EE21B005 - Week 4 Assignment

Aditya Mallick EE21B005

March 1, 2023

1 Combinational Circuit Evaluation

[52]:
```
import networkx as nx
import os
from timeit import default_timer as timer
```

Importing required libraries, the **networkx** module has been imported to enable easy implementation and manipulation of graphs. OS is simply used for checking file path validity, and timeit is used for precise measurement of time taken within code blocks. Later on, we will also import some more modules.

1.1 Reading in Netlist

[53]:
```
def netread(f_net):

    accepted_gates = ["AND2", "OR2", "NOT", "NOR2", "NAND2", "XOR2", "XNOR2",␣
 ↪"INV", "BUF"]
    gates = {}
    edges_from = []
    nodes = {}


    for line in f_net:
        split = line.split()
        if split[1].upper() not in accepted_gates:
            continue
        # elif not split[0].isalnum():
        #     continue
        #elif not split[1].isalnum():
        #     continue
        #elif not split[2].isalnum():
        #     continue
        #elif not split[-1].isalnum():
        #     continue
        else:
            gates[split[0]] = split[1:]

    for gate in gates:
```

```
        if gates[gate][0].upper() == 'NOT' or gates[gate][0].upper() == 'INV'␣
↪or gates[gate][0].upper() == 'BUF':
            nodes[gates[gate][2]] = gates[gate][0].upper()
            edges_from.append((gates[gate][1],gates[gate][2]))
        else:
            edges_from.append((gates[gate][1],gates[gate][3]))
            edges_from.append((gates[gate][2],gates[gate][3]))
            nodes[gates[gate][3]] = gates[gate][0].upper()

    g = nx.DiGraph()

    # add nodes and edges to the DAG
    g.add_edges_from(edges_from)
    nx.set_node_attributes(g, nodes,name="gateType")

    try:
        n_ordered = list(nx.topological_sort(g))
    except nx.NetworkXUnfeasible:
        return -1

    return [g, n_ordered, nodes]
```

A base data extractor function that analyzes the netlist. * This function takes the open file variable as an input, reads the content and returns a dictionary of nodes, a list of topologically sorted nodes, and the directed acyclic graph created using the input parameters.

- These will act as inputs to our solver functions.
- We have implemented topological sort to find the hierarchical order of nodes.
- The nodes dictionary will have the name of the node as a key and the type of the gate of which it is an output as the corresponding value.
- We have kept in mind that only certain types of gates are to be accepted and we have checked for single/dual input gates as well.
- We have implemented error handling for exceptions such as a cyclic graph.

1.2 Topological sort + Ordered evaluation

[80]:
```
def getstate(primary, g, n_ordered, nodes):

    finalstates = {}
    for node in n_ordered:
        if node in primary:
            finalstates[node] = primary[node]
        else:
            prenodes = list(g.predecessors(node))
            if nodes[node] == 'NAND2':
```

```
                finalstates[node] = int(not(finalstates[prenodes[0]] &␣
↪finalstates[prenodes[1]]))
            elif nodes[node] == 'AND2':
                finalstates[node] = int(finalstates[prenodes[0]] &␣
↪finalstates[prenodes[1]])
            elif nodes[node] == 'NOR2':
                finalstates[node] = int(not(finalstates[prenodes[0]] |␣
↪finalstates[prenodes[1]]))
            elif nodes[node] == 'XOR2':
                finalstates[node] = int(finalstates[prenodes[0]] ^␣
↪finalstates[prenodes[1]])
            elif nodes[node] == 'XNOR2':
                finalstates[node] = int(not(finalstates[prenodes[0]] ^␣
↪finalstates[prenodes[1]]))
            elif nodes[node] == 'OR2':
                finalstates[node] = int(finalstates[prenodes[0]] |␣
↪finalstates[prenodes[1]])
            elif nodes[node] == 'NOT' or nodes[node] == 'INV':
                finalstates[node] = int(not(finalstates[prenodes[0]]))
            elif nodes[node] == 'BUF':
                finalstates[node] = int(finalstates[prenodes[0]])
    return finalstates
```

In the above function, we have used the topological order of nodes to solve the circuit. The order of steps is as follows:

- We iterate over each node starting from the lowest topological level to the highest.
- For each node, we evaluate its state by passing the states of its predecessor nodes through the required logic gate and finding the output.
- We then update our 'finalstates' dictionary, which stores the current state of all nodes.
- By using topological sort, we ensure that all inputs to a logic gate are evaluated before finding its output, so that no problems arise.
- After all nodes have been evaluated, we return the final state of the circuit using a dictionary.

