**DAILY ASSESSMENT FORMAT**

|  |  |  |  |
| --- | --- | --- | --- |
| **Date:** | 30 May 2020 | **Name:** | Anupama J S |
| **Course:** | Logic design | **USN:** | 4AL16EC005 |
| **Topic:** | 1. Boolean equations for digital circuits. Combinational circuits: Conversion of MUX and Decoders to logic gates. 2. Analysis of clocked sequential circuits | **Semester & Section:** | 8th sem “A”section |
| **Github Repository:** | AnupamaJS |  |  |

|  |
| --- |
| **FORENOON SESSION DETAILS** |
| Boolean Algebra is an algebra, which deals with binary numbers & binary variables. Hence, it is also called as Binary Algebra or logical Algebra. A mathematician, named George Boole had developed this algebra in 1854. The variables used in this algebra are also called as Boolean variables. The range of voltages corresponding to Logic ‘High’ is represented with ‘1’ and the range of voltages corresponding to logic ‘Low’ is represented with ‘0’.  Postulates and Basic Laws of Boolean Algebra  In this section, let us discuss about the Boolean postulates and basic laws that are used in Boolean algebra. These are useful in minimizing Boolean functions.  Boolean Postulates  Consider the binary numbers 0 and 1, Boolean variable xx and its complement x′x′. Either the Boolean variable or complement of it is known as literal. The four possible logical OR operations among these literals and binary numbers are shown below.  x + 0 = x  x + 1 = 1  x + x = x  x + x’ = 1  Similarly, the four possible logical AND operations among those literals and binary numbers are shown below.  x.1 = x  x.0 = 0  x.x = x  x.x’ = 0  These are the simple Boolean postulates. We can verify these postulates easily, by substituting the Boolean variable with ‘0’ or ‘1’.  Basic Laws of Boolean Algebra  Following are the three basic laws of Boolean Algebra.  Commutative law  Associative law  Distributive law  Commutative Law  If any logical operation of two Boolean variables give the same result irrespective of the order of those two variables, then that logical operation is said to be Commutative. The logical OR & logical AND operations of two Boolean variables x & y are shown below  x + y = y + x  x.y = y.x  The symbol ‘+’ indicates logical OR operation. Similarly, the symbol ‘.’ indicates logical AND operation and it is optional to represent. Commutative law obeys for logical OR & logical AND operations.  Associative Law  If a logical operation of any two Boolean variables is performed first and then the same operation is performed with the remaining variable gives the same result, then that logical operation is said to be Associative. The logical OR & logical AND operations of three Boolean variables x, y & z are shown below.  x + y+zy+z = x+yx+y + z  x.y.zy.z = x.yx.y.z  Associative law obeys for logical OR & logical AND operations.  Distributive Law  If any logical operation can be distributed to all the terms present in the Boolean function, then that logical operation is said to be Distributive. The distribution of logical OR & logical AND operations of three Boolean variables x, y & z are shown below.  x.y+zy+z = x.y + x.z  x + y.zy.z = x+yx+y.x+zx+z  Distributive law obeys for logical OR and logical AND operations.  These are the Basic laws of Boolean algebra. We can verify these laws easily, by substituting the Boolean variables with ‘0’ or ‘1’.  Theorems of Boolean Algebra  The following two theorems are used in Boolean algebra.  Duality theorem  DeMorgan’s theorem  Duality Theorem  This theorem states that the dual of the Boolean function is obtained by interchanging the logical AND operator with logical OR operator and zeros with ones. For every Boolean function, there will be a corresponding Dual function.  DeMorgan’s Theorem  This theorem is useful in finding the complement of Boolean function. It states that the complement of logical OR of at least two Boolean variables is equal to the logical AND of each complemented variable.  DeMorgan’s theorem with 2 Boolean variables x and y can be represented as  x+yx+y’ = x’.y’  The dual of the above Boolean function is  x.yx.y’ = x’ + y’  Therefore, the complement of logical AND of two Boolean variables is equal to the logical OR of each complemented variable. Similarly, we can apply DeMorgan’s theorem for more than 2 Boolean variables also.  COMBINATIONAL CIRCUIT:  Combinational circuit is a circuit in which we combine the different gates in the circuit, for example encoder, decoder, multiplexer and demultiplexer. Some of the characteristics of combinational circuits are following −  The output of combinational circuit at any instant of time, depends only on the levels present at input terminals.  The combinational circuit do not use any memory. The previous state of input does not have any effect on the present state of the circuit.  A combinational circuit can have an n number of inputs and m number of outputs.  