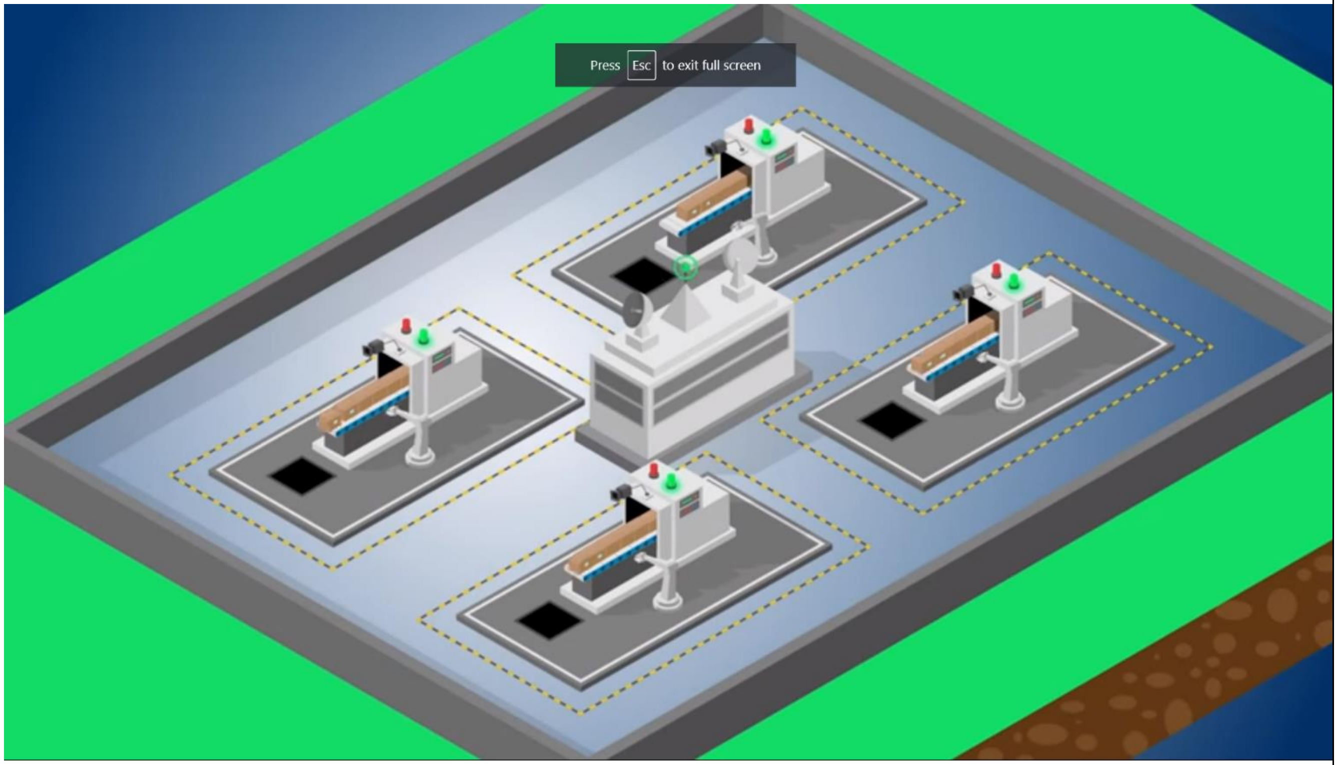
DAILY ASSESSMENT FORMAT

Date:	01/06/2020	Name:	Shilpa N
Course:	HDL	USN:	4AL16EC071
Topic:	DAY 1	Semester & Section:	8 “B”
Github Repository:	Shilpan-test		

FORENOON SESSION DETAILS
Image of session 
Report – Report can be typed or hand written for up to two pages. Intel FPGA are enabling industry 4.0. across an IOT across wide variety of industrial application. In industrial automation smart energy & intelligent vision. In todays reality intel is helping integral secure, programmable logic gates, motor drives and controller interfaces on a single chip. Intel SOC has 15years of good working and is cheapest. FPGA business fundamentals

the session gives a overview of FPGA market. The FPGA is frequently compared to ASIC, ASSP. ASIC-Application specific integrated circuit which is specific to one task, high upfront cash, large volume. ASSP- Application specific standard product has specific function and general enough that any one can use/purchase it. FPGA- field programmable gate array which flexible and customizable, IP is like a magnetic stirrer we can stick on the board.

ASIC/ASSP advantages and disadvantages

Advantages

- Low cost per unit
- Low power consumption
- Small unit size
- High performance/clock speed

Disadvantages

- High non-recurring engineer cost
- Complex design
- Long time to market
- Not flexible

Why FPGA?

- Reduced time to market
- Product longevity
- Market size optimization
- Reprogrammable & flexible

FPGA life cycle is of 10-15years. Developing and prototyping on FPGA s can reduce TTM, especially in emerging markets where standards have not been defined. Serving applications with increasingly large volume.

Software enables intel hardware

- Intel heavily invest on Quartus because it is paramount to the success of FPGAs.
- They have good tools in order to become competent in FPGA market.

- FPGA market is a duopoly because of the software and tool needed to develop FPGA.

FPGA versus ASIC design flow

The session by the end make us to describe key differences in ASIC and FPGA design flow, includes