NOTE: Regarding the states of the input nodes, we have found them separately using another function and passed them here as a parameter. This has been done to avoid taking the additional input file as a parameter for the base netlist reader. The 'nodes' dictionary is also first updated to include these input nodes, only after which it is passed as an argument to this function.

[65]:
```
def ordered_solve(netfile, inputfile):
    data = netread(netfile)
    if (data == -1):
        return -1
    else:
```

```
        graph = data[0]
        nodes = data[2]
        ordered = data[1]

    P_inputs = inputfile.readline().strip("\n").split()
    for ip in P_inputs:
        nodes[ip] = "PI"
    inputdict = dict.fromkeys(P_inputs)


    outputs=[]
    start = timer()
    for line in inputfile:
        line = line.split()
        for i in range(len(inputdict)):
            inputdict[list(inputdict.keys())[i]] = int(line[i])
        state = getstate(inputdict, graph, ordered, nodes)
        if (state == -1):
            return -1
        else:
            outputs.append(dict(state))
    end = timer()
    print(f"\n Time taken: {end - start:e} seconds")
    return outputs
```

This is the 'compiler' function. It calls the base netlist reading function to gather data about the circuit. It also extracts input data by reading the input file and creating an input state dictionary. Finally, all this data is passed to the circuit solver function and the final state of the circuit is stored. **We have also noted the time taken by the circuit solver, using the timer() function**. For each set of inputs, we get a corresponding dictionary of states. These are stored in the 'outputs' list. Finally, we return this list as the output.

To view the output, a separate function has been written later on.

1.3 Event-driven evaluation

[56]:
```
from collections import deque
```

For the event-driven evaluation, we will require a FIFO data structure (queue). For our case, a list will suffice, however, a better alternative is the **deque**, a double-ended queue that supports adding and removing elements from either end in O(1) time. In a list, removing or adding elements to the left end takes O(n) time, thus a deque is faster in these cases.

[57]:
```
def realtimestate(primary, g, nodes, state):

    q = deque()

    for node in primary:
        if primary[node] != state[node]:
```

```
            q.append(node)
    while (q):
        change = q.popleft()
        if change in primary:
            state[change] = primary[change]
            for node in list(g.successors(change)):
                q.append(node)
        else:
            temp = state[change]
            prenodes = list(g.predecessors(change))
            if len(prenodes) == 2:
                if state[prenodes[0]] != 'x' and state[prenodes[1]] != 'x':
                    if nodes[change] == 'NAND2':
                        state[change] = int(not(state[prenodes[0]] &␣
↪state[prenodes[1]]))
                    elif nodes[change] == 'AND2':
                        state[change] = int(state[prenodes[0]] &␣
↪state[prenodes[1]])
                    elif nodes[change] == 'NOR2':
                        state[change] = int(not(state[prenodes[0]] |␣
↪state[prenodes[1]]))
                    elif nodes[change] == 'XOR2':
                        state[change] = int(state[prenodes[0]] ^␣
↪state[prenodes[1]])
                    elif nodes[change] == 'XNOR2':
                        state[change] = int(not(state[prenodes[0]] ^␣
↪state[prenodes[1]]))
                    elif nodes[change] == 'OR2':
                        state[change] = int(state[prenodes[0]] |␣
↪state[prenodes[1]])
            else:
                if state[prenodes[0]] != 'x':
                    if nodes[change] == 'NOT' or nodes[change] == 'INV':
                        state[change] = int(not(state[prenodes[0]]))
                    elif nodes[change] == 'BUF':
                        state[change] = int(state[prenodes[0]])
            if state[change] != temp:
                for node in list(g.successors(change)):
                    q.append(node)

    return state
```

For the second part of the question, we have implemented an event-driven evaluation of the state of the circuit, directly using a queue to solve the circuit instead of using the topologically sorted list of nodes. Time is saved by only evaluating the parts of the circuit where inputs have been altered. The order of steps is as follows: * Like before, we use the directed acyclic graph to help us find the predecessors/successors of each node as well as tell us about the gates connected to each node.

