Week 4 Sujal Burad EE21B144

March 1, 2023

For errors, I have assumed that the netlist file follows the proper syntax as mentioned in moodle.

0.1 Libraries :

[1]:
```python
import queue # To work with queue
```

[2]:
```python
# All the variables that will be used throughout the code
global gate_id, gate_type, input1, input2, output, s, primary_inputs, states,␣
 ↪graph, nl, levels, final_state,start_time,end_time
global state_q, start_value_q, end_value_q, number_of_inputs_affected,␣
 ↪final_state_list, final_state_list_values
global check_list, output_done
```

1 1] Topological order evaluation

1.1 To open the file of input netlists and read it

[3]:
```python
def file_input_netlists(file_name):
    file = open(file_name, 'r') # Opening the netlist file

    netlist = ""
    for i in file.readlines():
        netlist = netlist + i

    netlist = netlist.splitlines()

    gate_id = []
    gate_type = []
    input1 = []
    input2 = []
    output = []

    for i in netlist:
        i = i.split()
        gate_id.append(i[0])
        gate_type.append(i[1].upper())
        input1.append(i[2])
        # I am assuming the input2 in case of 'INV' and 'BUF' gate to be 1,␣
↪instead of it being empty, for simplification
        if(gate_type[-1] == 'INV'):
            input2.append('1')
            output.append(i[3])

        elif(gate_type[-1] == 'BUF'):
            input2.append('1')
            output.append(i[3])

        else:
            input2.append(i[3])
            output.append(i[4])

    print("Gate_id : ", gate_id, "\n")
    print("Gate_type : ", gate_type, "\n")
    print("Input_1 : ", input1, "\n")
    print("Input_2 : ", input2, "\n")
    print("Output : ", output, "\n")

    return gate_id, gate_type, input1, input2, output
```

1.2 Checking for the cyclic condition of the .net file

[4]:
```
check_list = []
output_done = []

def continue_till(input1, input2, output, output_check, inp1, inp2):

#     print(check_list, '\t', output_check, '\t', inp1, '\t', inp2)

    # If output is in the input, it will be cyclic
    if(output_check == input1[output.index(output_check)] or output_check ==␣
↪input2[output.index(output_check)]):
#         print("Cyclic")
        return True

    # Iterating recursively through the function and for loop to get the␣
↪condition of cyclic nature.
    # If an output gets repeated more than once, then the graph is cyclic.
    # If all the outputs are present only once, then the graph is acyclic.
    if(output_check in input1):
# enumerate() is an in-built python function to that returns the index / indices␣
↪of an element in a list,
# by iterating over the list.
        index1 = [y for y, x in enumerate(input1) if x == output_check]
        output_done.append(output_check)
        for i1 in index1:
            check_list.append(inp1)
            check_list.append(inp2)
            # This if block is used to terminate the function when we first find␣
↪that the output is repeated more than once.
            if(output[i1] in output_done):
#                   print("Possible infinite loop, stopped by I1")
                return
            else:
                output_done.append(output[i1])
                continue_till(input1, input2, output, output[i1], input1[i1],␣
↪input2[i1])

    elif(output_check in input2):
        index2 = [y for y, x in enumerate(input2) if x == output_check]
        for i2 in index2:
            check_list.append(inp1)
            check_list.append(inp2)
            # This if block is used to terminate the function when we first find␣
↪that the output is repeated more than once.
            if(output[i2] in output_done):
#                   print("Possible infinite loop, stopped by I2")
                return
            else:
                output_done.append(output[i2])
                continue_till(input1, input2, output, output[i2], input1[i2],␣
↪input2[i2])

 # print(check_list, '\t', output_check, '\t', inp1, '\t', inp2)
    if(output_check in check_list):
#           print("Cyclic graph")
        return True

    else:
#           print("Acyclic")
# Clearing this lists for the next outputs, if none of the above cases gets␣
↪satisfied.
        check_list.clear()
        output_done.clear()
        return False

def error_in_net_file(netlist_file):
    check_list = []
    gate_id, gate_type, input1, input2, output =␣
↪file_input_netlists(netlist_file)
    for out in range(len(output)):
        if(continue_till(input1, input2, output, output[out], input1[out],␣
↪input2[out])):
#             print('Directed cyclic graph')
#             print("True")
            return True
            break
        else:
            continue

#     print("Acyclic graph")
#     print("False")
    return False
```

1.3 To open the file of input vectors and read it

[5]:
```
def file_input(file_name, gate_type):
    file = open(file_name, 'r')

    s = ""
    for i in file.readlines():
        s = s + i
    s = s.splitlines()
    if 'INV' in gate_type or 'BUF' in gate_type:
        for i in range(len(s)):
            s[i] = s[i] + ' 1'
    else:
        pass

    primary_inputs = []
    x = s
    for i in x[0].split():
        primary_inputs.append(i)


    return s, primary_inputs
```