- Design methodology
- Verification techniques
- Tools
- Test-generation logic

Design Flow

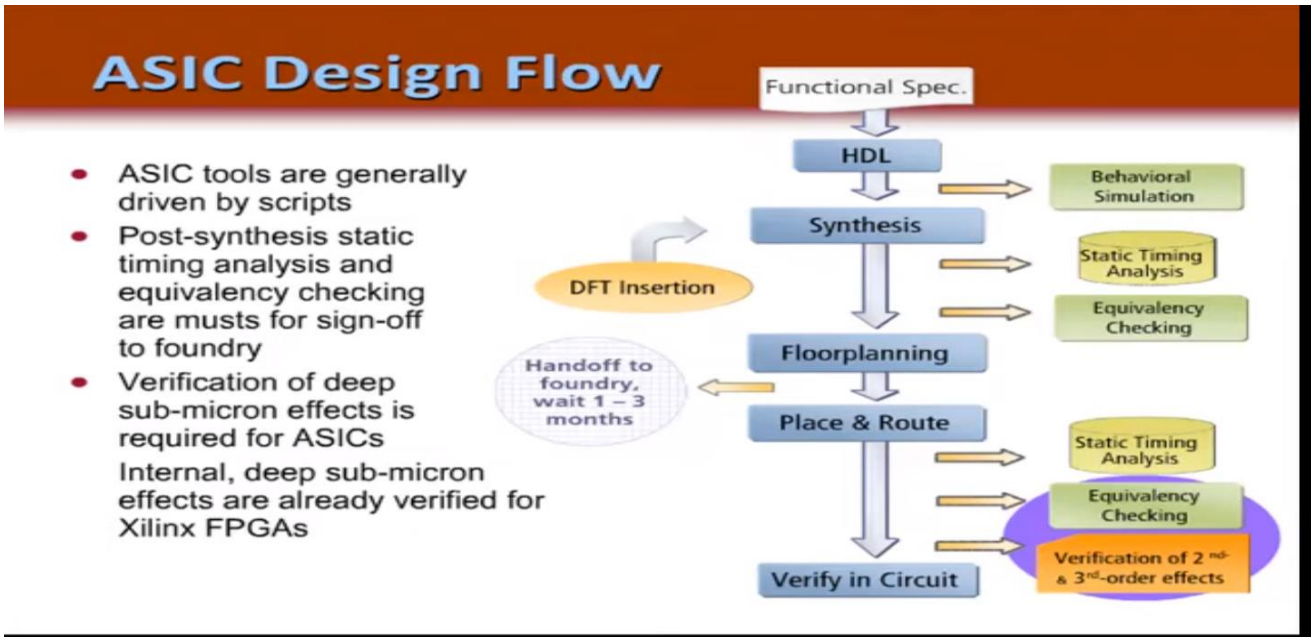
ASIC & FPGA design and implementation methodologies differ somewhat.

- Xilinx FPGAs provide for reduced design time and later bug fixes
  - No design for test logic is required
  - Deep sub-micron verification is done
  - Np waiting for prototypes

Coding style

- For high-performance designs, FPGAs may require some pipelining.
- When retargeting code from an ASIC to an FPGA, the code usually requires optimization.

ASIC DESIGN FLOW

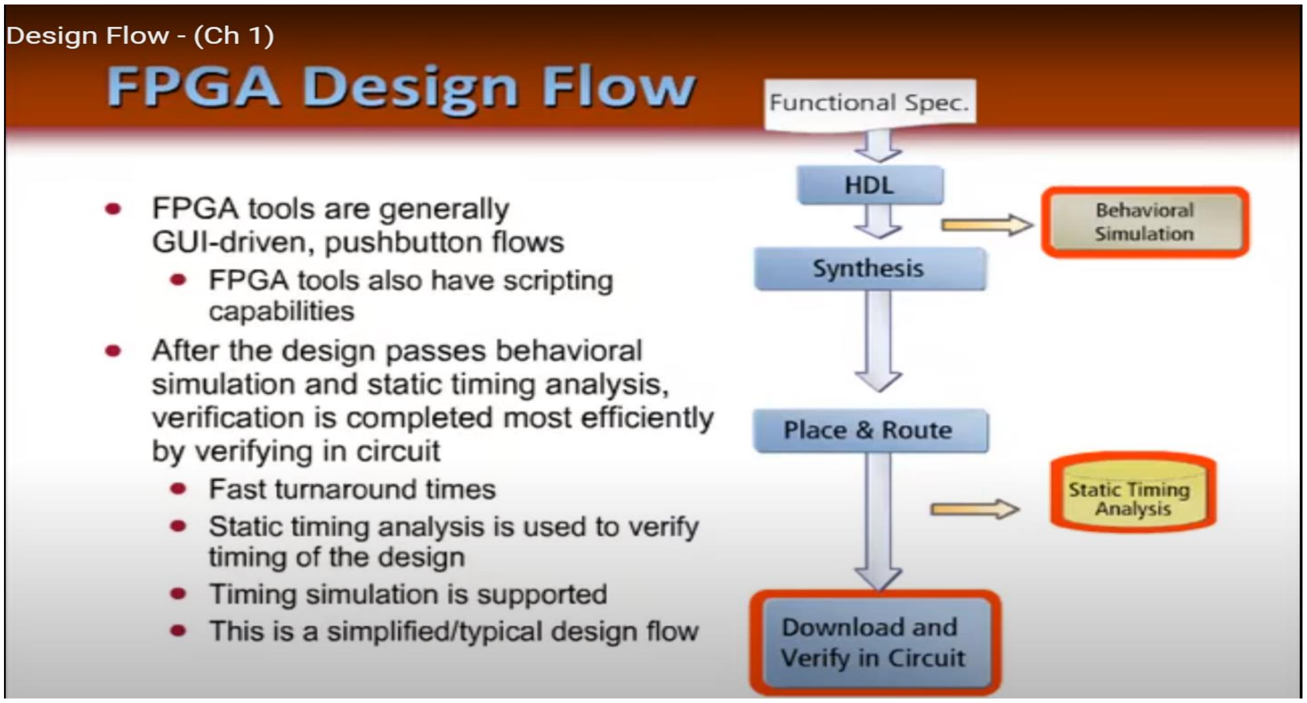


ASIC Design Flow

- ASIC tools are generally driven by scripts
- Post-synthesis static timing analysis and equivalency checking are musts for sign-off to foundry
- Verification of deep sub-micron effects is required for ASICs

