

DAILY ASSESSMENT FORMAT

Date:	21/6/2020	Name:	SAMRUDDI SHETTY
Course:	Python	USN:	4AL16IS047
Topic:	DAY 5	Semester & Section:	8 th sem
Github Repository:			

AFTERNOON SESSION DETAILS

Python Object Oriented Programming

Python has been an object-oriented language since it existed. Because of this, creating and using classes and objects are downright easy. This chapter helps you become an expert in using Python's object-oriented programming support.

Overview of OOP Terminology

- **Class** – A user-defined prototype for an object that defines a set of attributes that characterize any object of the class. The attributes are data members (class variables and instance variables) and methods, accessed via dot notation.
- **Class variable** – A variable that is shared by all instances of a class. Class variables are defined within a class but outside any of the class's methods. Class variables are not used as frequently as instance variables are.
- **Data member** – A class variable or instance variable that holds data associated with a class and its objects.
- **Function overloading** – The assignment of more than one behavior to a particular function. The operation performed varies by the types of objects or arguments involved.
- **Instance variable** – A variable that is defined inside a method and belongs only to the current instance of a class.
- **Inheritance** – The transfer of the characteristics of a class to other classes that are derived from it.
- **Instance** – An individual object of a certain class. An object obj that belongs to a class Circle, for example, is an instance of the class Circle.
- **Instantiation** – The creation of an instance of a class.
- **Method** – A special kind of function that is defined in a class definition.
- **Object** – A unique instance of a data structure that's defined by its class. An object comprises both data members (class variables and instance variables) and methods.
- **Operator overloading** – The assignment of more than one function to a particular operator.

Creating Classes

The `class` statement creates a new class definition. The name of the class immediately follows the keyword `class` followed by a colon as follows –

```
class ClassName:
```

```
    'Optional class documentation string'
```

```
    class_suite
```

- The class has a documentation string, which can be accessed via `ClassName.doc` .
- The `class_suite` consists of all the component statements defining class members, data attributes and functions.

Example

Following is the example of a simple Python class –

```
class Employee:
    'Common base class for all employees'
    empCount = 0

    def __init__(self, name, salary):
        self.name = name
        self.salary = salary
        Employee.empCount += 1

    def displayCount(self):
        print "Total Employee %d" % Employee.empCount

    def displayEmployee(self):
        print "Name : ", self.name, ", Salary: ", self.salary
```

- The variable `empCount` is a class variable whose value is shared among all instances of a this class. This can be accessed as `Employee.empCount` from inside the class or outside the class.
- The first method `init ()` is a special method, which is called class constructor or initialization method that Python calls when you create a new instance of this class.
- You declare other class methods like normal functions with the exception that the first argument to each method is `self`. Python adds the `self` argument to the list for you; you do not need to include it when you call the methods.

Creating Instance Objects

To create instances of a class, you call the class using `class name` and pass in whatever arguments its `__init__` method accepts.

"This would create first object of Employee class"

```
emp1 = Employee("Zara", 2000)
```

"This would create second object of Employee class"