Common Anode 7-Segment Display  For common anode apply +5 volts to vcc pin in series to a 510 ohm-1k ohm resistor. This resistor is very important always include it other wise your seven segment display will be damaged by over current. Note both the vcc pins are short so apply +5 volts on only one pin and leave other empty.  Ground the dp(decimal/display point) pin if you want it to illuminate for ever. If you to control dp(decimal/display point) led than connect it to some control system, microcontroller etc.  In common Anode the Cathode(-) side of led’s are connected to a,b,c,d,e,f,g pins of seven segment display.  In common anode seven segment display’s led becomes lit when we ground any a,b,c,d,e,f,g pin.  Common Anode seven segment display’s color is usually gray.    There are two types of input to the combinational logic :   * External inputs which not controlled by the circuit. * Internal inputs which are a function of a previous output states.   Secondary inputs are state variables produced by the storage elements, where as secondary outputs are excitations for the storage elements.  **Types of Sequential Circuits :**  There are two types of sequential circuit :   * Asynchronous sequential circuit – this circuit do not use a clock signal but uses the pulses of the inputs. These circuits are faster than synchronous sequential circuits because there is clock pulse and change their state immediately when there is a change in the input signal. We use asynchronous sequential circuits when speed of operation is important and independent of internal clock pulse.   But these circuits are more difficult to design and their output is uncertain.   * Synchronous sequential circuit – These circuit uses clock signal and level inputs (or pulsed) (with restrictions on pulse width and circuit propagation). The output pulse is the same duration as the clock pulse for the clocked sequential circuits. Since they wait for the next clock pulse to arrive to perform the next operation, so these circuits are bit slower compared to asynchronous. Level output changes state at the start of an input pulse and remains in that until the next input or clock pulse.   We use synchronous sequential circuit in synchronous counters, flip flops, and in the design of MOORE-MEALY state management machines.  We use sequential circuits to design Counters, Registers, RAM, MOORE/MEALY Machine and other state retaining machines.  The behaviour of a clocked sequential circuit is determined from   * The inputs * The outputs * The state of its flip-flops   The outputs and the next state are both a function of the inputs and the present state .To analyze a sequential circuit, we can use State equations, State table, State diagram, and Flip-Flop input equations. Digital Clock Circuit Design Using 7493 The 4 blocks of a digital clock are   * 1 Hz clock generator to generate 1 PPS (pulse per second) signal to the seconds block. * SECONDS block - contains a divide by 10 circuit followed by a divide by 6 circuit. Will generate a 1 PPM (pulse per minute) signal to the minutes block. The [BCD](http://electronics-course.com/number-systems) outputs connect to the [BCD to Seven Segment circuit](http://electronics-course.com/bcd-7-segment) to display the seconds values. * MINUTES block - identical to the seconds block it contains 2 dividers; a divide by 10 followed by a divide by 6. Will generate a 1 PPH (pulse per hour) signal to the HOURS block. The BCD outputs connects to the BCD to Seven Segment circuit to display the minutes values. * HOURS block - depending on whether it is a 12 or 24H clock, will have a divide 24 or divide by 12. For 24H, it will count from 00 to 23. For 12H, it will count from 00 to 11. The BCD outputs connects to the BCD to Seven Segment circuit to display the hours values.   The clock can be designed as a 24H or 12H clock. We will explain the steps to arrive at the combinational logic to obtain a 12H clock and we will leave it to the reader to design the 24H clock as an exercise. Click [hints](http://electronics-course.com/digital-clock#hide) if you need help to design the 24H clock.  12H Clock:   * In order to use all 4 bits of the IC1 (ones) counter, Q0 must be connected to CP1. Q0 is LSB and Q3 is MSB. The input clock is connected to CP0. * Since less than 3 bits are required for IC2 (tens), Q0 is not used. Q1 is LSB and Q3 is MSB. The input clock is connected to CP1. * The truth table of the counter is abbreviated - omitting those rows where the MR inputs to the counters are 0. Recall that for the 7493, a 1 to the MR will reset the counters to 0. * To simplify the table, K is Q0 of IC1 (ones), G is Q0 of IC2 (tens) and so on. * For the 12H clock, when the count in BCD reaches   + 0A, IC1 must be cleared (Y=1)   + 12, IC1 must be cleared (Y=1) and IC2 must be cleared (X=1) * Using [SOP (sum of products)](http://electronics-course.com/sum-of-products), we obtain   + Y = HJ + GJ where Y is the IC1 MR1, MR2 inputs connected together   + X = GJ where X is the IC2 MR1, MR2 inputs connected together |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Date:** | 30 May 2020 | **Name:** | Anupama J S |
