EE21B069_Keshaw

March 1, 2023

1 Assigement - 4

Week 4

[ ]:
```
import networkx as nx
import matplotlib.pyplot as plt
```

2 Reading the data file

[ ]:
```
# Reading the data file
def read_file(points):
    data = []
    for point in points:
        point = point.rstrip('\n')
        point = point.split()
        data.append(point)
    return data

#open the netlist file as a f
netList_file=input("Enter a Netlist File")
with open(netList_file) as f:
    lines = f.readlines()

netList = read_file(lines)

# print elements of the netlist
for i in netList:
    print(i)
```

[ ]:
```
# information dict or netlist dict
net_dict = {}
for i in netList:
    if(i[1]=='inv'):
        net_dict.update({i[3]:[i[1],i[2]]})
    elif(i[1]=='buf'):
        net_dict.update({i[3]:[i[1],i[2]]})
    else:
```

```
        net_dict.update({i[4]:[i[1],i[2],i[3]]})

print(net_dict)
```

3 Reading the input vector file

[ ]:
```
# Reading the input vector file

input_file=input("Enter a Input File")
with open(input_file) as f:
    ln = f.readlines()

Inputs = read_file(ln)
for i in Inputs:
    print(i)
```

[ ]:
```
input_dict = {}

for i in range (len(Inputs[0])):
    input_dict.update({Inputs[0][i]:[]})

for i in range(1, len(Inputs)):
    for j in range(len(Inputs[0])):
        input_dict[Inputs[0][j]].append(int(Inputs[i][j]))

print(input_dict)
```

4 Circuit Evaluation Using Topological Sort Algorithm

[ ]:
```
# Circuit Evaluation Using Topological Sort Algorithm

g = nx.DiGraph()

for i in netList:
    if(i[1]=='inv'):
        g.add_edge(i[2],i[3])
    elif(i[1]=='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))
print("Topologically ordered nodes are:- ", nl)
```

The topological sort algorithm is used in this above lines to evaluate a circuit represented as a

directed graph. The netList is made up of tuples that represent circuit components, with inverters and buffers having a single input and output and other components having two inputs and one output. The graph object is created, and edges are added according to the component type. The topological sort function is used to generate a topologically ordered list of nodes that is output to the console. To avoid circular dependencies, the code ensures that circuit components are evaluated in the correct order.

5 Define all Logical functions

[ ]:
```
#and
def and_fun(a,b):
    if(a==0 or b==0):
        return 0
    else:

        return 1
#or
def or_fun(a,b):
    if(a==1 or b==1):
        return 1
    else:
        return 0

#not/inv
def inv_fun(a):
    if(a==1):
        return 0
    else:
        return 1

#nor
def nor_fun(a,b):
    if(a==1 or b==1):
        return 0
    else:
        return 1

#nand
def nand_fun(a,b):
    if(a==0 or b==0):
        return 1
    else:
        return 0

#xor
def xor_fun(a,b):
    if ((a==1 and b==0) or (a==0 and b==1)):
```

```
        return 1
    else:
        return 0

#xnor
def xnor_fun(a,b):
    if ((a==1 and b==0) or (a==0 and b==1)):
        return 0
    else:
        return 1
```

The following logic gate functions are defined in the code: and_fun, or_fun, inv_fun, nor_fun, nand_fun, xor_fun, and xnor_fun. Each function accepts one or two inputs (a and b) and outputs a Boolean value based on the logic of the specific gate. The and fun function returns 1 if both inputs are 1, or 1 if at least one input is 1, inv_fun returns the complement of the input, nor fun returns 0 if at least one input is 1, nand_fun returns 0 if both inputs are 1, xor_fun returns 1 if the inputs are different, and xnor_fun returns 1.