1.4 Combined information about netlists and input vectors

[6]:
```
def info(gate_id, gate_type, input1, input2, output, s, primary_inputs):

    print("Primary inputs : ", primary_inputs)
# Adding an extra '1' at the end of primary_inputs depending on whether 'INV' or␣
↪'BUF' gate is there in gate_type or not.
    states = []
    print("\nStates of each primary inputs at different time instants ")
# Adding an '1' at the end of every states' line depending on whether 'INV' or␣
↪'BUF' gate is there in gate_type or not.
```

```
    for i in range(1, len(s)):
        states.append(s[i])
    # Considering the start time as T0
    for i in range(len(states)):
        print(f"T{i} = ", states[i])


    print("\nInput_1  Input_2 \t Output \t\t Gate_type")
    for i in range(len(input1)):
        print(input1[i], "\t", input2[i], "\t\t", output[i], "\t\t",␣
 ↪gate_type[i])

    print("\n")
    return states
```

1.5 Creating a DAG for the netlist and input vectors

[7]:
```
import networkx as nx

def DAG(gate_type, input1, input2, output, primary_inputs):
    # Create a DAG
    graph = nx.DiGraph()

    # Adding nodes and edges to DAG
    nodes_and_edges = []
    for i in range(len(input1)):
        if((input1[i], output[i]) not in nodes_and_edges):
            nodes_and_edges.append((input1[i], output[i]))

        if((input2[i], output[i]) not in nodes_and_edges):
            nodes_and_edges.append((input2[i], output[i]))

    graph.add_edges_from(nodes_and_edges)

    dictionary = {}
    for i in range(len(primary_inputs)):
        dictionary[primary_inputs[i]] = 'PI'

    for i in range(len(output)):
        dictionary[output[i]] = gate_type[i]


    nx.set_node_attributes(graph, dictionary, name = "gateType")

    # Sorting the nodes in topological order
    nl = list(nx.topological_sort(graph))
    return graph
```

1.6 Finding the output for various gates

[8]:
```
# A and B should be passed as integer
def gate(gate_type, A, B):
#     print(type(A), type(B))
    A = int(A)
    B = int(B)
    if(gate_type == 'AND2'):
        return A * B

    elif(gate_type == 'OR2'):
        if(A == 1 or B == 1):
            return 1
        else:
            return 0

    elif(gate_type == 'NOT' or gate_type == 'INV'):
        if(A == 1):
            return 0
        else:
            return 1

    elif(gate_type == 'BUF'):
        return A

    elif(gate_type == 'NOR2'):
        if(A == 1 or B == 1):
            return 0
        else:
            return 1

    elif(gate_type == 'NAND2'):
        return 1 - (A * B)

    elif(gate_type == 'XOR2'):
        if((A + B) == 1):
            return 1
        else:
            return 0

    elif(gate_type == 'XNOR2'):
        if((A + B) == 1):
            return 0
        else:
            return 1
```

```
[11]: # final_state is the dictionary containing all the nodes and their outputs
final_state = {}
# Combined function for the topological evaluation
def states_of_all_nets(netlist_file, input_file):

    if(not error_in_net_file(netlist_file)):
        print("\n\nIt's a Directed Acyclic graph \n\n")
        output_file = open("Output_file_topo.txt", 'w') # The output file for␣
 ↪storing the values
        onlyonce = 1
        gate_id, gate_type, input1, input2, output =␣
 ↪file_input_netlists(netlist_file)
        s, primary_inputs = file_input(input_file, gate_type)
        states = info(gate_id, gate_type, input1, input2, output, s,␣
 ↪primary_inputs)
        levels  = list(nx.topological_generations(DAG(gate_type, input1, input2,␣
 ↪output, primary_inputs)))
        print("Nodes in Topological order : ")

        k = 0
        for i in levels:
            print(f"Level_{k} :", i)
            k = k+1
#          # To get the topological levels of the nodes
        # Nodes with the same levels are put in a single list

        for time in range(len(states)):
            print("\n\n", "-"*50, f"Time = T{time}", "-"*50)
            for i in range(len(levels)):
                print(f"\nLevel {i}")
                for j in range(len(levels[i])):
                    node = levels[i][j]
                    if(i == 0): # Reading the primary inputs
                        val = states[time]
                        val = val.split()
                        print("State of", node, "=", val[primary_inputs.
 ↪index(node)])
                        final_state[node] = val[primary_inputs.index(node)]

                    else: # Calculating the value of nodes other than the␣
 ↪primary inputs
                        index = output.index(node)
                        logic_gate = gate_type[index]
                        A = (final_state.get(input1[index]))
                        B = (final_state.get(input2[index]))
                        print("State of", node, "=", gate(logic_gate, A, B))
                        final_state[node] = gate(logic_gate, A, B)
            # Writing the states and its values in output file
            final_state_list = list(final_state.keys())
            final_state_list.sort()

            if(onlyonce == 1): # To print the alphabetically ordered net names␣
↪only once in the file
                output_file.write(' '.join(final_state_list)) # .join() is used␣
↪to join all the elements of list as string with
    # seperator as ' '
                onlyonce = 0
            output_file.write('\n')
            final_state_list_values = []
            for finalstate in final_state_list:
                final_state_list_values.append(str(final_state[finalstate]))
            output_file.write(' '.join(final_state_list_values))
            output_file.write('\n')

    #          print("Sorted list = ", final_state_list)
    #          print("Sorted list's values = ", final_state_list_values)
    #          print("All nets and their outputs for time ", time,  " = ",␣
↪final_state)

        output_file.close()

    else:
#          pass
        print("It is a Cyclic graph, and will run into an infinite loop")
```