Internal, deep sub-micron effects are already verified for Xilinx FPGAs

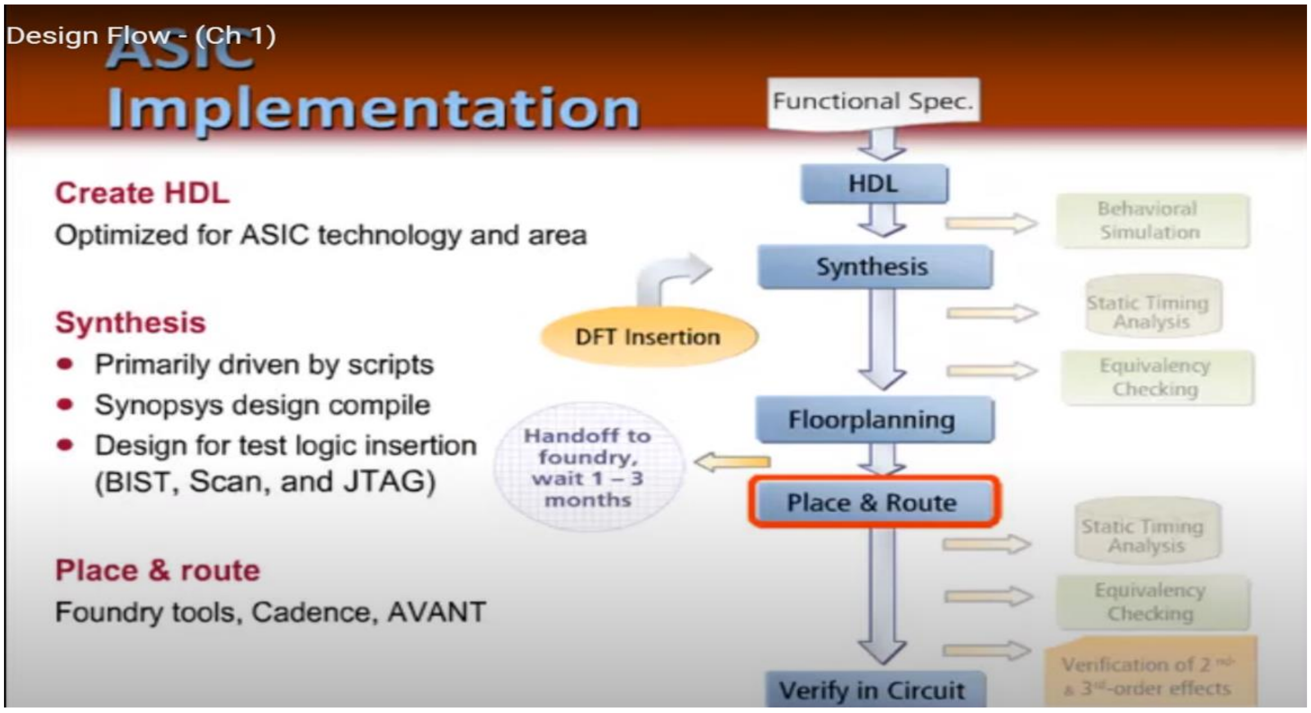
FPGA DESIGN FLOW


Design Flow - (Ch 1)

FPGA Design Flow

- FPGA tools are generally GUI-driven, pushbutton flows
  - FPGA tools also have scripting capabilities
- After the design passes behavioral simulation and static timing analysis, verification is completed most efficiently by verifying in circuit
  - Fast turnaround times
  - Static timing analysis is used to verify timing of the design
  - Timing simulation is supported
  - This is a simplified/typical design flow

ASIC IMPLEMENTATION


Design Flow - (Ch 1)

ASIC Implementation

Create HDL

Optimized for ASIC technology and area

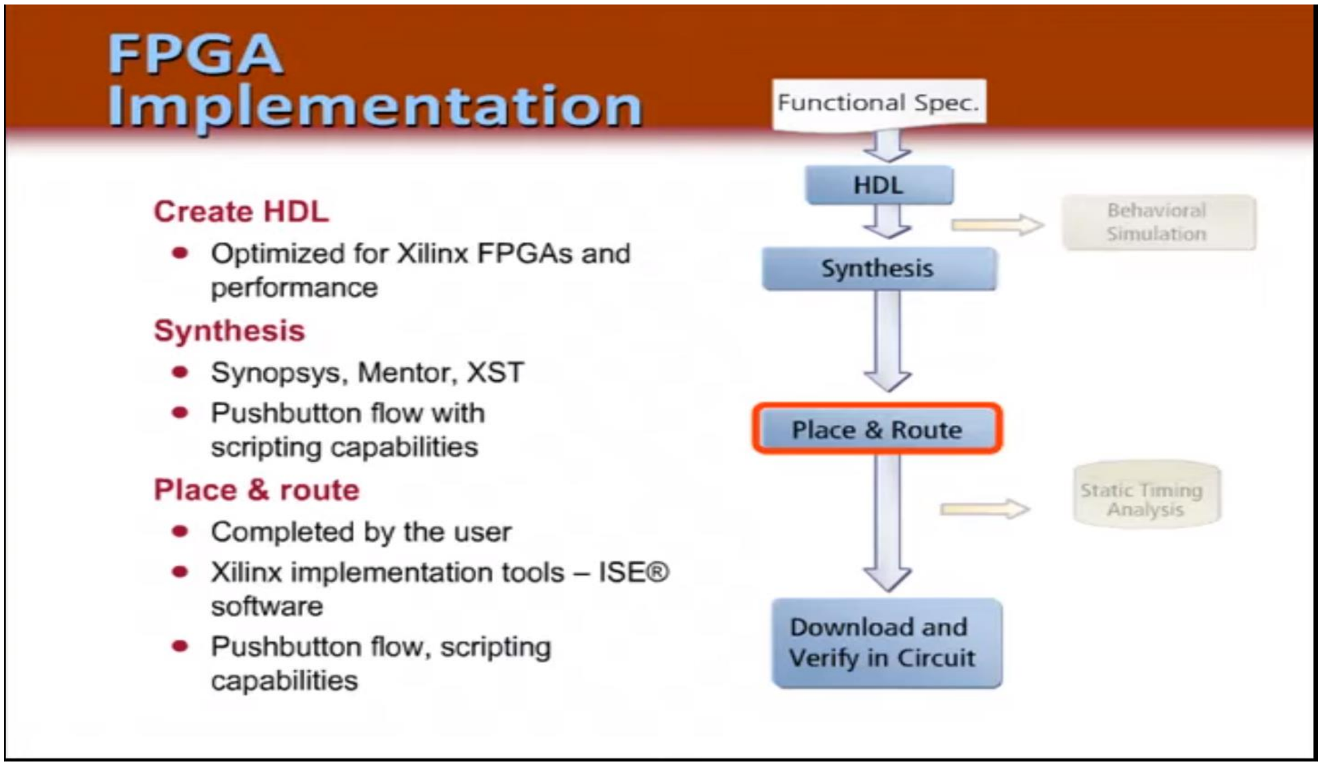
Synthesis

- Primarily driven by scripts
- Synopsys design compile
- Design for test logic insertion (BIST, Scan, and JTAG)

Place & route

Foundry tools, Cadence, AVANT

FPGA IMPLEMENTATION



FPGA Implementation

Create HDL

- Optimized for Xilinx FPGAs and performance

Synthesis

- Synopsys, Mentor, XST
- Pushbutton flow with scripting capabilities

Place & route

- Completed by the user
- Xilinx implementation tools – ISE® software
- Pushbutton flow, scripting capabilities

What are the most important things you should know right away?

Get out of the software mindset – You're not writing software. Let me say that again because this is the single most important point if you're thinking about working with FPGAs.

You-are-NOT-writing-software.

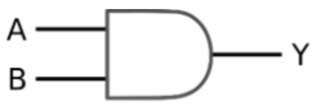
You’re designing a digital circuit. You’re using code to tell the chip how to configure itself.

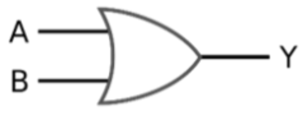
Plan for lots of bugs – yes, plan for them. They are going to happen. Way more than you expected. If you’re a newbie developer, you need to pull in someone that has experience with FPGA development to help with this estimate.
Application-specific realities – you ought to concern yourself with realities revolving around cyber security and safety, as FPGAs are a different animal than what you’re likely used to.