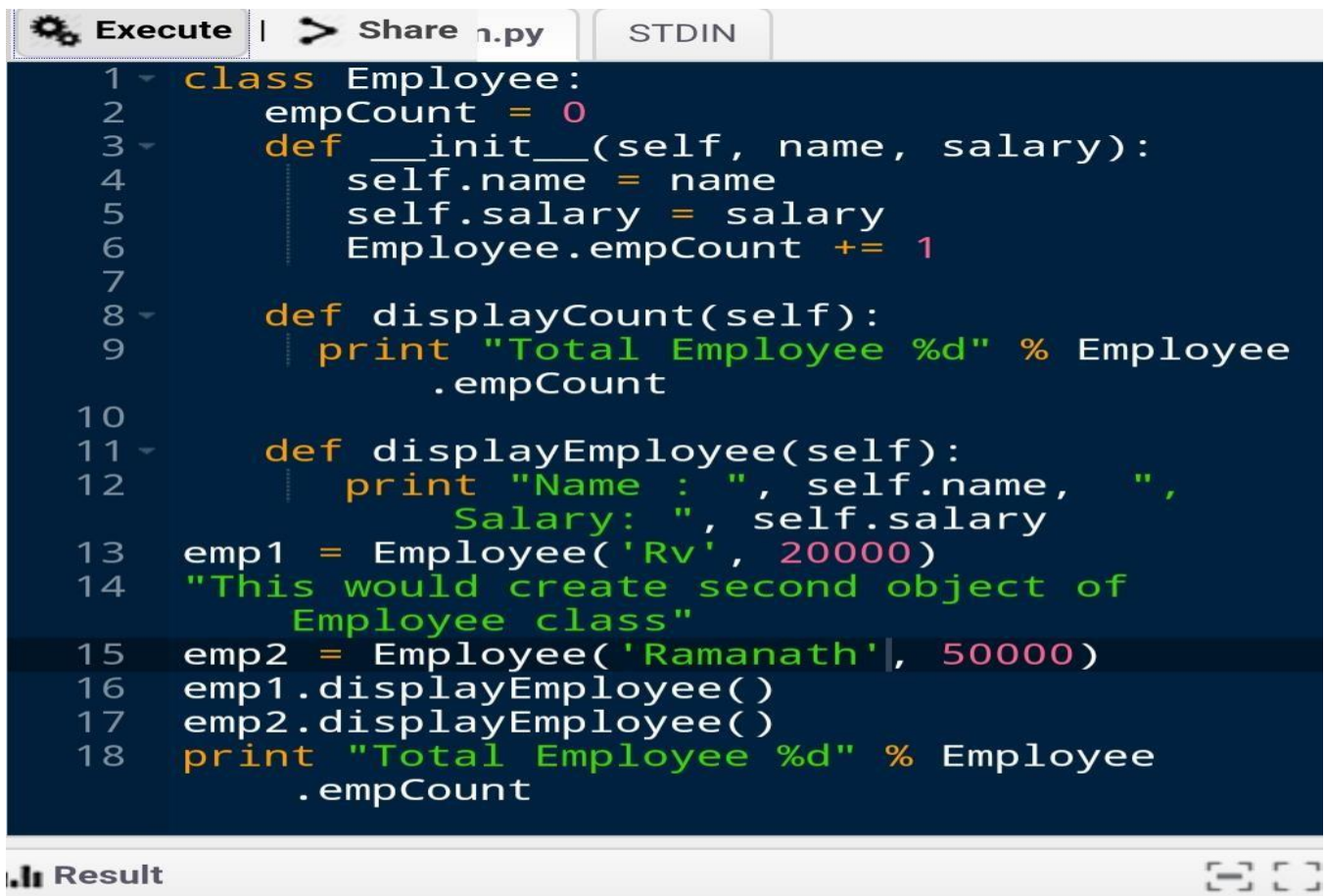
```
emp2 = Employee("Manni", 5000)
```

Accessing Attributes

You access the object's attributes using the dot operator with object. Class variable would be accessed using class name as follows –

```
emp1.displayEmployee()  
emp2.displayEmployee()  
print "Total Employee %d" % Employee.empCount
```

Now, putting all the concepts together –



The screenshot shows a Python IDE with a dark blue background. At the top, there are tabs for 'Execute', 'Share', 'n.py', and 'STDIN'. The code is as follows:

```
1 class Employee:  
2     empCount = 0  
3     def __init__(self, name, salary):  
4         self.name = name  
5         self.salary = salary  
6         Employee.empCount += 1  
7  
8     def displayCount(self):  
9         print "Total Employee %d" % Employee.  
            .empCount  
10  
11     def displayEmployee(self):  
12         print "Name : ", self.name, " ,  
            Salary: ", self.salary  
13 emp1 = Employee('Rv', 20000)  
14 "This would create second object of  
    Employee class"  
15 emp2 = Employee('Ramanath', 50000)  
16 emp1.displayEmployee()  
17 emp2.displayEmployee()  
18 print "Total Employee %d" % Employee.  
    .empCount
```

