in front of webcamera:- [“Airplane”, “Automobile”, “Bird ”, “Cat”, “Deer”, “Dog”, “Frog” , “Horse”,”Ship”, “Truck”] and a graphical analysis or report will come to detect object.

Theory:-
In this project, a kind of Tiny-Yolo model was developed based on the Darknet network to recognize multiple objects. Each input image is associated with a single neural network.

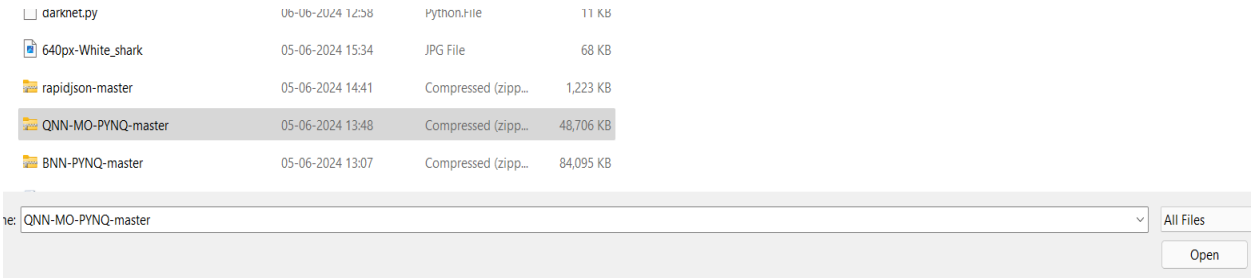
Procedure:-

1. For this experiment, we need to download XILINX/QNN-MO-PYNQ SOURCE CODE FROM GITHUB.

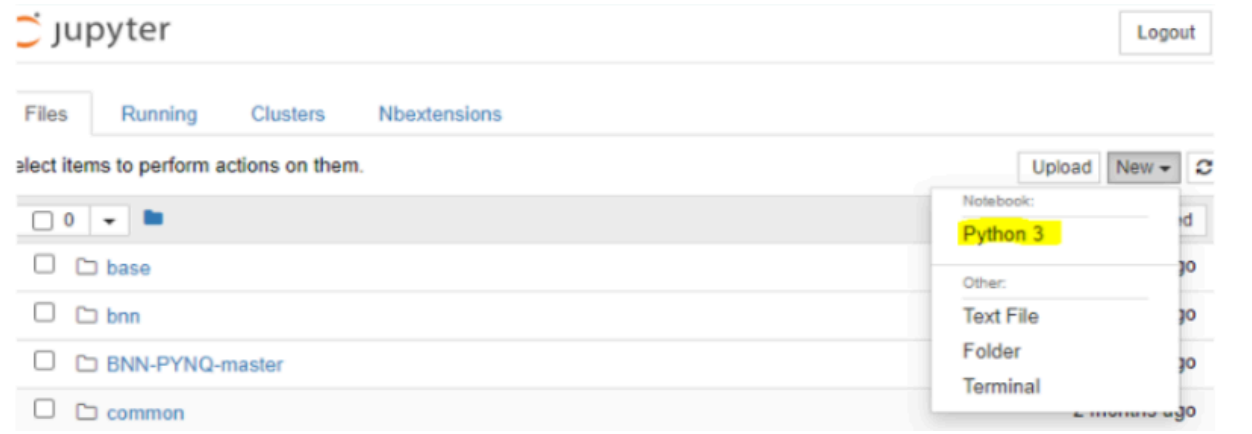


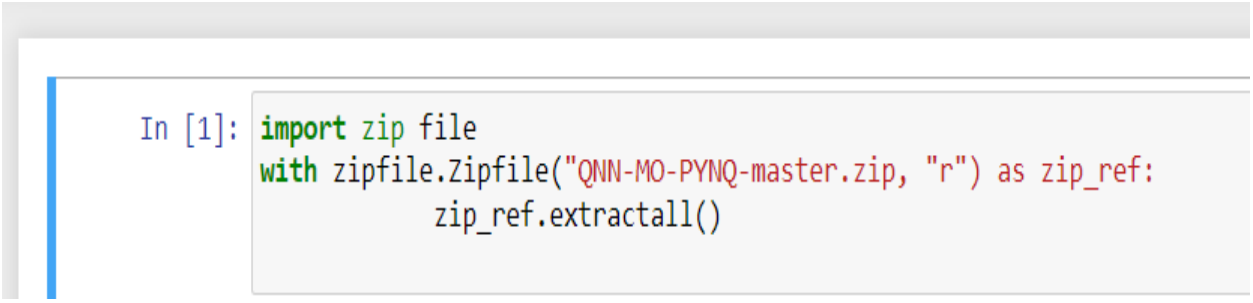
Go to code and download zip file.

2. Upload the code on jupyter notebook by selecting the downloaded zip folder.



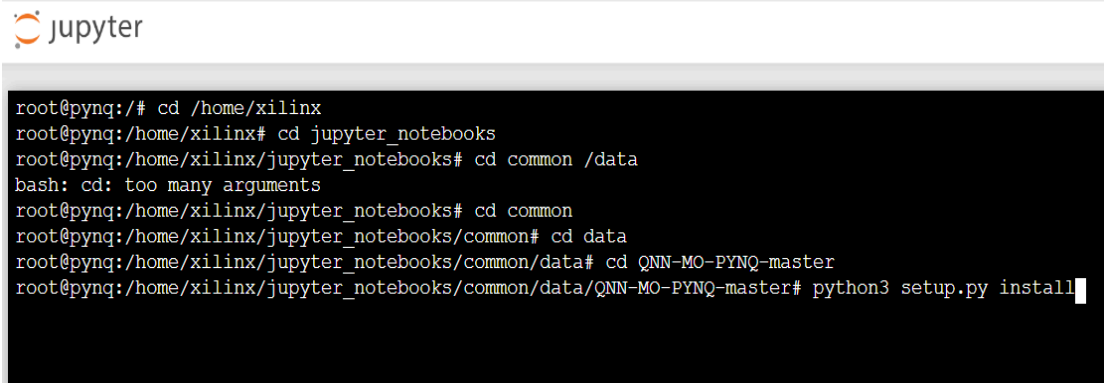
3. Open a new notebook and unzip the the file using following code:-



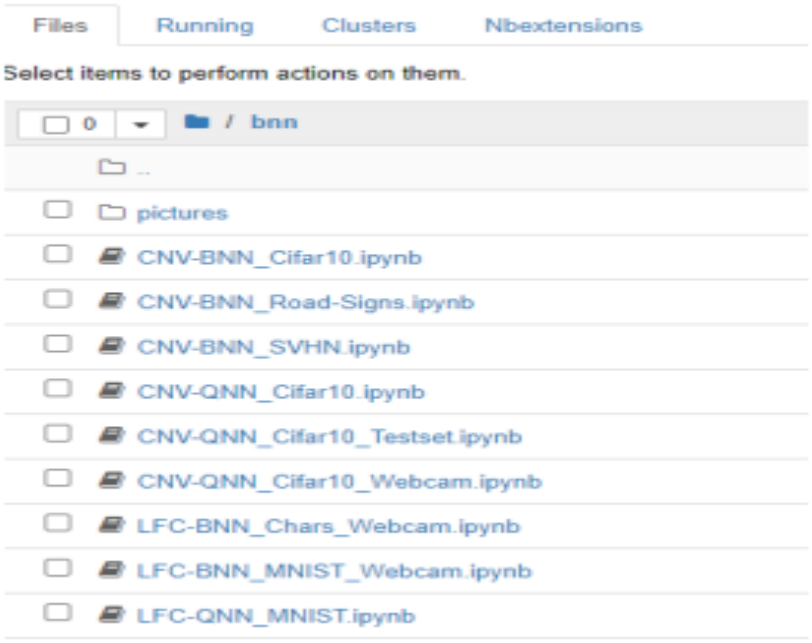


```
In [1]: import zip file
        with zipfile.Zipfile("QNN-MO-PYNQ-master.zip, "r") as zip_ref:
                    zip_ref.extractall()
```

4. Go to terminal and write:-



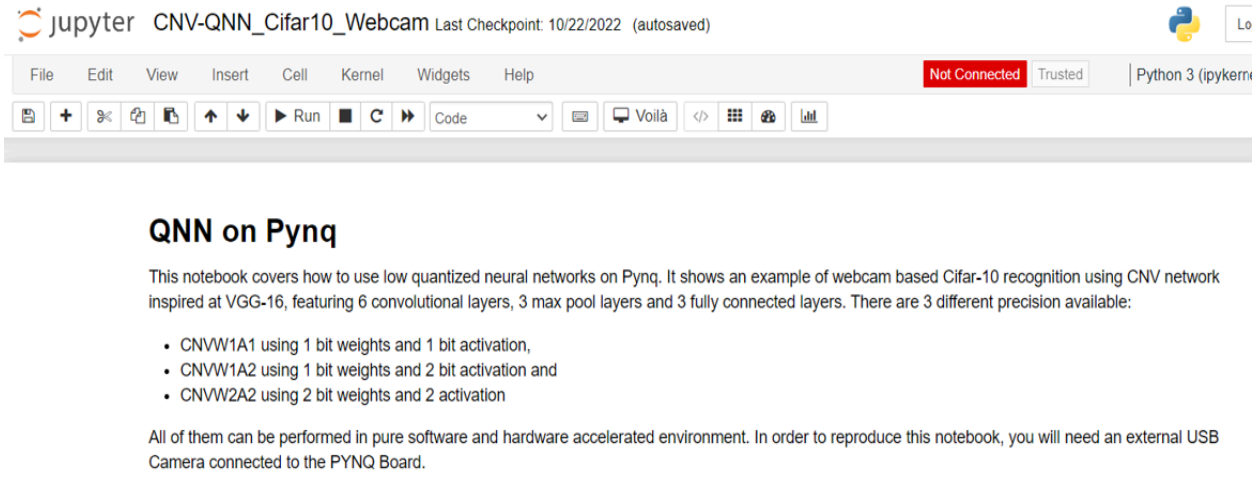
5. This will install the project package on board and create a directory in the jupyter home area



6. Go to CNV-QNN file using following path juptyter notebook:-

192.168.2.99:9090/notebooks/common/data/BNN-PYNQ-master/notebooks/CNV-QNN_Cifar10_Webcam.ipynb

7. The code will open like:-



8. Place the image in front of webcamera using your phone

1. Load image from the camera

The image is captured from the external USB camera and shown:

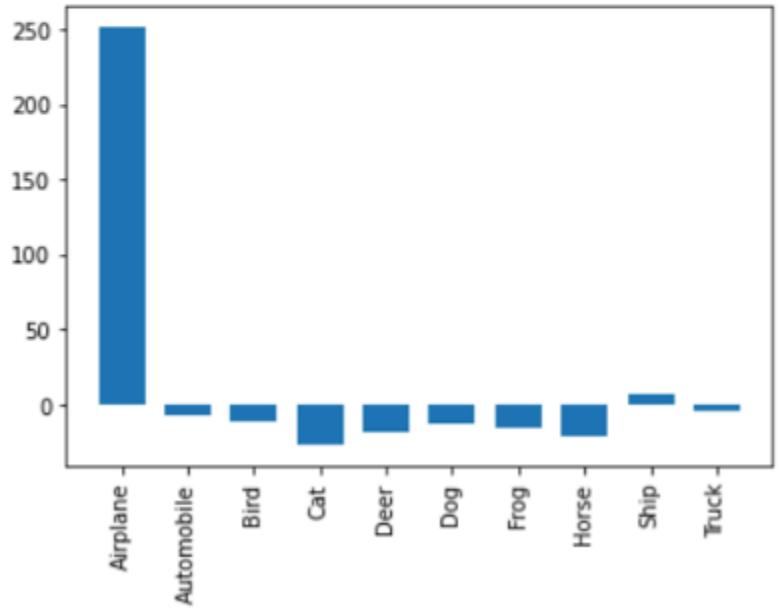
3]:
```
import cv2
from PIL import Image as PIL_Image
from PIL import ImageEnhance
from PIL import ImageOps

# says we capture an image from a webcam
cap = cv2.VideoCapture(0)
_ , cv2_im = cap.read()
cv2_im = cv2.cvtColor(cv2_im,cv2.COLOR_BGR2RGB)
img = PIL_Image.fromarray(cv2_im)
```

For eg here we used photo of Aeroplane:-

9. Result:-

```
In [9]: x_pos = np.arange(len(class_ranksW1A2))
        fig, ax = plt.subplots()
        ax.bar(x_pos, (class_ranksW1A2), 0.7)
        ax.set_xticklabels(hw_classifier.classes, rotation='vertical')
        ax.set_xticks(x_pos)
        ax.set
        plt.show()
```

```
/tmp/ipykernel_7969/3053073917.py:4: UserWarning: FixedFormatte
  ax.set_xticklabels(hw_classifier.classes, rotation='vertical'
```



The graph shows that the image put in front of webcam is of Aeroplane.

Getting familiar with Xilinx's FPGA Vivado Software

Xilinx's Vivado Webpack software provides a full-featured integrated design environment for circuit design, simulation, and implementation. Vivado works seamlessly with the Boolean board and supports both Verilog and VHDL design entry. The tool includes an HDL editing and debug environment, a simulator, a synthesizer, and an FPGA programming interface and hardware debugging tool (a logic analyser).

Part 1. Introduction to the XILINX Vivado

In this part we will learn how to use Xilinx Vivado software to design, simulate and implement a design. As an example, we are using a half adder circuit. Once completed, we will generate the bitstream and learn how to download that code on the Boolean board.
The following steps will guide you further: -

Step 1: Create a Vivado project

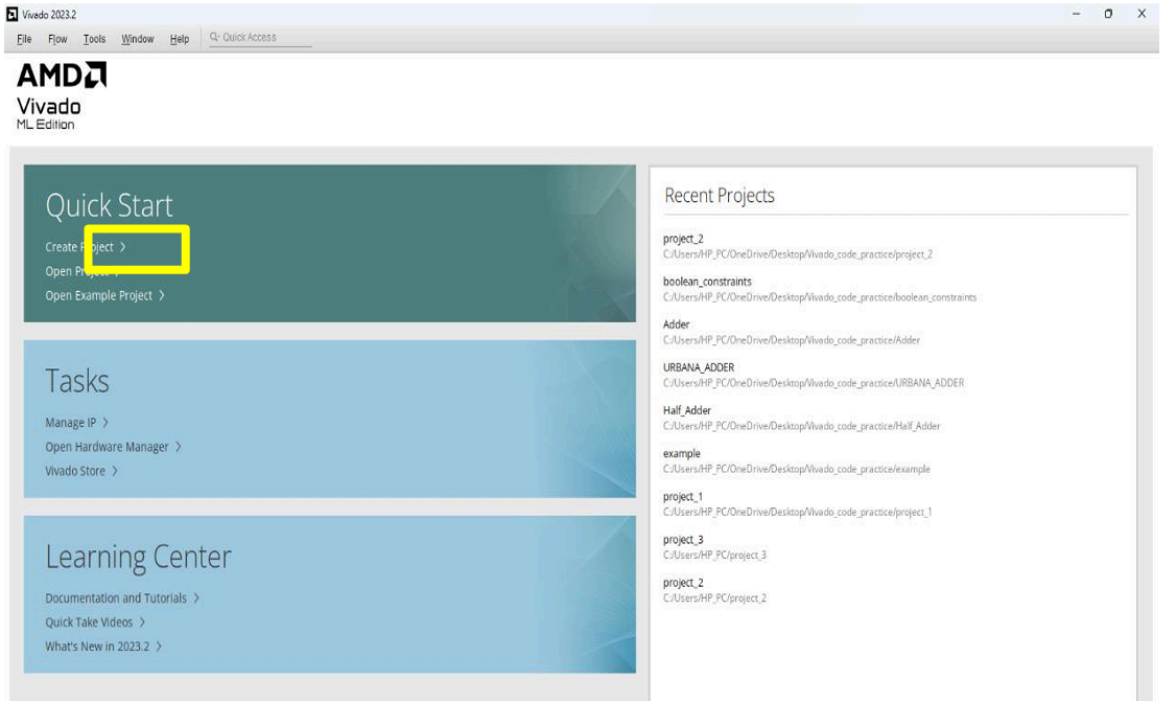
Vivado Projects

Vivado "projects" are directory structures that contain all the files needed by a particular design. Some of these files are user-created source files that describe and constrain the design, but many others are system files created by Vivado to manage the design, simulation, and implementation of projects. In a typical design, you will only be concerned with the user-created source files. But, in the future, if you need more information about your design, or if you need more precise control over certain implementation details, you can access the other files as well.