- After creating our queue, we first check the inputs. If any node has been altered, it is added to the queue.
- Now, we will start popping items from the queue.
- Any node obtained from the queue will either have its value re-evaluated or preset depending on whether it is a primary input or not. The evaluation happens as follows:
  - If the value(s) of its predecessor(s) is/are well-defined (i.e not 'x'), then it will be re-evaluated. Else no change.
  - Re-evaluation is done in the same manner as before, by checking the type of logic gate and calculating the output.
  - After re-evaluation, if there is a change in the value of the node, update the state of the circuit and add its successor node(s) to the queue for re-evaluation as well. Otherwise, no change to the queue.
- This process is repeated until the queue is empty, after which we can say that the entire circuit has been evaluated.
- Finally, we return the final state of the circuit using a dictionary.

NOTE: Primary inputs have been handled in the same way as before. Apart from this, the current state of the circuit is also passed as an input to the function, since now we are only trying to evaluate any changes that have occurred.

[64]:
```
def queue_solver(netfile, inputfile):

    data = netread(netfile)
    if (data == -1):
        return -1
    else:
        graph = data[0]
        nodes = data[2]
        ordered = data[1]

    P_inputs = inputfile.readline().strip("\n").split()
    for ip in P_inputs:
        nodes[ip] = "PI"
    inputdict = dict.fromkeys(P_inputs)

    cur_state = dict.fromkeys(nodes)
    for node in cur_state:
        cur_state[node] = 'x'

    outputs = []
    start = timer()
    for line in inputfile:
        line = line.split()
        for i in range(len(inputdict)):
```

```
            inputdict[list(inputdict.keys())[i]] = int(line[i])
        cur_state = realtimestate(inputdict, graph, nodes, cur_state)
        outputs.append(dict(cur_state))
        if (cur_state == -1):
            return -1
    end = timer()
    print(f"\n Time taken: {end - start:e} seconds")


    return outputs
```

Again, we use a 'compiler' function. As usual, it gathers data about the circuit and its inputs. This data is passed to the circuit solver function and the final state of the circuit is stored. **Once again, we have noted the time taken by the circuit solver, using the timer() function**. The only difference now is that we are passing the stored state of the circuit back into the solver to evaluate for the next set of inputs. This process is repeated for each set of inputs, and all the outputs are stored in a list.

To view the output, a separate function has been written later on.

[72]:
```
import csv

def display(values):
    if values == -1:
        print("\nGraph contains a cycle, cannot be evaluated!")
        return -1
    else:
        with open("outputdata.csv", 'w', newline='') as file:
            # Writing net names in alphabetical order
            sortednodes = sorted(list(values[0].keys()))
            writer = csv.DictWriter(file, fieldnames=sortednodes)
            writer.writeheader()
            # Printing net names in alphabetical order
            print("\n")
            for node in sortednodes:
                print(f" {node} ", end="")
            print("\n")

            # Printing outputs
            for state in values:
                writer.writerow(state)
                for node in sortednodes:
                    print(f" {state[node]:^{len(node)}} ", end="")
                print("\n")
            return 0
```

A simple function that takes the list of dictionaries as input and prints the states of the circuit for

each set of input vectors. The nodes have been sorted in alphabetical order before being displayed.

Apart from this, the complete state table has also been written to a csv file "output-data.csv" to view the outputs in a more readable format.

