**DAILY ASSESSMENT FORMAT**

|  |  |  |  |
| --- | --- | --- | --- |
| **Date:** | **01/06/2020** | **Name:** | **Nayanashree K S** |
| **Course:** | **HDL design** | **USN:** | **4AL16EC042** |
| **Topic:** | **Industry Applications of FPGA**  **FPGA vs ASIC Design Flow** | **Semester & Section:** | **8 A** |
| **Github Repository:** | **nayana\_online** |  |  |

|  |
| --- |
| **FORENOON SESSION DETAILS** |
| **Image of session** |
| **Report**  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.  FPGA-basics-gates   1. **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). 2. **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. 3. **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. 4. **Signal processing** – includes algorithms such as digital filtering, demodulation, detection algorithms, frequency domain processing, image processing, or control algorithms.   In the case of FPGAs, the following are generally the main disadvantages of FPGA-based solutions.   1. **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. 2. **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](http://bigocheatsheet.com/) and check out this article [here](http://www.embedded.com/design/configurable-systems/4006440/Sorting-data-in-two-clock-cycles) for some more info. 3. **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](http://www.xilinx.com/products/intellectual-property/floating_pt.html) and [Altera FP White Paper](https://www.altera.com/en_US/pdfs/literature/wp/wp-01028.pdf) ), 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](https://www.altera.com/products/fpga/features/dsp/arria10-dsp-block.html)). 4. **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](http://www.xilinx.com/support/documentation/application_notes/xapp480.pdf). There are both static and dynamic aspects to power consumption. Check out these [power estimation spreadsheets](https://www.xilinx.com/products/technology/power/xpe.html) 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. 5. **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.   ASIC stands for Application Specific Integrated Circuit. As the name implies, ASICs are application specific. They are designed for one sole purpose and they function the same their whole operating life. For example, the CPU inside your phone is an ASIC. It is meant to function as a CPU for its whole life. Its logic function cannot be changed to anything else because its digital circuitry is made up of permanently connected gates and flip-flops in silicon. The logic function of ASIC is specified in a similar way as in the case of FPGAs, using hardware description languages such as Verilog or VHDL.  FPGA vs ASIC:  **Speed** ASIC rules out FPGA in terms of speed. As ASIC are designed for a specific application they can be optimized to maximum, hence we can have high speed in ASIC designs. ASIC can have hight speed clocks.  **Cost** FPGAs are cost effective for small applications. But when it comes to complex and large volume designs (like 32-bit processors) ASIC products are cheaper.  **Size/Area** FPGA are contains lots of LUTs, and routing channels which are connected via bit streams(program). As they are made for general purpose and because of re-usability. They are in-general larger designs than corresponding ASIC design. For example, LUT gives you both registered and non-register output, but if we require only non-registered output, then its a waste of having a extra circuitry. In this way ASIC will be smaller in size.  **Power** FPGA designs consume more power than ASIC designs. As explained above the unwanted circuitry results wastage of power. FPGA wont allow us to have better power optimization. When it comes to ASIC designs we can optimize them to the fullest.  **TimeMarket** FPGA designs will till less time, as the design cycle is small when compared to that of ASIC designs. No need of layouts, masks or other back-end processes. Its very simple: Specifications -- HDL + simulations -- Synthesis -- Place and Route (along with static-analysis) -- Dump code onto FPGA and Verify. When it comes to ASIC we have to do floor planning and also advanced verification. The FPGA design flow eliminates the complex and time-consuming floor planning, place and route, timing analysis, and mask / re-spin stages of the project since the design logic is already synthesized to be placed onto an already verified, characterized FPGA device. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Date:** | **01/6/2020** | **Name:** | **Nayanashree K S** | |