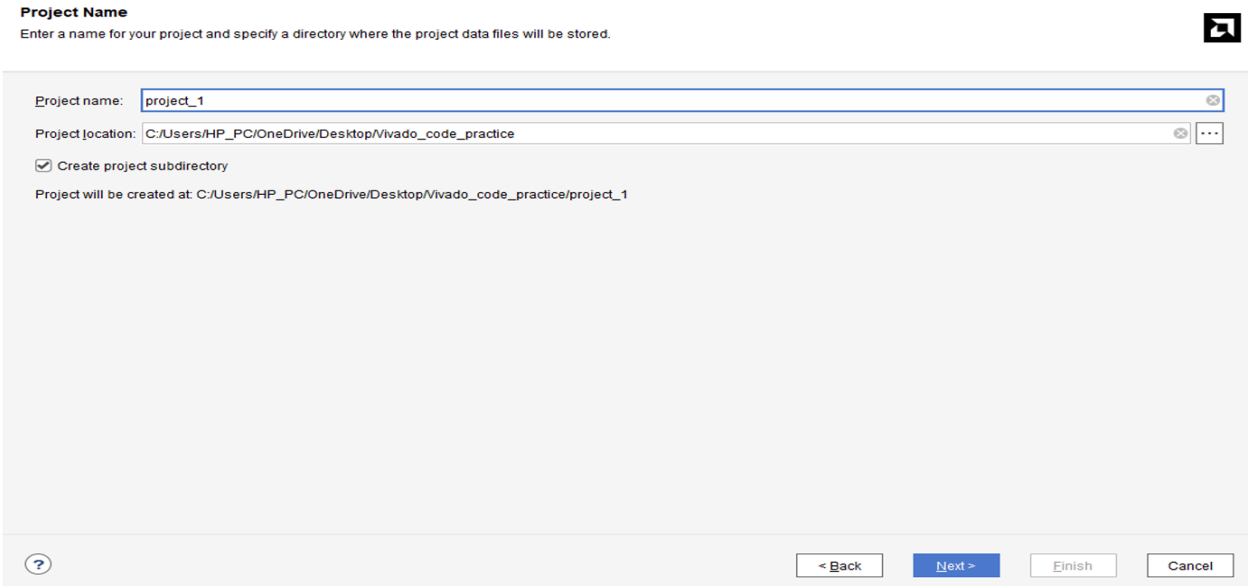
When setting up a project in Vivado, you must give the project a unique name, choose a location to store all the project files, specify the type of project you are creating, add any pre-existing source files or constraints files (you might add existing sources if you are modifying an earlier design, but if you are creating a new design from scratch, you won't add any existing files – you haven't written them yet), and finally, select which physical chip you are designing for. These steps are illustrated below.

1. **Start Vivado:** Double click on the Vivado icon on your desktop to open up the welcome window of the development tool (as shown below). Three main sections can be observed in this window: **"Quick Start", "Tasks",** and **"Learning Center".**

2. **Open create project dialog:** Click on **“Create Project”** in the **Quick Start panel**. This will open the New Project dialog as shown in Figure . Click Next to continue.

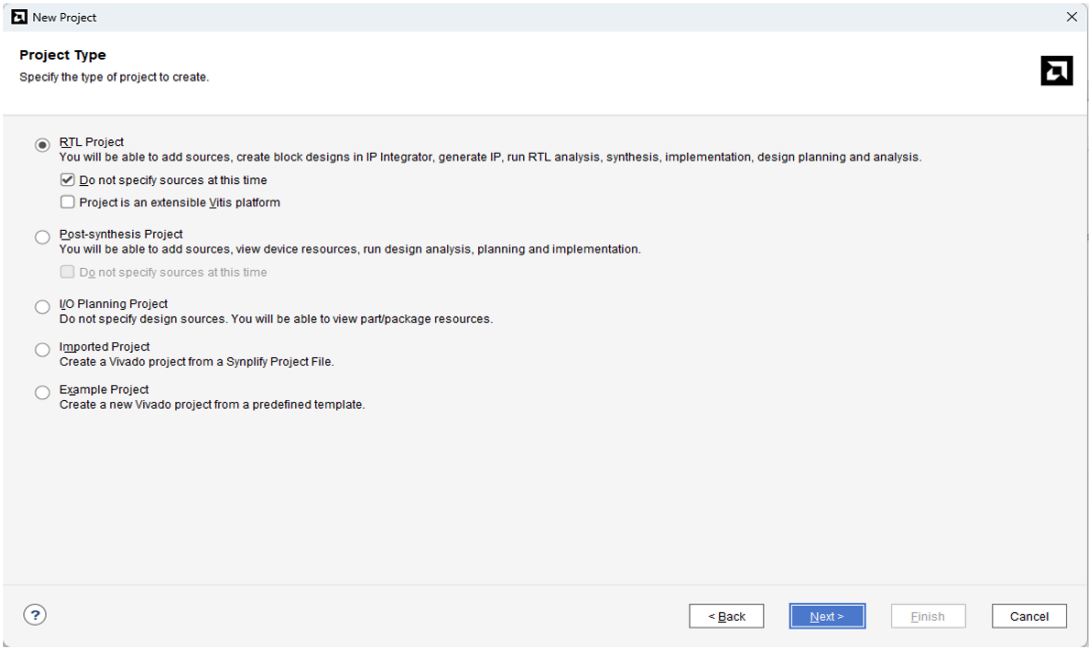


3. **Set project name:** Enter a name for the project. In the figure, the project name is “project_1”, which isn’t a particularly useful name. It’s usually a good idea to make the project name more descriptive, so you can more readily identify your designs in the future. For example, if you design a seven-segment controller, you might call the project **“seven_segment_controller”**. In our case we use project name

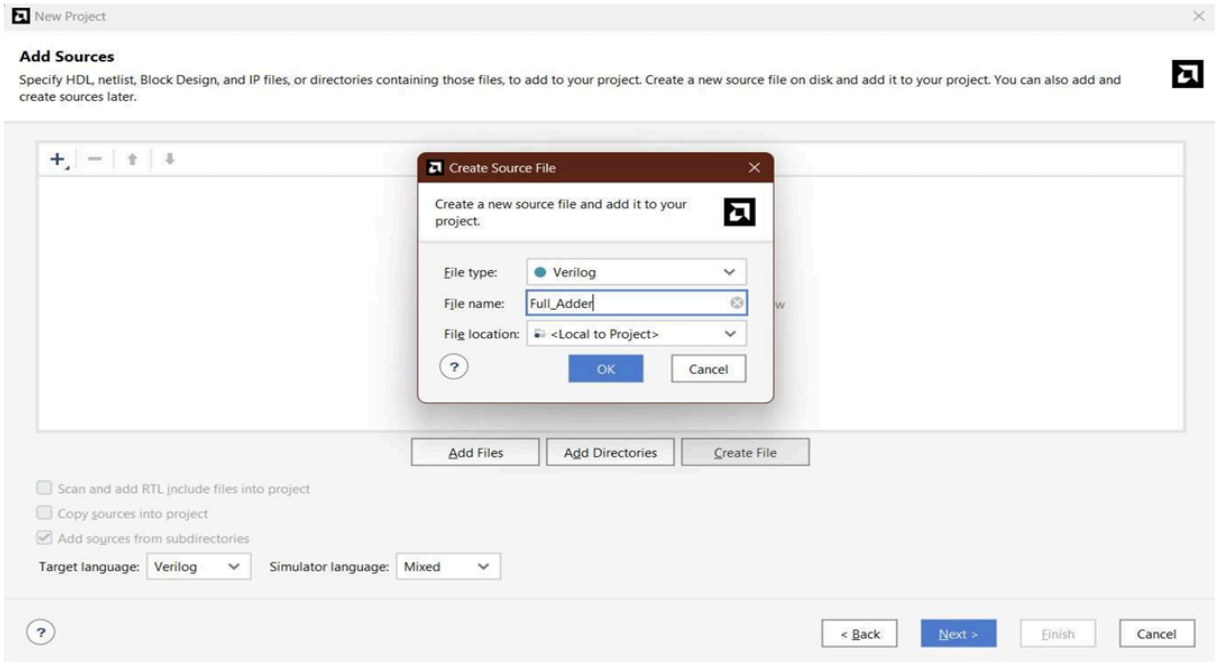


4.

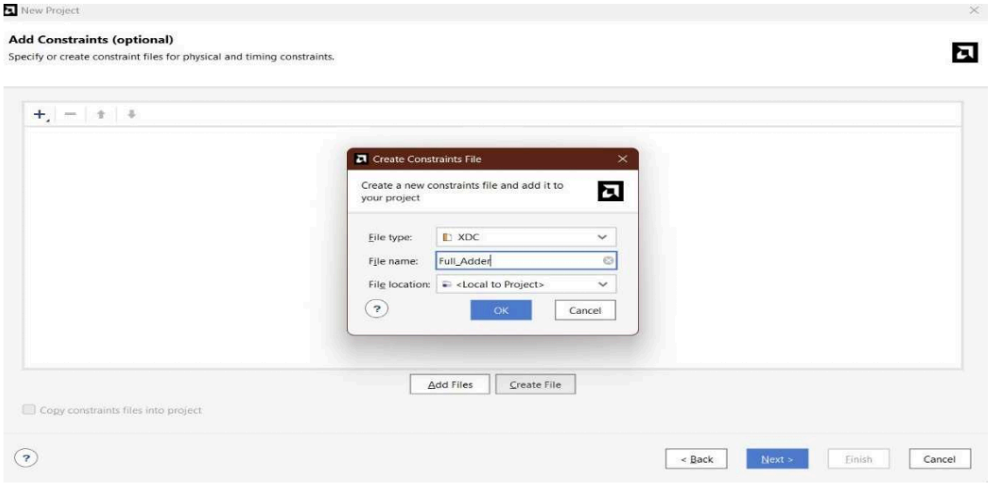
Make sure RTL project is selected, and click Next. You can skip the next two steps as we also have checked the checkbox for do not specifying sources.



5. Click on **Create File** option to create a Verilog file. A dialogue box will appear and select Verilog option and give a suitable name for your Verilog file for our project and click **Next**. You can also skip this step by just clicking **Next** and Verilog file by choosing Add Source option later.



6. Create a constraint file for our Full Addder by clicking on **Create File.** Constraint file help us to match the pins, led and switches of our board with our Verilog file for desired input and output ports of our project. You can skip this step also by just clicking **Next** and Verilog file by choosing Add Source option later.

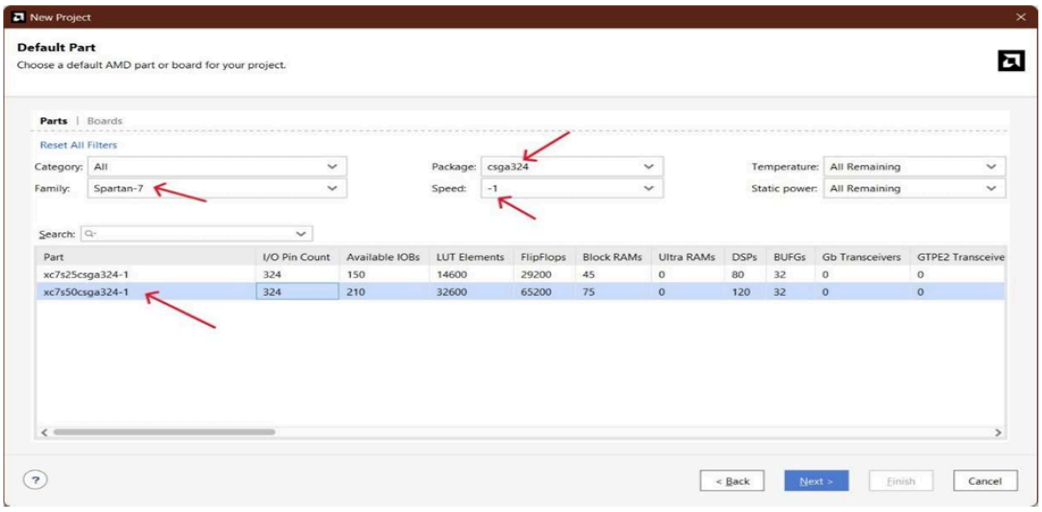


7. Now select the Board on which you want to work. To select the board first select the family and then package of the board then speed.