Below the code editor, there is a 'Result' section with a bar chart icon and a 'python main.py' command. The output is:

```
Name :  Rv , Salary:  20000  
Name :  Ramanath , Salary:  50000  
Total Employee 2
```

You can add, remove, or modify attributes of classes and objects at any time –

```
emp1.age = 7 # Add an 'age' attribute.  
emp1.age = 8 # Modify 'age' attribute.  
del emp1.age # Delete 'age' attribute.
```

Instead of using the normal statements to access attributes, you can use the following functions –

- The **getattr(obj, name[, default])** – to access the attribute of object.
- The **hasattr(obj,name)** – to check if an attribute exists or not.
- The **setattr(obj,name,value)** – to set an attribute. If attribute does not exist, then it would be created.
- The **delattr(obj, name)** – to delete an attribute.

hasattr(emp1, 'age') # Returns true if 'age' attribute exists

getattr(emp1, 'age') # Returns value of 'age' attribute

setattr(emp1, 'age', 8) # Set attribute 'age' at 8

delattr(emp1, 'age') # Delete attribute 'age'

Built-In Class Attributes

Every Python class keeps following built-in attributes and they can be accessed using dot operator like any other attribute –

- **__dict__** – Dictionary containing the class's namespace.
- **__doc__** – Class documentation string or none, if undefined.
- **__name__** – Class name.
- **__module__** – Module name in which the class is defined. This attribute is " main " in interactive mode.
- **__bases__** – A possibly empty tuple containing the base classes, in the order of their occurrence in the base class list.

For the above class let us try to access all these attributes –

```
#!/usr/bin/python
```

```
class Employee:
```

```
    'Common base class for all employees'
```

```
    empCount = 0
```

```
    def __init__(self, name, salary):
```

```
        self.name = name
```

```
        self.salary = salary
```

```
        Employee.empCount += 1
```

```
    def displayCount(self):
```

```
        print "Total Employee %d" % Employee.empCount
```

```
    def displayEmployee(self):
```

```
        print "Name : ", self.name, ", Salary: ", self.salary
```

```
print "Employee.__doc__:", Employee.__doc__
```

```
print "Employee.__name__:", Employee.__name__
```

```
print "Employee.__module__:", Employee.__module__  
print "Employee.__bases__:", Employee.__bases__  
print "Employee.__dict__:", Employee.__dict__
```

When the above code is executed, it produces the following result –

```
Employee._doc_: Common base class for all employees  
Employee._name_: Employee  
Employee.__module__: __main__  
Employee._bases_: ()  
Employee._dict_: {'_module_': '__main__', 'displayCount':  
<function displayCount at 0xb7c84994>, 'empCount': 2,  
'displayEmployee': <function displayEmployee at 0xb7c8441c>,  
'_doc_': 'Common base class for all employees',  
'__init__': <function __init__ at 0xb7c846bc>}
```

Class Inheritance

Instead of starting from scratch, you can create a class by deriving it from a preexisting class by listing the parent class in parentheses after the new class name.

The child class inherits the attributes of its parent class, and you can use those attributes as if they were defined in the child class. A child class can also override data members and methods from the parent

```
class Parent:      # define parent class  
    parentAttr = 100  
    def __init__(self):  
        print "Calling parent constructor"  
  
    def parentMethod(self):  
        print 'Calling parent method'  
  
    def setAttr(self, attr):  
        Parent.parentAttr = attr  
  
    def getAttr(self):  
        print "Parent attribute :", Parent.parentAttr  
  
class Child(Parent): # define child class  
    def __init__(self):  
        print "Calling child constructor"  
  
    def childMethod(self):  
        print 'Calling child method'  
  
c = Child()      # instance of child  
c.childMethod()  # child calls its method  
c.parentMethod() # calls parent's method
```

```
c.setAttr(200)    # again call parent's method  
c.getAttr()      # again call parent's method
```

