Digital Logic Circuit simulator in python

Meghanad Shingate - 09307608,
Nirbhay Rane - 09307905,
Bharat Kumar - 09307904
Group - 13

AE 663 Course Project Report

1 Introduction

We have implemented *pydlcs*, a Digital Logic Circuit simulator in python as a part of our course project. It is a nice simulation exercise to show power of object oriented programming in python. We have implemented following functionality in our simulator.

- Basic gates - NOT, AND, OR, NAND.
- Derived gates - XOR, XNOR.
- Combinational Circuits - Half adder, Full adder, 1x2 MUX, 2x1 DEMUX.
- Sequential Circuits - D, JK, T flip flops, latches, frequency divider, Shift register and counters.
- Signal Sources - clock, constant signal generator.

2 System Overview

Figure 1 shows the system model of the *pydlcs* simulator. Class *SIMU* is the central class of the *pydlcs* simulator. It monitors the whole circuit operations and provides clock to different elements of the circuit. Its main functions are,

- Provide system clock to circuit elements
- Plotting input and output graphs

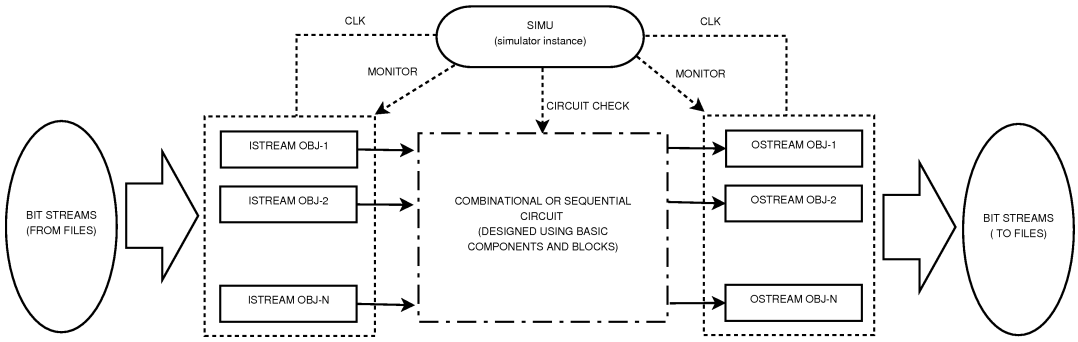

Figure 1: Simulator System Model

- Save the results and plots
- Monitor and control i/o streams
- Circuit debug option

Simulator takes the bit-streams as an input form files specified. *Istream* is the class defined for the providing the input facility from files. Each *Istream* object is connected to one file on one side and can supply bit-stream to any number of gates of the circuit on other side. *Ostream* is class defined for providing facility of writing result of simulation into the file specified. *Ostream* class object is connected to circuit pin from one side and pins data is logged into the file specified on other side. Both classes, *Istream* and *Ostream*, need to provide system clock for their operation. Their are different flags in *SIMU* class that we need to set for enabling different options *e.g.* for enabling annotation of plots we need to set flag *pannotate* in *SIMU* object. Flags details are as bellow,

- *plots* - Enable plots
- *pannotate* - Enable plot annotation
- *pclk* - Enable plotting system clock
- *start* - Start the simulation flag
- *stop* - Stop simulation flag
- *debug* - Enable debugging
- *step* - Enable step execution

There should be only one instance of *SIMU* class per circuit description file. Introduction to writing circuit description file is given in upcoming sections.