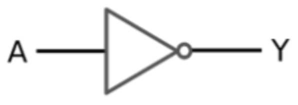
What is an FPGA?

An FPGA is a (mostly) digital, (re-)configurable ASIC. I say mostly because there are analog and mixed-signal aspects to modern FPGAs. For example, some have A/D converters and PLLs. I put re- in parenthesis because there are actually one-time-programmable FPGAs, where once you configure them, that’s it, never again. However, most FPGAs you’ll come across are going to be re-configurable. So what do I mean by digitally configurable ASIC?

I mean that at the core of it, you’re designing a digital logic circuit, as in AND, OR, NOT, flip-flops, etc. Of course that’s not entirely accurate and there’s much more to it than that, but that is the gist at its core.







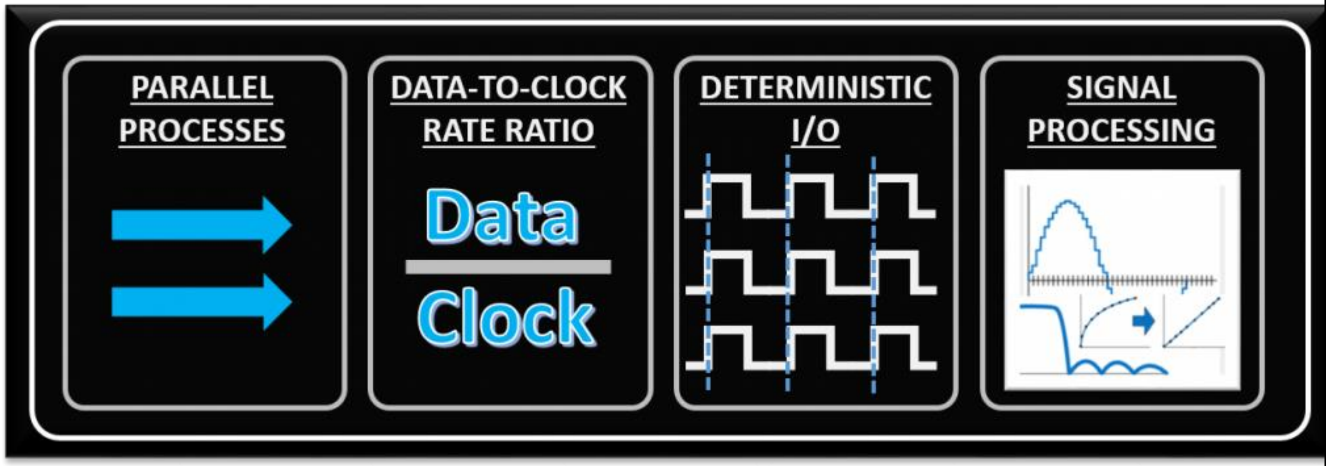
The players –

There are currently two big boys: Altera (part of Intel) and Xilinx, and some supporting players (e.g. Actel (owned by Microsemi)).

The main underlying technology options are SRAM-based (this is the most common technology), flash, and anti-fuse. As you might imagine, each option has its own pros and cons. Check this out for some more details.

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.



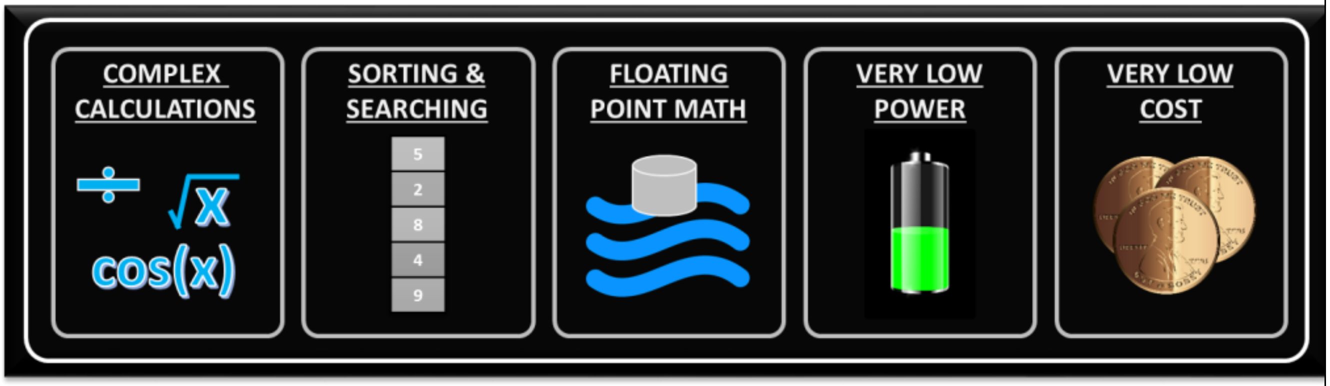
1. **Par**allel processes – if you need to process several input channels of information (e.g. many simultaneous A/D channels) or control several channels at once (e.g. several PID loops).
High data-to-clock-rate-ratio – if you've got lots of calculations that need to be executed over and over and over 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.

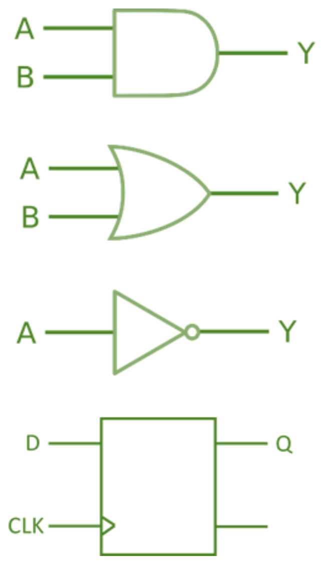
Sorting/searching – this really falls into the category of a sequential process. There are algorithms that attempt to reduce the number of computations involved, but in general, this is a sequential process that doesn't easily lend itself to efficient use of parallel logical resources. Check out the 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).

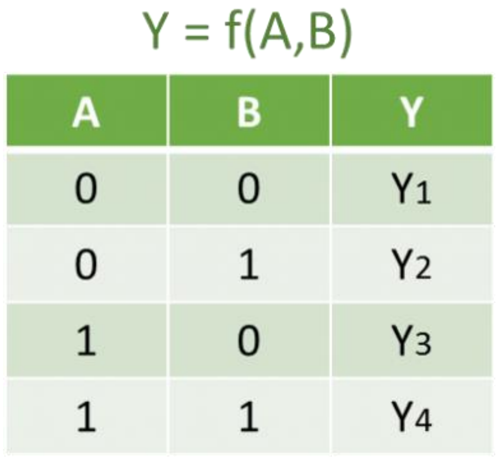
Very low power – Some FPGAs have low power modes (hibernate and/or suspend) to help reduce current consumption, and some may require external mode control ICs to get the most out of this. Check out an example low power mode FPGA here. There are both static and dynamic aspects to power consumption. Check 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.