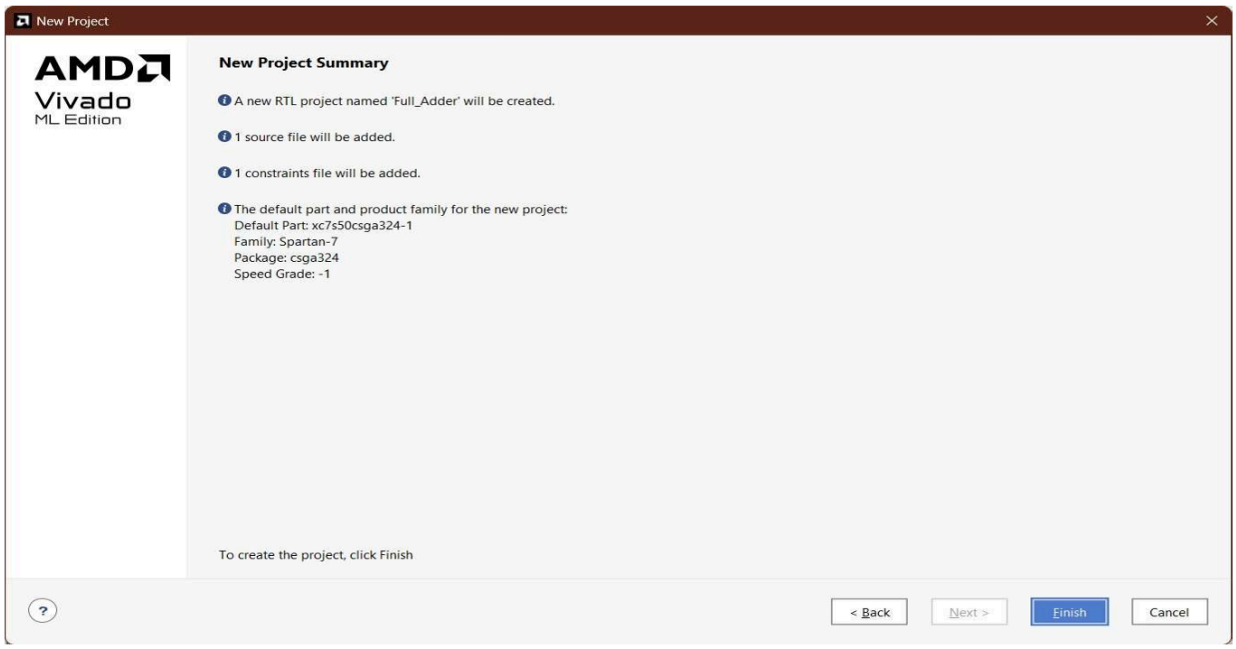
   For Urbana Board

   Family :- Spartan 7 Package:- csga324 speed:- -1

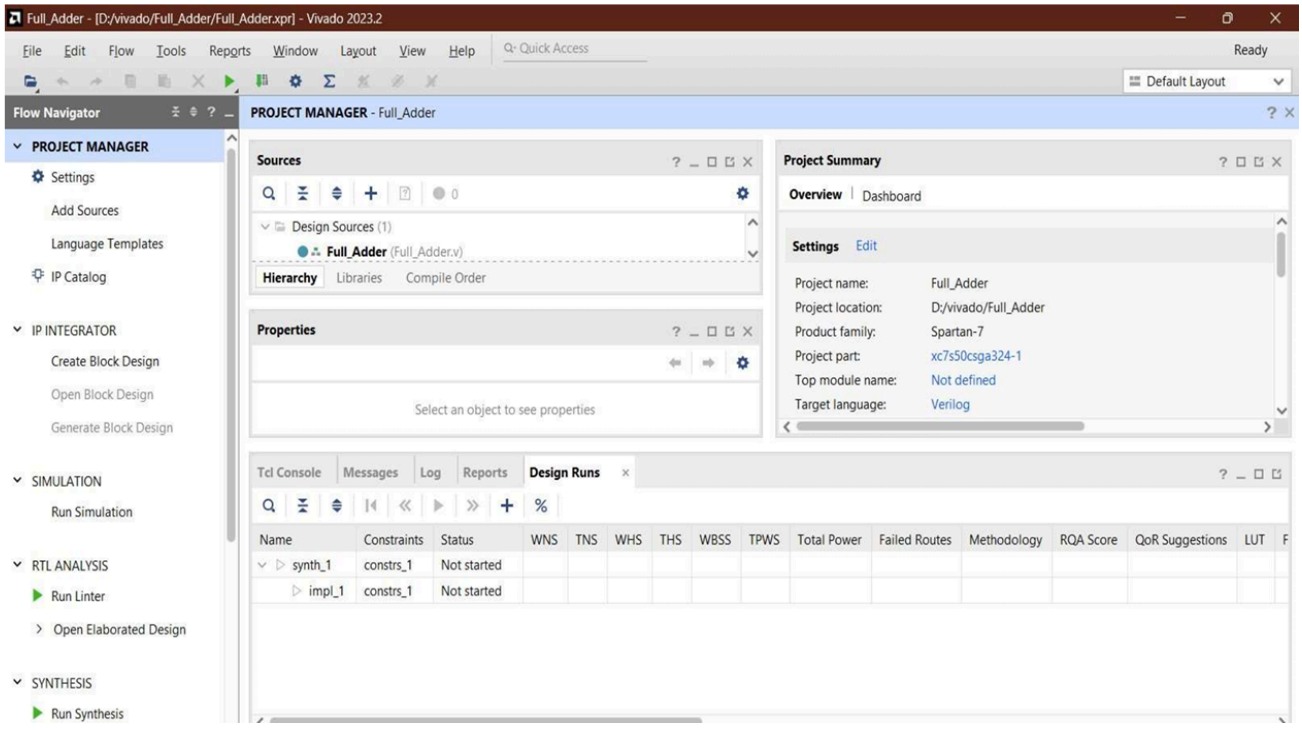
   then select part **xc7s50csga324-1**



8. Check project summary and then click **Finish**



9. The Final window appears



What is URBANA BOARD?

The Urbana board is an FPGA-based digital system development platform that brings state-of-the-art hardware design tools to the engineering desktop. The Urbana board is a complete, ready to use, stand-alone system that can host circuits and systems ranging from basic logic designs through RISC-V and Micrcoblaze embedded-core designs. The board is centered around AMD's Spartan-7 FPGA, and includes a 1Gbit DDR3 SDRAM, audio and video ports, a USB host controller, and many other features.

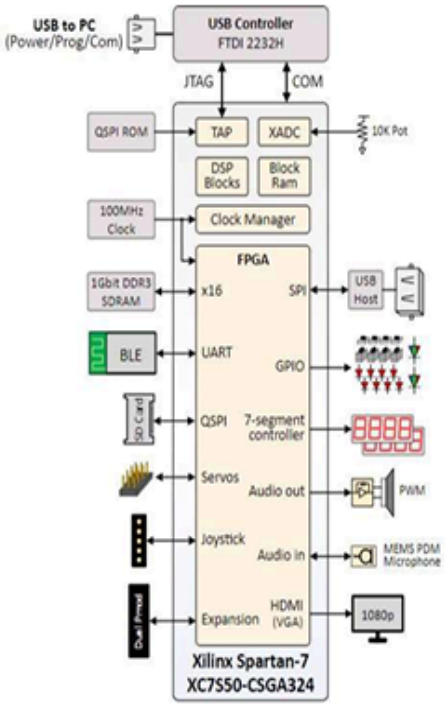
Typical designs implemented on the Urbana board include:

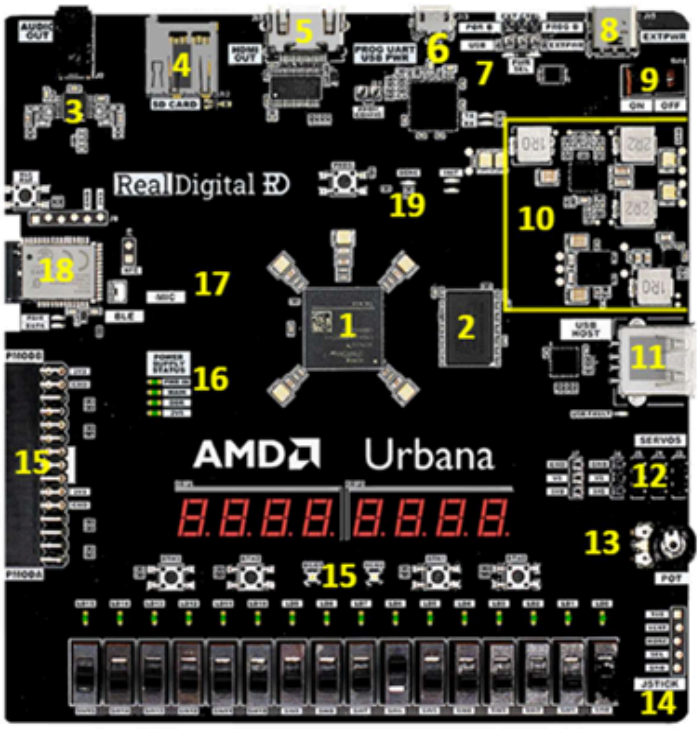
- Basic combinational logic circuits like muxes, decoders, adders, multipliers, etc.;
- Registers, counters, state machines and sequential circuits, bus controllers, etc.;
- IP blocks like pulse-width modulators, VGA controllers, FIFOs, digital filters, etc.;
- Processor cores including RISC-V and Microblaze;
- Hardware/software digital systems.

The Urbana board works seamlessly with AMD's freely downloadable Vivado and Vitis CAD software packages. Vivado provides a full-featured circuit design environment for Verilog, VHDL and/or HLS design entry, and it includes simulation, synthesis, implementation, and hardware debugger (logic analyzer) tools. Vitis supports software design in Assembly and C, and it includes a debugger and built-in terminal program for UART communications. Both tools can program and interact with Urbana's FPGA using the built-in USB programming port, and this same USB port also provides board power, a UART/COM port recognized by Windows and Linux, and a debugging port that drives the GNU software debugger and the hardware debugger.

Urbana Board Features	Spartan-7 FPGA Features
AMD Spartan-7 XCS50 FPGA	32K 6-input LUTs (52K logic cell equivalent)
1Gibt of fast DDR3 SDRAM (x16)	65K+ flip flops & 900Kb of distributed RAM/registers
1080p HDMI source with VGA2HDMI IP block	120 DSP slices with adders, multipliers & accumulators
USB port for power, programming & UART	2.7Mbits of fast, dual-port block RAM
Two Pmod ports, servo and joystick connectors	5 clock management tiles
16 slide switches, 4 pushbuttons, ADC w/pot	Integral dual 12-bit ADC
16 LEDs, 2 RGB LEDs, 8-digit 7seg display	Can host RISC-V and Microblaze embedded processors
Digital MEMs microphone and PWM audio out	Embedded logic analyzer tool for hardware debug
Bluetooth radio and SD card socket	FPGA can be programmed directly from Vivado

URBANA BOARD Components





1	Spartan-7 XC7S50-CSGA324 FPGA	10	Power supplies and control
2	1Gbit DDR3 memory (x16)	11	USB host connector and controller
3	Audio output circuit and 1/8" jack	12	Servo motor connectors
4	SD card socket	13	10K Potentiometer (connected to ADC)
5	HDMI source buffer and connector	14	Joystick connector
6	USB power/prog input and FTDI controller	15	GPIO (buttons, switches, LEDs)
7	Power and reset control select	16	Power supply status indicators
8	Optional external power jack (USBC)	17	PDM digital microphone
9	Main power switch	18	FPGA programming status and control

Description of URBANA BOARD Components:-

1. **Jumper Wires**

Power Select Jumper J16

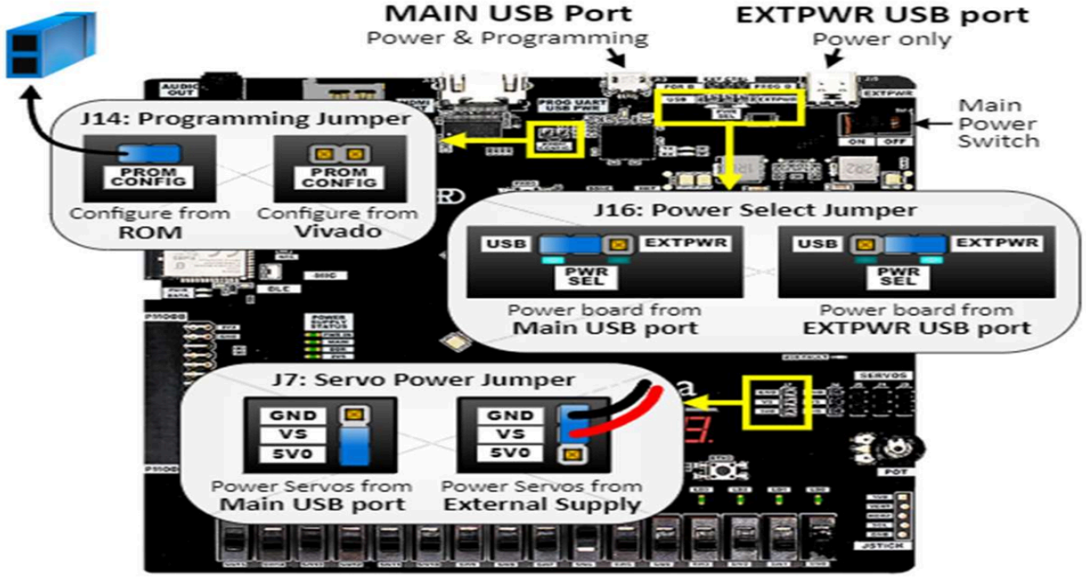
Jumper J16 selects whether board power is supplied by the main/programming micro-B USB port or by the type-C USB port - note that the type-C "EXTPWR" USB port supplies only power, and no data signals are connected. Most typically, J16 is set to draw power from the main/programming USB port. In applications requiring more than the 500mA, additional power can be delivered by connecting an external USB power supply using the type C connector and setting J16 to EXTPWR. One of two blue LEDs near J16 will illuminate to indicate which USB connector is providing power.

Programming Jumper J14

Jumper J14 selects whether the FPGA is programmed from Vivado using the USB/JTAG port, or from the on-board SPI ROM. If no jumper is loaded at J14, the FPGA will be held in reset until it is programmed from Vivado over the JTAG/USB programming port. If a jumper is loaded at J14, the FPGA will be automatically configured from the on-board ROM at power-on.

Servo Power Jumper J7

J7 selects the power source for any connected servo motors. If no servo motors are used, then no jumper is needed on J7. If a jumper connects the 5V0 and VS pins, then servo motor power will come from the main board supply (and the main supply can be sourced from either the main/programming USB port or the EXTPWR port as selected by J16). A single servo motor can draw up to 500mA, so when servo motors are being used, it is typical to use EXTPWR, or to use a separate bench supply connected to J7 using a two-pin MTE cable as shown in the figure below

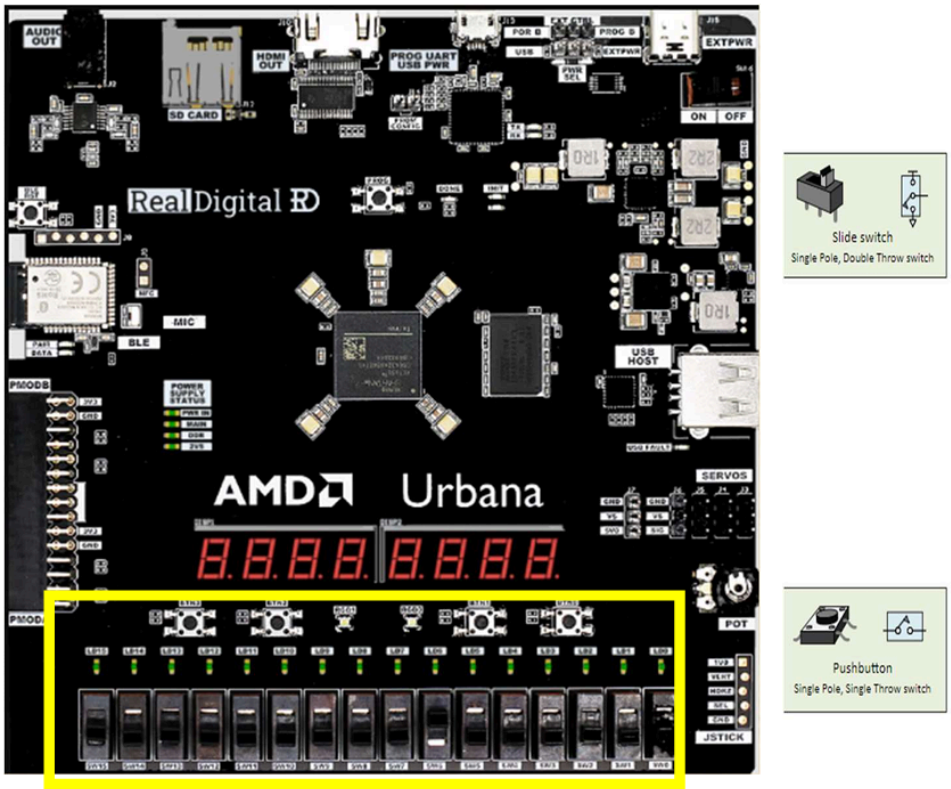


2. GPIO'S in URBANA Board

The term "General Purpose Input/Output" (GPIO) generally refers to digital input signals driven by two-state devices like slide switches or pushbuttons, or output signals that drive an indicator device (like an LED).

The Urbana board includes

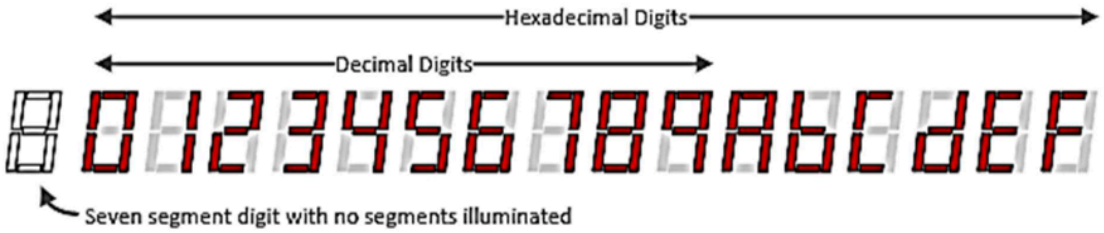
- 16 single-pole double-throw (SPDT) slide switches,
- 4 momentary single-pole single-throw (SPST)
- pushbutton switches,
- 16 single-color LEDs, and
- 2 tri-color (RGB) LEDs.
- An additional 22 GPIO signals drive the Pmod expansion connector.

