

DAILY ASSESSMENT FORMAT

Date:	1/6/2020	Name:	Shilpa S
Course:	Digital design using HDL	USN:	4AL14EC078
Topic:	Industry applications of FPGA, FPGA Business Fundamentals FPGA vs ASIC Design flow FPGA Basics-A look under the hood.	Semester & Section:	8 th sem A sec
Github Repository:	Shilpa_online		

FORENOON SESSION DETAILS

Image of session

Why FPGA?



Programmable Solutions Group



4

Reduced Time-to-Market

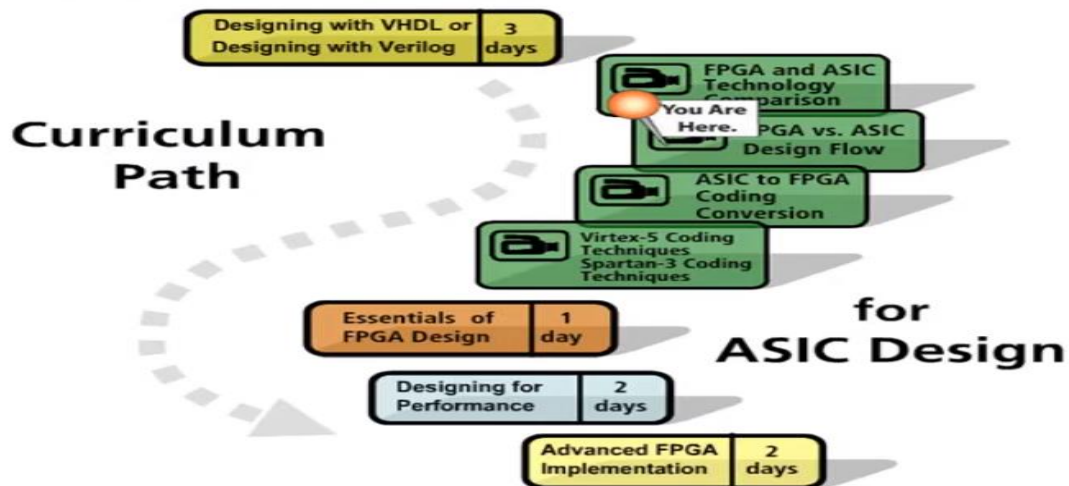
Developing and prototyping on FPGAs can reduce TTM, especially in emerging markets where standards have not yet been defined



Programmable Solutions Group

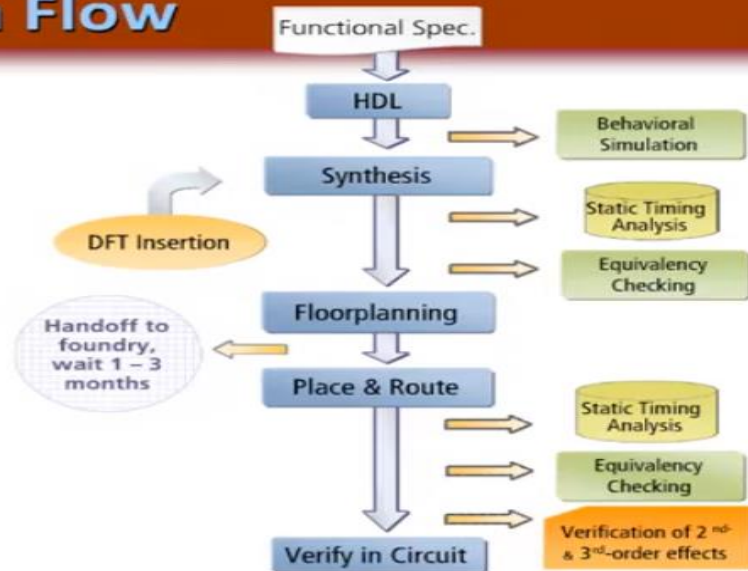


7



ASIC Design Flow

- ASIC tools are generally driven by scripts



Report –

ASIC:

ASIC stands for Application-Specific Integrated Circuit. Furthermore, as the name implies, it is a chip that serves the purpose for which it has been designed and does not permit reprogramming or modification. Which, in turn, means that it cannot perform another function or execute another application once programming is complete. Since the ASIC's design is for a specific function, this determines how the chip receives its programming. The programming process itself consists of drawing the resultant circuit permanently into the silicon. In terms of applications, ASIC chip technology is in

use in electronic devices such as laptops, smartphones, and TVs, to give you an idea of the scope of their use.