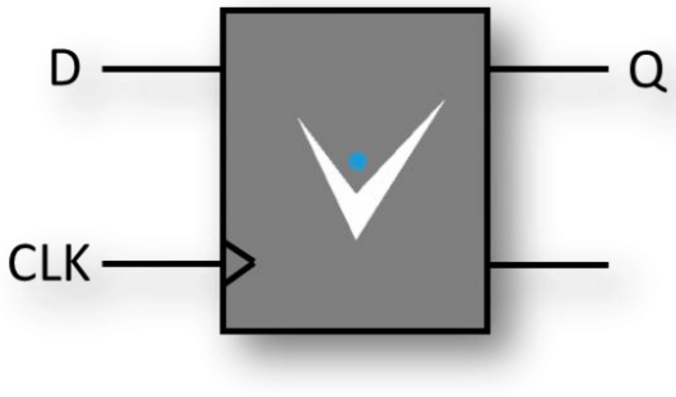
How Does an FPGA work?



You’re designing a digital circuit more than anything else, basically at one layer of abstraction above the logic gate (AND, OR, NOT) level. At the most basic level, you need to think about how you’re specifying the layout and equations at the level of LUTs (Look-Up Tables) and FFs (Flip-Flops).


Y = f(A,B)

A	B	Y
0	0	Y_1
0	1	Y_2
1	0	Y_3
1	1	Y_4



Otherwise you're circuit can get very large and slow very quickly. You've got a very detailed level of control at your fingertips, which is very powerful, but can be overwhelming, so start slow. You'll be determining the # of bits, and exact math / structure of each function.

An FPGA is a synchronous device, meaning that logical operations are performed on a clock cycle-by-cycle basis. Flip-flops are the core element to enabling this structure.

In general, you're going to put digital data into an FPGA and get digital data out of it through various low-voltage digital I/O lines, sometimes many bits in parallel (maybe through one or more A/D converter outputs or an external DRAM chip), sometimes through high-speed serial I/O (maybe connecting to an Ethernet PHY or USB chip).

What's inside – Core components (or at least what everyone likes to think about):

LUT (Look-Up Table) –

The name LUT in the context of FPGAs is actually misleading, as it doesn't convey the full power of this logical resource. The obvious use of a LUT is as a logic lookup table (see examples here and here), generally with 4 to 6 inputs and 1 to 2 outputs to specify any logical operation that fits within those bounds. There are however two other common uses for a LUT:

LUT as a shift register – shift registers are very useful for things like delaying the timing of an operation to align the outputs of one algorithm with another. Size varies based on FPGA.
LUT as a small memory – you can configure the LUT logic as a VERY small volatile random-access memory block. Size varies based on FPGA

FF (Flip-flop) –
Flip-flops store the output of a combinational logic calculation. This is a critical element in FPGA design because you can only allow so much asynchronous logic and routing to occur before it is registered by a synchronous resource (the flip-flop), otherwise the FPGA won't make timing. It's the core of how an FPGA works.

Flip-flops can be used to register data every clock cycle, latch data, gate off data, or enable signals.

Block Memory –
It's important to note that there are generally several types of memory in an FPGA. We mentioned the configuration of a LUT resource. Another is essentially program memory, which is intended to store the compiled version of the FPGA program itself (this may be part of the FPGA chip or as a separate non-volatile memory chip). What we're referring to here though, is neither of those types of memory. Here we're referring to dedicated blocks of volatile user memory within the FPGA. This memory block is generally on the order of thousands of bits of memory, is configurable in width and depth, and multiple blocks of memory can be chained together to create larger memory elements. They can generally be configured as either single-port or dual-port random access, or as a FIFO. There will generally be many block memory elements within an FPGA.

Multipliers or DSP blocks –
Have you ever seen the number of digital logic resources that it takes to create a 16-bit by 16-bit multiplier? It's pretty crazy, and would chew through your logical and routing resources pretty quickly. Check out the 2-bit by 2-bit example here: https://en.wikipedia.org/wiki/Binary_multiplier. FPGA vendors solve this problem with dedicated silicon to lay down something on the order of 18-bit multiplier blocks. Some architectures

have recognized the utility of digital signal processing taking place, and have taken it a step further with dedicated DSP (Digital Signal Processing) blocks, which can not only multiply, but add and accumulate as well.