[12]:
```
states_of_all_nets('parity.net', 'parity.inputs')
```

```
Gate_id :  ['g219_dummy0', 'g219', 'g220_dummy0', 'g220', 'g222_dummy0', 'g222',
'g221_dummy0', 'g221', 'g223_dummy0', 'g223', 'g224', 'g225', 'g226', 'g227',
'g228']

Gate_type :  ['XOR2', 'XNOR2', 'XOR2', 'XNOR2', 'XOR2', 'XNOR2', 'XOR2',
'XNOR2', 'XOR2', 'XNOR2', 'XOR2', 'XNOR2', 'XNOR2', 'XOR2', 'XNOR2']

Input_1 :  ['n_6', 'n_8', 'n_4', 'n_5', 'o', 'n_2', 'k', 'n_1', 'c', 'n_0', 'h',
'f', 'n', 'j', 'b']

Input_2 :  ['n_7', 'dummy_0', 'n_3', 'dummy_1', 'p', 'dummy_2', 'l', 'dummy_3',
'd', 'dummy_4', 'g', 'e', 'm', 'i', 'a']

Output :  ['dummy_0', 'q', 'dummy_1', 'n_8', 'dummy_2', 'n_7', 'dummy_3', 'n_6',
'dummy_4', 'n_5', 'n_4', 'n_3', 'n_2', 'n_1', 'n_0']
```

```
It's a Directed Acyclic graph


Gate_id :  ['g219_dummy0', 'g219', 'g220_dummy0', 'g220', 'g222_dummy0', 'g222',
'g221_dummy0', 'g221', 'g223_dummy0', 'g223', 'g224', 'g225', 'g226', 'g227',
'g228']

Gate_type :  ['XOR2', 'XNOR2', 'XOR2', 'XNOR2', 'XOR2', 'XNOR2', 'XOR2',
'XNOR2', 'XOR2', 'XNOR2', 'XOR2', 'XNOR2', 'XNOR2', 'XOR2', 'XNOR2']

Input_1 :  ['n_6', 'n_8', 'n_4', 'n_5', 'o', 'n_2', 'k', 'n_1', 'c', 'n_0', 'h',
'f', 'n', 'j', 'b']

Input_2 :  ['n_7', 'dummy_0', 'n_3', 'dummy_1', 'p', 'dummy_2', 'l', 'dummy_3',
'd', 'dummy_4', 'g', 'e', 'm', 'i', 'a']

Output :  ['dummy_0', 'q', 'dummy_1', 'n_8', 'dummy_2', 'n_7', 'dummy_3', 'n_6',
'dummy_4', 'n_5', 'n_4', 'n_3', 'n_2', 'n_1', 'n_0']

Primary inputs :  ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l',
'm', 'n', 'o', 'p']

States of each primary inputs at different time instants
T0 =  0 1 0 0 0 1 0 0 1 1 0 0 1 0 1 1
T1 =  0 0 1 0 0 0 0 0 0 1 0 0 1 1 1 1
T2 =  1 0 0 0 0 0 1 0 1 0 0 1 1 0 0 0
T3 =  0 0 1 1 1 1 0 0 0 1 0 1 1 1 1 1
T4 =  1 1 1 1 1 1 0 1 1 0 1 0 1 0 0 1

Input_1  Input_2         Output                  Gate_type
n_6      n_7             dummy_0                 XOR2
n_8      dummy_0                 q               XNOR2
n_4      n_3             dummy_1                 XOR2
n_5      dummy_1                 n_8             XNOR2
o        p               dummy_2                 XOR2
n_2      dummy_2                 n_7             XNOR2
k        l               dummy_3                 XOR2
n_1      dummy_3                 n_6             XNOR2
c        d               dummy_4                 XOR2
n_0      dummy_4                 n_5             XNOR2
h        g               n_4             XOR2
f        e               n_3             XNOR2
n        m               n_2             XNOR2
j        i               n_1             XOR2
b        a               n_0             XNOR2
```

```
Nodes in Topological order :
Level_0 : ['o', 'p', 'k', 'l', 'c', 'd', 'h', 'g', 'f', 'e', 'n', 'm', 'j', 'i',
'b', 'a']
Level_1 : ['dummy_2', 'dummy_3', 'dummy_4', 'n_4', 'n_3', 'n_2', 'n_1', 'n_0']
Level_2 : ['dummy_1', 'n_7', 'n_6', 'n_5']
Level_3 : ['dummy_0', 'n_8']
Level_4 : ['q']


 -------------------------------------------------- Time = T0
--------------------------------------------------

Level 0
State of o = 1
State of p = 1
State of k = 0
State of l = 0
State of c = 0
State of d = 0
State of h = 0
State of g = 0