FPGA:

Field Programmable Gate Array or FPGA is in direct competition with ASIC chip technology. Also, FPGA is, in essence, a chip that can be programmed and reprogrammed to perform numerous functions at any single point in time. Furthermore, a single chip is comprised of thousands of units called logic blocks, that are linked with programmable interconnects. The FPGA's circuit is made by connecting several configurable blocks, and it has a rigid internal structure. In summary, an FPGA is essentially a programmable version of an ASIC.

Overall, the FPGA affords general functionality that allows programming to your specifications. However, like most things in life, there are side effects of FPGA's versatility. In this case, it is an increased cost, increased internal delay, and limited analog functionality.

FPGA versus ASIC: A Side-By-Side Comparison

NRE: NRE stands for Non-Recurring Engineering costs. As you can imagine, with the words recurring and costs, in the same sentence, every business is concerned when they hear those two words. So, it is safe to say that this is an essential deciding factor. Moreover, in the case of ASIC, this is exceptionally high, whereas, with FPGA, it is nearly non-existent.

However, in the grand scheme, the total cost gets lower and lower the more significant the quantity you require in terms of ASIC. Furthermore, FPGA can cost you more overall since its individual costs are higher per unit than ASIC.

Design Flow: Every engineer and PCB designer prefer a more trouble-free and simplistic design process. Just because what you do is complex, does not mean that you want the process itself to be complicated. Therefore, in terms of the simplicity of design flow, FPGA is hands down less complicated than ASIC.

This is due to the FPGA's flexibility, versatility, shorter time to market, and the fact that it is reprogrammable. Whereas, with ASIC, it is more involved in terms of design flow because it is not reprogrammable, and it requires costly dedicated EDA tools for the design process.

Performance and Efficiency: In terms of performance, ASICs outperform FPGAs by a small margin, primarily due to lower power consumption and the various possible functionalities that you can layer onto a single chip. Also, FPGA has a more rigid internal structure, whereas, with an ASIC, you can design it to excel in power consumption or speed.

Cost: Even with the increased NRE cost, ASICs are thought to be more cost-effective, all things considered as compared to FPGAs, which are only profitable when developed in smaller quantities.

Power Consumption: As I mentioned previously, ASICs require less power and thus provide a better option than the higher power consumption FPGA. Especially with electronic devices that are battery operated.

Size: In terms of size, it is a matter of physics. With an ASIC, its design is for one functionality; therefore, it consists of precisely the number of gates required for the desired application. However, with FPGA's multifunctionality, a single unit will be significantly larger, because of its internal structure and a specific size that you cannot change.

Time to Market: Also, as mentioned earlier, FPGA affords a faster time to market than ASIC due to its simplicity in terms of the design flow. Moreover, ASICs also require layouts, back end processes, and advanced verification, all of which are time-consuming.

Configuration: Overall, the most apparent difference between FPGA and ASIC is programmability. Therefore, the logical conclusion here is FPGA offers more options in terms of flexibility. FPGAs are not only flexible, but they also provide "hot-swappable" functionality that allows modification even while in use.

Operating Frequency: In terms of design specifications, FPGAs have limited operating frequencies. This is one of those side effects of its flexibility (reprogrammable). However, with ASICs more focused approach to functionality, it can operate at higher frequencies.

Analog Designs: If your designs are analog, you will not be able to use FPGAs. However, in the case of ASICs, you can utilize analog hardware like RF blocks (Bluetooth and WiFi), analog to digital converters, and more to facilitate your analog designs.

FPGA versus ASIC in Terms of Applications

First of all, it is a fact that flexibility is FPGA's strong suit, which makes it ideal for devices and applications that require frequent modification, like prototyping. However, ASICs are best suited for more permanent applications that do not require modification. Overall, if you are designing a mass-production type project, the ASIC is the more cost-effective route to go, provided your devices do not require configuring or reconfiguring.