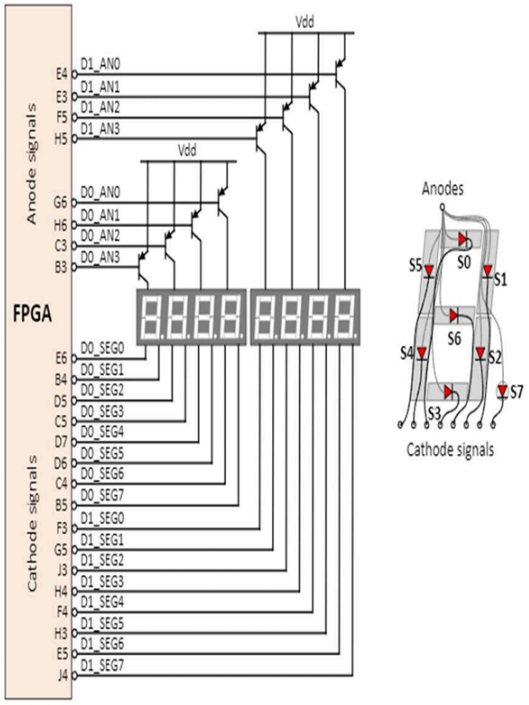


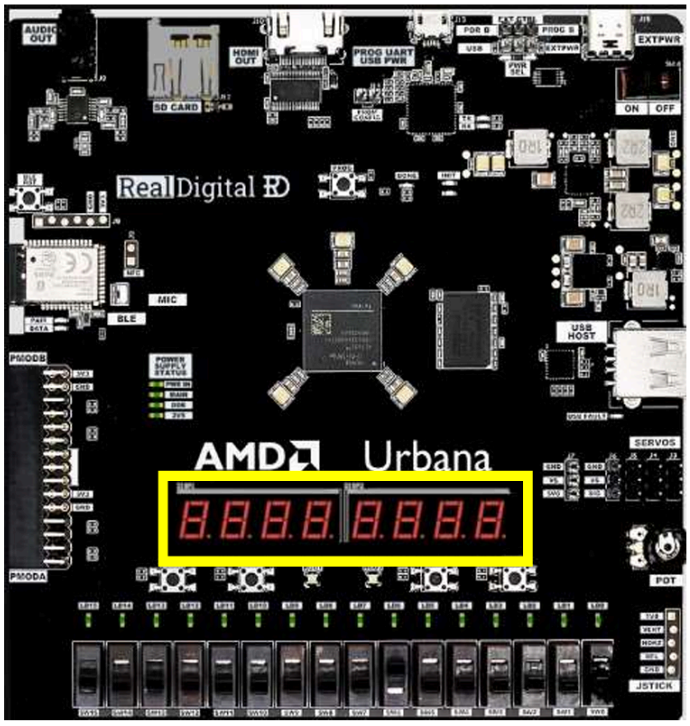
3. Seven Segment Display

A seven-segment display is built from individual LED's arranged in a figure-8 pattern as shown. Any LED/segment can be individually illuminated, so any one of 128 different patterns can be shown. The figure below shows segment illumination patterns for decimal and hexadecimal digits.

The Urbana board includes two 4-digit seven-segment displays (8 total digits) that use a common anode configuration. Segment LEDs consume about 3mA each (well within the current sourcing capability of the FPGA pins), so the cathodes are tied directly to FPGA pins. Since 24mA+ can flow through the anode signals, the anodes are driven from transistors that can provide the needed current (and the transistors are driven from the FPGA pins). All signals are active low.

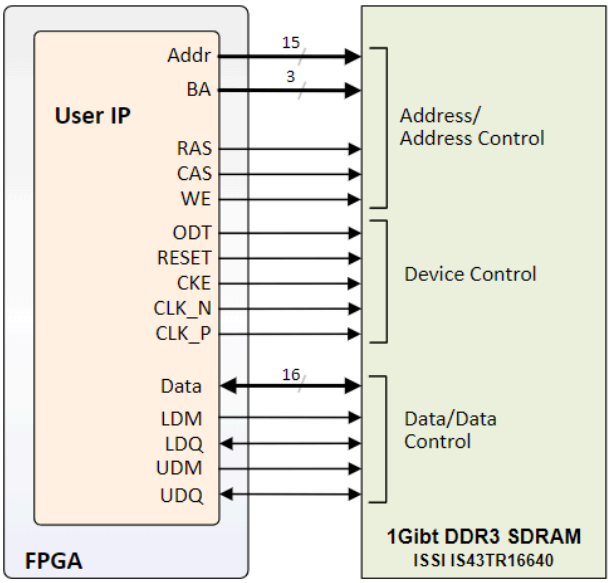






4. DDR

The Urbana board includes a 1Gbit DDR3 organized as a 64M x 16, single-rank memory that can operate up to 1066MHz (ISSI part number IS43TR16640CL-125JBL). All DDR signals have excellent signal integrity. They are length-matched to within 1mm, they all have unifrom characteristic impedance, and they include impedance-matching 36-ohm series termination resistors on all data signals near the FPGA, and on-die termination in the DDR.



DDR3 Pinout

Signal	Pin	Signal	Pin	Signal	Pin
Addr0	V3	DQ0	K2	BA0	V3
Addr1	R4	DQ1	M4	BA2	R4
Addr2	P6	DQ2	K3	BA3	P6
Addr3	T3	DQ3	L5	RAS	U2
Addr4	T6	DQ4	L6	CAS	U1
Addr5	T1	DQ5	M6	WE	T2
Addr6	V5	DQ6	L4	ODT	P7
Addr7	U7	DQ7	K6	RESET	M5
Addr8	R7	DQ8	N5	CKE	T5
Addr9	U6	DQ9	M1	CLK_N	T4
Addr10	U3	DQ10	P1	CLK_P	R5
Addr11	P5	DQ11	N1	LDM	K4
Addr12	V6	DQ12	R2	LDQ_N	L1
Addr13	V7	DQ13	N4	LDQ_P	K1
Addr14	R6	DQ14	P2	UDM	M3
		DQ15	M2	UDQ_N	N2
				UDQ_P	N2

5. CLOCK

The Urbana board includes an Abracon 100MHz 50PPM MEMS/CMOS oscillator that is always running. The oscillator is connected to a multi-region clock-capable (MRCC) FPGA I/O pin (pin N15). From that pin, the clock signal can route to any clock management tile, and/or to any flip-flop in the FPGA using the internal high-speed clock network.

Setting up the URBANA BOARD

Power

Power applied using one of two micro-USB connectors. The primary connector A (labelled PROG UART in the silkscreen) provides power as well as a USB port for JTAG programming and UART communications.

If more power needed, a plug-in power supply capable of proving more current can be connected to the EXTP USB port (B in the figure). Only power is drawn from this connector, so any USB charger plug can be used. Whichever USB connector is used, the power switch labeled C will turn the board on or off.

To draw power from the primary USB port, set jumper J16 to USB; to use external power set it to EXTP.

Programming Urbana Board

During programming, a ".bit" file is transferred to the FPGA to configure its internal programmable elements. Bit files are produced by Vivado during synthesis, and they can be used to program the FPGA directly, or they can be converted into an "mcs" file that can be used to program the on-board ROM. Jumper J14 directs the FPGA to configure from a live Vivado session or from the on-board ROM. When J14 is absent, the FPGA will be held in reset until it is programmed via the USB/JTAG port from a live Vivado session; when J14 is loaded, the FPGA will be programmed from the on-board ROM. The Hardware Manager tool, which is part of the Vivado tool set, is used to program the FPGA and on-board ROM

To program the FPGA from a live Vivado session, connect the computer to the Urbana board's programming USB port and power-on the board. From inside Vivado, **open the Hardware Manager** and select "**Open Target -> Autoconnect**". The Hardware Manager will scan available ports until the Urbana board is detected, and then display a small panel identifying the **xc7s50 FPGA has been detected**. To program the FPGA, click on "**Program Device**" under the Hardware Manager. Note the .bit file from the current project is selected by default, but you can navigate to any other .bit file as well.

Setting up the URBANA BOARD

Power

Power applied using one of two micro-USB connectors. The primary connector A (labelled PROG UART in the silkscreen) provides power as well as a USB port for JTAG programming and UART communications.

If more power needed, a plug-in power supply capable of proving more current can be connected to the EXTP USB port (B in the figure). Only power is drawn from this connector, so any USB charger plug can be used. Whichever USB connector is used, the power switch labeled C will turn the board on or off.

To draw power from the primary USB port, set jumper J16 to USB; to use external power set it to EXTP.

Programming Urbana Board

During programming, a ".bit" file is transferred to the FPGA to configure its internal programmable elements. Bit files are produced by Vivado during synthesis, and they can be used to program the FPGA directly, or they can be converted into an "mcs" file that can be used to program the on-board ROM. Jumper J14 directs the FPGA to configure from a live Vivado session or from the on-board ROM. When J14 is absent, the FPGA will be held in reset until it is programmed via the USB/JTAG port from a live Vivado session; when J14 is loaded, the FPGA will be programmed from the on-board ROM. The Hardware Manager tool, which is part of the Vivado tool set, is used to program the FPGA and on-board ROM

To program the FPGA from a live Vivado session, connect the computer to the Urbana board's programming USB port and power-on the board. From inside Vivado, **open the Hardware Manager** and select "**Open Target -> Autoconnect**". The Hardware Manager will scan available ports until the Urbana board is detected, and then display a small panel identifying the **xc7s50 FPGA has been detected**. To program the FPGA, click on "**Program Device**" under the Hardware Manager. Note the .bit file from the current project is selected by default, but you can navigate to any other .bit file as well.

Constraint file and bitstream

Constraint File (XDC):

- An XDC (eXtensible Design Constraints) file is a text-based file used to specify design requirements for your Verilog code in Vivado.
- It provides instructions to the Vivado tools regarding timing, placement, and routing of your design within the FPGA fabric.
- Common constraints defined in XDC files include:
    - **Clock Constraints:** Specifying clock frequencies, phases, and relationships between clocks.
    - **I/O Constraints:** Assigning specific physical pins on the FPGA to your design's input and output signals.
    - **Timing Constraints:** Defining minimum and maximum delays for signals within your design.
    - **Placement Constraints:** Suggesting or fixing the location of specific logic blocks in the FPGA fabric (advanced usage).

Benefits of Using XDC Files:

- **Improved Performance:** By providing timing constraints, you can ensure that your design meets the required speed.
- **Enhanced Reliability:** Proper placement and routing can help mitigate signal integrity issues.
- **Reduced Design Iteration Time:** Clear constraints can guide the Vivado tools to a better solution faster.

Creating and Using XDC Files:

- You can create XDC files manually using a text editor or leverage Vivado's graphical interface to generate them.
- Add the XDC file to your Vivado project and specify which constraint set to use during implementation.
- Vivado will consider the constraints during synthesis, placement, and routing to optimize your design for the target FPGA.

Bitstream:

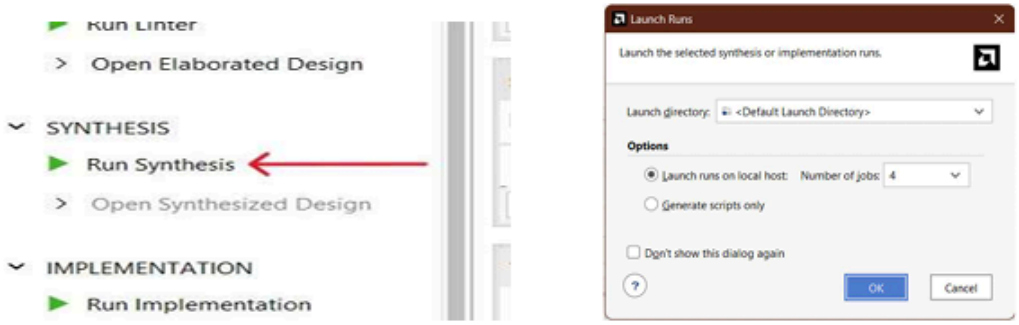
- A bitstream is a configuration file that contains the programming instructions for the FPGA.
- It's generated by Vivado after you've successfully implemented your design and created a Place and Route (PAR) file.
- The bitstream essentially tells the FPGA how to configure its internal logic blocks and routing resources to match your Verilog code.

Relationship Between XDC and Bitstream:

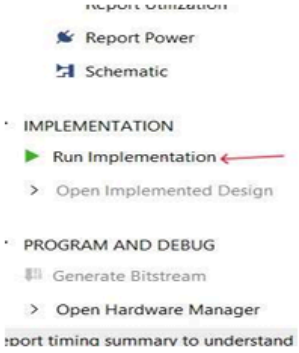
- The constraints specified in your XDC file influence the way the Vivado tools generate the bitstream.
- By providing timing, placement, and routing constraints, you can indirectly affect the final configuration of the FPGA through the bitstream.

Follow the steps given below to synthesize, implement and generate bitstream:-

1. After your Verilog and constraint files are complete, you can Synthesize the design project. In the synthesis process, Verilog code is translated into a "netlist" that defines all the required circuit components needed by the design (these components are the programmable parts of the targeted logic device). You can start the Synthesize process by clicking on Run Synthesis button in the Flow Navigator panel as shown.

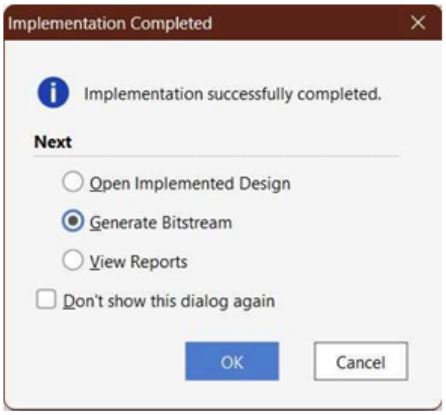


2. After synthesis is completed, continue with run implementation. After the design is synthesized, you must run the Implementation process. Implementation tool will take the netlist as input and does optimization, placement and routing. The implementation process maps the synthesized design onto the Xilinx chip targeted by the design. Click the Run Implementation button in the Flow Navigator panel as shown. When the implementation process is running, the log panel at the bottom of Project Manager will show details about any errors that occur. After implementation is done you can view your design schematic.



OR

3. Generate Bitstream: After the design is successfully implemented, you can create a .bit file by clicking on the Generate Bitstream process located in the Flow Navigator panel as shown. The process translates the implemented design into a bitstream which can be directly programmed into your board's device



In Summary:

- XDC files provide design guidance to Vivado, helping it optimize your Verilog code for the target FPGA.
- Bitstreams are the actual programming instructions used to configure the FPGA based on the implementation results influenced by constraints.
- They work together to ensure your Verilog design functions correctly and efficiently on the FPGA hardware.

Experiment-6

Objective: To create a **decade counter** that drives a **seven-segment display** and a set of LEDs.

Tools Required: Xilinx's VIVADO FPGA software and Urbana board along with its pin description.

Theory:- Counters are fundamental building blocks in digital circuits. They are sequential logic circuits that store and manipulate a count value.
Counters typically use flip-flops (memory elements) to store the count value. Each flip-flop can be in either a 0 or 1 state. By cascading multiple flip-flops and applying clock signals, the counter can increment or decrement the stored value.

Decade Counters: These count from 0 to 9 and are commonly used for displaying digits. They can be implemented using binary counters with additional logic (like the clock divider in the given code) to achieve the specific modulo-10 behavior.

Verilog code:-

```
module sevensegcounter(
  input clk,
  output [15:0] led,
  output [7:0] seg,
  output [3:0] an
);

  reg [25:0] counter = 26'd0;
  reg divclk = 1'b0;
  reg [9:0] round_counter = 4'd0;
  reg [15:0] led_reg;
  reg [7:0] seg_reg;
  reg [7:0] temp_reg [0:9];

  initial begin
    temp_reg[0] = 8'hC0;
    temp_reg[1] = 8'hF9;
    temp_reg[2] = 8'hA4;
    temp_reg[3] = 8'hB0;
    temp_reg[4] = 8'h99;
```

```
    temp_reg[5] = 8'h92;
    temp_reg[6] = 8'h82;
    temp_reg[7] = 8'hF8;
    temp_reg[8] = 8'h80;
    temp_reg[9] = 8'h90;
  end

  /* Clock Divider: 100MHz -> 10Hz (1s) */
  always @(posedge clk) begin
    if (counter == 26'd49999999) begin
      divclk <= ~divclk;
      counter <= 26'd0;
    end else begin
      divclk <= divclk;
      counter <= counter + 1'd1;
    end
  end

  always @(posedge divclk) begin
    round_counter <= round_counter + 1'd1;
    if (round_counter == 4'd0) begin
      led_reg <= 16'hFFFE; // Turn on all LEDs except the last one
    end
    seg_reg <= temp_reg[round_counter];
  end

  assign led = {led_reg[14:0], led_reg[15]}; // Shift last bit only once per cycle
  assign an = 4'h0;
  assign seg = seg_reg;

endmodule
```

Counter:

- The code uses a 26-bit register (`counter`) to keep track of a count.
- It has a clock divider circuit that generates a slower clock signal (`divclk`) from the main clock (`clk`). This slows down the counting process.
- In the `always @(posedge clk)` block, the counter increments every clock cycle (`counter <= counter + 1'd1`).
- When the counter reaches its maximum value (`26'd49999999`), it resets to zero and toggles the `divclk` signal.

