EE2703 - Week 4

Aayush Patel <ee21b003@smail.iitm.ac.in>

March 1, 2023

1 Instructions:

- This is a simple **.ipynb** file just run it in **jupyter notebook**
- The program is divided into different function so the previous block of code is required for getting the correct output.
- Run the code serially to avoid some unusual output.
- The function are well defined.
- The code is self explanatory because the variable name are choose as such.
- Since the Topological Sort and Event Driven Approach give same output I have written output to one file only.

1.1 Initialization

Import the required Libraries and define all the gate functions.

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

[2]:
```
def bufGate(a):
    return a

def andGate(a,b):
    if(a==0 or b==0):
        return 0
    else:
        return 1
def orGate(a,b):
    if(a==1 or b==1):
        return 1
    else:
        return 0
def invGate(a):
    if(a==1):
        return 0
    else:
        return 1
def norGate(a,b):
    if(a==1 or b==1):
        return 0
    else:
        return 1
def nandGate(a,b):
    if(a==0 or b==0):
        return 1
    else:
        return 0
def xorGate(a,b):
    if ((a==1 and b==0) or (a==0 and b==1)):
        return 1
    else:
        return 0
def xnorGate(a,b):
    if ((a==1 and b==0) or (a==0 and b==1)):
        return 0
    else:
        return 1
```

We define a function to print the netList in a clean manner

```
[3]: def displayNET(A):
         for i in A:
             print(i)
```

1.2 Reading the netlist

```
[4]: # Reading the data file
     def read_netlist(filename):
         filename+=".net"
         with open(filename) as f:
             lines = f.readlines()
             netList = []
             for line in lines:
                 line = line.split()
                 netList.append(line)
         return netList
```

1.3 Reading the inputs

```
[5]: # Reading the input file
     def read_input(filename):
         filename+=".inputs"
         # print(filename)
         with open(filename) as f:
             lines = f.readlines()
             netInput = []
             for line in lines:
                 line = line.split()
                 netInput.append(line)
         return netInput
```

1.4 Convert netInput to Dictionary for better access to values

```
[6]: def get_inputDict(netInput):
         inputDict = {}
         # print(netInput)
         for i in range (len(netInput[0])):
             inputDict.update({netInput[0][i]:[]})
         for i in range(1, len(netInput)):
             for j in range(len(netInput[0])):
                 inputDict[netInput[0][j]].append(int(netInput[i][j]))
         return inputDict
```

1.5 Defining the parameters of a Node

The dictionary `info` stores a node and its parameters i.e. its parent node and the Gate data. Note that it doesn't contain Primary Node parameters because there are no parameters to store.

[7]:
```
def get_infoDict(netList):
    info = {}
    # print(netList)
    for i in netList:
        if(i[1].lower() in ['inv','not','buf']):
            info.update({i[3]:[i[1],i[2]]})
        else:
            try:
                info.update({i[4]:[i[1],i[2],i[3]]})
            except:
                print("error at ",i)
    return info
```

1.6 TOPOLOGICAL SORT FOR CIRCUIT EVALUATION

To get the Linear Arrangement of Nodes from the Directed Acyclic Graph, I have used the function `topological_sort()` from the `networkx` module. This list will be iterated one by one to get the required values at the nodes in the net. If the Input Graph is not Acyclic the this function raises an error which is handelled later when this function is called.

[8]:
```
def topo_sort(netList):
    g = nx.DiGraph()
    for i in netList:
        if(i[1].lower() in ['inv','not','buf']):
            g.add_edge(i[2],i[3])
        else:
            g.add_edge(i[2],i[4])
            g.add_edge(i[3],i[4])

    nl =  list(nx.topological_sort(g))
    return nl
```

1.7 Solving the DAG (from Topological Sort)

Linearly iterate through the Topologically Sorted Array and **keep updating** the values in the child nodes.