The rivalry between FPGA and ASIC can be decided by your design type (analog or digital), configuration requirements, and budget. Regardless of choice, the most important deciding factor should be your design needs, and if you are still on the fence, try simulation first.

Regardless of your choice of FPGA or ASIC for your designs, Cadence's suite of design and analysis tools. Allegro is more than capable of providing you with the design environment to proliferate any FPGA, ASIC, or other designs.

Strengths / best suited for:

Much of what will make it worthwhile to utilize an FPGA comes down to the low-level functions being performed within the device. There are four processing/algorithm attributes defined below that FPGAs are generally well-suited for. While just one of these needs may drive you toward an FPGA, the more of these your application has, the more an FPGA-based solution will appeal.

Parallel processes – if you need to process several input channels of information (e.g. many simultaneous A/D channels) or control several channels at once (e.g. several PID loops).

High data-to-clock-rate-ratio – if you've got lots of calculations that need to be executed over and over and over again, essentially continuously. The advantage is that you're not tying up a centralized processor. Each function can operate on its own.

Large quantities of deterministic I/O – the amount of determinism that you can achieve with an FPGA will usually far surpass that of a typical sequential processor. If there are too many operations within your required loop rate on a sequential processor, you may not even have enough time to close the loop to update all of the I/O within the allotted time.

Signal processing – includes algorithms such as digital filtering, demodulation, detection algorithms, frequency domain processing, image processing, or control algorithms.

Weaknesses / not optimal for:

With any significant benefit, there's often times a corresponding cost. In the case of FPGAs, the following are generally the main disadvantages of FPGA-based solutions.

Complex calculations infrequently – If the majority of your algorithms only need to make a computation less than 1% of the time, you've generally still allocated those logic resources for a particular function (there are exceptions to this), so they're still sitting there on your FPGA, not doing anything useful for a significant amount of time.

Sorting/searching – this really falls into the category of a sequential process. There are algorithms that attempt to reduce the number of computations involved, but in general, this is a sequential process that doesn't easily lend itself to efficient use of parallel logical resources. Check out the sorting section here and check out this article here for some more info.

Floating point arithmetic – historically, the basic arithmetic elements within an FPGA have been fixed-point binary elements at their core. In some cases, floating point math can be achieved (see Xilinx FP Operator and Altera FP White Paper), but it will chew up a lot of logical resources. Be mindful of single-precision vs double-precision, as well as deviations from standards. However, this FPGA weakness appears to be starting to fade, as hardened floating-point DSP blocks are starting to be embedded within some FPGAs (see Altera Arria 10 Hard Floating Point DSP Block).

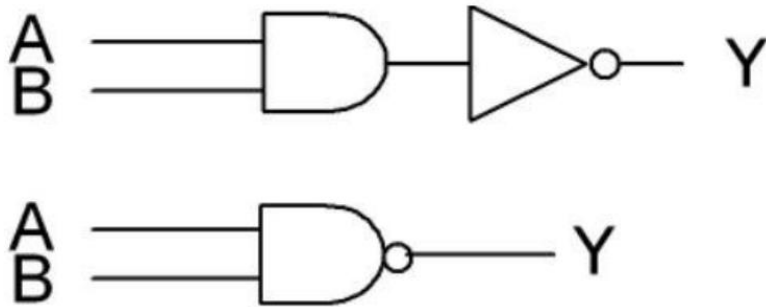
Very low power – Some FPGAs have low power modes (hibernate and/or suspend) to help reduce current consumption, and some may require external mode control ICs to get the most out of this. Check out an example low power mode FPGA here. There are both static and dynamic aspects to power consumption. Check out these power estimation spreadsheets to start to get a sense of power utilization under various conditions. However, if low power is critical, you can generally do better power-wise with low-power architected microprocessors or microcontrollers.

Very low cost – while FPGA costs have come down drastically over the last decade or so, they are still generally more expensive than sequential processors.

