EE2703 - Week 4

Santhosh S P ee21b119

March 1, 2023

1 Importing required modules

We import `networkx` to analyze graphs and `deque` from `collections` for the implementation of the queue data structure.

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

2 Defining the `makegraph()` function

We define a function `makegraph()` that takes in the name of the netlist file (along with the path). It parses through the file and collects:

- every pair of edge in a list `edge_pair_list`
- primary inputs in `primary_inputs`, outputs in `outputs`
- a dictionary `node_attribute_dict` stores the output nodes as key values and gate names as values

We use these parameters to create a DAG named `result_graph` and return it.

[ ]:
```
#function to make a graph by parsing through a netlist

def makegraph(netfile_path):

    #reading through the netlist file
    fh=open(netfile_path)
    data=fh.readlines()

    #some empty lists to store nodes and attributes
    edge_pair_list=[]
    node_attribute_dict={}
    primary_inputs=[]
    all_inputs=[]
    outputs=[]

    #each gate is written in this format [gatename, gatetype, inputs(s), output]
    for line in data:
        gate_inputs=line.split()[2:-1]
```

```
        gate_output=line.split()[-1]
        outputs.append(gate_output)

        for i in gate_inputs:
            all_inputs.append(i)

            #appending each pair of edges to this list
            edge_pair_list.append(tuple([i,gate_output]))

        #adding output nodes and gate names as key value pairs
        node_attribute_dict[gate_output]=line.split()[1]

    #finding out the primary inputs
    primary_inputs=list(set(all_inputs).difference(set(outputs)))
    all_inputs=list(set(all_inputs))

    #list of outputs
    outputs=list(set(outputs))

    #adding the primary nodes as keys to
    for j in primary_inputs:
        node_attribute_dict[j]="PI"

    #making a directional acyclic graph
    result_graph=nx.DiGraph()
    result_graph.add_edges_from(edge_pair_list)
    nx.set_node_attributes(result_graph,node_attribute_dict,name="gateType")

    #returning the DAG
    return result_graph
```

3 Collecting inputs from the input file: `collect_inputs()`

The cell below contains the `collect_inputs()` which takes the name of the `.inputs` as the argument, parses through the file and returns the data as a list of dictionaries, with the name of the primary input node as the key and the input as the value.

[ ]:
```
def collect_inputs(input_file):

    #storing the data of the file
    fh2=open(input_file)
    input_data=fh2.read().splitlines() #get rid of newlines
    fh2.close()

    current_inputs={}
    all_inputs2=[]
```

```
    primary_input_names=input_data[0].split()
    for j in primary_input_names:
        current_inputs[j]='x'

    for i in range(1,len(input_data)):
        for j in list(zip(primary_input_names, input_data[i].split())):
            current_inputs[j[0]]=j[1]
        all_inputs2.append(current_inputs.copy())

    #a list of dictionaries with key as primary node names and values of␣
↪primary inputs from file
    #is returned
    return all_inputs2
```

4 Simulating different gates

The cell below contains eight functions that implements the n-input version of AND, NAND, OR, NOR, XOR, XNOR and the single input gates NOT and BUF (buffer).

Each input/output is assumed to have three possible states 0, 1 and x (undefined).

[ ]:
```
#and gate
def and_result(*values):
    if 0 in values[0]:
        return 0
    elif all(elem==1 for elem in values[0]):
        return 1
    else:
        return 'x'

#or gate
def or_result(*values):
    if 1 in values[0]:
        return 1
    elif all(elem==0 for elem in values[0]):
        return 0
    else:
        return 'x'

#not gate
def not_result(*values):
    if 'x' in values[0]:
        return 'x'
    else:
```

```
        return int(not(values[0][0]))

#nand gate
def nand_result(*values):
    if 0 in values[0]:
        return 1
    elif all(elem==1 for elem in values[0]):
        return 0
    else:
        return 'x'

#nor gate
def nor_result(*values):
    if 1 in values[0]:
        return 0
    elif all(elem==0 for elem in values[0]):
        return 1
    else:
        return 'x'

#xor gate
def xor_result(*values):
    if 'x' in values[0]:
        return 'x'
    elif len([elem for elem in values[0] if elem ==1])%2:
        #checking if the number of 1's in the input is odd
        return 1
    else:
        return 0

#xnor gate
def xnor_result(*values):
    if 'x' in values[0]:
        return 'x'
    elif len([elem for elem in values[0] if elem ==1])%2:
        #checking if the number of 1's in the input is odd
        return 0
    else:
        return 1

#buffer
def buf_result(*values):
    return values[0][0]
```

5 Evaluating the circuit using topological evaluation

The below cell contains the function `topological_evaluation()` that takes the name of the netlist file and a dictionary containing primary inputs as arguments.