When the above code is executed, it produces the following result –

Calling child constructor

Calling child method

Calling parent method

Parent attribute : 200

Similar way, you can drive a class from multiple parent classes as follows –

```
class A:          # define your class A
```

```
.....
```

```
class B:          # define your class B
```

```
.....
```

```
class C(A, B):    # subclass of A and B
```

```
.....
```

You can use `issubclass()` or `isinstance()` functions to check a relationships of two classes and instances.

- The **`issubclass(sub, sup)`** boolean function returns true if the given subclass **`sub`** is indeed a subclass of the superclass **`sup`**.
- The **`isinstance(obj, Class)`** boolean function returns true if *obj* is an instance of class *Class* or is an instance of a subclass of *Class*

Date:	29/5/2020	Name:	Ramanath Naik
Course:	Logic design	USN:	4AL16EC054
Topic:	DAY 2	Semester & Section:	8 th sem B sec
Github Repository:			

FORENOON SESSION DETAILS

Sequential circuits

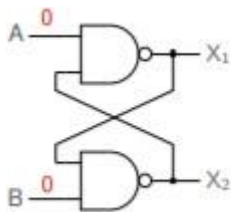
*The digital circuits we have seen so far (gates, multiplexer, demultiplexer, encoders, decoders) are combinatorial in nature, i.e., the output(s) depends only on the present values of the inputs and not on their past values.

* In sequential circuits, the “state” of the circuit is crucial in determining the output values. For a given input combination, a sequential circuit may produce different output values, depending on its previous state.

* In other words, a sequential circuit has a memory (of its past state) whereas a combinatorial circuit has no memory.

* Sequential circuits (together with combinatorial circuits) make it possible to build several useful applications, such as counters, registers, arithmetic/logic unit (ALU), all the way to microprocessors.

NAND latch (RS latch)



A	B	X ₁	X ₂
1	0	0	1
0	1	1	0
1	1	previous	
0	0		

* A, B: inputs, X₁, X₂: outputs

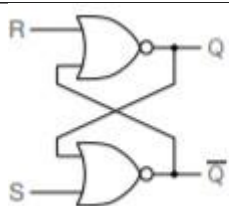
* Consider A = 1, B = 0. $B = 0 \Rightarrow X_2 = 1 \Rightarrow X_1 = A X_2 = 1 \cdot 1 = 0$. Overall, we have $X_1 = 0$, $X_2 = 1$.

* Consider A = 0, B = 1. $\rightarrow X_1 = 1$, $X_2 = 0$.

* Consider A = B = 1. $X_1 = A X_2 = X_2$, $X_2 = B X_1 = X_1 \Rightarrow X_1 = X_2$. If $X_1 = 1$, $X_2 = 0$ previously, the circuit continues to “hold” that state. Similarly, if $X_1 = 0$, $X_2 = 1$ previously, the circuit continues to “hold” that state. The circuit has “latched in” the previous state.

* For A = B = 0, X₁ and X₂ are both 1. This combination of A and B is not allowed for reasons that will become clear later.

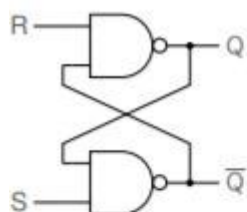
NOR latch (RS latch)



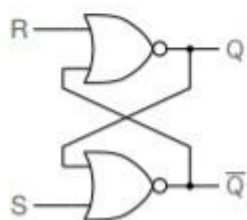
R	S	Q	\bar{Q}
1	0	0	1
0	1	1	0
0	0	previous	
1	1	invalid	

- * The NOR latch is similar to the NAND latch: When $R = 1, S = 0$, the latch gets reset to $Q = 0$. When $R = 0, S = 1$, the latch gets set to $Q = 1$.
- * For $R = S = 0$, the latch retains its previous state (i.e., the previous values of Q and \bar{Q}).
- * $R = S = 1$ is not allowed for reasons similar to those discussed in the context of the NAND latch.

