Submission

March 2, 2023

1 How to run this code:

- To run the code, simply open the provided .ipynb file in Jupyter Notebook. I've divided the program into different functions, so it's important to run the previous blocks of code before running the current block to avoid any unexpected output.
- I've added comments where necessary to make the functions well-defined and easy to understand. Although the code is quite lengthy, I've tried to name the variables in a way that matches their usage, so it should be self-explanatory.
- If you follow the code serially and manually, you should be able to understand it easily. Also, I've stored the output of both the topological sort and event-driven approaches separately in two output files: eventDrivenOutput.txt and topoDrivenOutput.txt, respectively.
- If you have any questions or difficulty understanding any part of the code, feel free to ask for clarification.

[1]:
```
import networkx as nx
import copy
```

1.1 The code below is to read the inputs and netlist

[2]:
```
global len_of_inputs

def readNetlis(filename_):
    filename= filename_
    with open(filename) as f:
        circuit= f.read().splitlines()
    circuitline=[]
    for lines in circuit:
        circuitline.append(lines.split())
    circuit= circuitline
    return circuitline

def input_list(filename_):
    global len_of_inputs
    global inputdetails
    filename= filename_
    temp=[]
    with open(filename) as f:
        input = f.read().splitlines()
        for line in input:
            elements = line.strip().split()
            temp.append(elements)
        input=temp
    inputdetails = input
    len_of_inputs= len(input)-1
    input_dict={}
    for index in range (len(input[0])):
        input_dict.update({input[0][index]:[]})
    for j in range(1,len(input)):
        for i in range(len(input[0])):
            value=int(input[j][i])
            input_dict[input[0][i]].append(value)

    return input_dict
```

[3]:
```
netlist= readNetlis("c17.net")
inputs= input_list("c17.inputs")
```

1.2 The following functons are the logical gates

[4]:
```
def ANDGATE(a,b):
    return int(a) and int(b)
def ORGATE(a,b):
    return int(a) or int(b)
def NOTGATE(a):
    return int(not(int(a)))
def NANDGATE(a,b):
    return int(not(ANDGATE(a,b)))
def NORGATE(a,b):
    return int(not(ORGATE(a,b)))
def XORGATE(a,b):
    return a^b
def XNORGATE(a,b):
    return int(not(XORGATE(a,b)))
def BUFFGATE(a):
    return a
```

The below code makes a graph using inbuilt function by adding edges.

[5]:
```
g= nx.DiGraph()
error=False
for entries in netlist:
    if(entries[1]=='inv' or entries[1]=='buf'):
        g.add_edge(entries[2],entries[3])
    else:
        g.add_edge(entries[2],entries[4])
        g.add_edge(entries[3],entries[4])

try:
    topoligical_sort=  list(nx.topological_sort(g))
    print("Nodes in topological order", topoligical_sort)
except:
    print("The combinational circuit is not in a combinational circuit format")
    error=1
```

```
Nodes in topological order ['N2', 'N7', 'N1', 'N3', 'N6', 'n_0', 'n_1', 'n_3',
'n_2', 'N22', 'N23']
```

[6]:
```
def out_in_():
    out_in={}
    for entries in netlist:
        out_in.update({entries[len(entries)-1]:[]})
        for i in range(1,len(entries)-1):
            out_in[entries[len(entries)-1]].append(entries[i])
    return out_in

out_in= out_in_()
```

[7]:
```
def dictionarykeysort(d):
    keyssorted= sorted(d.keys())
    sorted_dict= {}
    for key in keyssorted:
        sorted_dict[key]= d[key]
    return sorted_dict
```

1.3 Topological Approach

In the topological approach, we have a list called “nl” that contains the gates in topological order. To calculate the output values, we iterate through the list and check if the current entry is an input or if its predecessors have a value of zero. If it’s an input, we load its value from the “input_value” dictionary.

For non-primary inputs, we track the gate and node used to obtain their value using the “output_input” dictionary. Since we’re moving in topological order, the values of the input nodes will be known, so we use them to calculate the value of the node.

We repeat this process for all the input values and update our output list accordingly.