State of f = 1
State of e = 0
State of n = 0
State of m = 1
State of j = 1
State of i = 1
State of b = 1
State of a = 0

Level 1
State of dummy_2 = 0
State of dummy_3 = 0
State of dummy_4 = 0
State of n_4 = 0
State of n_3 = 0
State of n_2 = 0
State of n_1 = 0
State of n_0 = 0

Level 2
State of dummy_1 = 0
State of n_7 = 1
State of n_6 = 1
State of n_5 = 1

Level 3
```

```
State of dummy_0 = 0
State of n_8 = 0

Level 4
State of q = 1


 -------------------------------------------------- Time = T1
--------------------------------------------------

Level 0
State of o = 1
State of p = 1
State of k = 0
State of l = 0
State of c = 1
State of d = 0
State of h = 0
State of g = 0
State of f = 0
State of e = 0
State of n = 1
State of m = 1
State of j = 1
State of i = 0
State of b = 0
State of a = 0

Level 1
State of dummy_2 = 0
State of dummy_3 = 0
State of dummy_4 = 1
State of n_4 = 0
State of n_3 = 1
State of n_2 = 1
State of n_1 = 1
State of n_0 = 1

Level 2
State of dummy_1 = 1
State of n_7 = 0
State of n_6 = 0
State of n_5 = 1

Level 3
State of dummy_0 = 0
State of n_8 = 1
```

```
Level 4
State of q = 0


 -------------------------------------------------- Time = T2
--------------------------------------------------

Level 0
State of o = 0
State of p = 0
State of k = 0
State of l = 1
State of c = 0
State of d = 0
State of h = 0
State of g = 1
State of f = 0
State of e = 0
State of n = 0
State of m = 1
State of j = 0
State of i = 1
State of b = 0
State of a = 1

Level 1
State of dummy_2 = 0
State of dummy_3 = 1
State of dummy_4 = 0
State of n_4 = 1
State of n_3 = 1
State of n_2 = 0
State of n_1 = 1
State of n_0 = 0

Level 2
State of dummy_1 = 0
State of n_7 = 1
State of n_6 = 1
State of n_5 = 1

Level 3
State of dummy_0 = 0
State of n_8 = 0

Level 4
State of q = 1
```

```
 -------------------------------------------------- Time = T3
--------------------------------------------------

Level 0
State of o = 1
State of p = 1
State of k = 0
State of l = 1
State of c = 1
State of d = 1
State of h = 0
State of g = 0
State of f = 1
State of e = 1
State of n = 1
State of m = 1
State of j = 1
State of i = 0
State of b = 0
State of a = 0

Level 1
State of dummy_2 = 0
State of dummy_3 = 1
State of dummy_4 = 0
State of n_4 = 0
State of n_3 = 1
State of n_2 = 1
State of n_1 = 1
State of n_0 = 1

Level 2
State of dummy_1 = 1
State of n_7 = 0
State of n_6 = 1
State of n_5 = 0

Level 3
State of dummy_0 = 1
State of n_8 = 0

Level 4
State of q = 0


 -------------------------------------------------- Time = T4
--------------------------------------------------
```

```
Level 0
State of o = 0
State of p = 1
State of k = 1
State of l = 0
State of c = 1
State of d = 1
State of h = 1
State of g = 0
State of f = 1
State of e = 1
State of n = 0
State of m = 1
State of j = 0
State of i = 1
State of b = 1
State of a = 1

Level 1
State of dummy_2 = 1
State of dummy_3 = 1
State of dummy_4 = 0
State of n_4 = 1
State of n_3 = 1
State of n_2 = 0
State of n_1 = 1
State of n_0 = 1

Level 2
State of dummy_1 = 0
State of n_7 = 0
State of n_6 = 1
State of n_5 = 0

Level 3
State of dummy_0 = 1
State of n_8 = 1

Level 4
State of q = 1
```