TASK FOR DAY 1

Verilog code for NAND gate – All modeling styles

GATE LEVEL MODELLING



```
module NAND_2_gate_level (output Y, input A, B);  
  wire Yd;  
  and (Yd,A,B);  
  not(Y,Yd);  
endmodule
```

DATA FLOW MODELLING

```
module NAND_2_data_flow(output Y,input A,B);  
  assign Y = ~(A & B);  
endmodule
```

BEHAVIOURAL MODELLING

```
module NAND_2_behavioural(output reg Y,input A,B) ;  
  always @(A or B) begin  
    if( A == 1'b1 & B == 1'b1) begin  
      Y =1'b0;  
    end  
    else
```

```
Y = 1'b1;  
end  
endmodule
```

TESTBENCH OF THE NAND GATE USING VERILOG

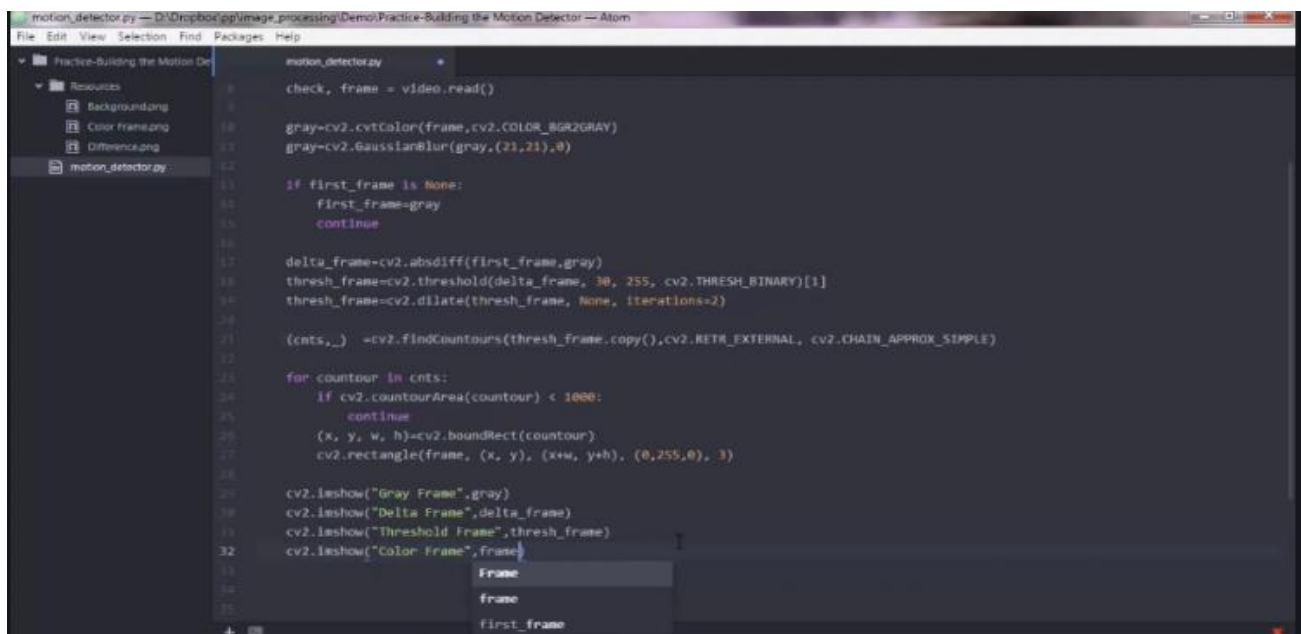
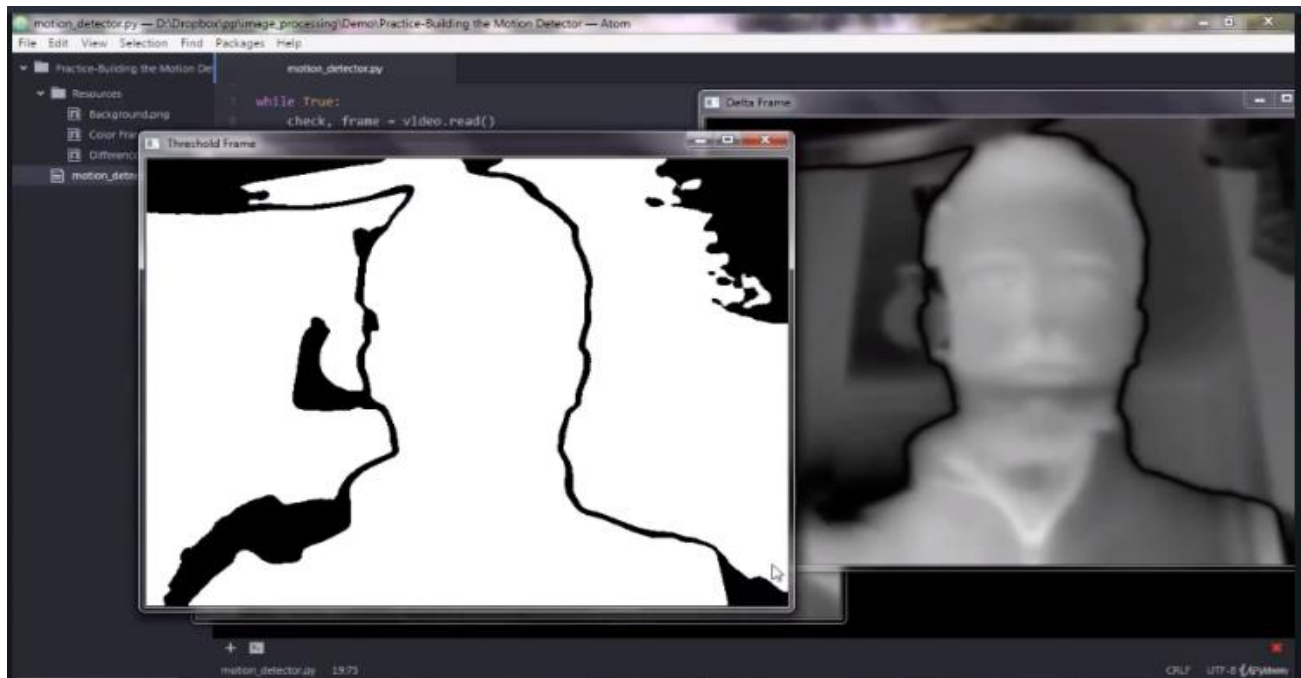
```
`include "NAND_2_behavioral.v"  
module NAND_2_behavioral_tb;  
reg A, B;  
wire Y;  
NAND_2_behavioral Indtance0 (Y, A, B);  
initial begin  
A = 0; B = 0;  
#1 A = 0; B = 1;  
#1 A = 1; B = 0;  
#1 A = 1; B = 1;  
end  
initial begin  
$monitor ("%t | A = %d| B = %d| Y = %d", $time, A, B, Y);  
$dumpfile("dump.vcd");  
$dumpvars();  
end  
endmodule
```

