EE2703 week4 Assignment

Nachiket Dighe EE21B093

March 1, 2023

1 Reading Netlist

[1]:
```python
def read_file(file):
    lines = file.readlines()
    for i in lines:
        print(i)
    return lines


file_name1 = open("c17.net",'r')
file_name2 = open("c8.net",'r')
file_name3 = open("c432.net",'r')
file_name4 = open("parity.net",'r')

lines = read_file(file_name1)
```

```
g51 nand2 n_3 n_0 N22

g52 nand2 n_3 n_2 N23

g53 nand2 n_1 N2 n_3

g54 nand2 n_1 N7 n_2

g55 nand2 N1 N3 n_0

g56 nand2 N3 N6 n_1
```

Here , I have Read the netlist given to us. One of the example i have considered is c_17 netlist

[2]:
```python
file_input1 = open("c17.inputs",'r')
file_input2 = open("c8.inputs",'r')
file_input3 = open("c432.inputs",'r')
file_input4 = open("parity.inputs",'r')
lines2 = read_file(file_input1) #reading the file
file_inputs = {} #a empty dictionary to store all input data
test_cases = 0 #to check how many tests we have in file
```

```
N1 N2 N3 N6 N7

0 1 0 0 0

0 0 1 0 0

1 0 0 0 0

0 0 1 1 1

1 1 1 1 1

1 1 1 0 0

1 1 1 1 0

1 1 0 0 0

0 1 1 0 1

0 0 1 1 0
```

Same as above , I have read the states of the nets given for that corresponding netlist

2 Generating Inputs

[3]:
```
def making_inputs(lines):
    for i in lines:
        keys = i.strip().split() #creating a list of all input variables
        break

    n = len(keys)
    for i in range(n):
        file_inputs[keys[i]] = [] #creating a empty list for given key

    for j in range(n): # a loop to store each corresponding input to that input␣
 ↪id
        for i in lines:
            if i.startswith("0") or i.startswith("1"):
                word = i.strip().split()
                file_inputs[keys[j]].append(word[j])
                test_cases = len(file_inputs[keys[j]])

    return test_cases

tests = making_inputs(lines2)
```

```
print(file_inputs , "\n") # creating a dictionary where each test case is taken␣
↪in the list
print(f"Number of test cases: " , tests)
```

```
{'N1': ['0', '0', '1', '0', '1', '1', '1', '1', '0', '0'], 'N2': ['1', '0', '0',
'0', '1', '1', '1', '1', '1', '0'], 'N3': ['0', '1', '0', '1', '1', '1', '1',
'0', '1', '1'], 'N6': ['0', '0', '0', '1', '1', '0', '1', '0', '0', '1'], 'N7':
['0', '0', '0', '1', '1', '0', '0', '0', '1', '0']}

Number of test cases:  10
```

2.1 making_inputs():

- This function is used to create a dictionary where each key contains all the states given in the input file as shown above

3 Getting Gates Info

[4]:
```
def gate_data(lines):
    gates = {} # a dictionary to store all info about the nets and gate
    gate_attribute = {}
    # Loop over each line in the file
    for line in lines:
        words = line.strip().split() # Split the line into words
        # Extract the gate ID, type, inputs, and output
        gate_id = words[0]
        gate_type = words[1]
        inputs = words[2:-1]
        output = words[-1]
        gates[gate_id] = {'type': gate_type, 'inputs': inputs, 'output': output}
        gate_attribute[output] = gate_type
    return gates , gate_attribute #for checking which gate is connected to output

gates , gate_attribute = gate_data(lines)
print(gates)
```

```
{'g51': {'type': 'nand2', 'inputs': ['n_3', 'n_0'], 'output': 'N22'}, 'g52':
{'type': 'nand2', 'inputs': ['n_3', 'n_2'], 'output': 'N23'}, 'g53': {'type':
'nand2', 'inputs': ['n_1', 'N2'], 'output': 'n_3'}, 'g54': {'type': 'nand2',
'inputs': ['n_1', 'N7'], 'output': 'n_2'}, 'g55': {'type': 'nand2', 'inputs':
['N1', 'N3'], 'output': 'n_0'}, 'g56': {'type': 'nand2', 'inputs': ['N3', 'N6'],
'output': 'n_1'}}
```

3.1 gate_data():

- This Function takes lines of the netlist as argument .
- using the line.strip().split() we have used to find the gate_id , gate_type , input and output of the given gate id.