| **Course:** | Python | **USN:** | 4AL16EC005 |
| **Topic:** | 1. Python for image and video process. | **Semester & Section:** | 8th sem “A”section |
| **Github Repository:** | AnupamaJS |  |  |
| **AFTERNOON SESSION DETAILS** | | | |
| **A BIT ABOUT THE RGB MODEL**  **C:\Users\User\Downloads\WhatsApp Image 2020-05-30 at 8.49.26 PM.jpeg**  Computer graphics often use the RBG model, which stands for Red, Green and Blue. These are the three primary colors that can be used to create other colors.  The main thing you need to know is that you can create different colors by combining these primary colors. RGB colors usually have values of 0-255, where 0 means the color isn’t present at all, and 255 means it’s present with full strength. So the Rgb values for the color red are:  255, 0 , 0  So the first part is 255, which is Red. The other two are zero. This will create pure red.  You can create other colors by mixing these three. For example, pink is:  255, 51, 255  Red part is 255, green  is 51 and blue is 255 again.  I found these values by Googling *rgb codes*, and opening one the dozens of results that come up.  The only other thing you need to know is OpenCv inverts this. So instead of RGB, you have BGR, or Blue, green, red. DISPLAY AN IMAGE So we are going to start really simple. How to display an image on the screen.  You might be surprised at how hard even this simple thing is. Try to search for how to display an image with Python, and you won’t find many results. I had to find a complicated example and extract the code from that.  Fire up a Python prompt and type:  import cv2  To our code:  import cv2  import sys  We import OpenCv and sys. sys will be used for reading from the command line.  # Read the image. The first command line argument is the image  image = cv2.imread(sys.argv[1])  The function to read from an image into OpenCv is imread(). We give it the arugment of sys.argv[1], which is just the first commandline argument. The image is read in a variable called image.  cv2.imshow("Image", image)  cv2.waitKey(0)  imshow() is the function that displays the image on the screen. The first value is the title of the window, the second is the image file we have previously read. cv2.waitKey(0) is required so that the image doesn’t close immediately. It will wait for a key press before closing the image.  python display.py ship.jpg  https://www.pythonforengineers.com/wp-content/uploads/2017/04/image1.png  And you should see the image. BLUR AND GRAYSCALE Two important functions in image processing are blurring and grayscale. Many image processing operations take place on grayscale (or black and white) images, as they are simpler to process (having just two colors).  Similarly, blurring is also useful in edge detection, as we will see in later examples. Open the   import cv2  import sys    # The first argument is the image  image = cv2.imread(sys.argv[1])  This is the same as before.  #convert to grayscale  gray\_image = cv2.cvtColor(image, cv2.COLOR\_BGR2GRAY)  First, we convert the image to gray. The function that does that is cvtColor(). The first argument is the image to be converted, the second is the color mode. COLOR\_BGR2GRAY stands  for Blue Green Red to Gray.  You must have heard of the RGB color scheme. OpenCv does it the other way round- so blue is first, then green, then red.  #blur it  blurred\_image = cv2.GaussianBlur(image, (7,7), 0)  If you have ever used Photoshop (or its ugly cousin Gimp), you may have heard of the Gaussian blur. It is the most popular function to blur images, as it offers good blurring at fairly fast speed. That’s what we’ll use.  The first argument is the image itself.  The second argument is the window size. Gaussian Blur works over a small window, and blurs all the pixels in that window (by averaging their values). The larger the window, the more blurring will be done, but the code will also be slower. I’m choosing a window of (7,7) pixels, which is a box 7 pixels long and 7 pixels wide. The last value is not important, so I’m setting it to the default(0).  # Show all 3 images  cv2.imshow("Original Image", image)  cv2.imshow("Gray Image", gray\_image)  cv2.imshow("Blurred Image", blurred\_image)  cv2.waitKey(0)  And now we show all images.  python blur.py ship.jpg  https://www.pythonforengineers.com/wp-content/uploads/2017/04/image2.png EDGE DETECTION Edge detection is a very useful function in image processing. Edge detection means detecting where the edges of an object in an image are. The algorithm looks for things like change in color, brightness etc to find the edges.  The most pioneering work in this domain was done by John Canny, and his algorithm is still the most popular. You don’t need to understand how the algorithms work under the hood to use them, but if you are interested in learning more, Wikipedia has good summaries:  <http://en.wikipedia.org/wiki/Edge\_detection>  <http://en.wikipedia.org/wiki/Canny\_edge\_detector>  We will jump straight into the code. Open edge\_detect.py.  