[ ]:
```
print(nl)
for i in range (1, len(Inputs)):
    output_list = []

    for j in range (len(nl)):
        output_list.append(None)

    output_dict = {}

    for j in range (len(nl)):
        if(nl[j] in input_dict.keys()):
            output_list[j] = input_dict[nl[j]][i-1]
            output_dict.update({nl[j]:output_list[j]})

        else:
            if (net_dict[nl[j]][0]=='and2'):
                output_list[j] =␣
 ↪and_fun(output_dict[net_dict[nl[j]][1]],output_dict[net_dict[nl[j]][2]])
                output_dict.update({nl[j]:output_list[j]})

            elif(net_dict[nl[j]][0]=='or2'):
                output_list[j] =␣
 ↪or_fun(output_dict[net_dict[nl[j]][1]],output_dict[net_dict[nl[j]][2]])
                output_dict.update({nl[j]:output_list[j]})

            elif (net_dict[nl[j]][0]=='inv'):
                output_list[j] = inv_fun(output_dict[net_dict[nl[j]][1]])
                output_dict.update({nl[j]:output_list[j]})
```

```
            elif (net_dict[nl[j]][0]=='nor2'):
                output_list[j] =␣
↪nor_fun(output_dict[net_dict[nl[j]][1]],output_dict[net_dict[nl[j]][2]])
                output_dict.update({nl[j]:output_list[j]})

            elif (net_dict[nl[j]][0]=='nand2'):
                output_list[j] =␣
↪nand_fun(output_dict[net_dict[nl[j]][1]],output_dict[net_dict[nl[j]][2]])
                output_dict.update({nl[j]:output_list[j]})

            elif (net_dict[nl[j]][0]=='xor2'):
                output_list[j] =␣
↪xor_fun(output_dict[net_dict[nl[j]][1]],output_dict[net_dict[nl[j]][2]])
                output_dict.update({nl[j]:output_list[j]})

            elif (net_dict[nl[j]][0]=='xnor2'):
                output_list[j] =␣
↪xnor_fun(output_dict[net_dict[nl[j]][1]],output_dict[net_dict[nl[j]][2]])
                output_dict.update({nl[j]:output_list[j]})

    print((output_list))
```

This code prints the list "nl", which represents a circuit's topologically ordered nodes. The code evaluates the circuit for each input in "Inputs" by iterating over the nodes in "nl". The code creates a list output list to store each node's output values and a dictionary output_dict to map each node to its output value. If the current node is an input, the value of its output is obtained from input_dict. If the current node is a logic gate, the output value is calculated using the logic gate function (e.g., and_fun) and the input values from output dict. The current input's output values are printed to the console.

6 Event-Driven Approach

[ ]:
```
# event-driven approach
node_type = set()
primary_nodes = set()
for i in netList:
    if(i[1]=='inv'):
        node_type.add(i[2])
        node_type.add(i[3])
    else:
        node_type.add(i[2])
        node_type.add(i[3])
        node_type.add(i[4])
```

```
for i in Inputs[0]:
    primary_nodes.add(i)

child = {}
answer_dict  = {}

for i in node_type:
    child.update({i:[]})
    answer_dict.update({i:-1})

for i in netList:
    if(i[1]=='inv'):
        child[i[2]].append(i[3])
    else:
        child[i[2]].append(i[4])
        child[i[3]].append(i[4])

print("All Nodes:-", node_type, "\n")
print("primary_nodes:-", primary_nodes, "\n")
print("Childs of a nodes:-", child, "\n")
print("The answer Dict:-", answer_dict, "\n")
print("The Netlist dict:-", net_dict,"\n")
print("Input Dict:-", input_dict)
```

This above lines defines an event-driven approach to circuit evaluation. It first identifies the circuit's node types and primary nodes. It then generates dictionaries for each node's children and answer. Finally, for debugging purposes, it prints these dictionaries along with the netlist and input dictionaries.