[9]:
```
def solve_DAG(netInput,inputDict,info,nl):
    all_outputs=[]

    Gates={
        'buf':bufGate,
        'inv':invGate,
        'not':invGate,
        'and2':andGate,
        'or2':orGate,
        'nand2':nandGate,
        'nor2':norGate,
        'xor2':xorGate,
        'xnor2':xnorGate
    }
    sorted_array=list(nl)
    sorted_array.sort()
    all_outputs.append(sorted_array)
    for i in range (1, len(netInput)):
        output = []
        outputDict = {}
        for j in range (len(nl)):
            output.append(None)
        for j in range (len(nl)):
            if(nl[j] in inputDict.keys()):
                output[j] = inputDict[nl[j]][i-1]
                outputDict.update({nl[j]:output[j]})
            else:
                if(info[nl[j]][0].lower() in ['inv','not']):
                    output[j] = invGate(outputDict[info[nl[j]][1]])
                    outputDict.update({nl[j]:output[j]})
                elif(info[nl[j]][0].lower()=='buf'):
                    output[j] = bufGate(outputDict[info[nl[j]][1]])
                    outputDict.update({nl[j]:output[j]})
                else:
                    output[j] = Gates[info[nl[j]][0].
↪lower()](outputDict[info[nl[j]][1]],outputDict[info[nl[j]][2]])
                    outputDict.update({nl[j]:output[j]})
        sorted_array_output=[]
        for i in range(len(sorted_array)):
            sorted_array_output.append(outputDict[sorted_array[i]])
        all_outputs.append(sorted_array_output)
    return all_outputs
```

1.8 EVENT DRIVEN APPROACH

Instead of using a topological sort, we could also use queues for updating the states in an event driven approach. To do this, we start with a state table that contains the current state of each net, initialized to -1 at the start of the simulation. We form a queue that tracks the nets whose state has to be re-evaluated. Firstly the initialization of -1 is done and child node sof each nodes are calculated.

[10]:
```python
def event_driven_initialization(netList,netInput):
    typesOfNodes = set()
    primaryNodes = set()
    for i in netList:
        if(i[1].lower() in ['inv','not','buf']):
            typesOfNodes.add(i[2])
            typesOfNodes.add(i[3])
        else:
            typesOfNodes.add(i[2])
            typesOfNodes.add(i[3])
            typesOfNodes.add(i[4])
    for i in netInput[0]:
        primaryNodes.add(i)
    child = {}
    answerDict  = {}
    for i in typesOfNodes:
        child.update({i:[]})
        answerDict.update({i:-1})
    for i in netList:
        if(i[1].lower() in ['inv','not','buf']):
            child[i[2]].append(i[3])
        else:
            child[i[2]].append(i[4])
            child[i[3]].append(i[4])
    return typesOfNodes,primaryNodes,child,answerDict
```

Now Firstly we get the node states. Then from the second input vector and so on, the states of the primary inputs needs to be updated so we add these to the queue. Then, we dequeue nets from the queue and evaluate the new state. If the new state is different from the current state, the sucessors of the net are added to the queue for evaluation, otherwise nothing is done with respect to that node.