Comparison of NAND and NOR latches

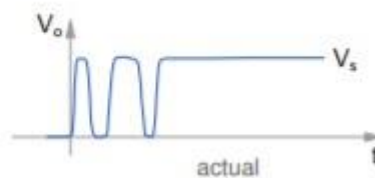
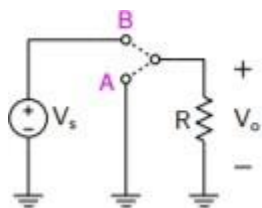


R	S	Q	\bar{Q}
1	0	0	1
0	1	1	0
1	1	previous	
0	0	invalid	

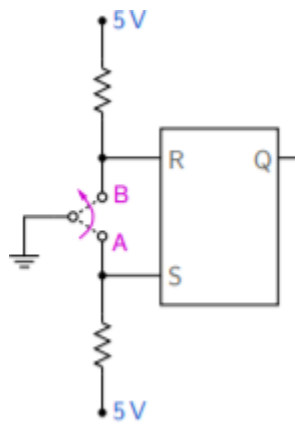


R	S	Q	\bar{Q}
1	0	0	1
0	1	1	0
0	0	previous	
1	1	invalid	

Chatter (bouncing) due to a mechanical switch



- * When the switch is thrown from A to B, V_o is expected to go from 0 V to V_s (say, 5 V).
- * However, mechanical switches suffer from "chatter" or "bouncing," i.e., the transition from A to B is not a single, clean one. As a result, V_o oscillates between 0 V and 5 V before settling to its final value (5 V).
- * In some applications, this chatter can cause malfunction \rightarrow need a way to remove the chatter.



R	S	Q	\bar{Q}
1	0	0	1
0	1	1	0
1	1	previous	
0	0	invalid	

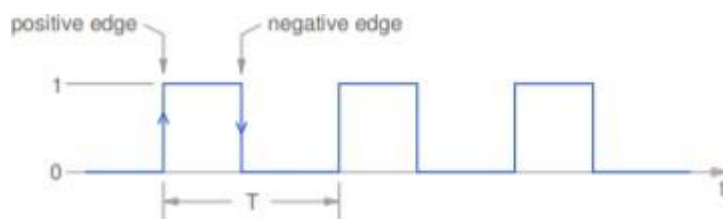


*

Because of the chatter, the S and R inputs may have multiple transitions when the switch is thrown from A to B.

* However, for $S = R = 1$, the previous value of Q is retained, causing a single transition in Q, as desired.

CLOCK



* Complex digital circuits are generally designed for synchronous operation, i.e., transitions in the various signals are synchronized with the clock.

* Synchronous circuits are easier to design and troubleshoot because the voltages at the nodes (both output nodes and internal nodes) can change only at specific times.