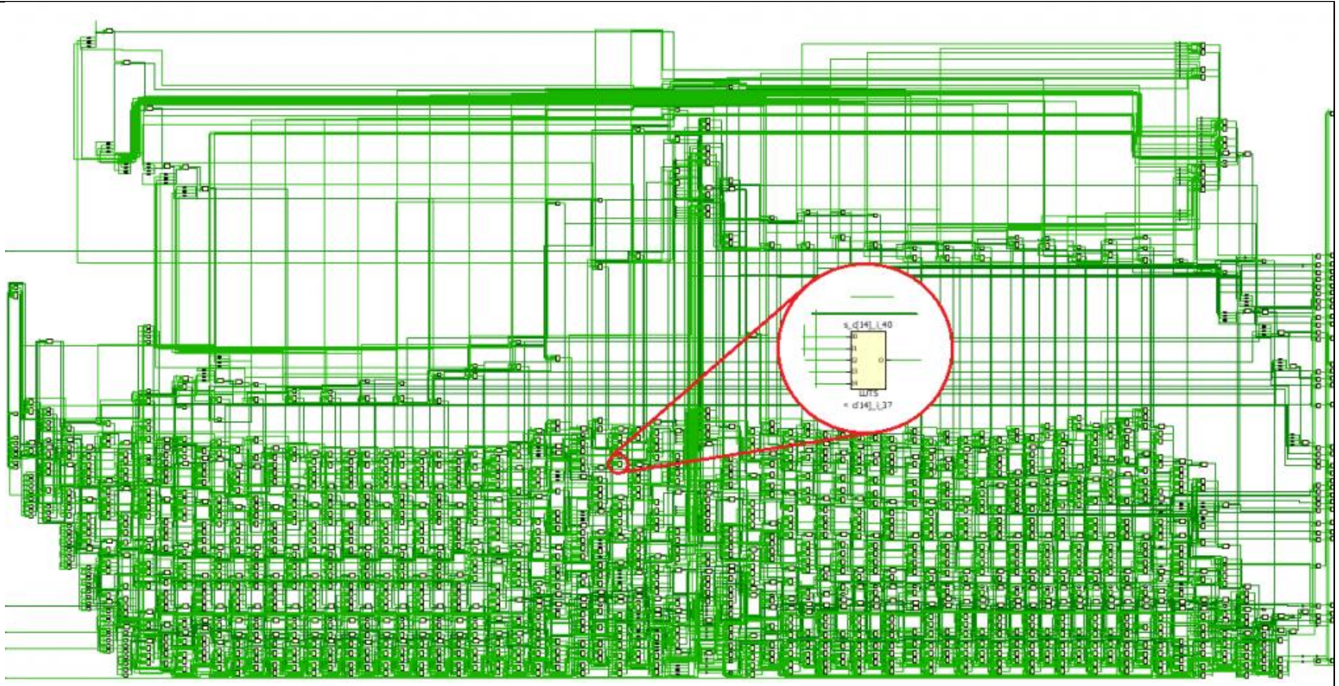
I/O (Input/Output) –

If you're going to do something useful with an FPGA, you generally have to get data from and/or provide data outside the FPGA. To facilitate this, FPGAs will include I/O blocks that allow for various voltage standards (e.g. LVCMOS, LVDS) as well as timing delay elements to help align multiple signals with one another (e.g. for a parallel bus to an external RAM chip).

Clocking and routing –

This is really a more advanced topic, but critical enough to at least introduce. You'll likely use an external oscillator and feed it into clocking resources that can multiply, divide, and provide phase-shifted versions of your clock to various parts of the FPGA.

Routing resources not only route your clock to various parts of the FPGA, but also your data. Routing resources within an FPGA are one of the most underappreciated elements, but so critical. Check out this sea of madness:



What’s Inside – Advanced components

Hard cores – These are functional blocks that (at least for the most part) have their own dedicated logical resources. In other words, they are already embedded into your FPGA silicon. You configure them with various parameters and tell the tools to enable them for you. This could include functions such as high-speed communications (e.g. high-speed serial, Ethernet), low-speed A/D converters for things like measuring slowly varying voltages, and microprocessor cores to handle some of the functions that FPGA logic is not as well suited for.

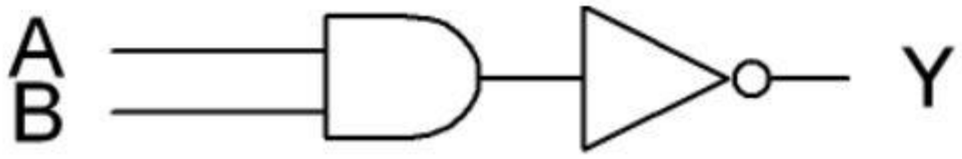
Soft cores – These are functional blocks that don’t have their own dedicated logical resources. In other words, they are laid out with your core logic resources. You configure them with various parameters and tell the tools to build them for you. This could include everything from DDR memory interfaces to FFT cores to FIR filters to microprocessors to CORDICs. The library of available soft cores can be impressive. On the plus side, you don’t have to take as much time to develop these cores. On the negative side, since you won't know the intricacies of the design, when you plop it down and it doesn't work, it will generally take you longer to figure out why.

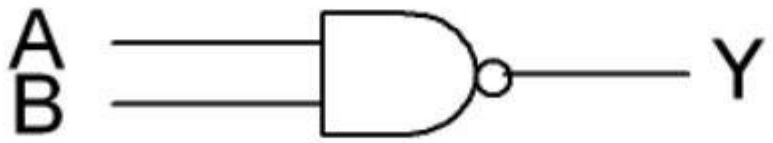
TASK

Write a verilog code to implement NAND gate in all different styles.

Logic Circuit of the NAND gate

The NAND gate is the complement of AND function. Its graphic symbol consists of an AND gate's graphic symbol followed by a small circle. Here's the logical representation of the NAND gate.





Verilog code for NAND gate using gate-level modeling

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

Data flow modeling

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

```
endmodule
```

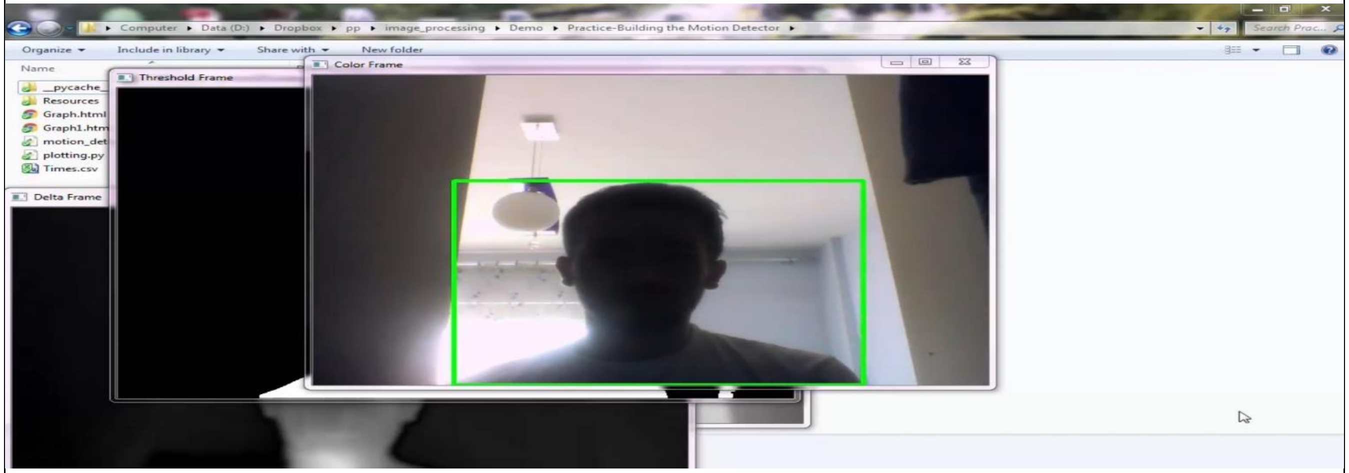
Behavioral Modeling

```
module NAND_2_behavioral (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
```