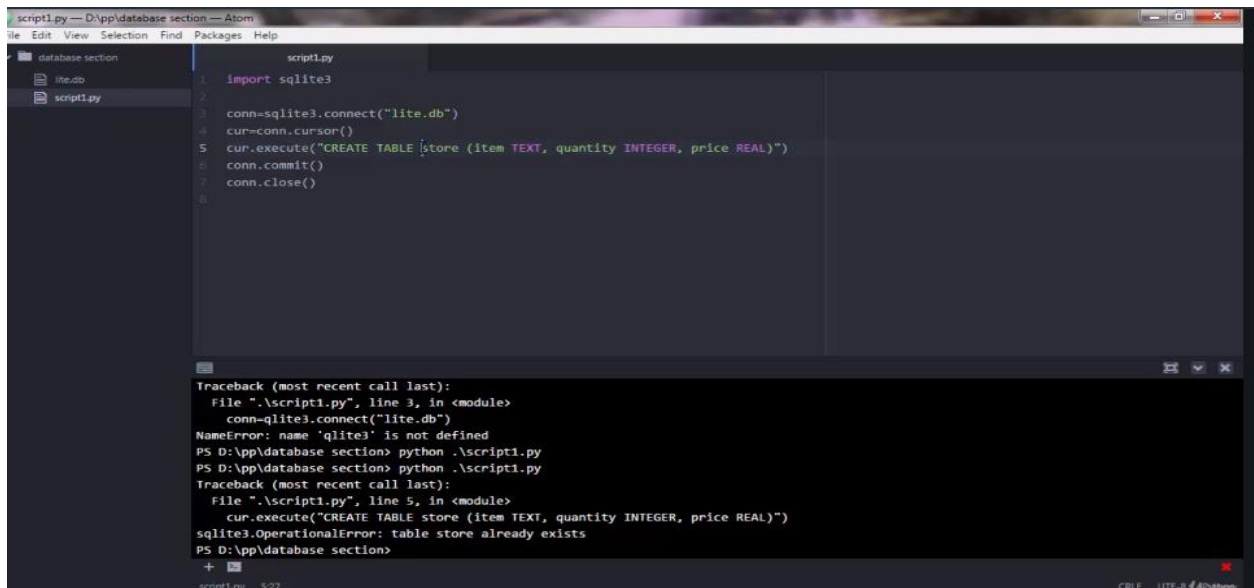

Date:1/6/2020
Course: Python
Topic:
Application 6:
Build a webcam
motion detector

Name: Shilpa S
USN:4AL14EC078
Sem :8th sem
Section: A sec

AFTERNOON SESSION DETAILS

SESSION IMAGE :





```
script1.py
1 import sqlite3
2
3 conn=sqlite3.connect("lite.db")
4 cur=conn.cursor()
5 cur.execute("CREATE TABLE store (item TEXT, quantity INTEGER, price REAL)")
6 conn.commit()
7 conn.close()
8

Traceback (most recent call last):
  File ".\script1.py", line 3, in <module>
    conn=sqlite3.connect("lite.db")
NameError: name 'sqlite3' is not defined
PS D:\pp\database section> python .\script1.py
PS D:\pp\database section> python .\script1.py
Traceback (most recent call last):
  File ".\script1.py", line 5, in <module>
    cur.execute("CREATE TABLE store (item TEXT, quantity INTEGER, price REAL)")
sqlite3.OperationalError: table store already exists
PS D:\pp\database section>
```

REPORT:

Code:

```
import cv2
```

```
import sys
```

```
cascPath = sys.argv[1]
```

```
faceCascade = cv2.CascadeClassifier(cascPath)
```

```
video_capture = cv2.VideoCapture(0)
```

```
while True:
```

```
    # Capture frame-by-frame
```

```
    ret, frame = video_capture.read()
```

```
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
```

```
    faces = faceCascade.detectMultiScale(
```

```
        gray,
```

```
        scaleFactor=1.1,
```

```
        minNeighbors=5,
```

```
minSize=(30, 30),
flags=cv2.cv.CV_HAAR_SCALE_IMAGE
)

# Draw a rectangle around the faces
for (x, y, w, h) in faces:
    cv2.rectangle(frame, (x, y), (x+w, y+h), (0, 255, 0), 2)

# Display the resulting frame
cv2.imshow('Video', frame)

if cv2.waitKey(1) & 0xFF == ord('q'):
    break
```

```
# When everything is done, release the capture
video_capture.release()
cv2.destroyAllWindows()
Now let's break it down...
```

```
import cv2
import sys
```

```
cascPath = sys.argv[1]
faceCascade = cv2.CascadeClassifier(cascPath)
```

This should be familiar to you. We are creating a face cascade, as we did in the image example.

```
video_capture = cv2.VideoCapture(0)
```

This line sets the video source to the default webcam, which OpenCV can easily capture.

NOTE: You can also provide a filename here, and Python will read in the video file. However, you need to have ffmpeg installed for that since OpenCV itself cannot decode compressed video. Ffmpeg acts as the front end for OpenCV, and, ideally, it should be compiled directly into OpenCV. This is not easy to do, especially on Windows.

```
while True:
```

```
    # Capture frame-by-frame
```

```
    ret, frame = video_capture.read()
```

Here, we capture the video. The `read()` function reads one frame from the video source, which in this example is the webcam. This returns:

1. The actual video frame read (one frame on each loop)
2. A return code

The return code tells us if we have run out of frames, which will happen if we are reading from a file. This doesn't matter when reading from the webcam, since we can record forever, so we will ignore it.

```
    # Capture frame-by-frame
```

```
    ret, frame = video_capture.read()
```

```
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
```

```
    faces = faceCascade.detectMultiScale(
```

```
        gray,
```

```
        scaleFactor=1.1,
```

```
        minNeighbors=5,
```

```
        minSize=(30, 30),
```

```
        flags=cv2.cv.CV_HAAR_SCALE_IMAGE
```

```
    )
```

```
    # Draw a rectangle around the faces
```

```
    for (x, y, w, h) in faces:
```

```
        cv2.rectangle(frame, (x, y), (x+w, y+h), (0, 255, 0), 2)
```

```
    # Display the resulting frame
```

```
    cv2.imshow('Video', frame)
```

