Assignment4

Varsha S P, EE21B154 <ee21b154@smail.iitm.ac.in>

March 1, 2023

1 Assignment 4

1.1 Read the netlist and sort the nets in the toplogical order

To first read the netlist we define a function called `ckt` that opens the netlist and stores an array of gate ID, gate type, the two inputs and its corresponding node at output.

[53]:
```
import numpy as np
import sys
import cmath
import networkx as nx
from queue import Queue
import sys
import timeit
def ckt(file):
    try:
        with open(file,'r') as f:
            data=f.read().split("\n")
    except Exception as ex:
        print (ex)
        return
    gate_id=[]
    gate_type=[]
    input_1=[]
    input_2=[]
    output=[]
    for l in data:
        if l=='':
            continue
        i=l.split()
        gate_id.append(i[0])
        gate_type.append(i[1])
        input_1.append(i[2])
        if len(i)==4:
            input_2.append(None)
            output.append(i[3])
        else:
            input_2.append(i[3])
```

```
            output.append(i[4])

    return gate_id,gate_type,input_1,input_2,output
```

Once the netlist is read we try to print the nodes in topological order. This is done by using `nx.topological_sort` from Networkx. Let's see what it looks like for `c17` circuit

[54]:
```
def topologicalorder(file):
    gate_id,gate_type,input_1,input_2,output=ckt(file) #Reading the netlist

    M=np.column_stack([gate_id,gate_type,input_1,input_2,output])

    edges=[]
    node_attributes={}
    primary=[]

    for o in range(len(output)):
        if M[o][3]!=None:
            edges.append((M[o][2],M[o][4]))
            edges.append((M[o][3],M[o][4]))
        else:
            edges.append((M[o][2],M[o][4]))


    for j in range(len(output)):
        if M[j][2] not in output:
            primary.append(M[j][2])

        if M[j][3] not in output:
            primary.append(M[j][3])
    primary_input=[*set(primary)]
    for l in range(len(primary_input)):
        node_attributes.__setitem__(primary_input[l],'PI')
    for l in range(len(output)):
        node_attributes.__setitem__(output[l],gate_type[l])
    g = nx.DiGraph()
    g.add_edges_from(edges)
    nx.set_node_attributes(g,node_attributes,name="gateType")
    cycle=not nx.algorithms.dag.is_directed_acyclic_graph(g)
    if cycle:
        print('A cycle was detected')
        sys.exit()
    nodes=g.nodes(data=True)
    nl = list(nx.topological_sort(g))
    nlo = list(nx.lexicographical_topological_sort(g))
    return nl,nlo,primary_input,nodes,g
nl,nlo,primary_input,nodes,g=topologicalorder('c17.net')
```

```
print("The topological order of c17 netlist is :",nl)
```

```
The topological order of c17 netlist is : ['N2', 'N7', 'N1', 'N3', 'N6', 'n_0',
'n_1', 'n_3', 'n_2', 'N22', 'N23']
```

Now we try to define functions for all possible combinational gates used in these netlist.

[55]:
```
def AND(x,y):
    return x and y
def NAND(x,y):
    if x == 1 and y == 1:
        return 0
    else:
        return 1

def OR(x,y):
    return x or y
def XOR(x,y):
    if x != y:
        return 1
    else:
        return 0
def NOT(x):
    if x == 0:
        return 1
    else:
        return 0
def BUF(x):
    return x
def NOR(x,y):
    return NOT(OR(x,y))
def XNOR(x,y):
    return NOT(XOR(x,y))
```

We define another function called `inp_ut` which returns all the input values in the `.input` file.

[56]:
```
def inp_ut(file_inp):
    try:
        with open(file_inp,'r') as f:
            data=f.read().split("\n")
    except Exception as ex:
        print (ex)
        return


    return data
```

1.2 Read the list of input vectors, evaluate the circuit and find the state of all nets in the circuit.

1.2.1 Using Topoligical sort

Now we define a fuction called `topologicalsort()` to solve and create a `.txt` file with the state of each node for all the inputs in `.inputs` file. Here, we first try to map the nodes read from `.input` file with their node type and then we run a loop across each line of input and write each nodes' state to the `txt` file.