The function: * First calls `makegraph()` to constuct the DAG * required primary inputs are added as attributes named `value` of the nodes * the list of nodes are sorted topologically * for non-primary input nodes, the `gateType` parameter is checked, and using the required gate, the value is evaluated * The dictionary `result` containing the final values of the nodes is sorted and the resulting dictionary `sorted_result` is returned

[ ]:
```
#a sample input to test topological evaluation of parity.net
input_dict={'a':1,'b':1,'c':1,'d':1,'e':1,'f':1,'g':1,'h':0,'i':1,'j':0,'k':
↪1,'l':0,'m':1,'n':0,'o':0,'p':1}

#add a new attribute called 'value' that stores the values of the nodes (as␣
↪integers)
#adding the initial values
def topological_evaluation(netlist_file, primary_inputs_dict):

    #constructing the graph
    graph=makegraph(netlist_file)

    #inputs from the dictionary as added as the value of a new attribute␣
↪'value' of the nodes
    for nodename in list(primary_inputs_dict.keys()):
        graph.nodes[nodename]['value']=primary_inputs_dict[nodename]

    #nodes are sorted topologically
    nl = list(nx.topological_sort(graph))

    #print('Nodes in topological order',nl)
    for i in nl:

        #if the node is not a primary input, it's evaluated according to the␣
↪gate output
        if graph.nodes[i]['gateType']!='PI':

            predecessor_list=list(graph.predecessors(i))
            value_list=list(graph.nodes[pre_node_name]['value'] for␣
↪pre_node_name in predecessor_list)

            #and gate
            if graph.nodes[i]['gateType']=="AND2" or graph.
↪nodes[i]['gateType']=="and2":
                graph.nodes[i]['value']=and_result(value_list)

            #or gate
```

```
            if graph.nodes[i]['gateType']=="OR2" or graph.
↪nodes[i]['gateType']=="or2":
                graph.nodes[i]['value']=or_result(value_list)

            #nand gate
            if graph.nodes[i]['gateType']=="NAND2" or graph.
↪nodes[i]['gateType']=="nand2":
                graph.nodes[i]['value']=nand_result(value_list)

            #nor gate
            if graph.nodes[i]['gateType']=="NOR2" or graph.
↪nodes[i]['gateType']=="nor2":
                graph.nodes[i]['value']=nor_result(value_list)

            #not gate
            if graph.nodes[i]['gateType']=="NOT" or graph.
↪nodes[i]['gateType']=="not" or graph.nodes[i]['gateType']=="inv" or graph.
↪nodes[i]['gateType']=="INV":
                graph.nodes[i]['value']=not_result(value_list)

            #xor gate
            if graph.nodes[i]['gateType']=="XOR2" or graph.
↪nodes[i]['gateType']=="xor2":
                graph.nodes[i]['value']=xor_result(value_list)

            #xnor gate
            if graph.nodes[i]['gateType']=="XNOR2" or graph.
↪nodes[i]['gateType']=="xnor2":
                graph.nodes[i]['value']=xnor_result(value_list)

            #buffer gate
            if graph.nodes[i]['gateType']=="buf" or graph.
↪nodes[i]['gateType']=="BUF":
                graph.nodes[i]['value']=buf_result(value_list)


    result={}
    sorted_result={}

    for i in nl:
        result[i]=graph.nodes[i]['value']

    for key in sorted(result.keys()):
        sorted_result[key]=result[key]
    #the output dictionary is sorted alphabetically and returned
    return sorted_result
```

```
print(topological_evaluation(netlist_file="benchmarks/parity.net",␣
↪primary_inputs_dict=input_dict))
%timeit topological_evaluation(netlist_file="benchmarks/parity.net",␣
↪primary_inputs_dict=input_dict)
```

```
{'a': 1, 'b': 1, 'c': 1, 'd': 1, 'dummy_0': 1, 'dummy_1': 0, 'dummy_2': 1,
'dummy_3': 1, 'dummy_4': 0, 'e': 1, 'f': 1, 'g': 1, 'h': 0, 'i': 1, 'j': 0, 'k':
1, 'l': 0, 'm': 1, 'n': 0, 'n_0': 1, 'n_1': 1, 'n_2': 0, 'n_3': 1, 'n_4': 1,
'n_5': 0, 'n_6': 1, 'n_7': 0, 'n_8': 1, 'o': 0, 'p': 1, 'q': 1}
296 µs ± 9.11 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

6 Writing the output to a file

- The function `write_to_output()` takes the name of the netlist file and input file as arguments.
- A file of the same name as the netlist and input files is created with the extension `.out`.
- Using the overall result of the `collect_inputs()` function, we call the function `topological_evaluation()` on each element and write the result into the newly created file.

[ ]:
```
def write_to_output(netlist,input):
    #output file name
    filename=netlist[0:netlist.find(".")]+".out"

    fh3=open(filename,"w")
    for i in collect_inputs(input_file=input):
        #writing the result into the output file
        fh3.write(f"{topological_evaluation(netlist_file=netlist,␣
↪primary_inputs_dict=i)}")
        fh3.write("\n")
    fh3.close()

write_to_output(netlist="benchmarks/parity.net",input="benchmarks/parity.
↪inputs")
```

7 Performing event-driven analysis

- The below cell contains the function `event_driven_evaluation()` that takes the names of the netlist and input file as the arguments.
- The DAG is made using the `makegraph()` function.
- `state_table_old` and `state_table_new` are dictionaries that are used to track the change in states of the nodes.
- The queue `q` stores the next nodes to be evaluated, and it's constantly updated.

[ ]:
```python
#a sample input to test event-driven simulation of parity.net
input_dict={'a':1,'b':1,'c':1,'d':1,'e':1,'f':1,'g':1,'h':0,'i':1,'j':0,'k':
 ↪1,'l':0,'m':1,'n':0,'o':0,'p':1}
def event_driven_evaluation(netlist_file,primary_inputs_dict):