[13]:
```
# states_of_all_nets('c432.net', 'c432.inputs')
```

[14]:
```
# states_of_all_nets('c17.net', 'c17.inputs')
```

[15]:
```
# states_of_all_nets('c8.net', 'c8.inputs')
```

[16]:
```
# It will first check for the netlist file, and then the inputs. If it runs␣
↪properly, then it will execute the input file.
# A random input file has to be provided, even if it is cyclic, to satisfy the␣
↪parameters of the function that I wrote.
# If it is cyclic, a proper input file has to be provided.
states_of_all_nets('c17_1.net', 'c179484.inputs')
```

```
Gate_id :  ['g51', 'g52', 'g53', 'g54', 'g55', 'g56']

Gate_type :  ['NAND2', 'NAND2', 'NAND2', 'NAND2', 'NAND2', 'NAND2']

Input_1 :  ['n_3', 'n_3', 'N22', 'n_1', 'N1', 'N3']

Input_2 :  ['n_0', 'n_2', 'N2', 'N7', 'N3', 'N6']

Output :  ['N22', 'N23', 'n_3', 'n_2', 'n_0', 'n_1']

It is a Cyclic graph, and will run into an infinite loop
```

2 Event driven evaluation

[17]:
```
final_state = {}

# This function evaluates the final_state value at given time i.e T0, T1, T2, ...
# In case no value of time is given, I have taken T0 as the start time for␣
↪initialization
# Also, if no value of end_time is given I have taken T1 as the end time to␣
↪calculate the final state

def current_state_values(netlist_file, input_file, start_time = 0, end_time = 1):
    gate_id, gate_type, input1, input2, output =␣
↪file_input_netlists(netlist_file)
    s, primary_inputs = file_input(input_file, gate_type)
    states = info(gate_id, gate_type, input1, input2, output, s, primary_inputs)
    levels  = list(nx.topological_generations(DAG(gate_type, input1, input2,␣
↪output, primary_inputs)))
    print("Nodes in Topological order : ")

    k = 0
    for i in levels:
        print(f"Level_{k} :", i)
        k = k+1

    for i in range(len(levels)):
        for j in range(len(levels[i])):
            node = levels[i][j]
            if(i == 0):
                val = states[start_time]
                val = val.split()
                final_state[node] = val[primary_inputs.index(node)]

            else:
                index = output.index(node)
                logic_gate = gate_type[index]
                A = (final_state.get(input1[index]))
                B = (final_state.get(input2[index]))
                final_state[node] = gate(logic_gate, A, B)

    print('\nFinal state at start time :\n', final_state, '\n')
    return gate_id, gate_type, input1, input2, output, s, primary_inputs,␣
↪states, levels, final_state
```

[18]:
```
# This prints out the primary values whose values has been changed and the total␣
↪number of affected primary values
def affected_primary_values(start_time, end_time, states, primary_inputs):
    time = []
    for i in range(len(states)):
        time.append(states[i].split())

    # Using queues to store the states, and values of the changed primary inputs
    state_q = queue.Queue() # To store the primary input states, whose value has␣
↪changed
    start_value_q = queue.Queue() # Value at the start time
    end_value_q = queue.Queue() # To store the values at end time
    number_of_inputs_affected = 0 # The store the number of primary input values␣
↪that got changed
    # The number_of_inputs_affected only calculates the changes in primary input␣
↪value at start time and at end time

    # Only did for start time and end time, remaining for other time states
    for i in range(len(time[start_time])):
        if(time[start_time][i] != time[end_time][i]):
            number_of_inputs_affected += 1
            print("Value of primary input ", primary_inputs[i], " changed from␣
↪", time[start_time][i], " to ", time[end_time][i])
            state_q.put(primary_inputs[i])
            start_value_q.put(time[start_time][i])
            end_value_q.put(time[end_time][i])

    print("Number of inputs affected = ", number_of_inputs_affected)

    return state_q, start_value_q, end_value_q, number_of_inputs_affected
```

The below function will update values in final_state(dictionary) that will be get affected by the change in primary input values. Rest of the values of final_state will remain the same as previous one. enumerate() is an in-built python function to that returns the index / indices of an element in a list, by iterating over the list.

To see how far the changed primary input values will affect the output values, the following cases can happen : 1] If 'inp' is present in any one of the input lists, then it will affect that respective output. 2] If 'inp' is not present in any one of the input lists, then the change in primary input values will not affect that and the further outputs.