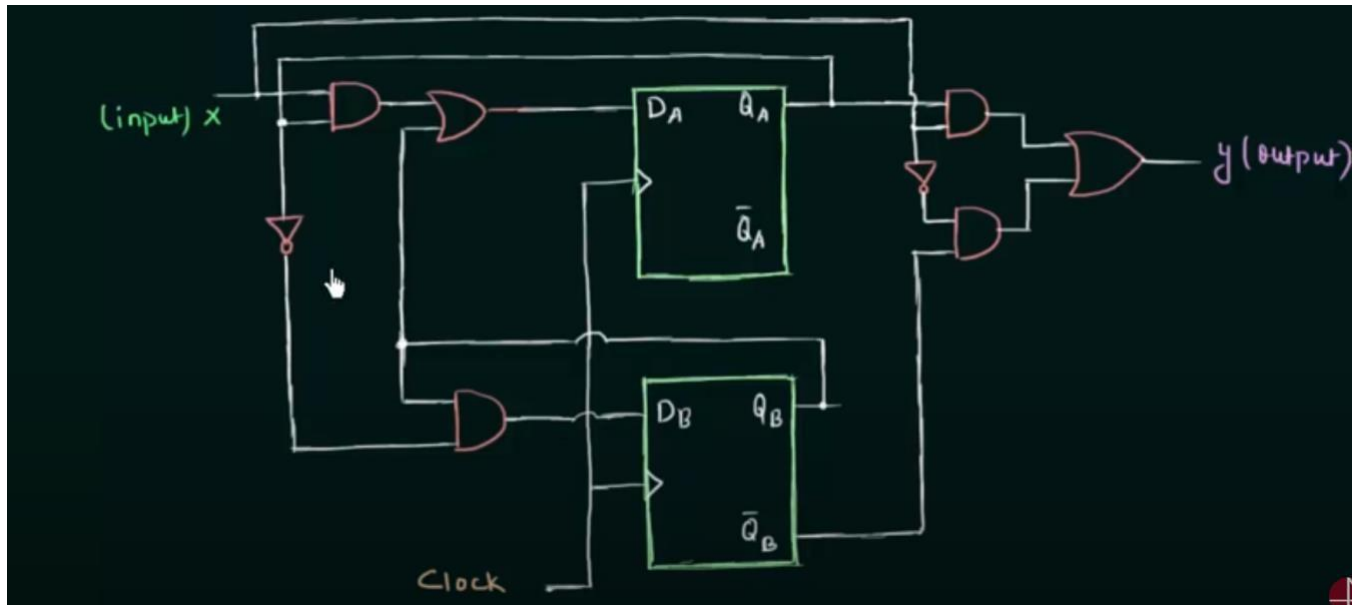
* A clock is a periodic signal, with a positive-going transition and a negative-going transition.

* The clock frequency determines the overall speed of the circuit. For example, a processor that operates with a 1 GHz clock is 10 times faster than one that operates with a 100 MHz clock. Intel 80286 (IBM PC-AT): 6 MHz Modern CPU chips: 2 to 3 GHz.

Analysis of clocked sequential circuits

Optimum goal of the presentation is to find state diagram of the circuits.

1) D-FLIP FLOP



Step1 – find out input/output equation

$$DA = XQA + QB$$

$$DB = QA'QB$$

$$Y = X'QB' + XQA$$

Step2 – state table

The value of $D = Q_{n+1}$ (next state)

$$QA_{+} = DA = XQA + QB$$

$$QB_{+} = DB = QA'QB$$

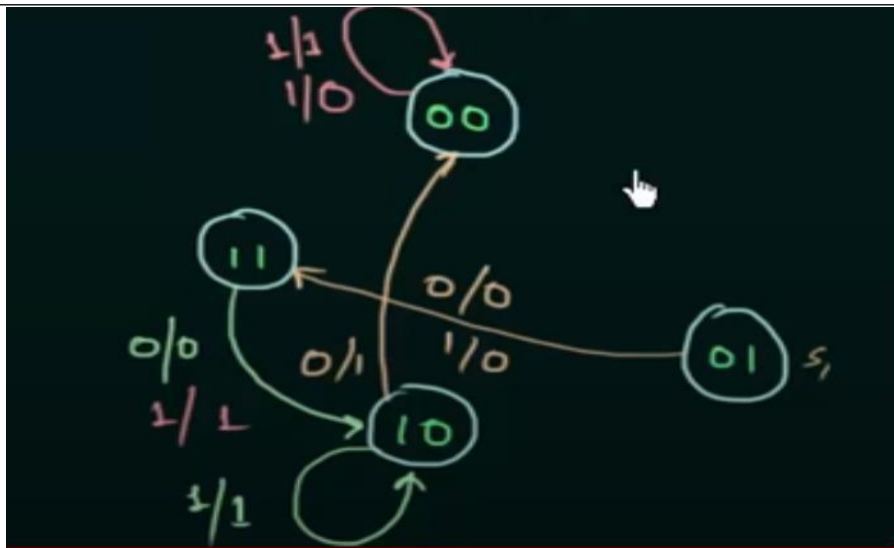
using these values, we find out next state