[11]:
```python
def␣
 ↪event_driven_approach(netInput,inputDict,info,child,answerDict,primaryNodes,typesOfNodes):
 ↪
    all_outputs=[]
    typesOfNodes=list(typesOfNodes)
    typesOfNodes.sort()
    all_outputs.append(typesOfNodes)
    inputsize = len(netInput)
```

```python
Gates={
    'buf':bufGate,
    'inv':invGate,
    'not':invGate,
    'and2':andGate,
    'or2':orGate,
    'nand2':nandGate,
    'nor2':norGate,
    'xor2':xorGate,
    'xnor2':xnorGate
}
noOfPrimaryNodes = len(netInput[0])
for i in range (1,inputsize):
    changeQueue = []
    if(i>=2):
        for j in range (noOfPrimaryNodes):
            if(netInput[i][j]!=netInput[i-1][j]):
                changeQueue.append(netInput[0][j])
                answerDict[netInput[0][j]]= -1
    else:
        for j in primaryNodes:
            changeQueue.append(j)
    while(len(changeQueue)!=0):
        node = changeQueue.pop(0)
        if(node in primaryNodes):
            answerDict[node] = inputDict[node][i-1]
            for j in child[node]:
                changeQueue.append(j)
                answerDict[j]=-1
        else:
            if(info[node][0].lower() in ['inv','not']):
                if(answerDict[info[node][1]]==-1):
                    continue
                else:
                    val = invGate(answerDict[info[node][1]])
                    answerDict[node] = val
                    for j in child[node]:
                        changeQueue.append(j)
                        answerDict[j]=-1
            elif(info[node][0].lower()=='buf'):
                if(answerDict[info[node][1]]==-1):
                    continue
                else:
                    val = bufGate(answerDict[info[node][1]])
                    answerDict[node] = val
                    for j in child[node]:
                        changeQueue.append(j)
```

```
                            answerDict[j]=-1
                else:
                    if(answerDict[info[node][1]]==-1 or␣
↪answerDict[info[node][2]]==-1):
                        continue
                    else:
                        val = Gates[info[node][0].
↪lower()](answerDict[info[node][1]],answerDict[info[node][2]])
                        answerDict[node] = val
                        for j in child[node]:
                            changeQueue.append(j)
                            answerDict[j]=-1
        output=list()
        for key in answerDict.keys():
            output.append(None)
        for i in range(len(typesOfNodes)):
            output[i]=answerDict[typesOfNodes[i]]
        all_outputs.append(output)
    return all_outputs
```

1.9 Compile All Functions

Here we compile all the functions that we have synthsized above. It runs those functions in the systematic order. It prints WRONG .netfile if the .net file has Cycles. Here a comparision of both approaches of the time taken is done. This `solve()` function also prints the summary and write the output to a file, if needed.

```
[12]: def solve(filename,show_summary=True,speed_analysis=True,write_to_file=False):

          netList=read_netlist(filename)
          try:
              nl=topo_sort(netList)
          except:
              print("Wrong .netfile")
              return

          netInput=read_input(filename)

          inputDict=get_inputDict(netInput)

          info=get_infoDict(netList)

          all_outputs_topoSort=solve_DAG(netInput,inputDict,info,nl)

        ␣
      ↪typesOfNodes,primaryNodes,child,answerDict=event_driven_initialization(netList,netInput)
        ␣
      ↪all_outputs_eventDriven=event_driven_approach(netInput,inputDict,info,child,answerDict,primar

          if(show_summary==True):
              print(filename)
              print()

              print("netList :")
              displayNET(netList)
              print()

              print("netInput :")
              displayNET(netInput)
              print()

              print(f"inputDict : \n{inputDict}")
              print()

              print(f"info :\n{info}")
              print()

              print('Nodes in topological order :', nl)
        print()

        print("Topological Sort Approach : ")
        display(all_outputs_topoSort)

        print("Event Driven Approach : ")
        display(all_outputs_eventDriven)

        if(all_outputs_topoSort==all_outputs_eventDriven):
            print("Successful")
        else:
            print("Wrong Somewhere")

    if(speed_analysis==True):
        print("Time taken in Topological Sort (DAG) Method : ")
        %timeit solve_DAG(netInput,inputDict,info,nl)
        print()

        print("Time taken in Event Driven Method : ")
        %timeit␣
 ↪event_driven_approach(netInput,inputDict,info,child,answerDict,primaryNodes,typesOfNodes)

    if(write_to_file==True):
        f=open(filename+".txt","w")
        # f.writelines(all_outputs_eventDriven)
        for row in all_outputs_eventDriven:
            for word in row:
                f.write(str(word))
                f.write(" ")
            f.write("\n")
        f.close()
        print("Written to File")

solve("myfile")
```

```
myfile

netList :
['G0', 'AND2', 'A', 'B', 'D']
['G1', 'XOR2', 'B', 'C', 'E']
['G2', 'NOT', 'C', 'F']
['G3', 'AND2', 'B', 'E', 'G']
['G4', 'AND2', 'E', 'F', 'H']
['G5', 'OR2', 'D', 'G', 'I']
['G6', 'OR2', 'G', 'H', 'J']

netInput :
['A', 'B', 'C']
```

```
['1', '1', '1']
['0', '1', '1']
['0', '1', '0']

inputDict :
{'A': [1, 0, 0], 'B': [1, 1, 1], 'C': [1, 1, 0]}

info :
{'D': ['AND2', 'A', 'B'], 'E': ['XOR2', 'B', 'C'], 'F': ['NOT', 'C'], 'G':
['AND2', 'B', 'E'], 'H': ['AND2', 'E', 'F'], 'I': ['OR2', 'D', 'G'], 'J':
['OR2', 'G', 'H']}

Nodes in topological order : ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J']

Topological Sort Approach :

[['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J'],
 [1, 1, 1, 1, 0, 0, 0, 0, 1, 0],
 [0, 1, 1, 0, 0, 0, 0, 0, 0, 0],
 [0, 1, 0, 0, 1, 1, 1, 1, 1, 1]]

Event Driven Approach :

[['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J'],
 [1, 1, 1, 1, 0, 0, 0, 0, 1, 0],
 [0, 1, 1, 0, 0, 0, 0, 0, 0, 0],
 [0, 1, 0, 0, 1, 1, 1, 1, 1, 1]]

Successful
Time taken in Topological Sort (DAG) Method :
25.2 µs ± 845 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)

Time taken in Event Driven Method :
39.7 µs ± 4.15 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

1.10 Writing the Outputs of the given .net files to .txt files

"Myfile" is a file created by me to verify the code. Other files were too large so to debug I need a relatievely smaller file.

[13]:
```
files=["myfile","c8","c17","c17_1","c432","parity"]
for file in files:
    print("filename :",file,end=" --> ")
    solve(file,show_summary=False,speed_analysis=False,write_to_file=True)
```