The function follows recursiveness until it finds the second condition, and exits the function. If the 1st condition keeps getting followed, it will go on till the last output of the netlist

[19]:
```
def output_for_given_input(inp, input1, input2, output, gate_type):
    output_list = []

# I have made two cases to check if 'inp' (the previous output) is in any one of␣
 ↪the input lists : input1, and input2.
    if(inp in input1):
        index1 = [y for y, x in enumerate(input1) if x == inp] # To find all the␣
 ↪indices of 'inp' in input1 list
        for l in index1:

            # Updating the values in final_state
            node = output[l]
            index = output.index(node)
            logic_gate = gate_type[index]
            A = (final_state.get(input1[index]))
            B = (final_state.get(input2[index]))
            final_state[node] = gate(logic_gate, A, B)

            output_list.append(output[l])

# Recursive function, to get all the outputs that will be affected by the change␣
 ↪in values of primary , until the 2nd condition
# occurs, as mentioned above
            output_for_given_input(output[l], input1, input2, output, gate_type)

            print("Output for index 1 = ", output_list)

    if(inp in input2):
        index2 = [y for y, x in enumerate(input2) if x == inp] # To find all the␣
 ↪indices of 'inp' in input1 list
        print("Index 2 = ", index2)
        for l in index2:

            # Updating the values in final_state
            node = output[l]
            index = output.index(node)
            logic_gate = gate_type[index]
            A = (final_state.get(input1[index]))
            B = (final_state.get(input2[index]))
            final_state[node] = gate(logic_gate, A, B)

            output_list.append(output[l])

# Recursive function, to get all the outputs that will be affected by the change␣
 ↪in values of primary , until the 2nd condition
# occurs, as mentioned above
            output_for_given_input(output[l], input1, input2, output, gate_type)

        print("Output for index 2 = ", output_list)
```

[20]:
```
check = []
# It goes through all the changed primary inputs, and will find all those␣
 ↪outputs whose value will change, and update it in the
# dictionary final_state
def the_loop(state_q, start_value_q, end_value_q, number_of_inputs_affected,␣
 ↪input1, input2, output, gate_type):

    for p in range(number_of_inputs_affected):
        i = state_q.get()
        final_state[i] = end_value_q.get()
        if(i not in check):
            check.append(i)
            if(i in input1):
                print("\n\nPrimary input is ", i)
                indices1 = [y for y, x in enumerate(input1) if x == i]
                print("Number of times ", i, ' is in input 1 = ',␣
 ↪len(indices1)," and is at ", indices1)
                for k in indices1:
                    print("Primary input's output ",output[k])

                    node = output[k]
                    index = output.index(node)
                    logic_gate = gate_type[index]
                    A = (final_state.get(input1[index]))
                    B = (final_state.get(input2[index]))
                    print("State of", node, "=", gate(logic_gate, A, B))
                    final_state[node] = gate(logic_gate, A, B)

                    output_for_given_input(output[k], input1, input2, output,␣
 ↪gate_type)

            if(i in input2):
                print("\n\nPrimary input is ", i)
                indices2 = [y for y, x in enumerate(input2) if x == i]
                print("Number of times ", i, ' is in input 2 = ', len(indices2),␣
↪" and is at ", indices2)
                for k in indices2:

                    node = output[k]
                    index = output.index(node)
                    logic_gate = gate_type[index]
                    A = (final_state.get(input1[index]))
                    B = (final_state.get(input2[index]))
                    print("State of", node, "=", gate(logic_gate, A, B))
                    final_state[node] = gate(logic_gate, A, B)

                    print("Primary input's output ",output[k])
                    output_for_given_input(output[k], input1, input2, output,␣
↪gate_type)

    print("\nThe final state at end time : \n", final_state)
```

[21]:
```
def event_driven_evaluation(netlist_file, input_file, start_time = 0, end_time =␣
↪1):
    if(not error_in_net_file(netlist_file)):
        print("\n\nIt's a Directed Acyclic graph\n\n")
        if(start_time == end_time):
            print("The values at end time will remain same as that of the start␣
↪time")
            gate_id, gate_type, input1, input2, output, s, primary_inputs,␣
↪states, levels, final_state = current_state_values(netlist_file, input_file,␣
↪start_time, end_time)

        elif(start_time != end_time):
            print("The values at the end time will be different than that of␣
↪start time, if there is change in primary input values")
            gate_id, gate_type, input1, input2, output, s, primary_inputs,␣
↪states, levels, final_state = current_state_values(netlist_file, input_file,␣
↪start_time, end_time)
            state_q, start_value_q, end_value_q, number_of_inputs_affected =␣
↪affected_primary_values(start_time, end_time, states, primary_inputs)
            the_loop(state_q, start_value_q, end_value_q,␣
↪number_of_inputs_affected, input1, input2, output, gate_type)

        output_file = open("Output_file.txt", 'w') # The output file for storing␣
↪the values
        onlyonce = 1

     # Writing the states and its values in output file
```

```
        final_state_list = list(final_state.keys())
        final_state_list.sort()

        if(onlyonce == 1): # To print the alphabetically ordered net names only␣
↪once in the file
            output_file.write(' '.join(final_state_list)) # .join() is used to␣
↪join all the elements of list as string with
    # seperator as ' '
            onlyonce = 0
        output_file.write('\n')
        final_state_list_values = []
        for finalstate in final_state_list:
            final_state_list_values.append(str(final_state[finalstate]))
        output_file.write(' '.join(final_state_list_values))
        output_file.write('\n')
        output_file.close()

    else:
        print("It is a Cyclic graph, and will run into an infinite loop")
```