- All this information is stored in a dictionary gates whose key is gate id and its value is adictionary which contains the type , inputs and output
- Also one more dictionary is used gate_attribute to check what gate type is connected to the output
- This dictionary contains output as key and gate type as value

4 Creating Topological Order

[5]:
```python
import networkx as nx # create a DAG

# create an empty directed graph
G = nx.DiGraph()

# iterate over each line in the netlist
for key, value in gates.items():
    # store the dictionary into its components
    gate_id = key

    gate_type = value['type']

    if len(value['inputs'])== 2:
        input1 , input2 = value['inputs'][0] , value['inputs'][1]
    else:
        input1 = value['inputs'][0]
        input2 = None

    output = value['output']

    # add the gate as a node to the graph
    nx.set_node_attributes(G,gate_attribute,name="gate_type")
    # add edges for each input and output connection
    if input1 not in G.nodes():
        G.add_node(input1)
    G.add_edge(input1, output)

    if input2 is not None:
        if input2 not in G.nodes():
            G.add_node(input2)
        G.add_edge(input2, output)

    if output not in G.nodes():
        G.add_node(output)

    if(nx.is_directed_acyclic_graph(G)==True):
        pass
    else:
        print("Error!")
        print("The given gate connections form a cycle.")
        print("Terminating.")
# get the topological order of the graph
topological_order = list(nx.topological_sort(G))
print(topological_order)

#print('Successors of node N3 :', list(G.successors("N3"))) # find children of a␣
 ↪node
#print('Parents of node n_0 :',list(G.predecessors("n_0"))) # find parents of␣
 ↪node
```

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

4.1 DAG maker:

- G.add_edge connects the input to the output
- G.add_node adds a node in the list
- Using the attribute nx.topological_sort we find the topological order of the given netlist
- By this we can tell which are the parents of the net and which are the children of the net

[6]:
```
def nandgate(a, b): #to find output for NAND gate
    return 2 + ~(a & b)

def orgate(a, b): #to find output for OR gate
    return (a | b)

def andgate(a, b): #to find output for AND gate
    return (a & b)

def norgate(a, b): #to find output for NOR gate
    return 2 + ~(a | b)

def xorgate(a, b): #to find output for XOR gate
    return (a ^ b)

def xnorgate(a, b): #to find output for XNOR gate
    return 2 + ~(a ^ b)

def invgate(a): #to find output for NOT gate
    return 2 + ~(a)

def bufgate(a): #to find output for Buffer gate
    return a
```

4.2 Gates output Calculator:

- Basic functions for finding the output of a AND , NAND , XOR gates , etc are used

[7]:
```
def gate_solver(x , key , node , inputs):
    if len(key) == 2: #check for 2 inputs
        a = key[0]
        b = key[1]
    else: c = key[0] #check for 1 input

    if node == 'nand2':
        x = nandgate(int(inputs[a]) , int(inputs[b]))
    if node == 'and2':
        x = andgate(int(inputs[a]) , int(inputs[b]))
    if node == 'nor2':
        x = norgate(int(inputs[a]) , int(inputs[b]))
    if node == 'or2':
        x = orgate(int(inputs[a]) , int(inputs[b]))
    if node == 'xnor2':
        x = xnorgate(int(inputs[a]) , int(inputs[b]))
    if node == 'xor2':
        x = xorgate(int(inputs[a]) , int(inputs[b]))
    if node == 'inv':
        x = invgate(int(inputs[c]))

    return x
```

4.3 Gate_solver():

- This function takes inputs as argument which is a dictionary which will tell us the final output and which contains all nets with some initial value
- Then we check what gatetype is connected to the output(x) using the argument node
- FInally by using the basic gates functions we return the output

[8]:
```
def topological_sort(output , to , gates):
    l1 = len(list(file_inputs.keys())) #to not consider the primary inputs
    n = len(to)
    for i in range(l1 , n):
        key = list(G.predecessors(to[i])) #get all predecessors of the net
        node = gate_attribute[to[i]]
        inputs[to[i]] = gate_solver(inputs[to[i]] , key , node , inputs)

    return inputs
```

4.4 Topological_sort():

- This is our main function which finds the states of all the nets using the gates dictionary and the topological order

```
[9]: inputs ={}
     to = list(nx.topological_sort(G))
     for i in to:
         inputs[i] = 0
     print(inputs)
```

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

Initialise all the net states to zero

```
[10]: output_file = open("c17_output" , 'w') #write file in c17_output
      for k in range(tests):
          for i in to:
              inputs[i] = 0
          for key in file_inputs.keys():
              if list(inputs.keys()).count(key):
                  inputs[key] = int(file_inputs[key][k]) #initialise the inputs
          inputs = topological_sort(inputs , to ,  gates)
          alpha_output = dict(sorted(inputs.items(), reverse=False)) #sort the␣
       ↪dictionary in alphabetical order
          l2 = list(alpha_output.keys())
          if(k==0):
              for i in l2:
                  output_file.write(f'{str(i)} ') #List of all nets in the circuit
              output_file.write("\n")
          for i in l2:
              output_file.write(f"{str(alpha_output[i])} ") #List of states for input␣
       ↪vector k
          output_file.write("\n")

      output_file.close() #closing a file
```

Here , we iterate through the test cases and change the inputs initial states . FInally , the dictionary Inputs which we get is sorted in alphabetical order using the sorted algorithm . And then it is written in the file c17_output in the format as mentioned in the assignment