Seven-Segment Display:

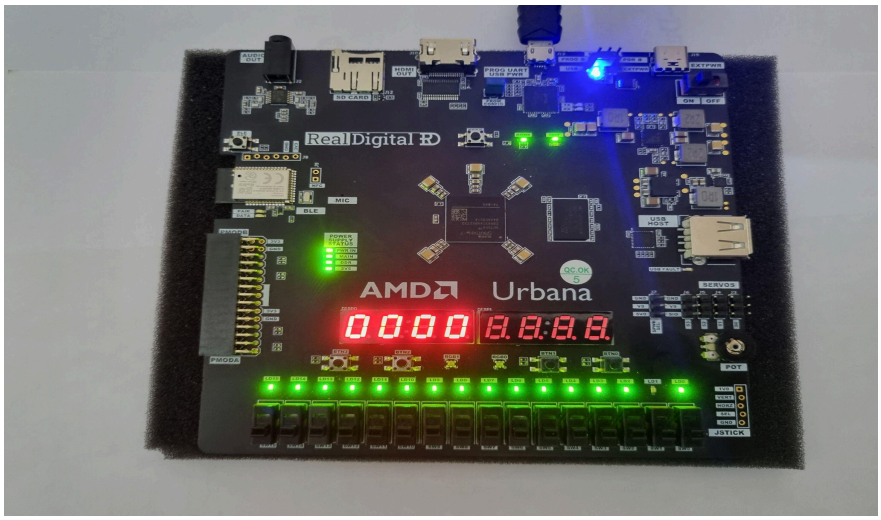
- The code uses a 10-element array (`temp_reg`) to store the binary patterns for displaying digits 0 to 9 on a seven-segment display.
- A 4-bit register (`round_counter`) keeps track of which digit to display on the seven-segment display.
- In the `always @(posedge divclk)` block, the `round_counter` increments.
- Every time `round_counter` resets to zero, it updates the `seg_reg` register with the next digit's pattern from the `temp_reg` array.
- Finally, the `seg` output is assigned the value of `seg_reg`, which drives the seven-segment display.

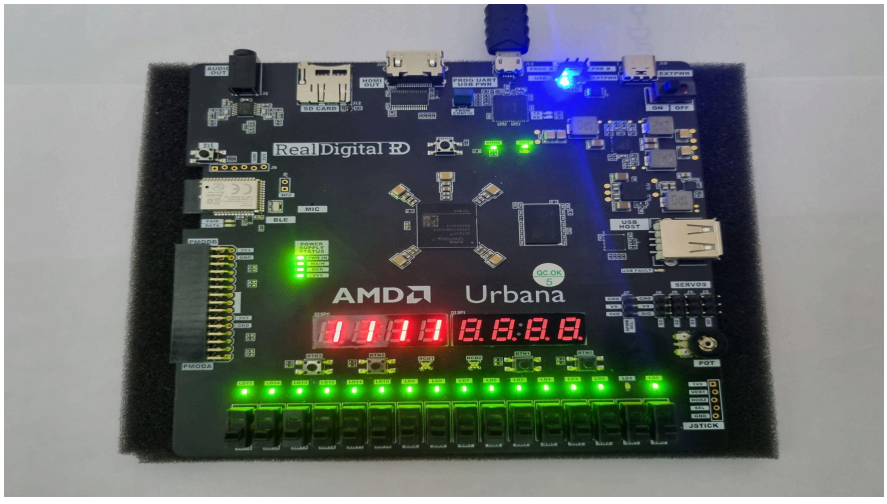
Constraint file:-

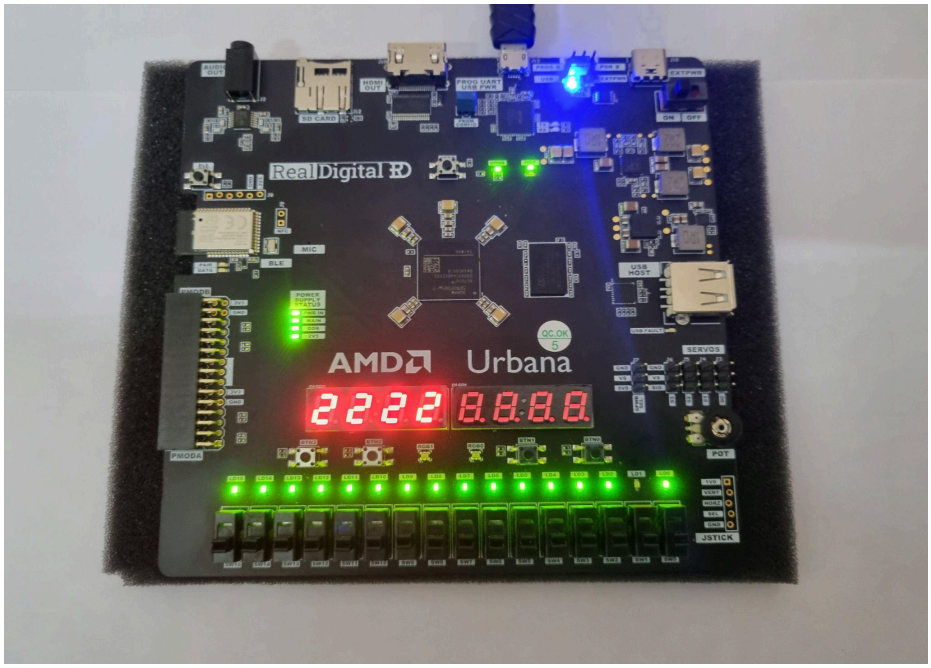
```
set_property -dict {PACKAGE_PIN N15 IOSTANDARD LVCMOS33} [get_ports {clk}]
set_property -dict {PACKAGE_PIN C13 IOSTANDARD LVCMOS33} [get_ports {led[0]}]
set_property -dict {PACKAGE_PIN C14 IOSTANDARD LVCMOS33} [get_ports {led[1]}]
set_property -dict {PACKAGE_PIN D14 IOSTANDARD LVCMOS33} [get_ports {led[2]}]
set_property -dict {PACKAGE_PIN D15 IOSTANDARD LVCMOS33} [get_ports {led[3]}]
set_property -dict {PACKAGE_PIN D16 IOSTANDARD LVCMOS33} [get_ports {led[4]}]
set_property -dict {PACKAGE_PIN F18 IOSTANDARD LVCMOS33} [get_ports {led[5]}]
set_property -dict {PACKAGE_PIN E17 IOSTANDARD LVCMOS33} [get_ports {led[6]}]
set_property -dict {PACKAGE_PIN D17 IOSTANDARD LVCMOS33} [get_ports {led[7]}]
set_property -dict {PACKAGE_PIN C17 IOSTANDARD LVCMOS33} [get_ports {led[8]}]
set_property -dict {PACKAGE_PIN B18 IOSTANDARD LVCMOS33} [get_ports {led[9]}]
set_property -dict {PACKAGE_PIN A17 IOSTANDARD LVCMOS33} [get_ports {led[10]}]
set_property -dict {PACKAGE_PIN B17 IOSTANDARD LVCMOS33} [get_ports {led[11]}]
set_property -dict {PACKAGE_PIN C18 IOSTANDARD LVCMOS33} [get_ports {led[12]}]
set_property -dict {PACKAGE_PIN D18 IOSTANDARD LVCMOS33} [get_ports {led[13]}]
set_property -dict {PACKAGE_PIN E18 IOSTANDARD LVCMOS33} [get_ports {led[14]}]
set_property -dict {PACKAGE_PIN G17 IOSTANDARD LVCMOS33} [get_ports {led[15]}]
set_property -dict {PACKAGE_PIN G6 IOSTANDARD LVCMOS25} [get_ports {an[0]}]; # Active
LOW
set_property -dict {PACKAGE_PIN H6 IOSTANDARD LVCMOS25} [get_ports {an[1]}]; # Active
LOW
set_property -dict {PACKAGE_PIN C3 IOSTANDARD LVCMOS25} [get_ports {an[2]}]; # Active
LOW
set_property -dict {PACKAGE_PIN B3 IOSTANDARD LVCMOS25} [get_ports {an[3]}]; # Active
LOW
set_property -dict {PACKAGE_PIN E6 IOSTANDARD LVCMOS25} [get_ports {seg[0]}]; # CA
Active LOW
```

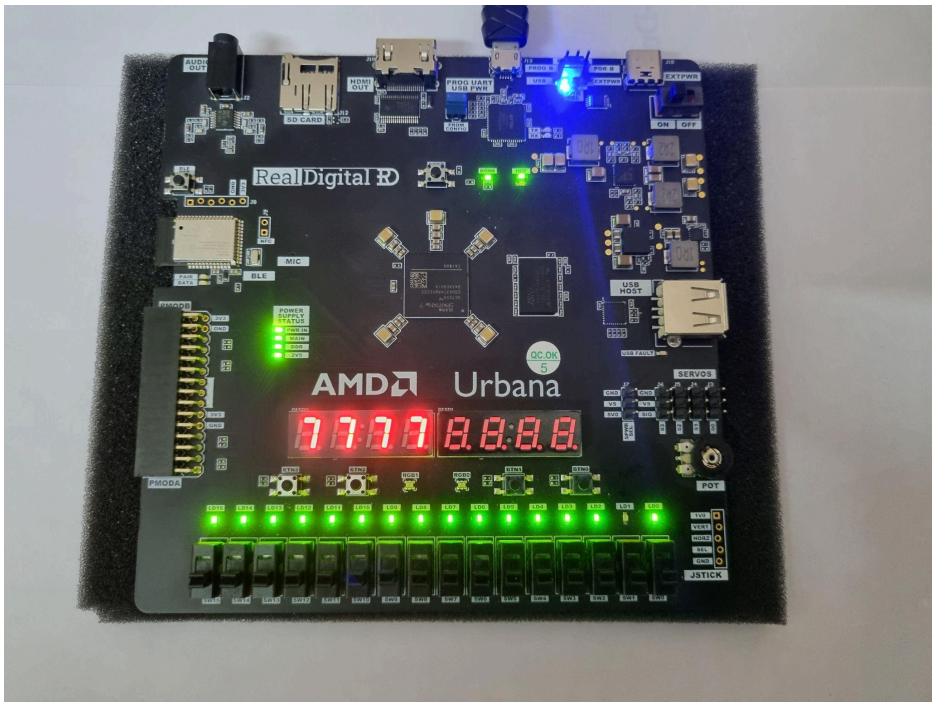
set_property -dict {PACKAGE_PIN B4 IOSTANDARD LVCMOS25} [get_ports {seg[1]}]; # CB Active LOW
set_property -dict {PACKAGE_PIN D5 IOSTANDARD LVCMOS25} [get_ports {seg[2]}]; # CC Active LOW
set_property -dict {PACKAGE_PIN C5 IOSTANDARD LVCMOS25} [get_ports {seg[3]}]; # CD Active LOW
set_property -dict {PACKAGE_PIN D7 IOSTANDARD LVCMOS25} [get_ports {seg[4]}]; # CE Active LOW
set_property -dict {PACKAGE_PIN D6 IOSTANDARD LVCMOS25} [get_ports {seg[5]}]; # CF Active LOW
set_property -dict {PACKAGE_PIN C4 IOSTANDARD LVCMOS25} [get_ports {seg[6]}]; # CG Active LOW
set_property -dict {PACKAGE_PIN B5 IOSTANDARD LVCMOS25} [get_ports {seg[7]}]; # CDP Active LOW

Urbana Board Outputs:









And so on till 9.

Experiment-7

Objective: To display a **sequence of numbers** (multiples of 5) on a **seven-segment display** with a refresh rate.

Tools Required: Xilinx's VIVADO FPGA software and Urbana board along with its pin description.

Theory:
In digital circuits, refresh rate is often achieved using counters to control the timing of how often a specific operation or display update occurs.
refresh rate refers to the frequency at which a display or data is updated. For example, a monitor with a 60Hz refresh rate updates the image on the screen 60 times per second.
Counters are sequential logic circuits that can increment or decrement their value based on a clock signal.

`display_counter`: This counter increments every clock cycle.
`refresh_counter`: This counter cycles through a specific range (2 bits in this case) to select which digit to display on the seven-segment display.

Verilog code:

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 30.06.2024 22:53:04
// Design Name:
// Module Name: fivetable
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
```

```
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module fivetable(
input clk,       // Clock signal
    output reg [7:0] seg,  // Segment control signals
    output [3:0] an   // Anode control signals
    );
    reg [7:0] segment_pattern [9:0];  // Segment patterns for digits 0-9
    reg [3:0] anode_pattern;  // Anode pattern
    reg [3:0] digit_high, digit_low;  // Digits to display (high and low)
    reg [27:0] display_counter;  // Counter to cycle through numbers
    reg [3:0] number_index;  // Index to cycle through the table of 5

    initial begin
        // Segment patterns for digits 0-9
        segment_pattern[0] = 8'b11000000;  // 0
        segment_pattern[1] = 8'b11111001;  // 1
        segment_pattern[2] = 8'b10100100;  // 2
        segment_pattern[3] = 8'b10110000;  // 3
        segment_pattern[4] = 8'b10011001;  // 4
        segment_pattern[5] = 8'b10010010;  // 5
        segment_pattern[6] = 8'b10000010;  // 6
        segment_pattern[7] = 8'b11111000;  // 7
        segment_pattern[8] = 8'b10000000;  // 8
        segment_pattern[9] = 8'b10010000;  // 9

        number_index = 0;  // Start with the first number in the table (5)
    end

    // Display counter to update the number every second or so (assuming 100MHz clock)
    always @(posedge clk) begin
        display_counter <= display_counter + 1;
        if (display_counter == 100000000) begin  // Update every second
            display_counter <= 0;
            number_index <= number_index + 1;
            if (number_index == 10) number_index <= 0;  // Wrap around after 10 numbers
        end
    end

    // Determine the high and low digits for the current number in the table of 5
    always @(*) begin
```

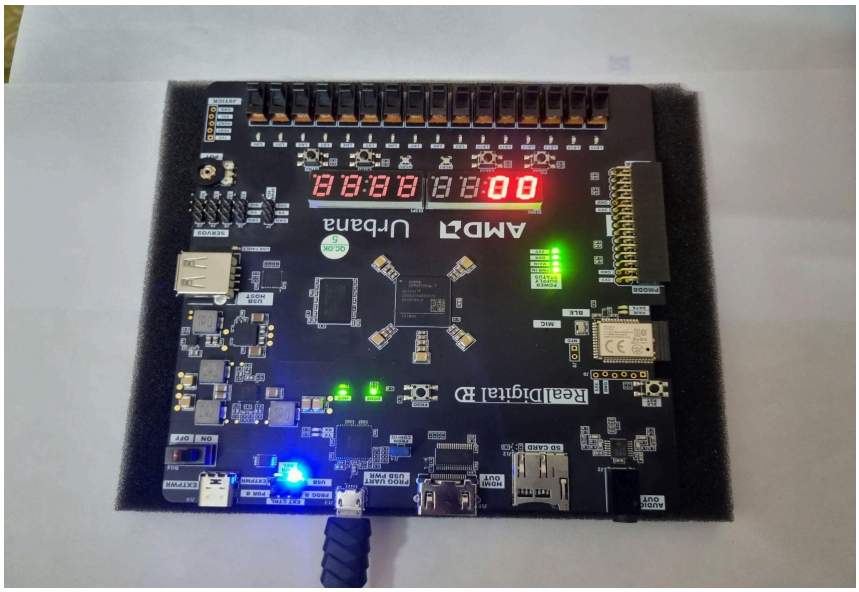
```
    case (number_index)
        0: begin digit_high = 0; digit_low = 5; end   // 5
        1: begin digit_high = 1; digit_low = 0; end   // 10
        2: begin digit_high = 1; digit_low = 5; end   // 15
        3: begin digit_high = 2; digit_low = 0; end   // 20
        4: begin digit_high = 2; digit_low = 5; end   // 25
        5: begin digit_high = 3; digit_low = 0; end   // 30
        6: begin digit_high = 3; digit_low = 5; end   // 35
        7: begin digit_high = 4; digit_low = 0; end   // 40
        8: begin digit_high = 4; digit_low = 5; end   // 45
        9: begin digit_high = 5; digit_low = 0; end   // 50
        default: begin digit_high = 0; digit_low = 0; end
    endcase
end

reg [18:0] refresh_counter;  // Counter for refresh rate control
wire [1:0] scan_select;  // 2-bit counter to cycle through the digits