[22]:
```
event_driven_evaluation('parity.net', 'parity.inputs')
```

```
Gate_id :  ['g219_dummy0', 'g219', 'g220_dummy0', 'g220', 'g222_dummy0', 'g222',
'g221_dummy0', 'g221', 'g223_dummy0', 'g223', 'g224', 'g225', 'g226', 'g227',
'g228']

Gate_type :  ['XOR2', 'XNOR2', 'XOR2', 'XNOR2', 'XOR2', 'XNOR2', 'XOR2',
'XNOR2', 'XOR2', 'XNOR2', 'XOR2', 'XNOR2', 'XNOR2', 'XOR2', 'XNOR2']

Input_1 :  ['n_6', 'n_8', 'n_4', 'n_5', 'o', 'n_2', 'k', 'n_1', 'c', 'n_0', 'h',
'f', 'n', 'j', 'b']

Input_2 :  ['n_7', 'dummy_0', 'n_3', 'dummy_1', 'p', 'dummy_2', 'l', 'dummy_3',
'd', 'dummy_4', 'g', 'e', 'm', 'i', 'a']

Output :  ['dummy_0', 'q', 'dummy_1', 'n_8', 'dummy_2', 'n_7', 'dummy_3', 'n_6',
'dummy_4', 'n_5', 'n_4', 'n_3', 'n_2', 'n_1', 'n_0']



It's a Directed Acyclic graph


The values at the end time will be different than that of start time, if there
is change in primary input values
Gate_id :  ['g219_dummy0', 'g219', 'g220_dummy0', 'g220', 'g222_dummy0', 'g222',
'g221_dummy0', 'g221', 'g223_dummy0', 'g223', 'g224', 'g225', 'g226', 'g227',
'g228']
```

```
Gate_type :  ['XOR2', 'XNOR2', 'XOR2', 'XNOR2', 'XOR2', 'XNOR2', 'XOR2',
'XNOR2', 'XOR2', 'XNOR2', 'XOR2', 'XNOR2', 'XNOR2', 'XOR2', 'XNOR2']

Input_1 :  ['n_6', 'n_8', 'n_4', 'n_5', 'o', 'n_2', 'k', 'n_1', 'c', 'n_0', 'h',
'f', 'n', 'j', 'b']

Input_2 :  ['n_7', 'dummy_0', 'n_3', 'dummy_1', 'p', 'dummy_2', 'l', 'dummy_3',
'd', 'dummy_4', 'g', 'e', 'm', 'i', 'a']

Output :  ['dummy_0', 'q', 'dummy_1', 'n_8', 'dummy_2', 'n_7', 'dummy_3', 'n_6',
'dummy_4', 'n_5', 'n_4', 'n_3', 'n_2', 'n_1', 'n_0']

Primary inputs :  ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l',
'm', 'n', 'o', 'p']

States of each primary inputs at different time instants
T0 =  0 1 0 0 0 1 0 0 1 1 0 0 1 0 1 1
T1 =  0 0 1 0 0 0 0 0 0 1 0 0 1 1 1 1
T2 =  1 0 0 0 0 0 1 0 1 0 0 1 1 0 0 0
T3 =  0 0 1 1 1 1 0 0 0 1 0 1 1 1 1 1
T4 =  1 1 1 1 1 1 0 1 1 0 1 0 1 0 0 1

Input_1  Input_2          Output                  Gate_type
n_6      n_7              dummy_0                 XOR2
n_8      dummy_0                  q               XNOR2
n_4      n_3              dummy_1                 XOR2
n_5      dummy_1                  n_8             XNOR2
o        p                dummy_2                 XOR2
n_2      dummy_2                  n_7             XNOR2
k        l                dummy_3                 XOR2
n_1      dummy_3                  n_6             XNOR2
c        d                dummy_4                 XOR2
n_0      dummy_4                  n_5             XNOR2
h        g                n_4             XOR2
f        e                n_3             XNOR2
n        m                n_2             XNOR2
j        i                n_1             XOR2
b        a                n_0             XNOR2


Nodes in Topological order :
Level_0 : ['o', 'p', 'k', 'l', 'c', 'd', 'h', 'g', 'f', 'e', 'n', 'm', 'j', 'i',
'b', 'a']
Level_1 : ['dummy_2', 'dummy_3', 'dummy_4', 'n_4', 'n_3', 'n_2', 'n_1', 'n_0']
Level_2 : ['dummy_1', 'n_7', 'n_6', 'n_5']
Level_3 : ['dummy_0', 'n_8']
Level_4 : ['q']
```

```
Final state at start time :
 {'o': '1', 'p': '1', 'k': '0', 'l': '0', 'c': '0', 'd': '0', 'h': '0', 'g':
'0', 'f': '1', 'e': '0', 'n': '0', 'm': '1', 'j': '1', 'i': '1', 'b': '1', 'a':
'0', 'dummy_2': 0, 'dummy_3': 0, 'dummy_4': 0, 'n_4': 0, 'n_3': 0, 'n_2': 0,
'n_1': 0, 'n_0': 0, 'dummy_1': 0, 'n_7': 1, 'n_6': 1, 'n_5': 1, 'dummy_0': 0,
'n_8': 0, 'q': 1}

Value of primary input  b  changed from  1  to  0
Value of primary input  c  changed from  0  to  1
Value of primary input  f  changed from  1  to  0
Value of primary input  i  changed from  1  to  0
Value of primary input  n  changed from  0  to  1
Number of inputs affected =  5


Primary input is  b
Number of times  b  is in input 1 =  1  and is at  [14]
Primary input's output  n_0
State of n_0 = 1
Output for index 1 =  ['q']
Output for index 1 =  ['n_8']
Output for index 1 =  ['n_5']


Primary input is  c
Number of times  c  is in input 1 =  1  and is at  [8]
Primary input's output  dummy_4
State of dummy_4 = 1
Index 2 =  [9]
Output for index 1 =  ['q']
Output for index 1 =  ['n_8']
Output for index 2 =  ['n_5']


Primary input is  f
Number of times  f  is in input 1 =  1  and is at  [11]
Primary input's output  n_3
State of n_3 = 1
Index 2 =  [2]
Index 2 =  [3]
Output for index 1 =  ['q']
Output for index 2 =  ['n_8']
Output for index 2 =  ['dummy_1']


Primary input is  i
Number of times  i  is in input 2 =  1  and is at  [13]
```

```
State of n_1 = 1
Primary input's output  n_1
Index 2 =  [1]
Output for index 2 =  ['q']
Output for index 1 =  ['dummy_0']
Output for index 1 =  ['n_6']


Primary input is  n
Number of times  n  is in input 1 =  1  and is at  [12]
Primary input's output  n_2
State of n_2 = 1
Index 2 =  [0]
Index 2 =  [1]
Output for index 2 =  ['q']
Output for index 2 =  ['dummy_0']
Output for index 1 =  ['n_7']

The final state at end time :
 {'o': '1', 'p': '1', 'k': '0', 'l': '0', 'c': '1', 'd': '0', 'h': '0', 'g':
'0', 'f': '0', 'e': '0', 'n': '1', 'm': '1', 'j': '1', 'i': '0', 'b': '0', 'a':
'0', 'dummy_2': 0, 'dummy_3': 0, 'dummy_4': 1, 'n_4': 0, 'n_3': 1, 'n_2': 1,
'n_1': 1, 'n_0': 1, 'dummy_1': 1, 'n_7': 0, 'n_6': 0, 'n_5': 1, 'dummy_0': 0,
'n_8': 1, 'q': 0}
```