```
filename : myfile --> Written to File
filename : c8 --> Written to File
filename : c17 --> Written to File
filename : c17_1 --> Wrong .netfile
filename : c432 --> Written to File
filename : parity --> Written to File
```

1.11 Comparision

- Speed or efficiency of above used methods depends on the number of input instances(N_i) and number of nets(N_n) in the given circuit.
- For N_n > N_i, method using topological sort and multiple round of circuit evaluation is more efficient/faster.
- Consider the case N_i = 2 and N_n > 2 such that in two instances, corresponding each primary input is complement of other. So in this case, 2(N_n-N_i) iterations will be taken by topological sort method but event driven simulation will take atleast 2N_n iterations.
- But if N_n < N_i, event driven simulation is more efficient/faster.
- For reasoning of above statement, we can state that since N_i is very large, almost all possible permutations of inputs are covered so on moving from one input instance to other, changed inputs(which would be less than N_n) will be quite less as N_n is already quite less.

[14]:
```
solve("c8",show_summary=False)
```

```
Time taken in Topological Sort (DAG) Method :
620 µs ± 19.5 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)

Time taken in Event Driven Method :
1.78 ms ± 62.6 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

[15]:
```
solve("c432",show_summary=False)
```

```
Time taken in Topological Sort (DAG) Method :
21 ms ± 2.69 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

Time taken in Event Driven Method :
617 ms ± 93.2 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

[16]:
```
solve("c17",show_summary=False)
```

```
Time taken in Topological Sort (DAG) Method :
88.4 µs ± 2.07 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)

Time taken in Event Driven Method :
129 µs ± 5.5 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

Note : This is completely opposite to our reasoning. But according to my inference I would say that since we are using the function topological sort function which is already very optimized the time take is lesser. But if we were to do DAG Topological Sorting from Scratch it would be taking more time.