[ ]:
```
queues = []

for i in primary_nodes:
    queues.append(i)

while(len(queues)!=0):
    print(queues)
    node = queues.pop(0)

    if(node in primary_nodes):
        answer_dict[node] = input_dict[node][0]
        for j in child[node]:
            queues.append(j)

    else:
        if(net_dict[node][0]=='inv'):
            if(answer_dict[net_dict[node][1]]==-1):
                continue
```

```
            else:
                val = inv_fun(answer_dict[net_dict[node][1]])
                answer_dict[node] = val
                for j in child[node]:
                    queues.append(j)
        else:
            if(net_dict[node][0]=='and2'):
                if(answer_dict[net_dict[node][1]]==-1 or␣
↪answer_dict[net_dict[node][2]]==-1):
                    continue
                else:
                    val =␣
↪and_fun(answer_dict[net_dict[node][1]],answer_dict[net_dict[node][2]])
                    answer_dict[node] = val
                    for j in child[node]:
                        queues.append(j)

            elif(net_dict[node][0]=='or2'):
                if(answer_dict[net_dict[node][1]]==-1 or␣
↪answer_dict[net_dict[node][2]]==-1):
                    continue
                else:
                    val =␣
↪or_fun(answer_dict[net_dict[node][1]],answer_dict[net_dict[node][2]])
                    answer_dict[node] = val
                    for j in child[node]:
                        queues.append(j)

            elif(net_dict[node][0]=='nor2'):
                if(answer_dict[net_dict[node][1]]==-1 or␣
↪answer_dict[net_dict[node][2]]==-1):
                    continue
                else:
                    val =␣
↪nor_fun(answer_dict[net_dict[node][1]],answer_dict[net_dict[node][2]])
                    answer_dict[node] = val
                    for j in child[node]:
                        queues.append(j)


            elif(net_dict[node][0]=='nand2'):
                if(answer_dict[net_dict[node][1]]==-1 or␣
↪answer_dict[net_dict[node][2]]==-1):
                    continue
                else:
```

```
                    val =␣
↪nand_fun(answer_dict[net_dict[node][1]],answer_dict[net_dict[node][2]])
                    answer_dict[node] = val
                    for j in child[node]:
                        queues.append(j)

            elif(net_dict[node][0]=='xor2'):
                if(answer_dict[net_dict[node][1]]==-1 or␣
↪answer_dict[net_dict[node][2]]==-1):
                    continue
                else:
                    val =␣
↪xor_fun(answer_dict[net_dict[node][1]],answer_dict[net_dict[node][2]])
                    answer_dict[node] = val
                    for j in child[node]:
                        queues.append(j)

            elif(net_dict[node][0]=='xnor2'):
                if(answer_dict[net_dict[node][1]]==-1 or␣
↪answer_dict[net_dict[node][2]]==-1):
                    continue
                else:
                    val =␣
↪xnor_fun(answer_dict[net_dict[node][1]],answer_dict[net_dict[node][2]])
                    answer_dict[node] = val
                    for j in child[node]:
                        queues.append(j)


print(answer_dict)
```

This above lines simulates a logic circuit using an event-driven approach. It begins by establishing sets for node types and primary nodes. Then it creates a dictionary with each node's child nodes. It then creates a queue with the primary nodes and loops through them. If the node is a primary node, its value is set to the value of its input. Otherwise, its value is determined by its function (for example, AND, OR, NOT) and the values of its input nodes. The calculated value is saved in a dictionary, and the processed node's child nodes are added to the queue. Finally, the dictionary is printed with the calculated values.