All this should be familiar, as it is similar to the last section.  import cv2  import sys    # The first argument is the image  image = cv2.imread(sys.argv[1])    #convert to grayscale  gray\_image = cv2.cvtColor(image, cv2.COLOR\_BGR2GRAY)    #blur it  blurred\_image = cv2.GaussianBlur(gray\_image, (7,7), 0)    cv2.imshow("Orignal Image", image)  canny = cv2.Canny(blurred\_image, 10, 30)  cv2.imshow("Canny with low thresholds", canny)  The function for Canny edge detection is, unsurprisingly, called Canny(). It takes three  arguments. The first is the image. The second and third are the lower and upper thresholds respectively.  The Canny edge detector detects edges by looking in the difference of pixel intensities. Now, I could spend hours explaining what that means, or I could just show you. So bear with me for a moment. For the first example above, I’m using low thresholds of 10, 30, which means a lot of thresholds will be detected.  canny2 = cv2.Canny(blurred\_image, 50, 150)  cv2.imshow("Canny with high thresholds", canny2)  In this second example, we will use higher thresholds. Let’s see what that means.  python edge\_detect.py ship.jpg  https://www.pythonforengineers.com/wp-content/uploads/2017/04/image3.png  The leftmost is the original image. The middle is the one with low thresholds. You can see it detected a lot of edges. Look inside the ship. The algorithm detected the windows of the ship, as well as a small hatch near the front. But it also detected a lot of unnecessary details in the sea. The rightmost image has the high thresholds. It didn’t detect the unneeded info in the sea, but it also failed to detect the windows in the ship.  So how will you choose the thresholds? One thing I will say repeatedly in this chapter- there are no fixed answers. Try different values till you find ones you like. This is because the values will depend on your application and the type of images you are working with.  Before I close this section, a bit of info about the image. I took the photo in Southampton when on a river cruise. The ship is at the exact place where the Titanic sailed from. That parking spot costs £1000 a day (around $1500). Still cheap for a £30 million ship. FACE DETECTIONC:\Users\User\Downloads\WhatsApp Image 2020-05-30 at 8.49.26 PM (1).jpeg **Face detection with OpenCV**  OpenCV uses machine learning algorithms to search for faces within a picture. For something as complicated as a face, there isn’t one simple test that will tell you if it found a face or not. Instead, there are thousands of small patterns/features that must be matched. The algorithms break the task of identifying the face into thousands of smaller, bite-sized tasks, each of which is easy to solve. These tasks are also called classifiers.  For something like a face, you might have 6,000 or more classifiers, all of which must match for a face to be detected (within error limits, of course). But therein lies the problem: For face detection, the algorithm starts at the top left of a picture and moves down across small blocks of data, looking at each block, constantly asking, “Is this a face? … Is this a face? … Is this a face?” Since there are 6,000 or more tests per block, you might have millions of calculations to do, which will grind your computer to a halt.  https://www.pythonforengineers.com/wp-content/uploads/2017/04/face1.jpg  The image above is a rough example of how face detection works. The algorithm breaks the image into small blocks of pixels, and does the face detection on each.  To get around this, OpenCV uses cascades. What’s a cascade? The best answer can be found from the dictionary: A waterfall or series of waterfalls  Like a series of waterfalls, the OpenCV cascade breaks the problem of detecting faces into multiple stages. For each block, it does a very rough and quick test. If that passes, it does a slightly more detailed test, and so on.  https://www.pythonforengineers.com/wp-content/uploads/2017/04/face2.jpg  The algorithm may have 30-50 of these stages or cascades, and it will only detect a face if all stages pass. The advantage is that the majority of the pictures will return negative during the first few stages, which means the algorithm won’t waste time testing all 6,000 features on it. Instead of taking hours, face detection can now be done in real time.  **Cascades in practice**  Though the theory may sound complicated, in practice it is quite easy. The cascades themselves are just a bunch of XML files that contain OpenCV data used to detect objects. You initialize your code with the cascade you want, and then it does the work for you.  Since face detection is such a common case, OpenCV comes with a number of built-in cascades for detecting everything from faces to eyes to hands and legs. There are even cascades for non-human things. For example, if you run a banana shop and want to track people stealing bananas, [this guy](http://coding-robin.de/2013/07/22/train-your-own-opencv-haar-classifier.html) has built one for that | | | |