[8]:
```
class topological_approach():
    def __init__(self,out_in,nl,len_in):
        self.out_in= out_in
        self.nl= nl
        self.len_in= len_in
```

```
def topoevaluate(self,nl,out_in,input):
    out= []
    trackout= {}
    for items in nl:
            if (items in inputs.keys()):
                out.append(inputs[items][input])
                trackout.update({items:inputs[items][input]})

            else:
                if(out_in[items][0]=='nand2'):
                    temp= trackout[out_in[items][1]]
                    temp1= trackout[out_in[items][2]]
                    ans= NANDGATE(temp,temp1)
                    out.append(ans)
                    trackout.update({items:ans})

                elif(out_in[items][0]=='or2'):
                    temp= trackout[out_in[items][1]]
                    temp1= trackout[out_in[items][2]]
                    ans= ORGATE(temp,temp1)
                    out.append(ans)
                    trackout.update({items:ans})

                elif(out_in[items][0]=='nor2'):
                    temp= trackout[out_in[items][1]]
                    temp1= trackout[out_in[items][2]]
                    ans= NORGATE(temp,temp1)
                    out.append(ans)
                    trackout.update({items:ans})

                elif(out_in[items][0]=='xor2'):
                    temp= trackout[out_in[items][1]]
                    temp1= trackout[out_in[items][2]]
                    ans= XORGATE(temp,temp1)
                    out.append(ans)
                    trackout.update({items:ans})

                elif(out_in[items][0]=='and2'):
                    temp= trackout[out_in[items][1]]
                    temp1= trackout[out_in[items][2]]
                    ans= ANDGATE(temp,temp1)
                    out.append(ans)
                    trackout.update({items:ans})

                elif(out_in[items][0]=='inv'):
                    temp= trackout[out_in[items][1]]
```

```
                        ans= NOTGATE(temp)
                        out.append(ans)
                        trackout.update({items:ans})

                    elif(out_in[items][0]=='buf'):
                        temp= trackout[out_in[items][1]]
                        ans= BUFFGATE(temp)
                        out.append(ans)
                        trackout.update({items:ans})
        return trackout


    def topoligical_output(self):
        to_return= []
        for input in range(self.len_in):
            outtrack= self.topoevaluate(self.nl, self.out_in,input)
            to_return.append(dictionarykeysort(outtrack))
        return to_return
```

[9]:
```
solver= topological_approach(out_in,topoligical_sort,len_of_inputs)
sol = solver.topoligical_output()
for i in sol:
    print(i)
```

```
{'N1': 0, 'N2': 1, 'N22': 1, 'N23': 1, 'N3': 0, 'N6': 0, 'N7': 0, 'n_0': 1,
'n_1': 1, 'n_2': 1, 'n_3': 0}
{'N1': 0, 'N2': 0, 'N22': 0, 'N23': 0, 'N3': 1, 'N6': 0, 'N7': 0, 'n_0': 1,
'n_1': 1, 'n_2': 1, 'n_3': 1}
{'N1': 1, 'N2': 0, 'N22': 0, 'N23': 0, 'N3': 0, 'N6': 0, 'N7': 0, 'n_0': 1,
'n_1': 1, 'n_2': 1, 'n_3': 1}
{'N1': 0, 'N2': 0, 'N22': 0, 'N23': 0, 'N3': 1, 'N6': 1, 'N7': 1, 'n_0': 1,
'n_1': 0, 'n_2': 1, 'n_3': 1}
{'N1': 1, 'N2': 1, 'N22': 1, 'N23': 0, 'N3': 1, 'N6': 1, 'N7': 1, 'n_0': 0,
'n_1': 0, 'n_2': 1, 'n_3': 1}
{'N1': 1, 'N2': 1, 'N22': 1, 'N23': 1, 'N3': 1, 'N6': 0, 'N7': 0, 'n_0': 0,
'n_1': 1, 'n_2': 1, 'n_3': 0}
{'N1': 1, 'N2': 1, 'N22': 1, 'N23': 0, 'N3': 1, 'N6': 1, 'N7': 0, 'n_0': 0,
'n_1': 0, 'n_2': 1, 'n_3': 1}
{'N1': 1, 'N2': 1, 'N22': 1, 'N23': 1, 'N3': 0, 'N6': 0, 'N7': 0, 'n_0': 1,
'n_1': 1, 'n_2': 1, 'n_3': 0}
{'N1': 0, 'N2': 1, 'N22': 1, 'N23': 1, 'N3': 1, 'N6': 0, 'N7': 1, 'n_0': 1,
'n_1': 1, 'n_2': 0, 'n_3': 0}
{'N1': 0, 'N2': 0, 'N22': 0, 'N23': 0, 'N3': 1, 'N6': 1, 'N7': 0, 'n_0': 1,
'n_1': 0, 'n_2': 1, 'n_3': 1}
```

