EE2703 Week 4

Srikar Babu Gadipudi EE21B138

March 1, 2023

1 Prerequisites to run this notebook

This notebook consists of all the inputs and netlists used for testing. The user is expected to have networkx and collections library along with a working jupyter environment. The folder `outputs` contains all the outputs generated by different netlists using the two mentioned methods (topologiaclly ordered evaluation and event-driven evaluation).

1.1 Imports

[1]:
```
import networkx as nx
from collections import deque
```

1.2 File reading fucntion

[2]:
```
def fr(name):
    f = open(name, "r")
    ls = f.readlines()
    for i in range(len(ls)):
        if ls[i][-1] == '\n':
            ls[i] = ls[i][:-1]
    arr = [[]]

    for i in range(len(ls)):
        arr.append(ls[i].split())
    f.close()
    return arr[1:]
```

1.3 Logic Gates

[3]:
```
def And(n1, n2):
    return n1*n2
```

[4]:
```
def Or(n1, n2):
    return n1 | n2
```

[5]:
```
def Not(n1):
    return int(not n1)
```

[6]:
```
def Nor(n1, n2):
    if(n1 == 0) and (n2 == 0):
        return 1
    else:
        return 0
```

[7]:
```
def Nand(n1, n2):
    if n1 == 1 and n2 == 1:
        return 0
    else:
        return 1
```

[8]:
```
def Xor(n1, n2):
    if n1 != n2:
        return 1
    else:
        return 0
```

[9]:
```
def Xnor(n1, n2):
    return Not(Xor(n1, n2))
```

2 Topologically Ordered Evaluation

[10]:
```
def dag(l):
    connections = []
    for i in range(len(l)):
        connections.append((l[i][2], l[i][-1]))
        if l[i][1] != 'inv' and l[i][1] != "buf":
            connections.append((l[i][3], l[i][-1]))
    return connections
```

2.1 Explanation

Creating the Directed Acyclic Graph.

[11]:
```
def topo(conn):
    g = nx.DiGraph()

    g.add_edges_from(conn)

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

2.2 Explanation

Sorting the nodes in topological order.

[12]:
```
def typ(nodes, l):
    dic = {}
    for i in range(len(nodes)):
        c = True
        for j in range(len(l)):
            if l[j][-1] == nodes[i]:
                dic[nodes[i]] = l[j][1]
                c = False
                break
        if c == True:
            dic[nodes[i]] = "PI"
    return dic
```

2.3 Explanation

To keep track of each type of gate, whether primary input or the logic gate it is associated with

2.4 Inputs

[13]:
```
def inputs(lis):
    ls = []
    for i in range(1, len(lis)):
        dic = {}
        for j in range(len(lis[0])):
            dic[lis[0][j]] = int(lis[i][j])
        ls.append(dic)
    return ls
```

2.5 Evaluation

[14]:
```
def outputs(top, l, ty, inpu):
    ans = {}
    for i in range(len(top)):
        if ty[top[i]] == "PI":
            ans[top[i]] = inpu[top[i]]
        elif ty[top[i]] == "and2":
            for j in range((len(l))):
                if l[j][-1] == top[i]:
                    ans[top[i]] = And(ans[l[j][2]], ans[l[j][3]])
                    break
        elif ty[top[i]] == "or2":
            for j in range((len(l))):
                if l[j][-1] == top[i]:
                    ans[top[i]] = Or(ans[l[j][2]], ans[l[j][3]])
                    break
        elif ty[top[i]] == "inv":
            for j in range((len(l))):
```

```
                if l[j][-1] == top[i]:
                    ans[top[i]] = Not(ans[l[j][2]])
                    break
        elif ty[top[i]] == "buf":
            for j in range((len(l))):
                if l[j][-1] == top[i]:
                    ans[top[i]] = ans[l[j][2]]
                    break
        elif ty[top[i]] == "nor2":
            for j in range((len(l))):
                if l[j][-1] == top[i]:
                    ans[top[i]] = Nor(ans[l[j][2]], ans[l[j][3]])
                    break
        elif ty[top[i]] == "nand2":
            for j in range((len(l))):
                if l[j][-1] == top[i]:
                    ans[top[i]] = Nand(ans[l[j][2]], ans[l[j][3]])
                    break
        elif ty[top[i]] == "xor2":
            for j in range((len(l))):
                if l[j][-1] == top[i]:
                    ans[top[i]] = Xor(ans[l[j][2]], ans[l[j][3]])
                    break
        elif ty[top[i]] == "xnor2":
            for j in range((len(l))):
                if l[j][-1] == top[i]:
                    ans[top[i]] = Xnor(ans[l[j][2]], ans[l[j][3]])
                    break
    return ans
```

2.6 Writing to a file

[15]:
```
def writetopo(file, l, inp):
    f = open(file, "w")
    alpha = sorted(outputs(topo(dag(l)), l, typ(topo(dag(l)), l),␣
 ↪inputs(inp)[0]))
    for i in alpha:
        f.write(i+" ")
    f.write("\n")
    for i in range(len(inputs(inp))):
        dic = outputs(topo(dag(l)), l, typ(topo(dag(l)), l), inputs(inp)[i])
        for j in alpha:
            f.write(str(dic[j])+" ")
        f.write("\n")
    f.close()
    return
```

[16]:
```
l = fr("c17.net")
```

[17]:
```
try:
    topo(dag(l))
except:
    print("The network is cyclic!!!")
```

[18]:
```
print("The topologically sorted list of nodes:", topo(dag(l)))
```

```
The topologically sorted list of nodes: ['N2', 'N7', 'N1', 'N3', 'N6', 'n_0',
'n_1', 'n_3', 'n_2', 'N22', 'N23']
```

2.7 Explanation

This is the topologically sorted list of nodes.

[19]:
```
inp = fr("c17.inputs")
```

[20]:
```
writetopo("c17topo.txt", l, inp)
```

[21]:
```
%timeit writetopo("c17topo.txt", l, inp)
```

```
1.52 ms ± 55.7 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

3 Event-Driven Evaluation

Note: When network and input changed, run all the subsequent cells again.