[59]:
```
def topologicalsort(file,file_inp,file_out):

    nl,nlo,primary_input,nodes,g=topologicalorder(file)


    data=inp_ut(file_inp)
    valin=[]
    for l in data:
            if l=='':
                continue
            i=l.split()
            valin.append(i)
    res = [i for i in nl if i not in primary_input]
    inpp=[i for i in nlo if i in primary_input]
    identifiers = inpp
    inp = sorted(identifiers)
    nodes_type={}
    for i in range(len(res)):
        for item in nodes:
            if res[i] in item:
                nodes_type.__setitem__(res[i],item[1]['gateType'])
    #print(nodes_type)
    biinp={}
    for l in range(1,len(valin)):
        inp_dict={}
        for k in range(len(inp)):
            inp_dict.__setitem__(valin[0][k],valin[l][k])
        for p in range(len(inp)):
            biinp.__setitem__(inpp[p],inp_dict.get(inp[p]))
        for i in range(len(res)):
            if nodes_type[res[i]]=='nand2':
                biinp.__setitem__(res[i],NAND(int(biinp[list(g.
↪predecessors(res[i]))[0]]),int(biinp[list(g.predecessors(res[i]))[1]])))

            if nodes_type[res[i]]=='and2':
                biinp.__setitem__(res[i],AND(int(biinp[list(g.
↪predecessors(res[i]))[0]]),int(biinp[list(g.predecessors(res[i]))[1]])))
```

```
            if nodes_type[res[i]]=='or2':
                biinp.__setitem__(res[i],OR(int(biinp[list(g.
 ↪predecessors(res[i]))[0]]),int(biinp[list(g.predecessors(res[i]))[1]])))

            if nodes_type[res[i]]=='nor2':
                biinp.__setitem__(res[i],NOR(int(biinp[list(g.
 ↪predecessors(res[i]))[0]]),int(biinp[list(g.predecessors(res[i]))[1]])))

            if nodes_type[res[i]]=='inv':
                biinp.__setitem__(res[i],NOT(int(biinp[list(g.
 ↪predecessors(res[i]))[0]])))

            if nodes_type[res[i]]=='buf':
                biinp.__setitem__(res[i],BUF(int(biinp[list(g.
 ↪predecessors(res[i]))[0]])))

            if nodes_type[res[i]]=='xor2':
                biinp.__setitem__(res[i],XOR(int(biinp[list(g.
 ↪predecessors(res[i]))[0]]),int(biinp[list(g.predecessors(res[i]))[1]])))

            if nodes_type[res[i]]=='xnor2':
                biinp.__setitem__(res[i],XNOR(int(biinp[list(g.
 ↪predecessors(res[i]))[0]]),int(biinp[list(g.predecessors(res[i]))[1]])))


        if l==1:
            with open(file_out, 'a') as f:
        # Write the headers to the file
                headers =sorted(list(biinp.keys()))
                f.write('\t'.join(headers) + '\n')
        headers =sorted(list(biinp.keys()))
        outsort={}
        for i in range(len(biinp)):
            outsort.__setitem__(headers[i],biinp.get(headers[i]))
        with open(file_out, 'r+') as f:
            contents = f.read()
            values=list(outsort.values())
            f.write('\t'.join(str(val) for val in values) + '\n')


topologicalsort('c17.net','c17.inputs','output.txt')

#%timeit topologicalsort('c17.net','c17.inputs','output.txt')
```

```
3.56 ms ± 81.9 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

1.2.2 Using Event driven

Here, we first define lists of all the nodes, gate type and dictionaries of gate type. And define a dictionary initially as `final_set` with all the nodes as keys amd random values say -1 as values. We run a for loop for every input from `.input` file and assign it's values to corresponding 'PI'. We define a function called `check()` which checks for the changed inputs in inp_dict. Initailly as all the inputs are diffrent from final_set, we add all the inputs to the Queue. And for each `.get()` we append its successors to the queue. Once all the primary inputs are omitted out of the queue, as the `g.predecessors` of levels above zero is not None, the nodes start performing their respective the gate operations. Once all the elements of the queue are out, the states of the nodes are added to `.txt` file. Again we check for changed keys and do the gate opperations to only it's successors.

[62]:
```
def check(inp_dict,final_set,Graph):
    changed_keys = []
    for key in inp_dict.keys():
        if inp_dict[key] != final_set[key]:
            tep=list(Graph.successors(key))
            for i in tep:
                pred=list(Graph.predecessors(i))
                if key in pred:
                    inp_keys=inp_dict.keys()
                    if all(elem in inp_keys for elem in pred):
                        for i in pred:
                            if i!= key and final_set[i]!=final_set[key]:
                                changed_keys.append(key)
                            elif i!=key and final_set[key]!=inp_dict[i]:
                                changed_keys.append(key)
                            elif i!=key and final_set[i]!=inp_dict[key]:
                                changed_keys.append(key)
                            else:
                                continue
                    else:
                        changed_keys.append(key)
    changed_keys=[*set(changed_keys)]

    return changed_keys
```

[64]:
```
def eventdriven(file,file_inp,file_out):
    gate_id,gate_type,input_1,input_2,output=ckt(file)
    M=np.column_stack([gate_id,gate_type,input_1,input_2,output])
    edges=[]
    node_attributes={}
    primary=[]
    valin=[]
    final_set={}
    inp_dict={}
```