3 Implementation Details

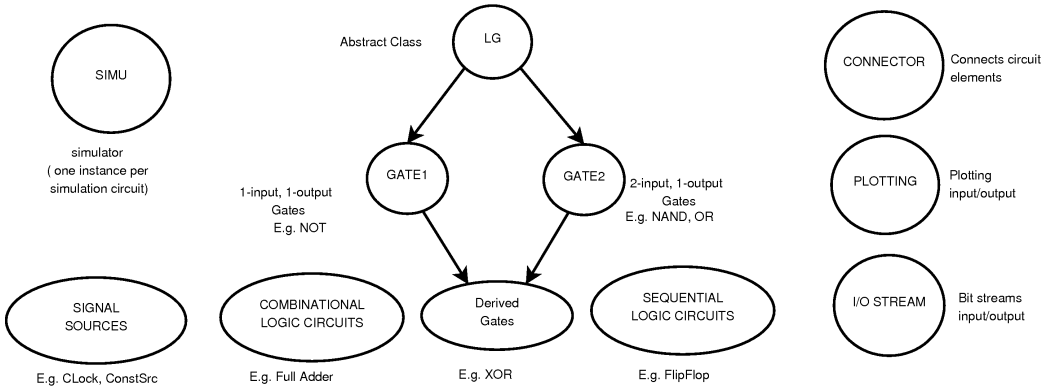

Figure 2: Class Structure

The implementation of Logic simulator is done using a modular and test driven approach. Each of the module is implement using a set of classes. The Logic simulator consists of the following classes.

- Class Logic Gate
- Class Gate
- Classes implementing basic gates.
- Class Connector.
- Classes implementing derived gates.
- Classes implementing combinational logic circuits.
- Classes implementing sequential elements.
- Classes implementing signal sources and clocks.
- Class I/O stream
- Class Simulator

3.1 Class Logic Gate

This is an abstract class which implements a method to name the gate that is created. This name is then used to refer to the pins of this gate and define the the pins to which it is connected. It also has the definition of an evaluate function. This function is overloaded by any class derived from this class. All the class which implement basic logic gates are derived from this class.

This code snippet shows its implementation.

```
class LG :
    # Logic Circuits have names and an evaluation function
       defined in child classes
    # They will also contain a set of inputs and outputs
    def __init__ (self, name) :
        self.name = name
    def evaluate (self) : return
```

3.2 Class Gate

This class is derived from the abstract class Logic Gate. The purpose of this class is to create objects that represent the pins of a gate. In our implementation we have a class named GATE2 which creates terminals of a two input and one output logic gate. The input terminals are named A and B, Output terminal is named C.This implementation can be extended to create gate having arbitrary number of inputs and outputs.The Connector call creates a pin with a particular name and defines weather a change in this pin's value will activate the gate or not. Further details about Connector are provided subsection Connector. The code below shows the implementation.

```
class Gate2 (LG) :          # two input gates. Inputs A and B.
   Output C.
    def __init__ (self, name) :
        LG.__init__ (self, name)
        self.A = Connector(self,'A', activates=1)
        self.B = Connector(self,'B', activates=1)
        self.C = Connector(self,'C')
```

3.3 Classes implementing basic gates

All the basic gates ie AND, OR, NAND etc are derived from the class GATE2 (except NOT which has a single input and output.). Each of these classes overloads the function evaluate defined in the class Logic Gate. The evaluate function

is used to calculate the output of the gate by reading the corresponding inputs.

3.3.1 NOT gate

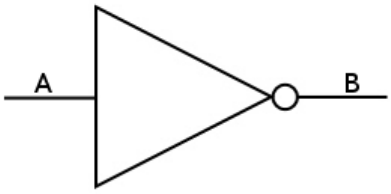

Figure 3: NOT gate

```
# ==========================
# CLASS : NOT GATE (Not)
# ==========================
class Not (LG) :          # Inverter. Input A. Output B.
    def __init__ (self, name) :
        LG.__init__ (self, name)
        self.A = Connector(self,'A', activates=1)
        self.B = Connector(self,'B')
    def evaluate (self) : self.B.set(not self.A.value)
```

We can see that class NOT derived from LG class. It has two attributes for input and output pin. When a event happens on the input pin, evaluate method of this class is called which performs NOT operation.

The code below shows the implementation of AND gate.

```
class And (Gate2) :       # two input AND Gate
    def __init__ (self, name) :
        Gate2.__init__ (self, name)
    def evaluate (self) : self.C.set(self.A.value and self.B.v
```

Similarly we have implemented other basic gates like OR, NAND using above approach.