PRESENT STATE			NEXT STATE		
QA	QB	X	QA+	QB+	Y
0	0	0	0	0	1
0	0	1	0	0	0
0	1	0	1	1	0
0	1	1	1	1	0
1	0	0	0	0	1
1	0	1	1	0	1
1	1	0	1	0	0
1	1	1	1	0	1

Step3 – State Diagram

there are 2 flip flops so there are 4 possible states

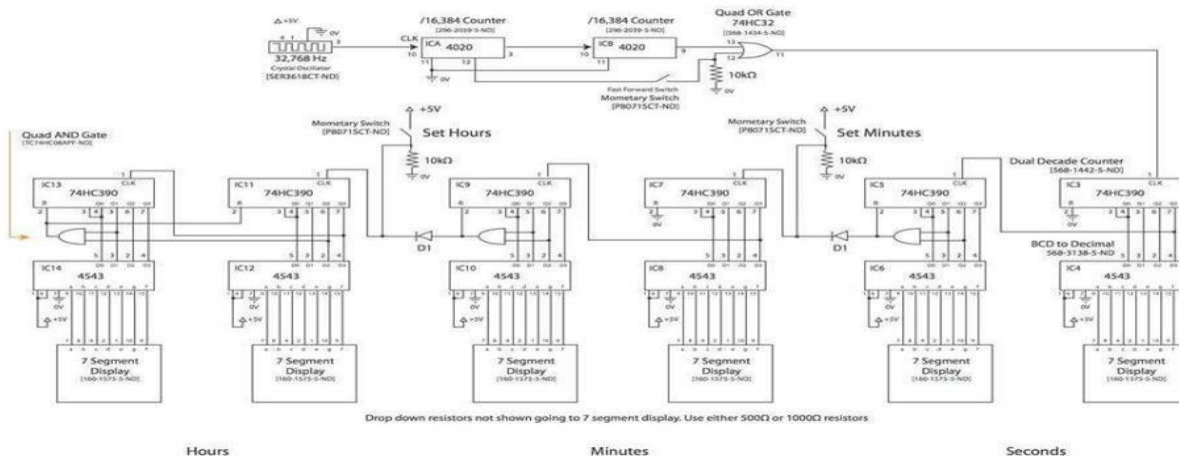
	QA	QB
S0	0	0
S1	0	1
S2	1	0
S3	1	1



24h /12h Digital clock using ic555 ,7490-decade counters and 7447 bcd-7seg decoders

Block Diagram of Circuit

Part Numbers from Digikey are in brackets
by mattosx@me.com



main parts of the circuit are as follows:

- 1- Timer 555: Responsible for generating the clock pulses for the counters, the frequency of the output should be 1 hz which means 1 second for each pulse.
 - 2- Counters: Responsible for generating the time in BCD (Binary Coded decimal).
 - 3- Decoders: Takes the BCD of the counter as input and produces 7 segment output.
 - 4- 7 segments: Displays the time, of course!
- * Seconds have 2 displays, 2 decoders and 2 counters. The same for minutes and hours.

we should know before how the circuit is going to work is that 7490-decade counters respond only to the negative going edge of 555 pulses, which means it will change its state only when the clock goes from 1 to 0.

The circuit works as follows:

555 timer produces 1 second pulses to the clock input of the first counter which is responsible for the first column of seconds, so its output will change every second.

The counter produces numbers from 0 to 9 in BCD form and automatically resets to 0 after that.

so, the output of the first counter will count from 0 to 9 every second and that's exactly what we want from it, so we are done here. let's move to the next one.