// Refresh counter to cycle through the digits
always @(posedge clk) begin
    refresh_counter <= refresh_counter + 1;
end
// Select which digit to display based on the refresh counter
assign scan_select = refresh_counter[18:17];
// Select the digit and the corresponding segment pattern
always @(*) begin
    case(scan_select)
        2'b00: begin
            anode_pattern = 4'b1110;  // Enable the first digit
            seg = segment_pattern[digit_low];  // Display the low digit
        end
        2'b01: begin
            anode_pattern = 4'b1101;  // Enable the second digit
            seg = segment_pattern[digit_high];  // Display the high digit
        end
        2'b10: begin
            anode_pattern = 4'b1011;  // Disable the third digit
            seg = 8'b11111111;  // Turn off all segments
        end
        2'b11: begin
            anode_pattern = 4'b0111;  // Disable the fourth digit
            seg = 8'b11111111;  // Turn off all segments
        end
        default: begin
```

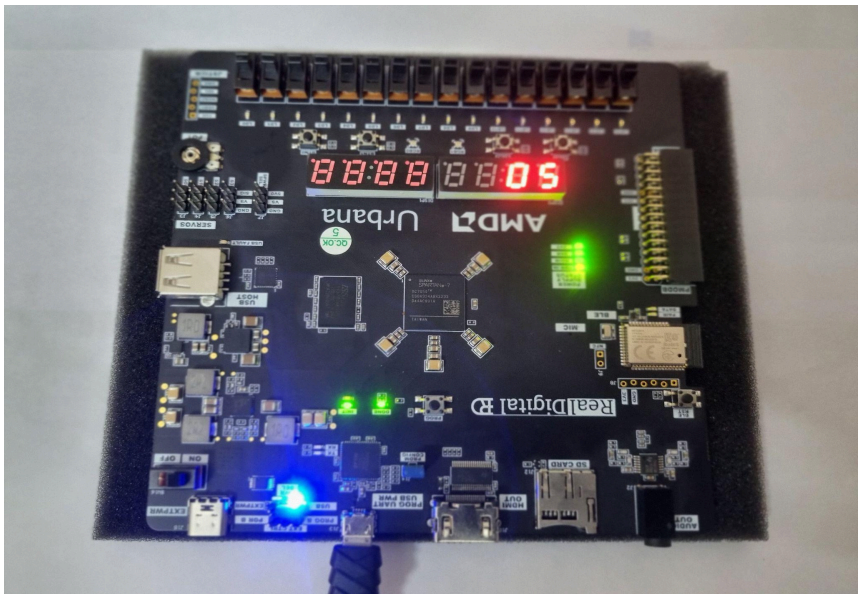
```
            anode_pattern = 4'b1111;  // Disable all digits
            seg = 8'b11111111;  // Turn off all segments
          end
       endcase
    end
   // Assign the anode pattern to the output
    assign an = anode_pattern;
endmodule
```

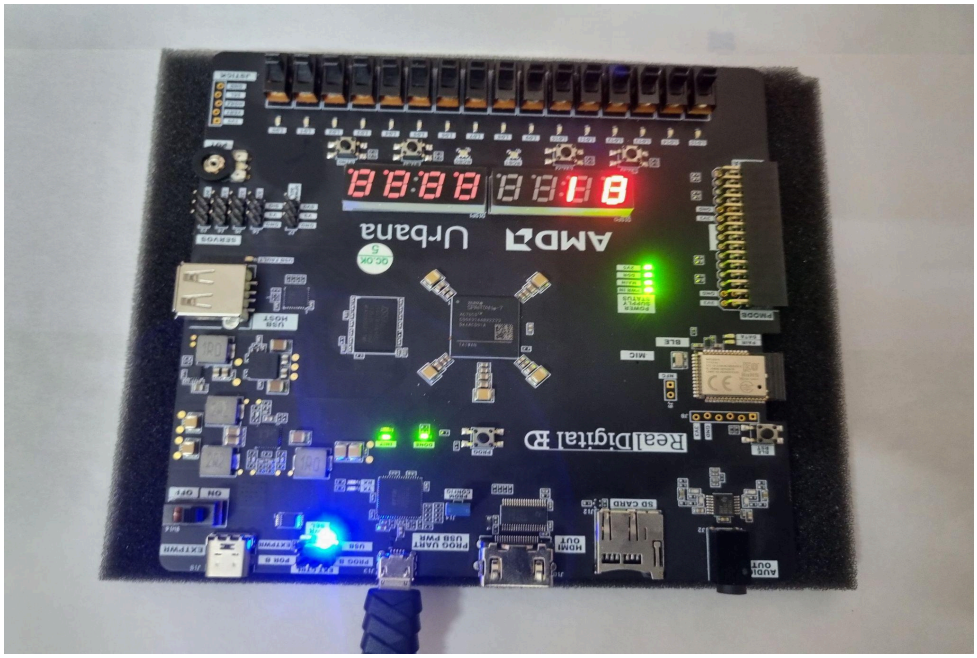
Constraint File:

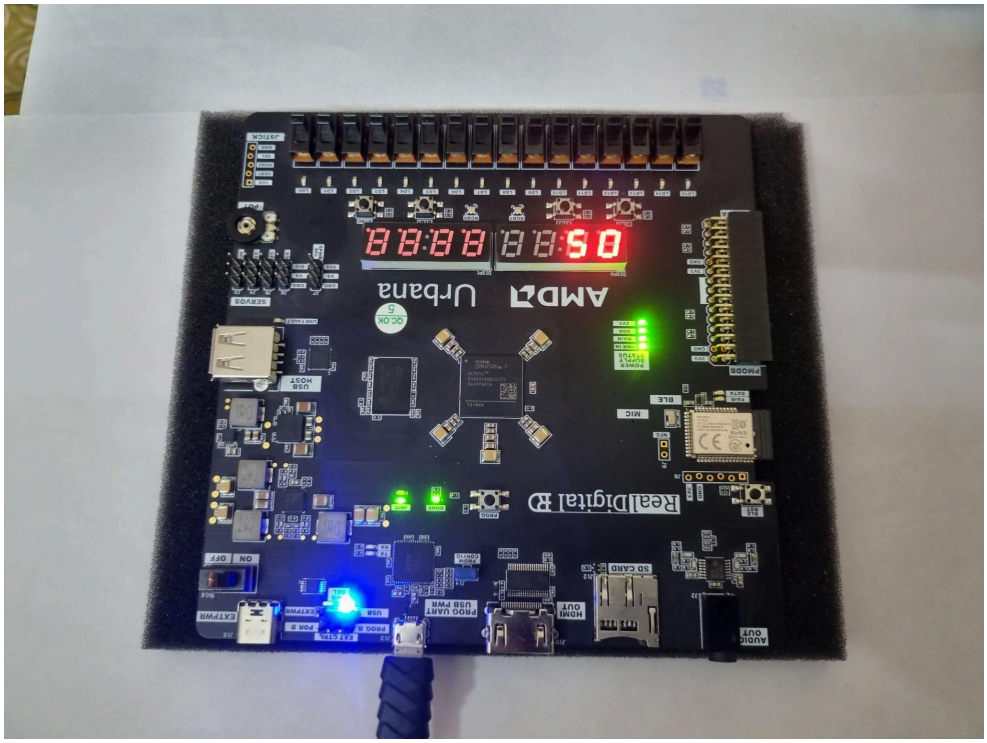
```
# Clock Pin
set_property -dict {PACKAGE_PIN N15 IOSTANDARD LVCMOS33} [get_ports {clk}]
# Anode Pins (Active LOW)
set_property -dict {PACKAGE_PIN G6 IOSTANDARD LVCMOS25} [get_ports {an[0]}]
set_property -dict {PACKAGE_PIN H6 IOSTANDARD LVCMOS25} [get_ports {an[1]}]
set_property -dict {PACKAGE_PIN C3 IOSTANDARD LVCMOS25} [get_ports {an[2]}]
set_property -dict {PACKAGE_PIN B3 IOSTANDARD LVCMOS25} [get_ports {an[3]}]
# Segment Pins (Active LOW likely)
set_property -dict {PACKAGE_PIN E6 IOSTANDARD LVCMOS25} [get_ports {seg[0]}]
set_property -dict {PACKAGE_PIN B4 IOSTANDARD LVCMOS25} [get_ports {seg[1]}]
set_property -dict {PACKAGE_PIN D5 IOSTANDARD LVCMOS25} [get_ports {seg[2]}]
set_property -dict {PACKAGE_PIN C5 IOSTANDARD LVCMOS25} [get_ports {seg[3]}]
set_property -dict {PACKAGE_PIN D7 IOSTANDARD LVCMOS25} [get_ports {seg[4]}]
set_property -dict {PACKAGE_PIN D6 IOSTANDARD LVCMOS25} [get_ports {seg[5]}]
set_property -dict {PACKAGE_PIN C4 IOSTANDARD LVCMOS25} [get_ports {seg[6]}]
set_property -dict {PACKAGE_PIN B5 IOSTANDARD LVCMOS25} [get_ports {seg[7]}]
```

Urbana Board Outputs:






AMD
Urbana
RealDigital


AMD
Urbana
RealDigital

Experiment-8

Objective: To check password entered is correct or not by matching it with a preset password and display result on seven segment display

Tools Required: Xilinx's VIVADO FPGA software and Urbana board along with its pin description.

Theory:

Segment Patterns:

- `segment_pattern`: An array holding the patterns for displaying `0` and `1` on the 7-segment display.

Driving the 7-Segment Display

The display is controlled by a combination of segment and anode signals:

- **Segment Control (`seg`)**: 8-bit control signals that drive each of the 7 segments and the decimal point.
- **Anode Control (`an`)**: 4-bit control signals that enable the specific digit to be displayed.

To create a password locker using the Urbana board with Vivado, we'll follow these steps:

1. **Define the Preset Password**: This will be hard-coded in the Verilog code.
2. **Read User Input**: Use the switches on the Urbana board to input the password.
3. **Compare the Input**: Check the input against the preset password.
4. **Display the Result**: Show `1` on the seven-segment display if the password is correct, otherwise show `0`.

Practical Considerations

- Switch Debouncing: Physical switches may produce noise (bouncing) which can cause erratic behavior. A debouncing mechanism or filtering should be considered for a robust design.
- Dynamic Passwords: For a more complex design, you could implement a system that allows the user to set or change the password dynamically.

Verilog code:

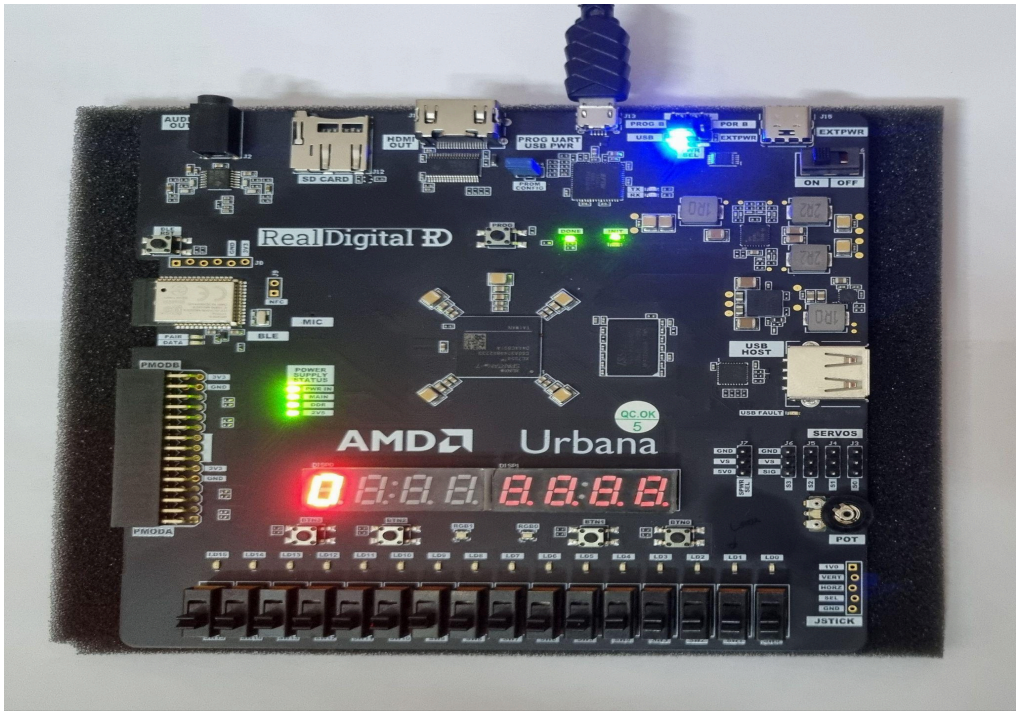
```
module passwordlocker(
    input clk,                  // Clock signal
    input [3:0] switches,        // 4 switches for input
    output reg [7:0] seg,        // 7-segment display segments
    output [3:0] an              // 7-segment display anodes
);
   // Preset password
    reg [3:0] preset_password = 4'b1010;  // Example password: 1010 (binary for
10 in decimal)
       // Digit patterns for 7-segment display
    reg [7:0] segment_pattern [1:0];  // Segment patterns for 0 and 1

// Initialize segment patterns
    initial begin
        segment_pattern[0] = 8'b11000000;  // Display 0
        segment_pattern[1] = 8'b11111001;  // Display 1
    end
   // Always enabled anode
    assign an = 4'b1110; // Enable the first digit (adjust as necessary for your
board setup)
   always @(posedge clk) begin
        if (switches == preset_password) begin
            seg <= segment_pattern[1];  // Display 1 if password is correct
        end else begin
            seg <= segment_pattern[0];  // Display 0 if password is incorrect
        end
    end
endmodule
```

Constraint File:

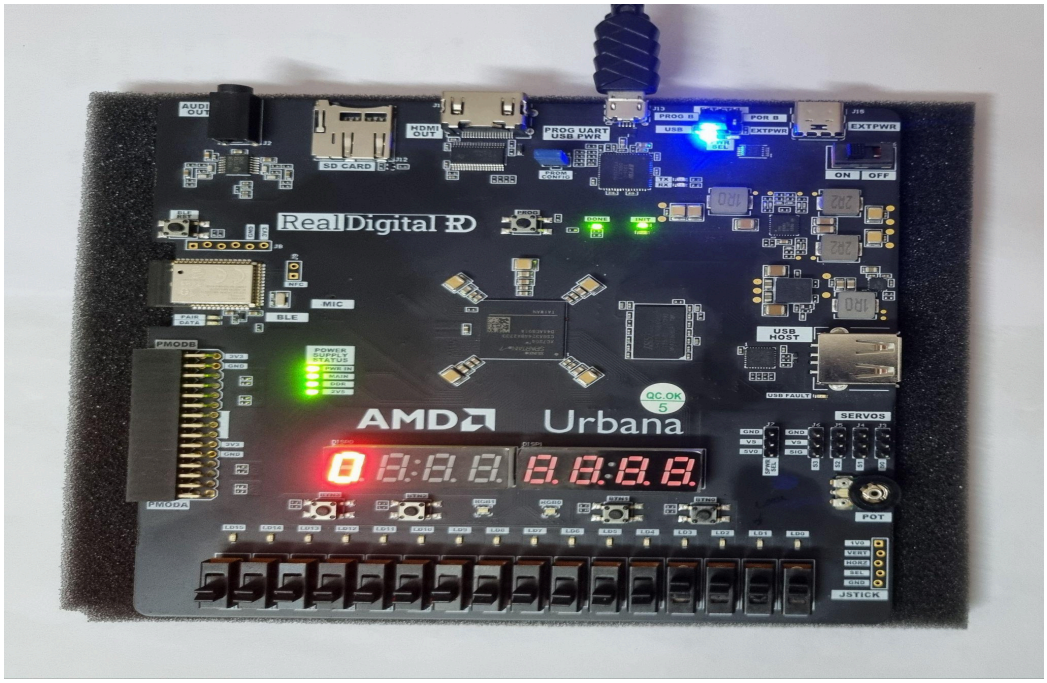
```
## Clock Pin
set_property -dict {PACKAGE_PIN N15 IOSTANDARD LVCMOS33} [get_ports clk]
create_clock -add -name sys_clk_pin -period 10.000 -waveform {0 5} [get_ports clk]