[23]:
```
# event_driven_evaluation('c432.net', 'c432.inputs')
```

[24]:
```
# event_driven_evaluation('c17.net', 'c17.inputs')
```

[25]:
```
# event_driven_evaluation('c8.net', 'c8.inputs')
```

[ ]:
```
# Run time for Event driven evaluation
# The below are for start_time = 0, and end_time =1

# %timeit event_driven_evaluation('parity.net', 'parity.inputs')
# %timeit event_driven_evaluation('c432.net', 'c432.inputs')
# %timeit event_driven_evaluation('c17.net', 'c17.inputs')
# %timeit event_driven_evaluation('c8.net', 'c8.inputs')
# We will have to comment all the print statements, otherwise the runtime will␣
 ↪be high
```

[ ]:
```
# Run time for topological ordered evaluation
# The below runs for all the time instances
# %timeit states_of_all_nets('parity.net', 'parity.inputs')
# %timeit states_of_all_nets('c432.net', 'c432.inputs')
# %timeit states_of_all_nets('c17.net', 'c17.inputs')
# %timeit states_of_all_nets('c8.net', 'c8.inputs')
```

The time found by %timeit for event driven approach and topological sort, highly depends on the circuit and the number of inputs. Event driven approach is better for only a single change in time instant than topological sort.

The variation of time taken by both the approaches can be understood by their respective working.
1] Topological sort : It is a graph based algorithm, whose time complexity depends on the N = number of nodes and E = number of edges. It evaluates a node only after all of its inputs have been evaluated. It is an efficient approach for circuits with small number of nodes and large number of gates. It is an organized approach, but the circuit is evaluated everytime, when the time is changing.

2] Event driven approach : It evaluates the circuit only when there is a change in the inputs. This approach is more efficient with large number of inputs and small number of gates. It responds more quickly to the changes in inputs, but it might happen that when moving in the queue we do not know both the inputs of the gates, so is efficient with less number of gates.

```
[ ]:
```