2 Event Driven

2.0.1 Explanation of the function for Event-driven Approach

The function takes three arguments:

- link: a dictionary that stores information about the circuit, such as the type of gate and its input and output nodes
- in_details: a dictionary that stores information about the input nodes, such as their values at different time instants
- primary: a dictionary that stores information about the primary inputs and their values at different time instants

Inside the function, a queue is used to store and perform the operations needed for the event-driven approach. The function then iterates over the different time instants of input and checks if the input has changed. If the input has changed, the corresponding element is appended to the queue.

Then the function iterates over the queue till it is empty, and assigns the updated output of different nodes. The input_connection dictionary is used to store the output that will be affected by changing that particular input.

The final output is stored in the final_output variable.

[10]:
```
class Event_Driven():
    def __init__(self,links,in_details,primary,error):
        self.links= links
        self.in_details= in_details
        self.primary= primary
        self.error=error

    def Event_Driven_sol(self):

        if(self.error):
            print('This is not combinational circuit!! ')
            return

        final_out=[]
        final_out.clear()
        out_updated={}
        out_updated.clear()
        queue=[]

        connect_in={}
        for items in self.links:
            if(self.links[items][0]!='inv' and self.links[items][0]!='buf'):
                connect_in.update({self.links[items][1]:[]})
                connect_in.update({self.links[items][2]:[]})
            else:
                connect_in.update({self.links[items][1]:[]})
        for items in self.links:
            # print(ele)
            if(self.links[items][0]!='inv' and self.links[items][0]!='buf'):
                I_ele1=connect_in[self.links[items][1]]
                I_ele1=I_ele1+[items]

                current_ele2=connect_in[self.links[items][2]]
                current_ele2=current_ele2+[items]

                connect_in.update({self.links[items][1]:I_ele1})
                connect_in.update({self.links[items][2]:current_ele2})
            else:
                I_ele1=connect_in[self.links[items][1]]
                I_ele1=I_ele1+[items]
                connect_in.update({self.links[items][1]:I_ele1})
        queue.clear()
        for i in range(len(self.in_details)-1):
            for items in self.primary:
                if(i==0):
                    queue.append(items)
                    out_updated.update({items:self.primary[items][i]})
                else:
                    if(self.primary[items][i]!=self.primary[items][i-1]):
                        queue.append(items)
                        out_updated.update({items:self.primary[items][i]})
                    else:
                        out_updated.update({items:self.primary[items][i]})
            while(len(queue)>0):

                curr_vertex=queue.pop(0)
                if curr_vertex in self.primary.keys():
                    if curr_vertex in connect_in.keys():
                        for all_output in connect_in[curr_vertex]:
                            queue.append(all_output)
                else:
                    if self.links[curr_vertex][0]!='inv' and self.
↪links[curr_vertex][0]!='buf':
                        inp1=self.links[curr_vertex][1]
                        inp2=self.links[curr_vertex][2]
                        if inp1 in out_updated.keys() and inp2 in out_updated.
↪keys():
                            if self.links[curr_vertex][0]=='and2':
                                out_updated.update({curr_vertex:
↪ANDGATE(out_updated[inp1],out_updated[inp2])})
                            elif self.links[curr_vertex][0]=='or2':
                                out_updated.update({curr_vertex:
↪ORGATE(out_updated[inp1],out_updated[inp2])})
```

```
                            elif self.links[curr_vertex][0]=='nand2':
                                out_updated.update({curr_vertex:
↪NANDGATE(out_updated[inp1],out_updated[inp2])})
                            elif self.links[curr_vertex][0]=='nor2':
                                out_updated.update({curr_vertex:
↪NORGATE(out_updated[inp1],out_updated[inp2])})
                            elif self.links[curr_vertex][0]=='xor2':
                                out_updated.update({curr_vertex:
↪XORGATE(out_updated[inp1],out_updated[inp2])})
                            elif self.links[curr_vertex][0]=='xnor2':
                                out_updated.update({curr_vertex:
↪XNORGATE(out_updated[inp1],out_updated[inp2])})
                        else:
                            queue.append(curr_vertex)
                            continue
                        if curr_vertex in connect_in.keys():
                            for all_output in connect_in[curr_vertex]:
                                queue.append(all_output)

                    else:
                        inp1=self.links[curr_vertex][1]
                        if(inp1 in out_updated.keys()):
                            if self.links[curr_vertex][0]=='inv':
                                out_updated.update({curr_vertex:
↪NOTGATE(out_updated[inp1])})
                            elif self.links[curr_vertex][0]=='buf':
                                out_updated.update({curr_vertex:
↪out_updated[inp1]})
                        else:
                            queue.append(curr_vertex)
                            continue
                        if curr_vertex in connect_in.keys():
                            for all_output in connect_in[curr_vertex]:
                                queue.append(all_output)
            final_out.append(copy.deepcopy(out_updated))
        return final_out


event_driven_Output=Event_Driven(out_in,inputdetails,inputs,error)  #Calling of␣
↪Event_Driven function for getting output
solution_Event= event_driven_Output.Event_Driven_sol()
```