Again, this code should be familiar. We are merely searching for the face in our captured frame.

```
if cv2.waitKey(1) & 0xFF == ord('q'):
```

```
break
```

We wait for the 'q' key to be pressed. If it is, we exit the script.

```
# When everything is done, release the capture
```

```
video_capture.release()
```

```
cv2.destroyAllWindows()
```

Object Classification

Colour Threshold

Plastic Play Pit Ball as Object to Detect

I opted to detect the plastic ball by colour so I need to set the colour range that I can use to classify each coloured ball.

```
1 blue = (104, 117, 222, 121, 255, 255)
```

I use an array to hold the values of the lower and upper colour threshold. The colour threshold uses the HSV (Hue Saturation Value) colour profile.

Colour Conversion

```
1 frameHSV = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
```

Before I can use the HSV profile values that define the coloured ball, I must convert the captured webcam frame to the HSV colour profile.

Create a Mask

Object Detection Mask

```
1 colorLow = np.array([lowHue,lowSat,lowVal])
```

```
2 colorHigh = np.array([highHue,highSat,highVal])
```

```
3 mask = cv2.inRange(frameHSV, colorLow, colorHigh)
```

Here I use the colour range I set for the ball to create a mask. The mask will make it more efficient to find contours around the detected object. The white area is the area of interest that was found within the colour range set above. I can use various image filters to improve the image mask. However, applying filters to get the perfect mask can be expensive in regards to processing power.

OpenCV Contours

Find Contours

```
1 im2, contours, hierarchy = cv2.findContours(mask, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
```

Now that I have an image mask to work with I can proceed with finding contours. The third parameter, contour approximation method, will collect only the endpoint coordinates of straight lines. All the white blobs in the mask will have contours applied. The array list of found contours will be in the contours variable.

(Optional) Draw all Contours

OpenCV Find Contours Example

```
1 cv2.drawContours(frame, contours, -1, (0,255,0), 3)
```

Two objects are detected, and some of the darker blue colour is out of range of the threshold set earlier. If the quality of the mask is noisy, there is going to be a lot of contours – many small contours. To get useful object detection, I need to remove the contours I do not need.

The image above shows a contour circling the plastic ball; I will aim to isolate that in the next step.

(Optional) Finding The Largest Contour

Finding Largest Contour Example – OpenCV Method

```
1 contour_sizes = [(cv2.contourArea(contour), contour) for contour in contours]
2 biggest_contour = max(contour_sizes, key=lambda x: x[0])[1]
3 cv2.drawContours(frame, biggest_contour, -1, (0,255,0), 3)
```

The above OpenCV Python code finds the biggest contour out of all the contours found. And then draw the biggest contour on to the original image. Since the ball is the largest blue object, I can reliably detect the ball. However, the ball must remain the dominant blue object and remain in focus to be tracked reliably.

So at this point, I was able to improve OpenCV object detection. I am now only detecting one item. The quality of the object detection is very good in this case. However, moving webcam or object or even changing light conditions can make the quality of detection unpredictable.

Bounding Rectangle

OpenCV Method – Bounding Rectangle Example

```
1 contour_sizes = [(cv2.contourArea(contour), contour) for contour in contours]
2 biggest_contour = max(contour_sizes, key=lambda x: x[0])[1]
3 x,y,w,h = cv2.boundingRect(biggest_contour)
4 cv2.rectangle(frame,(x,y),(x+w,y+h),(0,255,0),2)
```

As you can see from the example image, with very little Python code, I got good OpenCV object detection. The third line of the above Python code reveals how I can pull useful data about the detected object. Furthermore, I can see how this data is being used to draw a bounding box around the detected object.

If I am using OpenCV in an embedded device like the Raspberry Pi, I will only use the first three lines of the above code. And the above code should follow the find contours method.

The x, y coordinates and the width, height dimensions will constantly change between webcam frame updates. Also, the coordinates and dimensions will change more dramatically with larger objects where environmental conditions are variable.