What do we want in the circuit is,

First, we want the 2nd counter to start counting when the 1st one moves for 9 to 0 (that makes 10 seconds!)

This can be done as shown,

let's check the output of the first counter in BCD:

MSB---LSB

0: 0000

1: 0001

2: 0010

3: 0011

4: 0100

5: 0101

6: 0110

7: 0111

8: 1000

9: 1001

0: 0000

Remember that 7490-decade counters respond only to the pulses that go from 1 to 0 and notice that this case only happens in the BCD code above when the output changes from 9 to 0 (the Most significant bit changes from 1 to 0). So, we'll just connect the clock input of the 2nd counter to the most significant bit of the output of the first counter.

Second, Since we have 60 seconds in the minute we want the 2nd counter to count only to 5, that makes 59 seconds maximum, when it take another pulse it doesn't count to 60, instead it resets itself to 0 and send a pulse to the first counter in minutes to tell it to count 1 minute

This can be done, From the BCD code above (6: 0110) when the output is 6 the two middle bits are 1 (Q1,Q2),

So By ANDing these two bits the output will be 1, This output will be connected to the reset pin of the same counter (2nd one) and the clock input of the next one(3rd).

When the output is 6 the AND gate output (1) will reset the same counter and its outputs goes 0000 so the output of the and gate again goes to 0 (1 --- >0), that will clock the next counter. Beautiful!

*Notice that the output of the counters are named: Q0 , Q1 , Q2 , Q3

The 4th counter will be the same as the second one so we are clocking it using the Most Significant Bit of the output of the previous one.

Again, the 5th counter is the same as the 3rd one and takes its clock from the AND gate.

The 5th and the 6th counters are responsible for hours so they are limited to 23, and resets themselves to 00 when the 5th counter is 4 and the last one is 2 (24).

This is done using and gate with Q2 (3rd bit) of the 5th counter as one input and Q1 (second bit) of the last counter as the other input, and the output of this AND gate will be connected to both resets of the last 2 counters.

When the last counter is 0(0000) or 1(0001), Q1 which is one of the inputs to the AND gate will be 0 so the output of the AND gate will be zero. when it counts to 2 this bit will be 1 so the output of the and gate will depend on the the other input which is Q2 of the previous counter, and this bit will be zero until it reaches 4 (0100),So, the output of the and gate will be 1 (0--->1) resetting both counters to 00,

The output of these counters is converted to 7 segment output using 7447 decoders, then to the 7 segments, we won't get into the details of their datasheets.

BONUS TUTORIAL-SIMPLIFYING THE BRAIN

The session was by V. Srinivasa Chakravarthy, department of biotechnology, IIT madras, The numbers of neurons in the brain are more than number of stars in the milky way galaxy. The brain is the most complex object in the universe. Reasons for neuroscience include Large projects are there in neuroscience, which needs large number of data. Descriptive tradition of biology is also reason for neuroscience. We do structural analysis then functional analysis to fix a radio by observing its PCB. Discussed about a architecture of a radio in the engineer point of view. Data as the fourth pillar of science. The first 3 are theory, experiment, computation and last is data pillar. Neurons come in different shapes. The updated technologies which uses big data is the reason the neuroscience make brain look complexed. Briefed about neuromorph. The connectome project aims at working on the graph structure of the brain. Human brain has 100 billion neurons. Human connectome project includes 1200 individuals, IMRI+dmRI+MRI+MEG, Washington U+ U. Minnesota. Discussed about history of astronomy like Newtonian astronomy and Ptolemaic astronomy. The problem with computational modelling include includes more biology-style modelling, less physics style modelling, a thin rapper over data, need deep integrative theories that relate diverse phenomena. Discussed about contemporary neuroscience and cortex-subcortex. Modeling the basal ganglia which has multiple structure and has 7 nuclei. Cortex has many areas like visual cortex, motor cortex, sensory cortex. Explained about Functional anatomy of basal ganglia and Basal ganglia functions.