## Switches (input for password)
set_property -dict {PACKAGE_PIN G1 IOSTANDARD LVCMOS33} [get_ports
switches[0]];  # Switch 1
set_property -dict {PACKAGE_PIN F2 IOSTANDARD LVCMOS33} [get_ports
switches[1]];  # Switch 2
set_property -dict {PACKAGE_PIN F1 IOSTANDARD LVCMOS33} [get_ports
switches[2]];  # Switch 3
set_property -dict {PACKAGE_PIN E2 IOSTANDARD LVCMOS33} [get_ports
switches[3]];  # Switch 4
## Anode Pins (Active LOW)
set_property -dict {PACKAGE_PIN G6 IOSTANDARD LVCMOS33} [get_ports an[0]];  #
Anode 1
set_property -dict {PACKAGE_PIN H6 IOSTANDARD LVCMOS33} [get_ports an[1]];  #
Anode 2
set_property -dict {PACKAGE_PIN C3 IOSTANDARD LVCMOS33} [get_ports an[2]];  #
Anode 3
set_property -dict {PACKAGE_PIN B3 IOSTANDARD LVCMOS33} [get_ports an[3]];  #
Anode 4
## Segment Pins (Active LOW)
set_property -dict {PACKAGE_PIN E6 IOSTANDARD LVCMOS33} [get_ports seg[0]];  #
Segment a
set_property -dict {PACKAGE_PIN B4 IOSTANDARD LVCMOS33} [get_ports seg[1]];  #
Segment b
set_property -dict {PACKAGE_PIN D5 IOSTANDARD LVCMOS33} [get_ports seg[2]];  #
Segment c
set_property -dict {PACKAGE_PIN C5 IOSTANDARD LVCMOS33} [get_ports seg[3]];  #
Segment d
set_property -dict {PACKAGE_PIN D7 IOSTANDARD LVCMOS33} [get_ports seg[4]];  #
Segment e
set_property -dict {PACKAGE_PIN D6 IOSTANDARD LVCMOS33} [get_ports seg[5]];  #
Segment f
set_property -dict {PACKAGE_PIN C4 IOSTANDARD LVCMOS33} [get_ports seg[6]];  #
Segment g
set_property -dict {PACKAGE_PIN B5 IOSTANDARD LVCMOS33} [get_ports seg[7]];  #
Decimal Point (dp)
```

Urbana Board output:

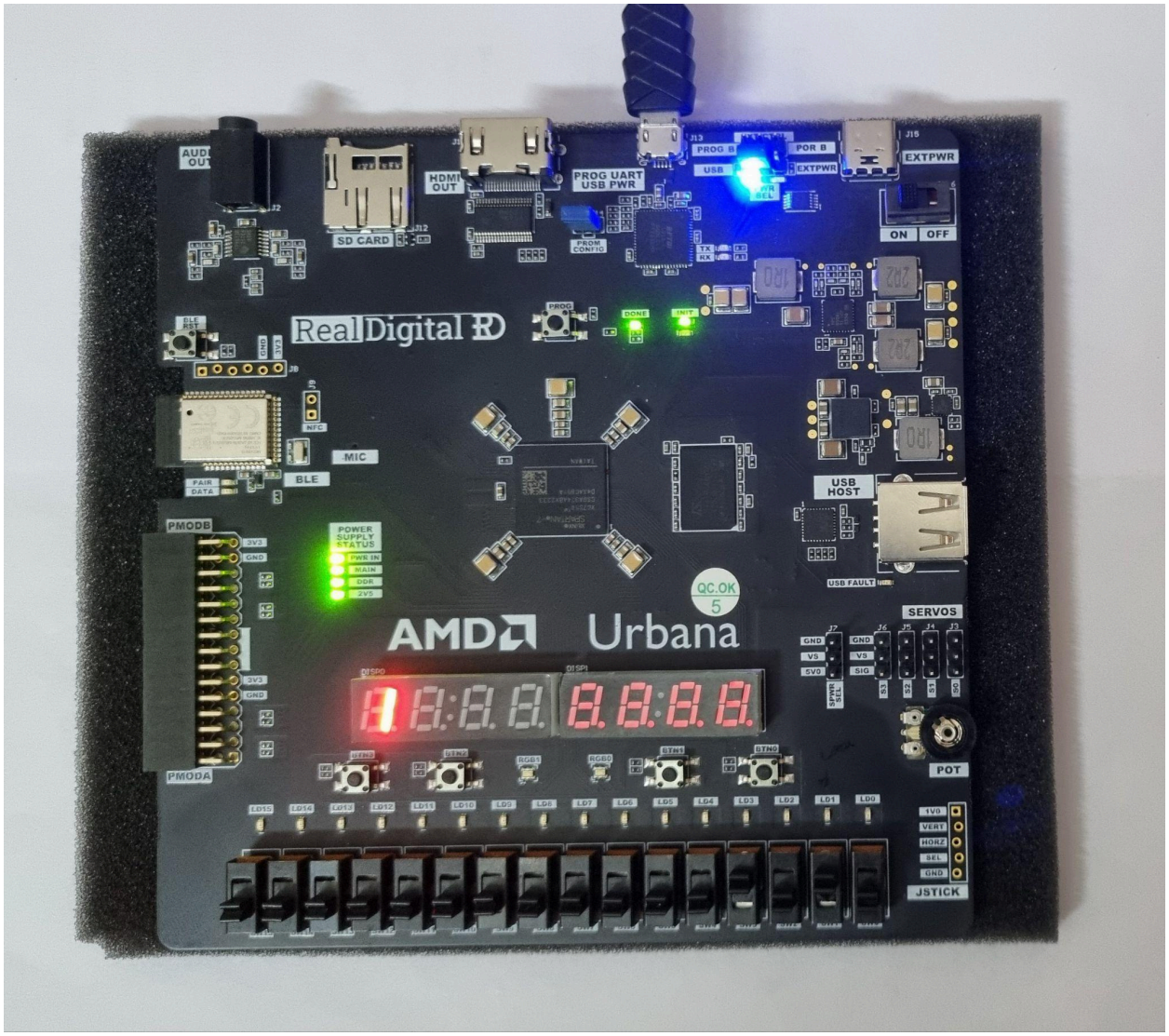
1. Initially when no password entered:-



2. When entered wrong password eg- 1111



3. When entered correct password- 1010



The seven segment display 1 after getting correct password.

INTRODUCTION

What is object detection:-

Object detection is computer vision technology that involves identifying and localizing objects of interest within an image or a video.

Its a challenging task as it involves not only the recognizing the presence of an object but can also the size and location of object within an image.

Some key points:-

Goal: Object detection aims to find and locate objects of interest in an image or video. This goes beyond just classifying what kind of object it is, but also pinpointing its whereabouts within the image.

https://www.analyticsvidhya.com/blog/2022/03/a-basic-introduction-to-object-detection/

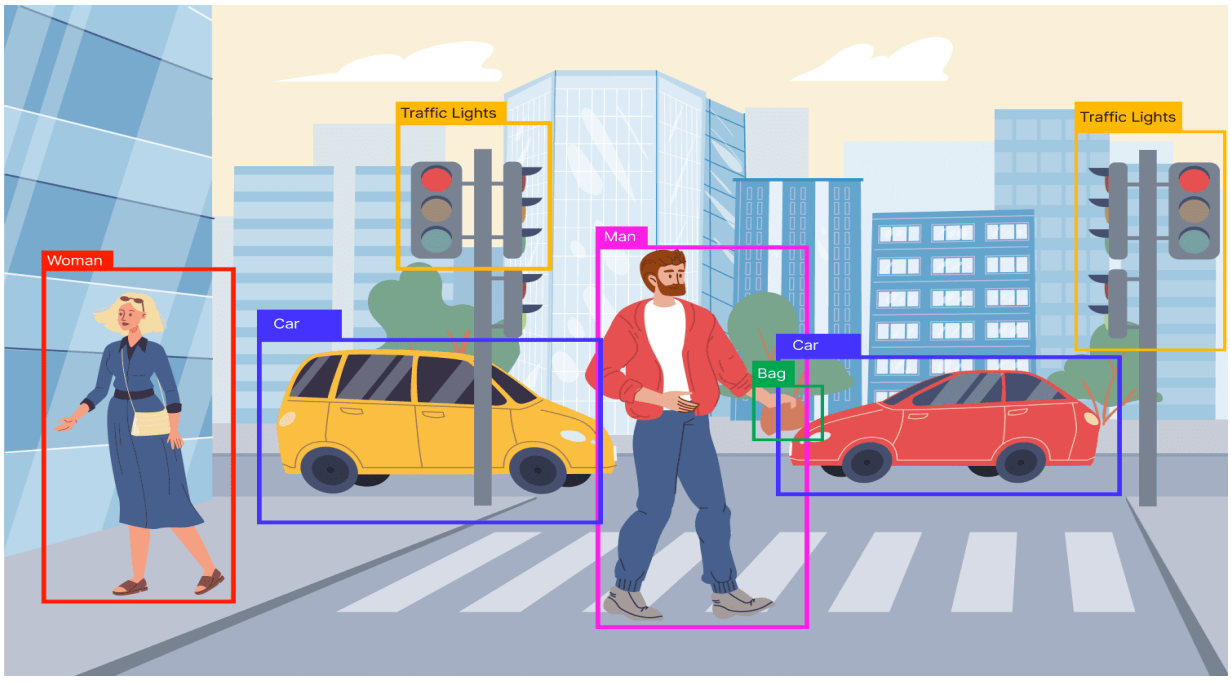
Applications: Object detection is a crucial technology in computer vision and has a wide range of applications. Some examples include self-driving cars (detecting lanes and pedestrians), video surveillance systems, and image retrieval systems.

https://blog.roboflow.com/object-detection/

Techniques: Object detection algorithms leverage machine learning or deep learning to identify objects. These algorithms are trained on massive amounts of data to be able to recognize and locate objects accurately. https://www.mathworks.com/help/vision/object-detection.html

Outputs: When an object detection system runs on an image, it typically outputs two things: bounding boxes around the detected objects. The bounding box indicates the location of the object, and represents how certain the system is about the detection.

https://blog.roboflow.com/object-detection/



Object Detection using open-cv

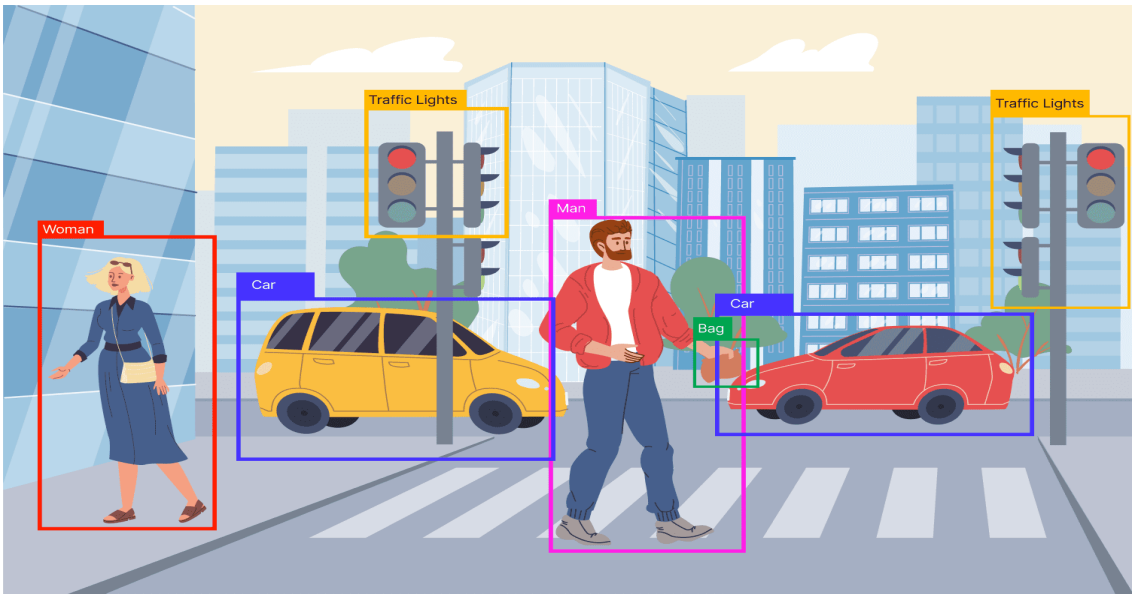
OpenCV (Open Source Computer Vision Library) is a powerful and widely used library for real-time computer vision applications. It offers a comprehensive set of functions for image and video processing, object detection, feature extraction, and more.

Key Features in Python open cv:

- Ease of Use: OpenCV's Python bindings make it accessible for programmers with experience in Python. The syntax is clear and concise, allowing you to focus on the computer vision task at hand.
- Rich Functionality: OpenCV provides a vast array of algorithms and functions for various computer vision tasks. These include:
    - Image loading, display, and manipulation (resizing, cropping, color space conversions)
    - Feature detection and description (finding and characterizing keypoints in images)
    - Object detection and recognition (identifying and classifying objects in images and videos)

Advantages:

- Higher accuracy and can detect a broader variety of objects.
- More adaptable to variations in object appearance.



Experiment

Objective:- To do object detection using open cv

Tool used:- Python and its libraries

Prerequisite:- We have to install open-cv in python jupyter

```
!pip install opencv-python
```

```
pip show opencv-python
```

```
Version: 4.10.0.84
Summary: Wrapper package for OpenCV python bindings.
Home-page: https://github.com/opencv/opencv-python
Author:
Author-email:
License: Apache 2.0
Location: C:\Users\Lenovo\AppData\Roaming\Python\Python311\site-packages
Requires: numpy, numpy, numpy, numpy, numpy
Required-by:
```

We have to download 3 files from the given link drive files named:- labels, frozen and mobilenet

Project Overview:- In this we will do object detection by reading an image and we can detect around 80 objects from the image.
['person', 'bicycle', 'car', 'motorbike', 'aeroplane', 'bus', 'train', 'truck', 'boat', 'traffic light', 'fire hydrant', 'stop sign', 'parking meter', 'bench', 'bird', 'cat', 'dog', 'horse', 'sheep', 'cow', 'elephant', 'bear', 'zebra', 'giraffe', 'backpack', 'umbrella', 'handbag', 'tie', 'suitcase', 'frisbee', 'skis', 'snowboard', 'sports ball', 'kite', 'baseball bat', 'baseball glove', 'skateboard', 'surfboard', 'tennis racket', 'bottle', 'wine glass', 'cup', 'fork', 'knife', 'spoon', 'bowl', 'banana', 'apple', 'sandwich', 'orange', 'broccoli', 'carrot', 'hot dog', 'pizza', 'donut', 'cake', 'chair', 'sofa', 'pottedplant', 'bed', 'diningtable', 'toilet', 'tvmonitor', 'laptop', 'mouse', 'remote', 'keyboard', 'cell phone', 'microwave', 'oven', 'toaster', 'sink', 'refrigerator', 'book', 'clock', 'vase', 'scissors', 'teddy bear', 'hair drier', 'toothbrush']

Theory:- Object detection is a crucial technology in computer vision and has a wide range of applications. Object detection algorithms leverage machine learning or deep learning to identify objects.
Object detection is widely used in a variety of applications, such as autonomous vehicles, surveillance systems, and medical imaging. It is also used for tasks such as face detection, pedestrian detection, and traffic sign recognition.

Config_file and frozen_model:-These two variables define the configuration file and frozen model weights needed to use a pre-trained MobileNet v3 Large object detection model.
The configuration file describes the model's architecture, and the frozen model contains the trained weights for object detection.

Procedure:- Use the link for full code:- code

Step 1:- We have to import open cv library for using it . `matplotlib.pyplot` module, commonly referred to as `plt`, from the Matplotlib library. Matplotlib is a versatile library for creating visualizations of data, including plots (line plots, scatter plots, histograms, etc.). The `plt` submodule provides a convenient interface for creating static plots.