3.4 Class Connector

This class has no base class that is it is not a derived class. The pourpose of this class is to create pins , to name and initialize them and to attach them to a particular gate. Each of the pin has an attribute called owner which is used to denote the gate to which a pin is connected. The connector also takes arguments called activates and monitor. The activates is a attribute of the pin which tells if a change in the value of a pin activates the gate or not. By default activates is set to zero meaning that the pin will not activate the gate. The activation of the gate cause the function evaluate to execute which produces the output of the gate. The attribute monitor tell if we want to print the value of a pin or not. By default it is set to 0 which means it will not print the value of the pin. Apart from the initialization mentioned above the Connector class is also responsible to

- connect the pins of one gate to another gate
- print the value of a pin if its monotor attribute is set to 1
- set the value of the pins and to call the evaluate function of the appropriate object when it sees a change in the input pins of a particular gate.

The above three functions are implemented using the functions Set and Connect. Set is used to set the value of a pin and call the evaluate function of appropriate gate. This is done by simply calling the evaluate function of the owner of the pin. Connect is used to connect the pins of one gate to another , it dose this by storing all the pins which are connected to that pin in a list. whenever the value of a particular pin changes set reads the connect list and it calls the evaluate function of all the gates corresponding to the pins stored in the connect list.

The code below shows the implementation of the connect function.

```
class Connector :
    # Connectors are inputs and outputs. Only outputs should
       connect
    # to inputs. Be careful NOT to have circular references
    # As an output is changed it propagates the change to its
       connected inputs

    def __init__ (self, owner, name, activates=0, monitor=0) :
        self.value = 0
        self.owner = owner
        self.name  = name
        self.monitor  = monitor
        self.connects = []
        self.activates= activates   # If true change kicks
           evaluate function
```

```
    def connect (self, inputs) :
        if type(inputs) != type([]) : inputs = [inputs]
        for input in inputs : self.connects.append(input)


    def set (self, value) :
        if self.value == value : return      # Ignore if no
            change
        self.value = value
        if self.activates : self.owner.evaluate()
        if self.monitor :
                        print "Connector: %s-%s set to %d" % (
                            self.owner.name,self.name,self.value)
        for con in self.connects : con.set(value)
```

3.5 Classes implementing derived gates

We can see that class XOR is derived from class LG. It uses the basic gate objects of AND, NOT and OR to implement XOR functionality. XNOR gate can be easily implemented with little change in above logic.

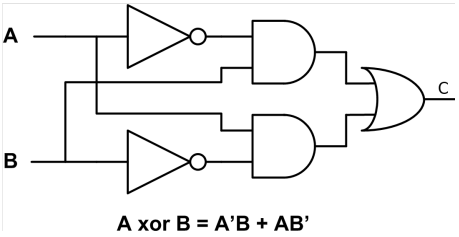

Figure 4: XOR gate

```
# =========================
# CLASS : XOR GATE (Xor)
# =========================
class Xor (Gate2) :
    def __init__ (self, name) :
        Gate2.__init__ (self, name)
        self.A1 = And("A1") # See circuit drawing to follow
            connections
        self.A2 = And("A2")
        self.I1 = Not("I1")
        self.I2 = Not("I2")
        self.O1 = Or ("O1")
        self.A.connect    ([ self.A1.A, self.I2.A])
        self.B.connect    ([ self.I1.A, self.A2.A])
```

```
        self.I1.B.connect ([ self.A1.B ])
        self.I2.B.connect ([ self.A2.B ])
        self.A1.C.connect ([ self.O1.A ])
        self.A2.C.connect ([ self.O1.B ])
        self.O1.C.connect ([ self.C ])
```

3.6 Classes implementing combinational logic circuits

In this section we describes the implementation of combinational logic circuits like half adder, full adder, MUX/DEMUX.