[ ]:
```
inputsz = len(Inputs)
for i in range (2,inputsz):
    changed_queue = []
    no_Primary_nodes = len(Inputs[0])
    for j in range (no_Primary_nodes):
        if(Inputs[i][j]!=Inputs[i-1][j]):
            changed_queue.append(Inputs[0][j])
            answer_dict[Inputs[0][j]]= -1
```

```
    while(len(changed_queue)!=0):
        # print(changed_queue)
        node = changed_queue.pop(0)

        if(node in primary_nodes):
            answer_dict[node] = input_dict[node][i-1]
            for j in child[node]:
                changed_queue.append(j)
                answer_dict[j]=-1

        else:

            if(net_dict[node][0]=='inv'):
                if(answer_dict[net_dict[node][1]]==-1):
                    continue
                else:
                    val = inv_fun(answer_dict[net_dict[node][1]])
                    answer_dict[node] = val
                    for j in child[node]:
                        changed_queue.append(j)
                        answer_dict[j]=-1

            else:
                if(net_dict[node][0]=='and2'):
                    if(answer_dict[net_dict[node][1]]==-1 or␣
↪answer_dict[net_dict[node][2]]==-1):
                        continue
                    else:
                        val =␣
↪and_fun(answer_dict[net_dict[node][1]],answer_dict[net_dict[node][2]])
                        answer_dict[node] = val
                        for j in child[node]:
                            changed_queue.append(j)
                            answer_dict[j]=-1

                elif(net_dict[node][0]=='or2'):
                    if(answer_dict[net_dict[node][1]]==-1 or␣
↪answer_dict[net_dict[node][2]]==-1):
                        continue
                    else:
                        val =␣
↪or_fun(answer_dict[net_dict[node][1]],answer_dict[net_dict[node][2]])
                        answer_dict[node] = val
                        for j in child[node]:
                            changed_queue.append(j)
```

```
                        answer_dict[j]=-1

            elif(net_dict[node][0]=='nor2'):
                if(answer_dict[net_dict[node][1]]==-1 or␣
↪answer_dict[net_dict[node][2]]==-1):
                    continue
                else:
                    val =␣
↪nor_fun(answer_dict[net_dict[node][1]],answer_dict[net_dict[node][2]])
                    answer_dict[node] = val
                    for j in child[node]:
                        changed_queue.append(j)
                        answer_dict[j]=-1


            elif(net_dict[node][0]=='nand2'):
                if(answer_dict[net_dict[node][1]]==-1 or␣
↪answer_dict[net_dict[node][2]]==-1):
                    continue
                else:
                    val =␣
↪nand_fun(answer_dict[net_dict[node][1]],answer_dict[net_dict[node][2]])
                    answer_dict[node] = val
                    for j in child[node]:
                        changed_queue.append(j)
                        answer_dict[j]=-1

            elif(net_dict[node][0]=='xor2'):
                if(answer_dict[net_dict[node][1]]==-1 or␣
↪answer_dict[net_dict[node][2]]==-1):
                    continue
                else:
                    val =␣
↪xor_fun(answer_dict[net_dict[node][1]],answer_dict[net_dict[node][2]])
                    answer_dict[node] = val
                    for j in child[node]:
                        changed_queue.append(j)
                        answer_dict[j]=-1

            elif(net_dict[node][0]=='xnor2'):
                if(answer_dict[net_dict[node][1]]==-1 or␣
↪answer_dict[net_dict[node][2]]==-1):
                    continue
                else:
                    val =␣
↪xnor_fun(answer_dict[net_dict[node][1]],answer_dict[net_dict[node][2]])
```

```
                        answer_dict[node] = val
                        for j in child[node]:
                            changed_queue.append(j)
                            answer_dict[j]=-1

    print(answer_dict)
```

Both approaches to evaluating a circuit have benefits as well as drawbacks, and their effectiveness is determined by the specific characteristics of the input vectors and the size of the circuit.

Topological sorting and multiple rounds of circuit evaluation are generally faster and more efficient for circuits with a small number of input vectors and a relatively large number of gates. This method involves sorting the gates in the circuit according to their dependencies, so that each gate is evaluated only after all of its input gates have been evaluated. The process is repeated until the output nets stabilise. This approach can take advantage of the fact that many gates in a circuit will not change their output values for certain input vectors and thus do not need to be evaluated for every input vector.

Event-driven simulation, on the other hand, is generally faster and more efficient for circuits with a large number of input vectors and a small number of gates. This method involves simulating the circuit by tracking the events that cause changes in the output nets and updating the circuit only when necessary. This method can take advantage of the fact that many gates in a circuit will not change their output values for consecutive input vectors and thus do not need to be evaluated for each input vector.

In summary, the most efficient approach for evaluating a circuit will be determined by the circuit's specific characteristics and the input vectors. Event-driven simulation is likely to be the more efficient approach for large circuits with a large number of input vectors. Topological sort and multiple rounds of circuit evaluations may be more efficient for smaller circuits with fewer input vectors.