    #making the graph
    graph=makegraph(netlist_file)

    #state table is defined as a dictionary
    state_table_old={'t':0}

    #all inital values of the dictionary is 'x'
    for nodename in graph.nodes:
        state_table_old[nodename]='x'

    #a second state table to track the inital position and the position after␣
 ↪change
    state_table_new=state_table_old.copy()

    #the queue contains the primay input nodes initally
    q=deque(list(primary_inputs_dict.keys()))

    while len(q)>0:
        state_table_old=state_table_new.copy()
        num=0
        element=q[num]

        #adding the values of the primary inputs
        if graph.nodes[element]['gateType']=='PI':

            #adding the values of primary inputs
            state_table_new[element]=primary_inputs_dict[element]

            #the new version has one higher t value than the old state table
            state_table_new['t']=state_table_old['t']+1

            #if there is a change in the value, its successors are appended,␣
 ↪and the node is popped
            if state_table_new[element]!=state_table_old[element]:
                for i in list(graph.successors(element)):
                    q.append(i)
                q.popleft()

            #if there's no change in value the node is popped
            else:
                q.popleft()
```

```
        #for non-primary nodes, the value is evaluated based on the gate output␣
↪connected
        else:

            predecessor_list=list(graph.predecessors(element))
            value_list=list(state_table_new[pre_node_name] for pre_node_name in␣
↪predecessor_list)

            #and gate
            if graph.nodes[element]['gateType']=="AND2" or graph.
↪nodes[element]['gateType']=="and2":
                state_table_new[element]=and_result(value_list)

            #or gate
            if graph.nodes[element]['gateType']=="OR2" or graph.
↪nodes[element]['gateType']=="or2":
                state_table_new[element]=or_result(value_list)

            #nand gate
            if graph.nodes[element]['gateType']=="NAND2" or graph.
↪nodes[element]['gateType']=="nand2":
                state_table_new[element]=nand_result(value_list)

            #nor gate
            if graph.nodes[element]['gateType']=="NOR2" or graph.
↪nodes[element]['gateType']=="nor2":
                state_table_new[element]=nor_result(value_list)

            #not gate
            if graph.nodes[element]['gateType']=="NOT" or graph.
↪nodes[element]['gateType']=="not" or graph.nodes[i]['gateType']=="inv" or␣
↪graph.nodes[i]['gateType']=="INV":
                state_table_new[element]=not_result(value_list)

            #xor gate
            if graph.nodes[element]['gateType']=="XOR2" or graph.
↪nodes[element]['gateType']=="xor2":
                state_table_new[element]=xor_result(value_list)

            #xnor gate
            if graph.nodes[element]['gateType']=="XNOR2" or graph.
↪nodes[element]['gateType']=="xnor2":
                state_table_new[element]=xnor_result(value_list)

            #buffer gate
```

```
            if graph.nodes[element]['gateType']=="buf" or graph.
 ↪nodes[element]['gateType']=="BUF":
                state_table_new[element]=buf_result(value_list)

            state_table_new['t']=state_table_old['t']+1

            #if there is a change in the value, its successors are appended,␣
 ↪and the node is popped
            if state_table_new[element]!=state_table_old[element]:
                for i in list(graph.successors(element)):
                    q.append(i)
                q.popleft()

            #if there's no change in value the node is popped
            else:
                q.popleft()

    sorted_result={}
    for key in sorted(state_table_new.keys()):
        sorted_result[key]=state_table_new[key]

    #returning the sorted result
    return sorted_result

print(event_driven_evaluation(netlist_file="benchmarks/parity.net",␣
 ↪primary_inputs_dict=input_dict))
%timeit event_driven_evaluation(netlist_file="benchmarks/parity.net",␣
 ↪primary_inputs_dict=input_dict)
```

```
{'a': 1, 'b': 1, 'c': 1, 'd': 1, 'dummy_0': 1, 'dummy_1': 0, 'dummy_2': 1,
'dummy_3': 1, 'dummy_4': 0, 'e': 1, 'f': 1, 'g': 1, 'h': 0, 'i': 1, 'j': 0, 'k':
1, 'l': 0, 'm': 1, 'n': 0, 'n_0': 1, 'n_1': 1, 'n_2': 0, 'n_3': 1, 'n_4': 1,
'n_5': 0, 'n_6': 1, 'n_7': 0, 'n_8': 1, 'o': 0, 'p': 1, 'q': 1, 't': 46}
377 µs ± 24.8 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```