3.6.1 Half Adder

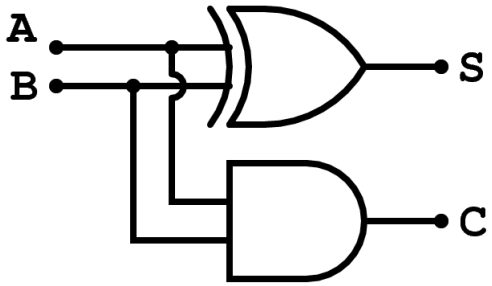

Figure 5: Half Adder

```
# ==================================
# CLASS : HALF ADDER (HalfAdder)
# ==================================
class HalfAdder (LG) :          # One bit adder, A,B in. Sum and
    Carry out
     def __init__ (self, name) :
         LG.__init__ (self, name)
         self.A = Connector(self,'A',1)
         self.B = Connector(self,'B',1)
         self.S = Connector(self,'S')
         self.C = Connector(self,'C')
         self.X1= Xor("X1")
         self.A1= And("A1")
         self.A.connect    ([ self.X1.A, self.A1.A])
         self.B.connect    ([ self.X1.B, self.A1.B])
         self.X1.C.connect ([ self.S])
         self.A1.C.connect ([ self.C])
```

We can see that class HalfAdder is derived from class LG. It uses the gate objects of XOR, AND to implement half adder functionality.

3.6.2 Full Adder

Similarly we have implemented Full Adder class using HalfAdder class objects and OR gate object.

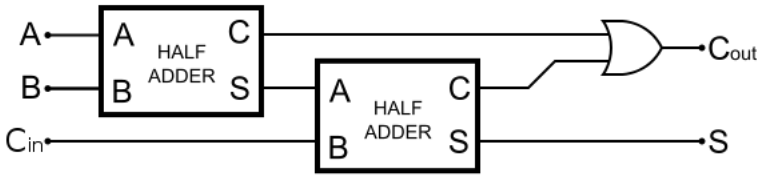

Figure 6: Full Adder

```
# ================================
# CLASS : FULL ADDER (FullAdder)
# ================================

class FullAdder (LG) :          # One bit adder, A,B,Cin in. Sum
    and Cout out
     def __init__ (self, name) :
         LG.__init__ (self, name)
         self.A    = Connector(self,'A',activates = 1)
         self.B    = Connector(self,'B',activates = 1)
         self.Cin  = Connector(self,'Cin',activates = 1)
         self.S    = Connector(self,'S')
         self.Cout = Connector(self,'Cout')
         self.H1= HalfAdder("H1")
         self.H2= HalfAdder("H2")
         self.O1= Or("O1")
         self.A.connect    ([ self.H1.A ])
         self.B.connect    ([ self.H1.B ])
         self.Cin.connect  ([ self.H2.A ])
         self.H1.S.connect ([ self.H2.B ])
         self.H1.C.connect ([ self.O1.B])
         self.H2.C.connect ([ self.O1.A])
         self.H2.S.connect ([ self.S])
         self.O1.C.connect ([ self.Cout])
```

In addition we have implemented 2x1 MUX and 1x2 DEMUX classes using the basic gate objects.

3.7 Classes implementing sequential logic elements

In this section we describes the implementation of sequential logic circuits like Flip-Flops, counters.

3.7.1 JK Flip Flop

As an example we show implementation of JK Flip Flop class which is derived from LG class.

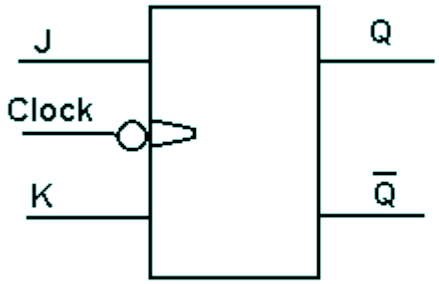

Figure 7: Full Adder

```
# ================
# JK - FlipFlop
# ================

class JKFlipFlop (LG):

  def __init__(self, name):
    LG.__init__(self,name)
    self.J = Connector(self,'J')
    self.K = Connector(self,'K')
    self.Q = Connector(self,'Q')
    self.C = Connector(self,'C', activates = 1)
    self.Q.value = 0
    self.prev = 0

  def evaluate (self):
    if (not self.C.value) and self.prev and self.J.value and (
       not self.K.value):
      self.Q.set(1)
    if (not self.C.value) and self.prev and not(self.J.value)
       and self.K.value:
      self.Q.set(0)
```

```
    if (not self.C.value) and self.prev and self.J.value and
       self.K.value:
      self.Q.set(not self.Q.value)
    self.prev = self.C.value
```

It has four attributes for J, K input pins, clcok input and Q output. When a event happens on clock pin, evaluate method of this class is called which performs JK flip flop operation.