[81]:
```
try:
    net_filename = input("Enter name of netlist file in benchmarks: ")
    inputs_filename = input("Enter name of input file in benchmarks: ")
    if (net_filename[-4:] != '.net' or inputs_filename[-7:] != '.inputs'):
        raise ValueError
    if not (os.path.isfile(f"benchmarks/{net_filename}") and os.path.
 ↪isfile(f"benchmarks/{inputs_filename}")):
        raise FileNotFoundError
except ValueError:
    print("\nPlease enter valid filenames!")
except FileNotFoundError:
    print('\nPlease use files present in the benchmarks folder!')
else:
    with open(f"benchmarks/{net_filename}", "r") as netfile, open(f"benchmarks/
 ↪{inputs_filename}", "r") as inputfile:
        display(ordered_solve(netfile, inputfile))
```

```
Enter name of netlist file in benchmarks:  c17.net
Enter name of input file in benchmarks:  c17.inputs
```

```
 Time taken: 1.179571e-04 seconds
```

N1	N2	N22	N23	N3	N6	N7	n_0	n_1	n_2	n_3
0	1	1	1	0	0	0	1	1	1	0
0	0	0	0	1	0	0	1	1	1	1
1	0	0	0	0	0	0	1	1	1	1
0	0	0	0	1	1	1	1	0	1	1
1	1	1	0	1	1	1	0	0	1	1
1	1	1	1	1	0	0	0	1	1	0
1	1	1	0	1	1	0	0	0	1	1
1	1	1	1	0	0	0	1	1	1	0
0	1	1	1	1	0	1	1	1	0	0

0	0	0	0	1	1	0	1	0	1	1

[82]:
```
try:
    net_filename = input("Enter name of netlist file in benchmarks: ")
    inputs_filename = input("Enter name of input file in benchmarks: ")
    if (net_filename[-4:] != '.net' or inputs_filename[-7:] != '.inputs'):
        raise ValueError
    if not (os.path.isfile(f"benchmarks/{net_filename}") and os.path.
 ↪isfile(f"benchmarks/{inputs_filename}")):
        raise FileNotFoundError
except ValueError:
    print("\nPlease enter valid filenames!")
except FileNotFoundError:
    print('\nPlease use files present in the benchmarks folder!')
else:
    with open(f"benchmarks/{net_filename}", "r") as netfile, open(f"benchmarks/
 ↪{inputs_filename}", "r") as inputfile:
        display(queue_solver(netfile, inputfile))
```

```
Enter name of netlist file in benchmarks:  c17.net
Enter name of input file in benchmarks:  c17.inputs
```

```
Time taken: 1.695440e-04 seconds
```

N1	N2	N22	N23	N3	N6	N7	n_0	n_1	n_2	n_3
0	1	1	1	0	0	0	1	1	1	0
0	0	0	0	1	0	0	1	1	1	1
1	0	0	0	0	0	0	1	1	1	1
0	0	0	0	1	1	1	1	0	1	1
1	1	1	0	1	1	1	0	0	1	1
1	1	1	1	1	0	0	0	1	1	0
1	1	1	0	1	1	0	0	0	1	1
1	1	1	1	0	0	0	1	1	1	0
0	1	1	1	1	0	1	1	1	0	0

```
0   0    0    0   1   1   0    1    0    1    1
```

The user is prompted for filename inputs. Validity of filenames are checked by ensuring they end in '.net' and '.inputs'. Then, if the files exist within the 'benchmarks' directory, they are opened and passed on for solving. If any of these conditions aren't passed, then we raise exceptions and exit with an error message.

1.4 Time Comparison

On running the two functions on the given datasets, we can make some clear observations:

- For smaller netlists, roughly the same time is taken for both the functions, and it is on the order of 0.10-0.20 ms. However the topological sort method is slightly faster.
- As the size of the netlists increase, the time difference is more noticeable. The queue solver starts taking almost double (sometimes even more) the amount of time taken by the topological solver. This is because even though the circuit is re-evauluated at each stage in the topological solver, the number of steps always stays the same, and it is equal to the number of nodes in the graph. However, for the queue solver the initial queue itself is very large and this takes up most of the time in the evaluation.
- Ultimately, the time taken depends on the size of the graph and the degree of variance of input vectors. If input vectors vary a lot, then the event-driven method offers no advantage, and is in fact slower. Only for very large circuits in which we change only a few outputs at each stage, the queue method will be faster. Otherwise it is better to stick with the topological sort method.