**DAILY ASSESSMENT FORMAT**

|  |  |  |  |
| --- | --- | --- | --- |
| **Date:** | **29-May-2020** | **Name:** | **Raziya Banu** |
| **Course:** | **LD** | **USN:** | **4AL16EC058** |
| **Topic:** | **Boolean equations for digital circuits. Combinational circuits: Conversion of MUX and Decoders to logic gates.** | **Semester & Section:** | **8th sem & ‘B’ section** |
| **Github Repository:** |  |  |  |

|  |
| --- |
| **FORENOON SESSION DETAILS** |
| **Image of session** |
| **Report –**  In my first session today I have studied about the **Boolean equations for digital circuits. Combinational circuits: Conversion of MUX and Decoders to logic gates.**  **Binary Adder**  The most basic arithmetic operation is addition. The circuit, which performs the addition of two binary numbers is known as **Binary adder**. First, let us implement an adder, which performs the addition of two bits.  **Half Adder**  Half adder is a combinational circuit, which performs the addition of two binary numbers A and B are of **single bit**. It produces two outputs sum, S & carry, C.  The **Truth table** of Half adder is shown below.   |  |  |  |  | | --- | --- | --- | --- | | **Inputs** | | **Outputs** | | | A | B | C | S | | 0 | 0 | 0 | 0 | | 0 | 1 | 0 | 1 | | 1 | 0 | 0 | 1 | | 1 | 1 | 1 | 0 |   When we do the addition of two bits, the resultant sum can have the values ranging from 0 to 2 in decimal. We can represent the decimal digits 0 and 1 with single bit in binary. But, we can’t represent decimal digit 2 with single bit in binary. So, we require two bits for representing it in binary.  Let, sum, S is the Least significant bit and carry, C is the Most significant bit of the resultant sum. For first three combinations of inputs, carry, C is zero and the value of S will be either zero or one based on the **number of ones** present at the inputs. But, for last combination of inputs, carry, C is one and sum, S is zero, since the resultant sum is two.  From Truth table, we can directly write the **Boolean functions** for each output as  S=A⊕BS=A⊕B  C=ABC=AB  We can implement the above functions with 2-input Ex-OR gate & 2-input AND gate. The **circuit diagram** of Half adder is shown in the following figure.  Half Adder  In the above circuit, a two input Ex-OR gate & two input AND gate produces sum, S & carry, C respectively. Therefore, Half-adder performs the addition of two bits.  **Full Adder**  Full adder is a combinational circuit, which performs the **addition of three bits** A, B and Cin. Where, A & B are the two parallel significant bits and Cin is the carry bit, which is generated from previous stage. This Full adder also produces two outputs sum, S & carry, Cout, which are similar to Half adder.  The **Truth table** of Full adder is shown below.   |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Inputs** | | | **Outputs** | | | **A** | **B** | **Cin** | **Cout** | **S** | | 0 | 0 | 0 | 0 | 0 | | 0 | 0 | 1 | 0 | 1 | | 0 | 1 | 0 | 0 | 1 | | 0 | 1 | 1 | 1 | 0 | | 1 | 0 | 0 | 0 | 1 | | 1 | 0 | 1 | 1 | 0 | | 1 | 1 | 0 | 1 | 0 | | 1 | 1 | 1 | 1 | 1 |   When we do the addition of three bits, the resultant sum can have the values ranging from 0 to 3 in decimal. We can represent the decimal digits 0 and 1 with single bit in binary. But, we can’t represent the decimal digits 2 and 3 with single bit in binary. So, we require two bits for representing those two decimal digits in binary.  Let, sum, S is the Least significant bit and carry, Cout is the Most significant bit of resultant sum. It is easy to fill the values of outputs for all combinations of inputs in the truth table. Just count the **number of ones** present at the inputs and write the equivalent binary number at outputs. If Cin is equal to zero, then Full adder truth table is same as that of Half adder truth table.  We will get the following **Boolean functions** for each output after simplification.  S=A⊕B⊕CinS=A⊕B⊕Cin  cout=AB+(A⊕B)cincout=AB+(A⊕B)cin  The sum, S is equal to one, when odd number of ones present at the inputs. We know that Ex-OR gate produces an output, which is an odd function. So, we can use either two 2input Ex-OR gates or one 3-input Ex-OR gate in order to produce sum, S. We can implement carry, Cout using two 2-input AND gates & one OR gate. The **circuit diagram** of Full adder is shown in the following figure.  Full Adder  This adder is called as **Full adder** because for implementing one Full adder, we require two Half adders and one OR gate. If Cin is zero, then Full adder becomes Half adder. We can verify it easily from the above circuit diagram or from the Boolean functions of outputs of Full adder.  **4-bit Binary Adder**  The 4-bit binary adder performs the **addition of two 4-bit numbers**. Let the 4-bit binary numbers, A=A3A2A1A0A=A3A2A1A0 and B=B3B2B1B0B=B3B2B1B0. We can implement 4-bit binary adder in one of the two following ways.   * Use one Half adder for doing the addition of two Least significant bits and three Full adders for doing the addition of three higher significant bits. * Use four Full adders for uniformity. Since, initial carry Cin is zero, the Full adder which is used for adding the least significant bits becomes Half adder.   For the time being, we considered second approach. The **block diagram** of 4-bit binary adder is shown in the following figure.  Four Bit Binary Adder  Here, the 4 Full adders are cascaded. Each Full adder is getting the respective bits of two parallel inputs A & B. The carry output of one Full adder will be the carry input of subsequent higher order Full adder. This 4-bit binary adder produces the resultant sum having at most 5 bits. So, carry out of last stage Full adder will be the MSB.  In this way, we can implement any higher order binary adder just by cascading the required number of Full adders. This binary adder is also called as **ripple carry**binarybinary**adder** because the carry propagates ripplesripples from one stage to the next stage.  **4-bit Binary Adder / Subtractor**  The 4-bit binary adder / subtractor produces either the addition or the subtraction of two 4-bit numbers based on the value of initial carry or borrow, 𝐶0. Let the 4-bit binary numbers, A=A3A2A1A0A=A3A2A1A0 and B=B3B2B1B0B=B3B2B1B0. The operation of 4-bit Binary adder / subtractor is similar to that of 4-bit Binary adder and 4-bit Binary subtractor.  Apply the normal bits of binary numbers A and B & initial carry or borrow, C0 from externally to a 4-bit binary adder. The **block diagram** of 4-bit binary adder / subtractor is shown in the following figure.  Adder and Subtractor  If initial carry, 𝐶0 is zero, then each full adder gets the normal bits of binary numbers A & B. So, the 4-bit binary adder / subtractor produces an output, which is the **addition of two binary numbers** A & B.  If initial borrow, 𝐶0 is one, then each full adder gets the normal bits of binary number A & complemented bits of binary number B. So, the 4-bit binary adder / subtractor produces an output, which is the **subtraction of two binary numbers** A & B. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Date:** | **29-May-2020** | **Name:** | **Raziya Banu** | |
| **Course:** | **Udemy** | **USN:** | **4AL16EC058** | |
| **Topic:** | **Tuples in Python** | **Semester & Section:** | **8th sem & ‘B’ section** | |
| **AFTERNOON SESSION DETAILS** | | | |
| **Image of session** | | | |
| **Python Tuples:**Tuple A tuple is a collection which is ordered and **unchangeable**. In Python tuples are written with round brackets.  **Example**  Create a Tuple:  thistuple = ("apple", "banana", "cherry") print(thistuple) Access Tuple Items You can access tuple items by referring to the index number, inside square brackets:  **Example**  Print the second item in the tuple:  thistuple = ("apple", "banana", "cherry") print(thistuple[1]) Negative Indexing Negative indexing means beginning from the end, -1 refers to the last item, -2 refers to the second last item etc.  **Example**  Print the last item of the tuple:  thistuple = ("apple", "banana", "cherry") print(thistuple[-1]) Range of Indexes You can specify a range of indexes by specifying where to start and where to end the range.  When specifying a range, the return value will be a new tuple with the specified items.  **Example**  Return the third, fourth, and fifth item:  thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango") print(thistuple[2:5]) Range of Negative Indexes Specify negative indexes if you want to start the search from the end of the tuple:  **Example**  This example returns the items from index -4 (included) to index -1 (excluded)  thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango") print(thistuple[-4:-1]  **Change Tuple Values**  Once a tuple is created, you cannot change its values. Tuples are **unchangeable**, or **immutable** as it also is called.  But there is a workaround. You can convert the tuple into a list, change the list, and convert the list back into a tuple.  **Example**  Convert the tuple into a list to be able to change it:  x = ("apple", "banana", "cherry") y = list(x) y[1] = "kiwi" x = tuple(y)  print(x) | | | |