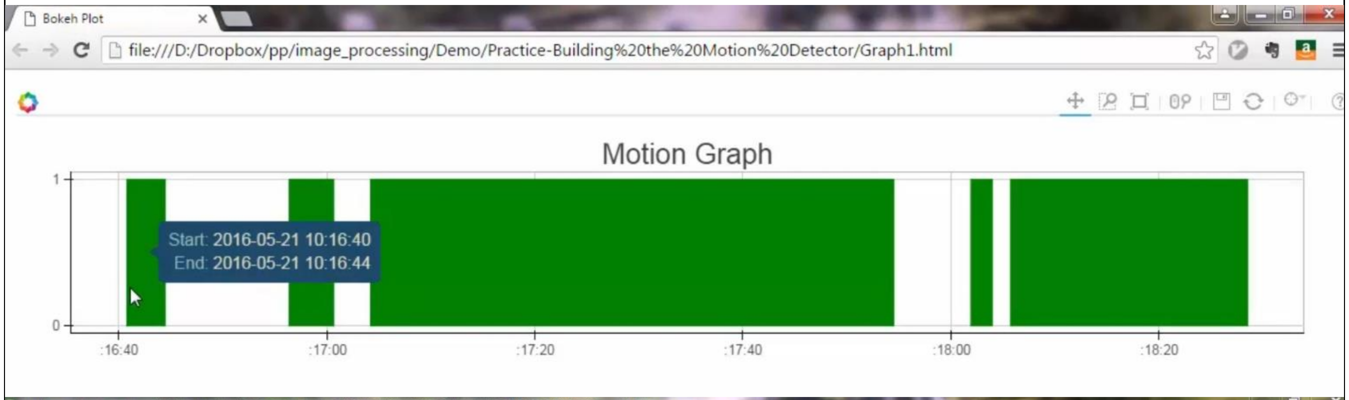
Date:	106/2020	Name:	Shilpa N
Course:	PYTHON	USN:	4AL16EC071
Topic:	DAY 13	Semester & Section:	8 “B”

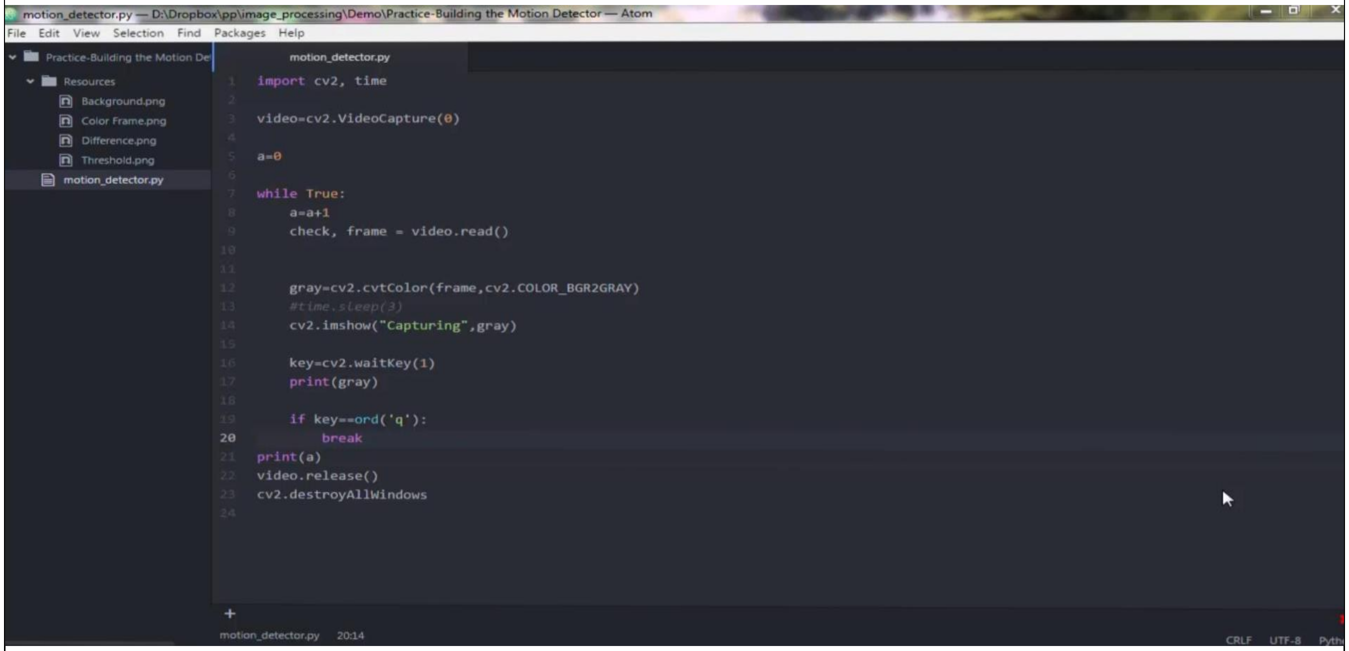
AFTERNOON SESSION DETAILS

We are going to build is a program that detects objects, moving objects in front of a computer webcam and then what it does it record the time that the object enters the webcam, so the video frame and the time when the object exited the video frame, so now there is a moving object in frame and if I go away you see that there's no green rectangle now in the video. So we're going to build this application from scratch using Python and we have a lot of things to learn such as image processing and video processing as well, and so on. And lastly what we do then if you press q for quit, what you get is a graph and as I said the graph will show the times where the object entered the frame so we're dealing with an interactive graph here and yeah, this are the times when the object entered the webcam like this one, this one here, and here it lasted longer, and so on. This can be a great application if you want to detect objects. You get throw, you can put this application of this Python program in a Raspberry Pi server for instance which is a small server and you can put it somewhere and maybe want to detect animals. You



may want to know when this animal is entering the frame and when it is exiting, or with people as well if there is the case. we have quite a lot of codes to write,




motion_detector.py — D:\Dropbox\pp\image_processing\Demo\Practice-Building the Motion Detector — Atom
File Edit View Selection Find Packages Help
Practice-Building the Motion De
Resources
Background.png
Color Frame.png
Difference.png
Threshold.png
motion_detector.py
motion_detector.py 20:14
CRLF UTF-8