Similarly we have implemented D and T Flip Flops. Also counter and shift register circuits are designed using this flip flop objects.

3.8 Class I/O stream

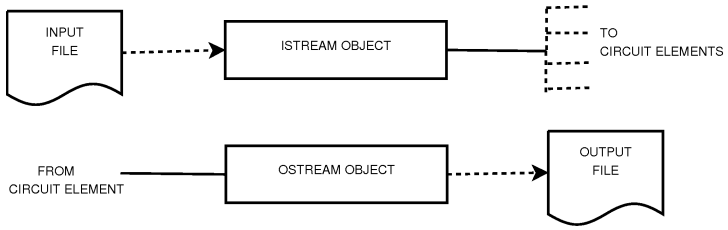

Figure 8: I/O Stream Model

3.9 Class Simulator

We have implemented the simulator in class SIMU. The following diagram shows the functional overview of the simulator. Figure 2 shows the overall class structure of the simulator design.

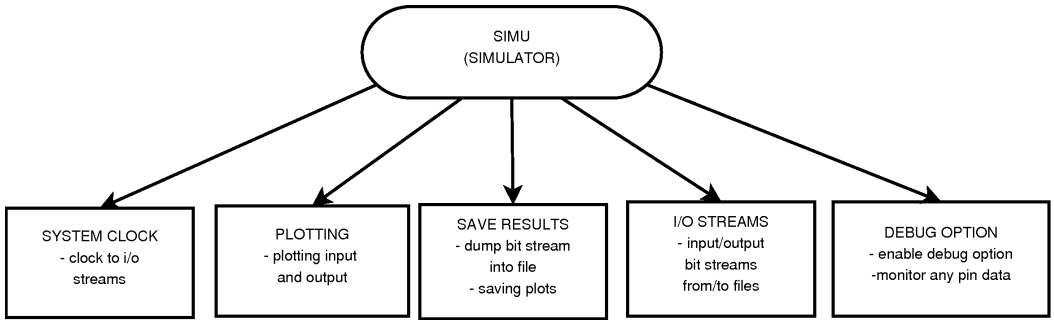

Figure 9: Simulator Class Details

3.10 Clocks & signal sources classes

The clocks and sources are nothing but classes which create instances of the Ostream class and perform required operations to generate the output. The output is generally a sequence of alternating ones and zeros or data read from a file.

4 Writing a circuit description for given logic circuit

In this section we present the our method of writing scripts to simulate given logic circuit using above mentioned modules like basic gate, combinational and sequential circuit modules. We will take a following example of 4 bit synchronous binary counter for this purpose.