| **Course:** | **Python** | **USN:** | **4al16ec042** | |
| **Topic:** | **Application 6: Build a Webcam Motion Detector** | **Semester & Section:** | **8 A** | |
| **AFTERNOON SESSION DETAILS** | | | |
| REPORT  **Main Logic :** Videos can be treated as stack of pictures called frames. Here I am comparing different frames(pictures) to the first frame which should be static(No movements initially). We compare two images by comparing the intensity value of each pixels. In python we can do it easily as you can see in following code:  # Python program to implement  # Webcam Motion Detector  # importing OpenCV, time and Pandas library  import cv2, time, pandas  # importing datetime class from datetime library  from datetime import datetime  # Assigning our static\_back to None  static\_back = None  # List when any moving object appear  motion\_list = [ None, None ]  # Time of movement  time = []  # Initializing DataFrame, one column is start  # time and other column is end time  df = pandas.DataFrame(columns = ["Start", "End"])  # Capturing video  video = cv2.VideoCapture(0)  # Infinite while loop to treat stack of image as video  while True:  # Reading frame(image) from video  check, frame = video.read()  # Initializing motion = 0(no motion)  motion = 0  # Converting color image to gray\_scale image  gray = cv2.cvtColor(frame, cv2.COLOR\_BGR2GRAY)  # Converting gray scale image to GaussianBlur  # so that change can be find easily  gray = cv2.GaussianBlur(gray, (21, 21), 0)  # In first iteration we assign the value  # of static\_back to our first frame  if static\_back is None:  static\_back = gray  continue  # Difference between static background  # and current frame(which is GaussianBlur)  diff\_frame = cv2.absdiff(static\_back, gray)  # If change in between static background and  # current frame is greater than 30 it will show white color(255)  thresh\_frame = cv2.threshold(diff\_frame, 30, 255, cv2.THRESH\_BINARY)[1]  thresh\_frame = cv2.dilate(thresh\_frame, None, iterations = 2)  # Finding contour of moving object  cnts,\_ = cv2.findContours(thresh\_frame.copy(),  cv2.RETR\_EXTERNAL, cv2.CHAIN\_APPROX\_SIMPLE)  for contour in cnts:  if cv2.contourArea(contour) < 10000:  continue  motion = 1  (x, y, w, h) = cv2.boundingRect(contour)  # making green rectangle arround the moving object  cv2.rectangle(frame, (x, y), (x + w, y + h), (0, 255, 0), 3)  # Appending status of motion  motion\_list.append(motion)  motion\_list = motion\_list[-2:]  # Appending Start time of motion  if motion\_list[-1] == 1 and motion\_list[-2] == 0:  time.append(datetime.now())  # Appending End time of motion  if motion\_list[-1] == 0 and motion\_list[-2] == 1:  time.append(datetime.now())  # Displaying image in gray\_scale  cv2.imshow("Gray Frame", gray)  # Displaying the difference in currentframe to  # the staticframe(very first\_frame)  cv2.imshow("Difference Frame", diff\_frame)  # Displaying the black and white image in which if  # intensity difference greater than 30 it will appear white  cv2.imshow("Threshold Frame", thresh\_frame)  # Displaying color frame with contour of motion of object  cv2.imshow("Color Frame", frame)  key = cv2.waitKey(1)  # if q entered whole process will stop  if key == ord('q'):  # if something is movingthen it append the end time of movement  if motion == 1:  time.append(datetime.now())  break  # Appending time of motion in DataFrame  for i in range(0, len(time), 2):  df = df.append({"Start":time[i], "End":time[i + 1]}, ignore\_index = True)  # Creating a CSV file in which time of movements will be saved  df.to\_csv("Time\_of\_movements.csv")  video.release()  # Destroying all the windows  cv2.destroyAllWindows()  **Difference Frame :** Difference frame shows the difference of intensities of first frame to the current frame.    **Threshold Frame :**If the intensity difference for a particular pixel is more than 30(in my case) then that pixel will be white and if the difference is less than 30 that pixel will be black    **Color Frame :**In this frame you can see the color images in color frame along with green contour around the moving objects | | | |
|  | | | |