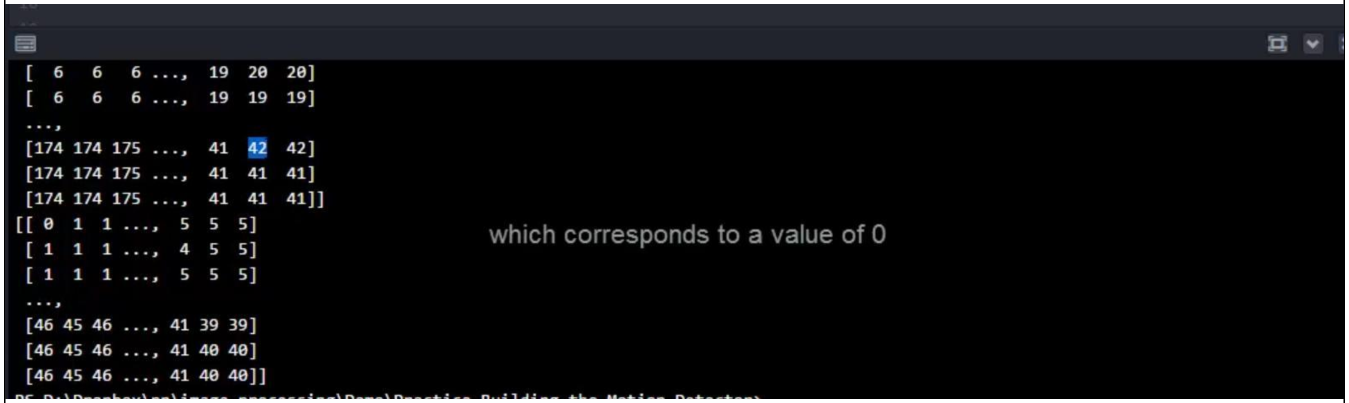
```
import cv2, time

video=cv2.VideoCapture(0)

a=0

while True:
    a=a+1
    check, frame = video.read()


    gray=cv2.cvtColor(frame,cv2.COLOR_BGR2GRAY)
    #time.sleep(3)
    cv2.imshow("Capturing",gray)

    key=cv2.waitKey(1)
    print(gray)

    if key==ord('q'):
        break
print(a)
video.release()
cv2.destroyAllWindows
```

Motion_detector program:

Motion_detector program:

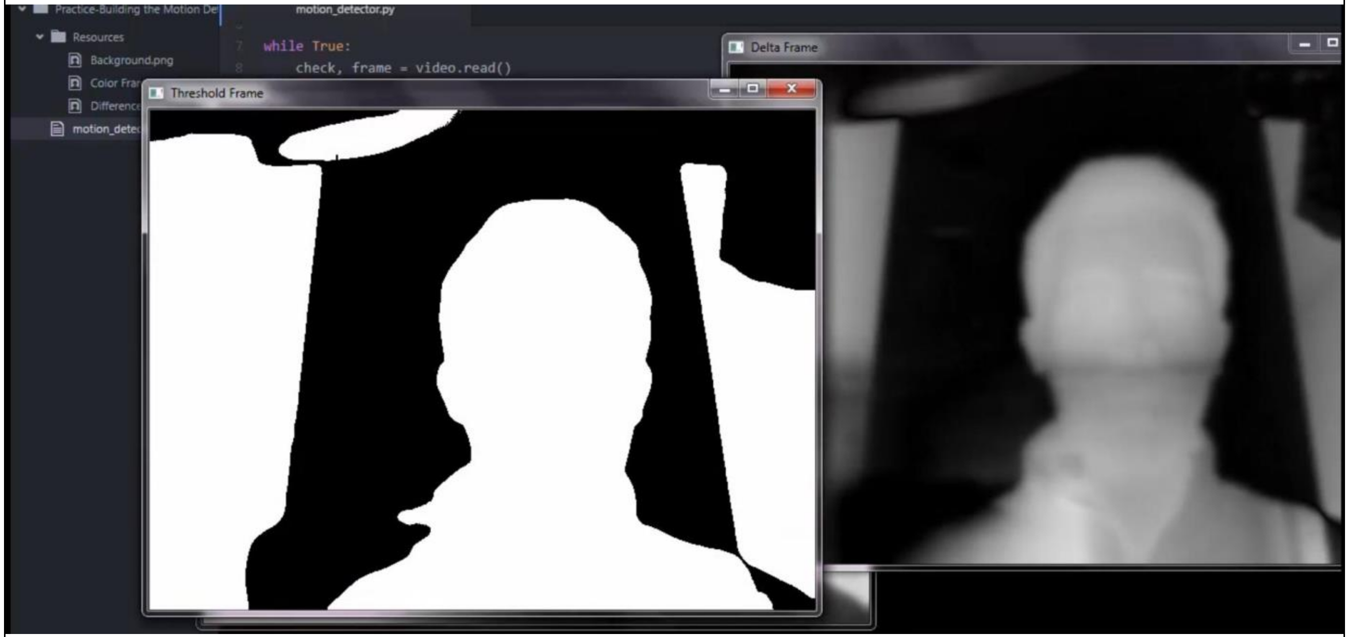
```
import cv2, time, pandas
from datetime import datetime



first_frame=None
status_list=[None,None]
times=[]
df=pandas.DataFrame(columns=["Start","End"])



video=cv2.VideoCapture(0)
```

```
while True:
    check, frame = video.read()
    status=0
    gray=cv2.cvtColor(frame,cv2.COLOR_BGR2GRAY)
    gray=cv2.GaussianBlur(gray,(21,21),0)


    if first_frame is None:
        first_frame=gray
        continue


    delta_frame=cv2.absdiff(first_frame,gray)
    thresh_frame=cv2.threshold(delta_frame, 30, 255, cv2.THRESH_BINARY)[1]
    thresh_frame=cv2.dilate(thresh_frame, None, iterations=2)


    (_,cnts,_)=cv2.findContours(thresh_frame.copy(),cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)


    for contour in cnts:
        if cv2.contourArea(contour) < 10000:
            continue
        status=1

        (x, y, w, h)=cv2.boundingRect(contour)
        cv2.rectangle(frame, (x, y), (x+w, y+h), (0,255,0), 3)
    status_list.append(status)
    status_list=status_list[-2:]



    if status_list[-1]==1 and status_list[-2]==0:
        times.append(datetime.now())
    if status_list[-1]==0 and status_list[-2]==1:

        times.append(datetime.now())



    cv2.imshow("Gray Frame",gray)

    cv2.imshow("Delta Frame",delta_frame)

    cv2.imshow("Threshold Frame",thresh_frame)

    cv2.imshow("Color Frame",frame)



    key=cv2.waitKey(1)



    if key==ord('q'):
        if status==1:
            times.append(datetime.now())

        break


print(status_list)
print(times)

for i in range(0,len(times),2):
    df=df.append({"Start":times[i],"End":times[i+1]},ignore_index=True)

df.to_csv("Times.csv")


video.release()

cv2.destroyAllWindows
```



```
 [  6   6   6 ...,  19  20  20]
 [  6   6   6 ...,  19  19  19]
 ...,
 [174 174 175 ...,  41  42  42]
 [174 174 175 ...,  41  41  41]
 [174 174 175 ...,  41  41  41]]
[[ 0  1  1 ...,  5  5  5]
 [ 1  1  1 ...,  4  5  5]
 [ 1  1  1 ...,  5  5  5]
 ...,
 [46 45 46 ..., 41 39 39]
 [46 45 46 ..., 41 40 40]
 [46 45 46 ..., 41 40 40]]
```

which corresponds to a value of 0



```
while True:
    check, frame = video.read()
```