```
    nodes_type={}
    q=Queue()
    for o in range(len(output)):
        if M[o][3]!=None:
            edges.append((M[o][2],M[o][4]))
            edges.append((M[o][3],M[o][4]))
        else:
            edges.append((M[o][2],M[o][4]))


    for j in range(len(output)):
        if M[j][2] not in output:
            primary.append(M[j][2])

        if M[j][3] not in output:
            primary.append(M[j][3])
    primary_input=[*set(primary)]
    for l in range(len(primary_input)):
        node_attributes.__setitem__(primary_input[l],'PI')
    for l in range(len(output)):
        node_attributes.__setitem__(output[l],gate_type[l])
    g1 = nx.DiGraph()
    g1.add_edges_from(edges)
    cycle=not nx.algorithms.dag.is_directed_acyclic_graph(g1)
    if cycle:
        print('A cycle was detected')
        sys.exit()
    nx.set_node_attributes(g1,node_attributes,name="gateType")
    nodes=g1.nodes(data=True)
    data=inp_ut(file_inp)
    for l in data:
            if l=='':
                continue
            i=l.split()
            valin.append(i)
    nlo=[*set(input_1+input_2+output)]
    res = [i for i in nlo if i not in primary_input]
    inpp=[i for i in nlo if i in primary_input]


    nlo=list(filter(lambda x: x is not None, nlo))
    identifiers= list(filter(lambda x: x is not None, inpp))
    inp = sorted(identifiers)

    for i in range(len(res)):
        for item in nodes:
            if res[i] in item:
```

```
                nodes_type.__setitem__(res[i],item[1]['gateType'])

    for i in nlo:
        final_set.__setitem__(i,-1)
    for l in range(1,len(valin)):
        inp_dict={}
        for k in range(len(inp)):
            inp_dict.__setitem__(valin[0][k],valin[l][k])

        changed_keys=check(inp_dict,final_set,g1)
        if len(changed_keys)>0:
            q=Queue()
            for pi_inputs in changed_keys:
                q.put(pi_inputs)

        while q.qsize()>0:
            temp=q.get()
            if  len(list(g1.predecessors(temp)))>0:
                pre_keys=final_set.keys()
                if all(elem in pre_keys for elem in list(g1.
↪predecessors(temp))):

                    for inputs in list(g1.successors(temp)):
                        q.put(inputs)
                    if nodes_type[temp]=='nand2':
                        final_set[temp]=NAND(int(final_set[list(g1.
↪predecessors(temp))[0]]),int(final_set[list(g1.predecessors(temp))[1]]))

                    if nodes_type[temp]=='and2':
                        final_set[temp]=AND(int(final_set[list(g1.
↪predecessors(temp))[0]]),int(final_set[list(g1.predecessors(temp))[1]]))

                    if nodes_type[temp]=='or2':
                        final_set[temp]=OR(int(final_set[list(g1.
↪predecessors(temp))[0]]),int(final_set[list(g1.predecessors(temp))[1]]))

                    if nodes_type[temp]=='nor2':
                        final_set[temp]=NOR(int(final_set[list(g1.
↪predecessors(temp))[0]]),int(final_set[list(g1.predecessors(temp))[1]]))

                    if nodes_type[temp]=='inv':
                        final_set[temp]=NOT(int(final_set[list(g1.
↪predecessors(temp))[0]]))

                    if nodes_type[temp]=='buf':
```

```
                        final_set[temp]=BUF(int(final_set[list(g1.
↪predecessors(temp))[0]]))

                    if nodes_type[temp]=='xor2':
                        final_set[temp]=XOR(int(final_set[list(g1.
↪predecessors(temp))[0]]),int(final_set[list(g1.predecessors(temp))[1]]))

                    if nodes_type[temp]=='xnor2':
                        final_set[temp]=XNOR(int(final_set[list(g1.
↪predecessors(temp))[0]]),int(final_set[list(g1.predecessors(temp))[1]]))
                else:

                    continue

            else:
                final_set.__setitem__(temp,inp_dict.get(temp))
                for inputs in list(g1.successors(temp)):
                    q.put(inputs)
        if l==1:
            with open(file_out, 'a') as f:
        # Write the headers to the file
                headers =sorted(list(final_set.keys()))
                f.write('\t'.join(headers) + '\n')
        headers =sorted(list(final_set.keys()))
        outsort={}
        for i in range(len(final_set)):
            outsort.__setitem__(headers[i],final_set.get(headers[i]))
        with open(file_out, 'r+') as f:
            contents = f.read()
            values=list(outsort.values())
            f.write('\t'.join(str(val) for val in values) + '\n')
    #This writes the new states of each node to the .txt file



eventdriven('c17.net','c17.inputs','output.txt')



#%timeit  eventdriven('c17.net','c17.inputs','output.txt')
```

```
1.23 ms ± 24.2 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

1.3 Briefly discuss your results: which approach is faster/more efficient, and for what types of inputs.

As we understand from both the methods, topological and event driven, we can come to a conclusion that event driven is supposed to more faster, efficient compared to topological. This is because, in

topological, we are using networkx to sort the nodes and running new set of inputs eventime we wanna know the change in state. However, in event driven we are applying gate operation only to the inputs with a change in their value. Therefore, we conclude that for any gven netlist, event driven methond is the most efficient and faster way to accquire state of each node.