[42]:
```
# read the netlist
l = fr("c17.net")
```

[43]:
```
try:
    topo(dag(l))
except:
    print("Loser")
```

[44]:
```
# get the types of gates
types = typ(topo(dag(l)), l)

# list of all primary inputs
new_list = []
for i in types.keys():
    if types[i] == "PI":
        new_list.append(i)

#read the input file
inp = fr("c17.inputs") # change input file here
inpe = inputs(inp)
```

```
#initializing the output dictionary
diction = {}
for i in types.keys():
    diction[i] = 0

#list of successors
prop = {}

for i in types.keys():
    prop[i] = []
    for j in range(len(l)):
        if l[j][1] != 'inv' and l[j][1] != 'buf':
            if i == l[j][2] or i == l[j][3]:
                prop[i].append(l[j][4])
        else:
            if i == l[j][2]:
                prop[i].append(l[j][3])

# list of precursors
prec = {}

for i in types.keys():
    prec[i] = []
    if types[i] != "PI":
        for j in range(len(l)):
            if l[j][-1] == i:
                if l[j][1] != 'inv' and l[j][1] != 'buf':
                    prec[i].append(l[j][2])
                    prec[i].append(l[j][3])
                else:
                    prec[i].append(l[j][2])
```

3.1 Explanation

Reading inputs, initializing the output dictionary, keeping track of precursors and succssors.

3.2 Evaluation

[45]:
```
def event(types, inp1, queue):
    a_prev = "dum"

    while(len(queue) != 0):

        a = queue[0]
        queue.popleft()
        if a != a_prev:
```

```
            for i in prop[a]:
                queue.append(i)


            if types[a] == "nand2":
                diction[a] = Nand(diction[prec[a][0]], diction[prec[a][1]])
            elif types[a] == "nor2":
                diction[a] = Nor(diction[prec[a][0]], diction[prec[a][1]])
            elif types[a] == "and2":
                diction[a] = And(diction[prec[a][0]], diction[prec[a][1]])
            elif types[a] == "or2":
                diction[a] = Or(diction[prec[a][0]], diction[prec[a][1]])
            elif types[a] == "xor2":
                diction[a] = Xor(diction[prec[a][0]], diction[prec[a][1]])
            elif types[a] == "xnor2":
                diction[a] = Xnor(diction[prec[a][0]], diction[prec[a][1]])
            elif types[a] == "PI":
                diction[a] = inp1[a]
            elif types[a] == "inv":
                diction[a] = Not(diction[prec[a][0]])
            elif types[a] == "buf":
                diction[a] = diction[prec[a][0]]

        a_prev = a

    return
```

3.3 Writing to a file

[46]:
```
def writeevent(file, inpe):
    f = open(file, "w")
    queue = deque()
    for i in types.keys():
        if types[i] == "PI":
            queue.append(i)
    event(types, inpe[0], queue)
    alpha = sorted(diction.keys())
    for i in alpha:
        f.write(i+" ")
    f.write("\n")
    for j in alpha:
        f.write(str(diction[j])+" ")
    f.write("\n")

    for i in range(1, len(inpe)):
        for j in new_list:
            if diction[j] != inpe[i][j]:
```

```
                queue.append(j)
        event(types, inpe[i], queue)
        for j in alpha:
            f.write(str(diction[j])+" ")
        f.write("\n")
    f.close()
    return
```

[47]:
```
writeevent("c17event.txt", inpe)
```

[48]:
```
%timeit writeevent("c17event.txt", inpe)
```

```
361 µs ± 62.4 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

4 Time Analysis

The time taken by each network using different methods is given below.

Network	Topological	Event-Driven
c17	1.16 ms ± 32.8 µs	181 µs ± 13 µs
c8	8.3 ms ± 204 µs	511 µs ± 87.9 µs
parity	1.3 ms ± 59.5 µs	214 µs ± 2.84 µs
c432	398 ms ± 12.9 ms	14.3 ms ± 2.69 ms

In all the cases, we observe event-driven evalutaion to run **faster** than topologically ordered evalutaion.

As we are evaluating the netlists, in topological we pass all the nodes every time a new set of input is encountered. In contrast, event-driven evaluation only computes the nodes which are changed and the subsequent successors of the changed nodes. This way all the nodes need not be evaluated.

Topologically ordered evaluation is achieved through an in-built function in the library `networkx` called `topological_sort()`, which sorts all the nodes in the order you want them to be evaluated.

Event-driven evaluation is achieved by using queues, which are appended and popped at every evualtion step. All the nodes that are to be changed are initialized into the queue, and popped while being evaluated along with appending the queue with its corresponding successors.