5 Method 2 : Event-driven evaluation

```
[11]: import queue
      # create a queue
      my_queue = queue.Queue()
```

```
[12]: dict1 ={}
      to = list(nx.topological_sort(G))
      for i in to:
          dict1[i] = None
      print(dict1)
```

```
{'N2': None, 'N7': None, 'N1': None, 'N3': None, 'N6': None, 'n_0': None, 'n_1':
None, 'n_3': None, 'n_2': None, 'N22': None, 'N23': None}
```

A dictionary is created in this case where all the nnets are initialised to some arbitrary value such as None

[13]:
```
def event_driven(my_queue , dict1 , dict2 = None):
    while not my_queue.empty():
        key = my_queue.get()
        a = list(G.successors(key))
        for succ in a:
            my_queue.put(succ)
            p = list(G.predecessors(succ))
            try:
                node =  gate_attribute[succ]
                dict1[succ] = gate_solver(dict1[succ] , p , node , dict1)
                if dict2[succ] != dict1[succ]:
                    dict2[succ] = dict1[succ]
            except:
                pass
    return
```

5.1 Event_driven():

- In this function first we iterate through the queue untill the queue becomes empty
- Then we find the successors of a particular element of the key
- After this, the successors are added in the queue and the parents of that successor are poped out

[14]:
```
for key in file_inputs.keys():
    count = 0
    dict1[key] = file_inputs[key][0]
    n = list(file_inputs.keys())
    if  count < len(n):
        my_queue.put(key)
    count += 1

event_driven(my_queue , dict1)
dict2 = dict1.copy()
alpha = dict(sorted(dict1.items(), reverse=False))
print(alpha)
```

```
{'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}
```

[15]:
```
print(dict1)
for i in range(1 , tests):
    dict2 = dict1.copy()
    for key in file_inputs.keys():
        count = 0
        dict1[key] = file_inputs[key][i]
        n = list(file_inputs.keys())
        if  count < len(n):
            my_queue.put(key)
        count += 1
        event_driven(my_queue , dict1 , dict2)
    print(dict1)
```

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

[16]:
```
event_driven(my_queue , dict1)
dict2 = dict1.copy()
alpha = dict(sorted(dict1.items(), reverse=False))

output_file = open("c17_output_queue" , 'w') #write file in c17_output
lp = list(alpha.keys())
for i in lp:
    output_file.write(f'{str(i)} ') #List of all nets in the circuit
output_file.write("\n")

for i in lp:
    output_file.write(f'{str(alpha[i])} ')
output_file.write(f'\n')



for k in range(1 , tests):
    dict2 = dict1.copy()
    for key in file_inputs.keys():
        count = 0
        dict1[key] = file_inputs[key][k]
        n = list(file_inputs.keys())
        if  count < len(n):
            my_queue.put(key)
        count += 1
        event_driven(my_queue , dict1 , dict2)
    alpha = dict(sorted(dict1.items(), reverse=False))
    for i in lp:
        output_file.write(f'{str(alpha[i])} ')
    output_file.write(f'\n')

output_file.close()
```

- In this Code , firstly we create a copy of the initial dictionary which we acquired . Then we use this dictionnary to compare the outputs of the new dictionary which is made for each test case.
- After the resulting dictionary is obtained we sort it using the sorted() function , and then we write it in the file using the write() function and close the file

[17]:
```
%timeit topological_sort(inputs , to ,  gates)
%timeit event_driven(my_queue , dict1 , dict2)
```

```
10.8 µs ± 440 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
727 ns ± 24.1 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
```

6 Conclusion

- As seen by the time it operations it can be seen as the topological method is faster than the queuing method in terms of solving and taking down the values for all the states which are present in the logic gate circuit. It can be seen from the inputs that topological sort with multiple rounds of circuit evaluations may be more efficient for small and simple circuits, while event-driven simulation using queues may be more efficient for larger and more complex circuits.
- As can be seen, topological sort requires an initial pass to determine the order of the nodes, followed by multiple rounds of circuit evaluations to determine the output values of the gates. The number of rounds required is determined by the depth of circuit that is the number of levels of states which are present in the circuit. Topological sort can be very efficient for small circuits with few gates, but for larger circuits, the number of rounds can become a heck as it will start calculating output for start as each new level comes into play. As just imagine we are calculating output for level 100, it will start from level 1 come to level 99 and then it will evaluate results for level 100 which is a lot of time taking process.
- Event-driven simulation with queues, on the other hand, can handle circuits of any size and complexity, but requires a more neat implementation. Each gate is represented as an event

that is added to a queue when its inputs change. When an event is processed, the gate's output value is calculated, and events for its successor gates are added to the queue when there is a change in event because the logic is if input gate changes there is a high possibility that the output might chage. Because it only processes events that are affected by input changes rather than computing all gates in each round, this approach can be very efficient for large circuits with many gates and high nodes.

- In summary, both approaches have advantages and disadvantages, and the choice depends on the specific problem and the characteristics of the input data.