Writing the Output in file eventDrivenOutput.tx and in topoDrivenOutput.txt

[11]:
```
def toposave(nl,solution):
    with open('topoDrivenOutput.txt','w') as f:
        for nodes in sorted(nl):
            f.write(f'{nodes:<10}')
        f.write("\n")
        for element in solution:
            for ele in sorted(element.keys()):
                f.write(f'{element[ele]:<10}')
            f.write('\n')
def eventsave(nl,solution):
    with open('eventDrivenOutput.txt','w') as f:
        for nodes in sorted(nl):
            f.write(f'{nodes:<10}')
        f.write("\n")
        for element in solution:
            for ele in sorted(element.keys()):
                f.write(f'{element[ele]:<10}')
            f.write('\n')

toposave(topoligical_sort,sol)
eventsave(topoligical_sort, solution_Event)
```

2.0.2 Using timit for the two functions

[12]:
```
print('The time taken by Event_driven_solution : ')
%timeit event_driven_Output.Event_Driven_sol()
print('the time taken by topo_sort_function : ')
%timeit solver.topoligical_output()
```

```
The time taken by Event_driven_solution :
277 µs ± 9.52 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
the time taken by topo_sort_function :
64.5 µs ± 692 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

In some cases, topological sort may take less time than an event-driven approach. However, in many cases, we may observe that the event-driven approach is faster, especially when the input changes only slightly. This is because with the event-driven approach, we only need to update the corresponding output, whereas with topological sort, we need to iterate through all vertices each time to update the output at each terminal.

The below code is the Master Code of the entire documentation. So the mastercode takes three arguments. These are:

- netlist: This is asking for the location of the netlist file
- input_filename: This is asking for the file where the inpt array is given
- method: in this, there will be either of the two, ‘topo’ or ‘event’ and it is caseinsensitive.

The output will be in the same text file

[13]:
```
def MasterCode(netlist,input_filename, method):
    netlist= readNetlis(netlist)
    inputs= input_list(input_filename)
    g= nx.DiGraph()
    error=False
    for entries in netlist:
        if(entries[1]=='inv' or entries[1]=='buf'):
            g.add_edge(entries[2],entries[3])
        else:
            g.add_edge(entries[2],entries[4])
            g.add_edge(entries[3],entries[4])

        try:
            topoligical_sort=  list(nx.topological_sort(g))
        except:
            print("The combinational circuit is not in a combinational circuit␣
↪format")
            return
    out_in= out_in_()

    if method.lower()=="topo":
        solver= topological_approach(out_in,topoligical_sort,len_of_inputs)
        sol = solver.topoligical_output()
        toposave(topoligical_sort,sol)
    elif method.lower()=="event":
        event_driven_Output=Event_Driven(out_in,inputdetails,inputs,error) ␣
↪#Calling of Event_Driven function for getting output
        solution_Event= event_driven_Output.Event_Driven_sol()
        eventsave(topoligical_sort, solution_Event)
```

Example

```
[14]: MasterCode("c17.net","c17.inputs","event")
      MasterCode("c17.net","c17.inputs","topo")
```