```
:  import cv2
   import matplotlib.pyplot as plt
```

Step 2:-

```
config_file="mobilenet_v3_large_coco_2020_01_14.pbtxt"
frozen_model="frozen_inference_graph.pb"
```

```
model=cv2.dnn_DetectionModel(frozen_model,config_file)
```

```
classLabels=[]
file_name='labels.txt'
with open(file_name,'rt') as fpt:
    classLabels= fpt.read().rstrip('\n').split('\n')
```

`config_file`: The variable name indicates it holds the configuration file path.
`"mobilenet_v3_large_coco_2020_01_14.pbtxt"`: The filename suggests several details:

- `mobilenet_v3_large`: The model architecture (MobileNet v3 Large).
- `coco_2020_01_14`: Likely trained on the COCO (Common Objects in Context) dataset in 2020 (January 14th).
- `.pbtxt`: Protocol Buffer text format, defining the model's architecture.

`frozen_model`: The variable name indicates it holds the frozen model path.
`"frozen_inference_graph.pb"`: Likely a TensorFlow frozen graph in `.pb` format. This file contains the trained weights and biases of the model's layers.

`model`: The variable name suggests it holds the created object representing the loaded model.
`cv2.dnn_DetectionModel`: This is a function from OpenCV's DNN module used for loading pre-trained object detection models.
`(frozen_model, config_file)`: These are the arguments passed to the function. It loads the frozen model weights (`frozen_model`) and the model configuration (`config_file`) to create the detection model object

This code prepares for using a pre-trained object detection model (MobileNet v3 Large in this case) for image processing tasks. The loaded model and the retrieved class labels will be used in subsequent code that can detect 80 objects in images written in text file called labels.

Step 3:-

```
In [34]: model.setInputSize(350,350)
         model.setInputScale(1.0/127.5)
         model.setInputMean((127.5,127,5,127.5))
         model.setInputSwapRB(True)

Out[34]: < cv2.dnn.Model 000002131E9459F0>
```

This line sets the expected input image size for the model. In this case, the model expects images to be resized to 350 pixels wide and 350 pixels high before feeding them into the network for prediction.
Defines how pixel values in the input image are scaled before feeding them into the model. Here, each pixel value is divided by 127.5. This normalization step is often used with pre-trained models to bring pixel values into a specific range that the model's internal calculations expect.

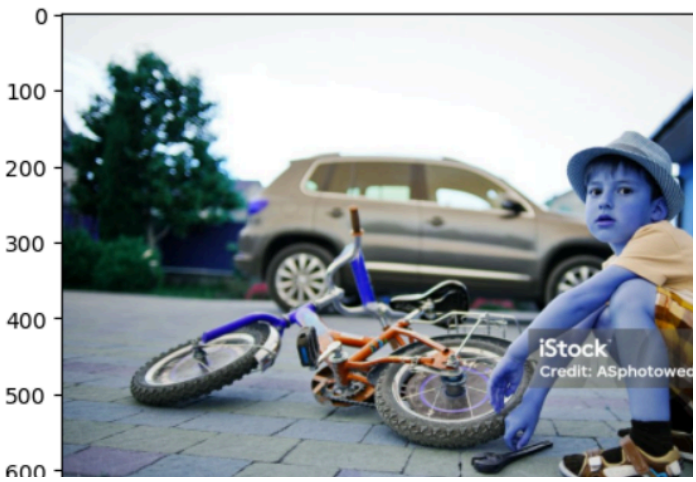
By resizing, scaling, subtracting the mean, and potentially swapping channels, the input data is prepared for accurate object detection.

The specific values used in these lines (image size, scaling factor, mean values) might vary depending on the pre-trained model you're using. It's essential to consult the model's documentation for recommended input pre-processing steps

Step4:- Now use any image to detect objects present in it.
To read that image use

```
In [35]: img=cv2.imread('boy.jpg')
         plt.imshow(img)
```

For eg:- Here i used image whose name is boy



Step5:- `ClassIndex`: This variable will hold a list of integers representing the predicted class labels for the detected objects. The index of each integer corresponds to the class label in the `classLabels` list you created earlier (assuming the model outputs class IDs).

```
: ClassIndex, confidence, bbox=model.detect(img , confThreshold =0.5)
```

```
: print(ClassIndex)
```

```
[3 1 4]
```

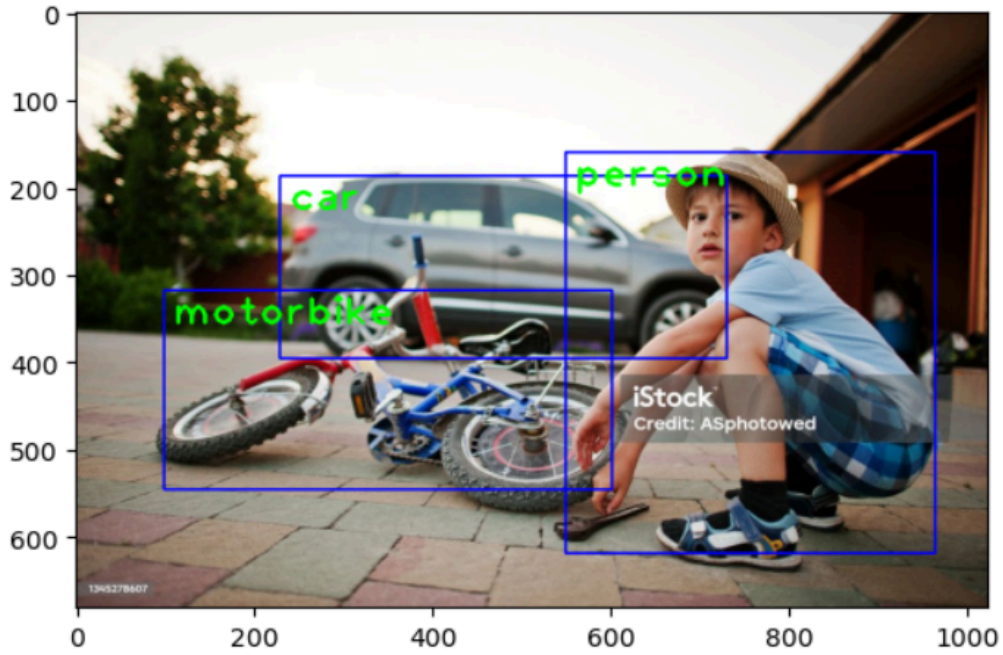
`confidence`: This variable will hold a list of floating-point numbers representing the confidence scores for the detected objects. Each confidence score indicates how likely the model believes an object belongs to the predicted class.

model.detect(img, confThreshold=0.5):

- `model`: This refers to the `cv2.dnn_DetectionModel` object you created earlier, which encapsulates the loaded pre-trained object detection model.
- `img`: This is the input image on which you want to detect objects. It's likely a NumPy array representing the image data in BGR format (Blue, Green, Red).
- `confThreshold=0.5`: This is a keyword argument that sets the confidence threshold for detected objects. Any object with a predicted confidence score lower than 0.5 in this case will be discarded. You can adjust this threshold based on your desired balance between accuracy (higher threshold) and detection rate (lower threshold).

Step6:- The object will get detected with a frame and and name on it.

```
In [39]: plt.imshow(cv2.cvtColor(img,cv2.COLOR_BGR2RGB))
```



Screenshot of full code:-

In [1]:
```
!pip install opencv-python
```

```
Defaulting to user installation because normal site-packages is not writeable
Requirement already satisfied: opencv-python in c:\users\lenovo\appdata\roaming\python\python311\site-packages (4.10.0.84)
Requirement already satisfied: numpy>=1.21.2 in c:\programdata\anaconda3\lib\site-packages (from opencv-python) (1.24.3)
```

In [2]:
```
import cv2
import matplotlib.pyplot as plt
```

In [3]:
```
pip show opencv-python
```

```
Name: opencv-pythonNote: you may need to restart the kernel to use updated packages.

Version: 4.10.0.84
Summary: Wrapper package for OpenCV python bindings.
Home-page: https://github.com/opencv/opencv-python
Author:
Author-email:
License: Apache 2.0
Location: C:\Users\Lenovo\AppData\Roaming\Python\Python311\site-packages
Requires: numpy, numpy, numpy, numpy, numpy
Required-by:
```

In [4]:
```
config_file="mobilenet_v3_large_coco_2020_01_14.pbtxt"
frozen_model="frozen_inference_graph.pb"
```

In [7]:
```
model=cv2.dnn_DetectionModel(frozen_model,config_file)
```

In [8]:
```
classLabels=[]
file_name='labels.txt'
with open(file_name,'rt') as fpt:
    classLabels= fpt.read().rstrip('\n').split('\n')
```

In [9]:
```
print(classLabels)
```

```
['person', 'bicycle', 'car', 'motorbike', 'aeroplane', 'bus', 'train', 'truck', 'boat', 'traffic light', 'fire hydrant', 'stop
sign', 'parking meter', 'bench', 'bird', 'cat', 'dog', 'horse', 'sheep', 'cow', 'elephant', 'bear', 'zebra', 'giraffe', 'backpa
ck', 'umbrella', 'handbag', 'tie', 'suitcase', 'frisbee', 'skis', 'snowboard', 'sports ball', 'kite', 'baseball bat', 'baseball
glove', 'skateboard', 'surfboard', 'tennis racket', 'bottle', 'wine glass', 'cup', 'fork', 'knife', 'spoon', 'bowl', 'banana',
'apple', 'sandwich', 'orange', 'broccoli', 'carrot', 'hot dog', 'pizza', 'donut', 'cake', 'chair', 'sofa', 'pottedplant', 'be
d', 'diningtable', 'toilet', 'tvmonitor', 'laptop', 'mouse', 'remote', 'keyboard', 'cell phone', 'microwave', 'oven', 'toaste
r', 'sink', 'refrigerator', 'book', 'clock', 'vase', 'scissors', 'teddy bear', 'hair drier', 'toothbrush']
```

In [10]:
```
print(len(classLabels))
```

```
80
```

In [34]:
```
model.setInputSize(350,350)
model.setInputScale(1.0/127.5)
model.setInputMean((127.5,127,5,127.5))
model.setInputSwapRB(True)
```

Out[34]: < cv2.dnn.Model 000002131E9459F0>

In [35]:
```
img=cv2.imread('boy.jpg')
plt.imshow(img)
```

Out[35]:
```
<matplotlib.image.AxesImage at 0x2131e9d5f90>
```

In [36]:
```
ClassIndex, confidence, bbox=model.detect(img , confThreshold =0.5)
```

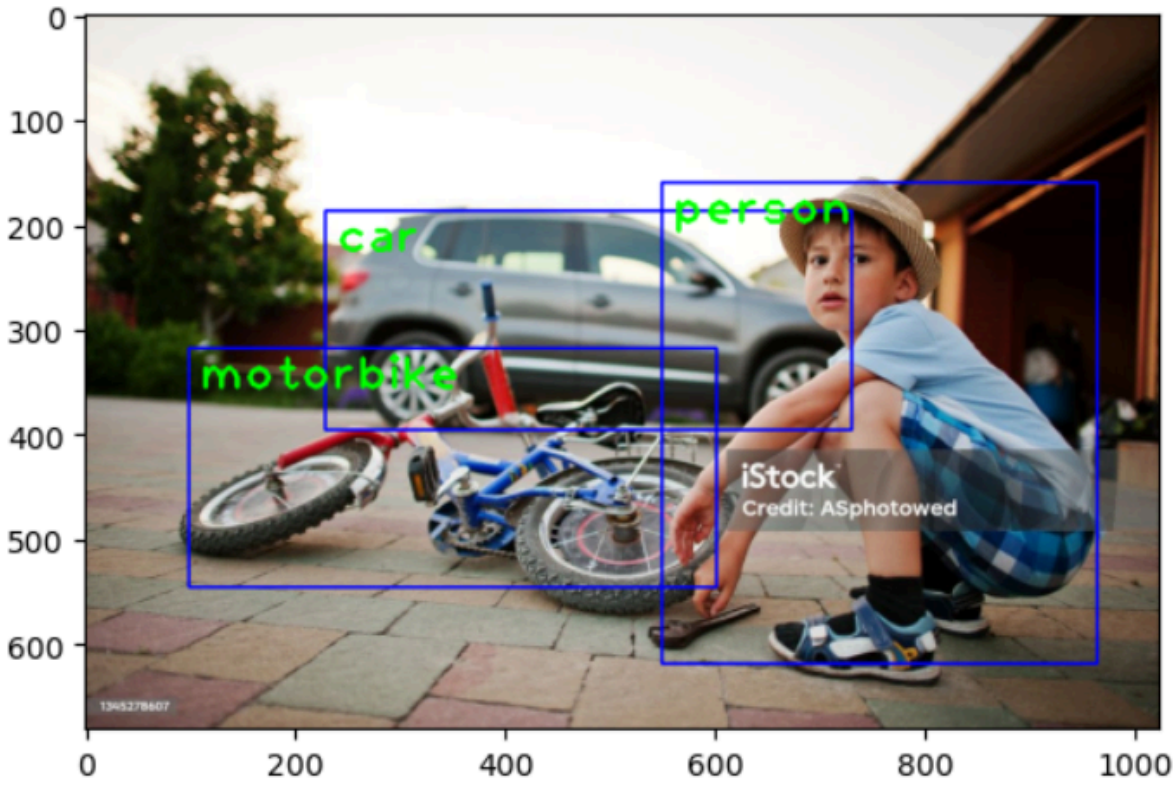
In [37]:
```
print(ClassIndex)
```

```
[3 1 4]
```

In [38]:
```
font_scale=3
font=cv2.FONT_HERSHEY_PLAIN
for ClassInd, conf, boxes in zip(ClassIndex.flatten(),confidence.flatten(), bbox):
    cv2.rectangle(img, boxes,(255,0,0),2 )
    cv2.putText(img, classLabels[ClassInd-1], (boxes[0] +10, boxes[1] +40), font, fontScale = font_scale,color=(0,255,0),
```

```
plt.imshow(cv2.cvtColor(img,cv2.COLOR_BGR2RGB))
```

```
<matplotlib.image.AxesImage at 0x2131130df90>
```



References

[1] Xilinx. PYNQ: Python Productivity for Zynq, https://www.pynq.io/

[2] TUL Corporation, “TUL PYNQ-Z2 Board Based on Xilinx Zynq SoC.” 2020, url: http://www.tul.com.tw/ProductsPYNQ-Z2.html.

[3] Object Detection Tutorial: https://sceweb.sce.uhcl.edu/xiaokun/doc/OpenIC/OpenProject/SaralaCaps

[4] https://pynq.readthedocs.io/en/v2.5/pynq_overlays/pynqz2/pynqz2_base_

[5] https://www.realdigital.org/doc/496fed57c6b275735fe24c85de5718c2#ddr

[6] https://drive.google.com/drive/folders/1GrFlJNaQ9eAKcFo9MdBa7lgC0xxl-PRw

[7] https://pynq.readthedocs.io/en/v2.5/pynq_overlays/pynqz2/pynqz2_base_