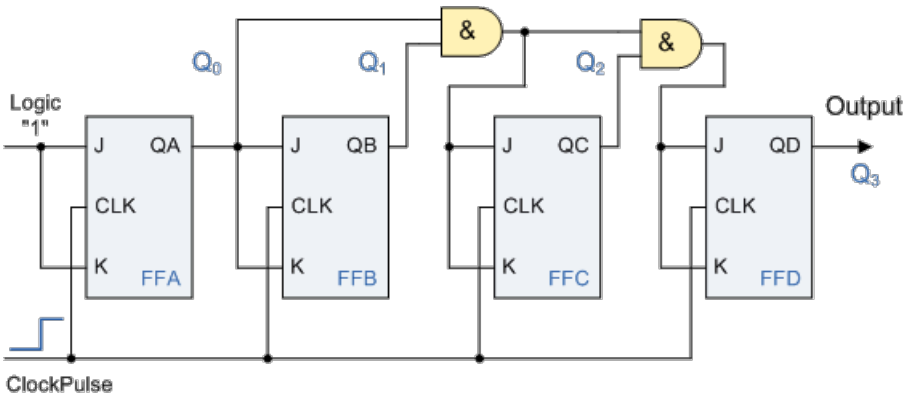

Figure 10: Binary 4-bit Synchronous Counter

To simulate this logic circuit we need 4 JK Flip Flops, 2 AND gates, constant source (CO) of generating input 1, clock source and 4 output streams. We generate above required objects different classes described in previous sections. Here we have to connect clock such that all flip flops are clocked simultaneously. We do that by output of clock (sim.clk.out) to each flip flops clock. We have to maintain the right order in connecting the clock. Using connector class objects we have connected each component of circuit as shown in figure. Finally we start the simulation using object sim of SIMU class.

```
# CIRCUIT:
# A 4 Bit Syncronous Binary Counter using JK Flip Flpos

from libpydlcs import *

sim = SIMU('sim1',start = 0,plots = 1, debug =1, pclk = 1 , step
    =0, clocks=33)
```

```
CO = ConstSrc('CS',value =1)

O1 = Ostream('OUT1', stream = 1)
O2 = Ostream('OUT2', stream = 1)
O3 = Ostream('OUT3', stream = 1)
O4 = Ostream('OUT4', stream = 1)

JK1 = JKFlipFlop('JK1')
JK2 = JKFlipFlop('JK2')
JK3 = JKFlipFlop('JK3')
JK4 = JKFlipFlop('JK4')

A1 = And('A1')
A2 = And('A2')


sim.clk_out.connect([JK4.C, O4.clk_in, JK3.C, O3.clk_in, JK2.C,
   O2.clk_in,CO.clk_in, JK1.C, O1.clk_in])
CO.data_out.connect([JK1.J, JK1.K])
JK1.Q.connect([JK2.J, JK2.K, A1.A, O1.data_in])
JK2.Q.connect([A1.B, O2.data_in])
A1.C.connect([JK3.J, JK3.K, A2.A])
JK3.Q.connect([A2.B, O3.data_in])
A2.C.connect([JK4.J, JK4.K])
JK4.Q.connect([O4.data_in])

sim.addplot([O1.data, O2.data, O3.data, O4.data])
sim.addpname(["Q1", "Q2", "Q3", "Q4"])

sim.start = 1
sim.simulate()
```

The counter output is shown in the Fig. 11.

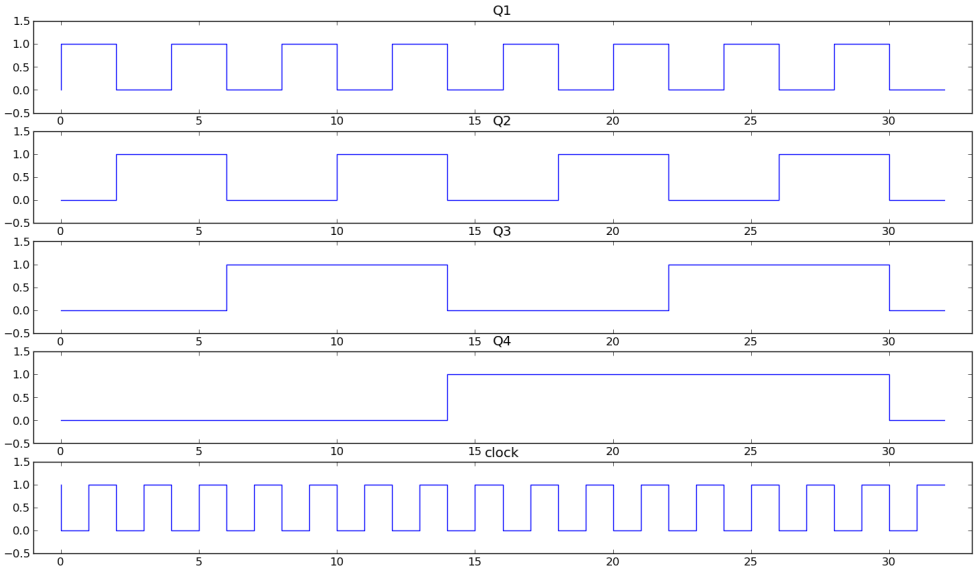

Figure 11: Binary 4-bit Synchronous Counter Output

5 Conclusion & future work

We have successfully implemented a digital logic simulator in python. We have demonstrated simulation of logic circuit containing combinational and sequential elements using our simulator. Potential future work includes GUI implementation. Feedback that activates the evaluation function is not possible in current release, this can be done as future work.