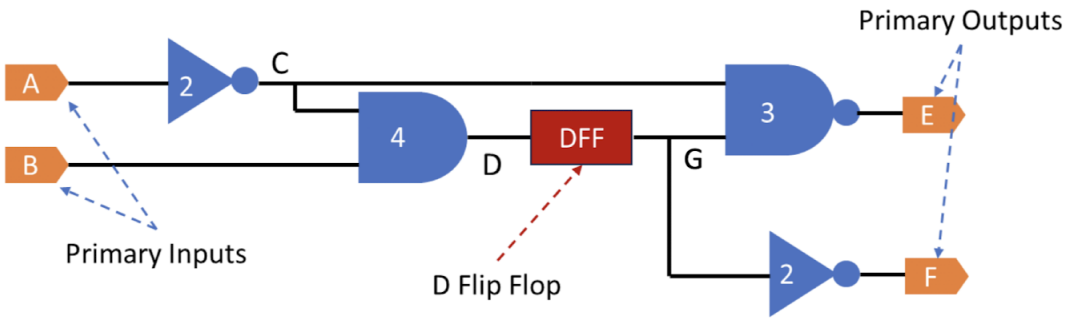
COL215P Software Assignment 2

Circuits with D Flip Flops
and
Area Optimisation

October 2023

Abhinav Rajesh Shripad (2022CS11596)
Aneeket Yadav (2022CS11116)



Contents

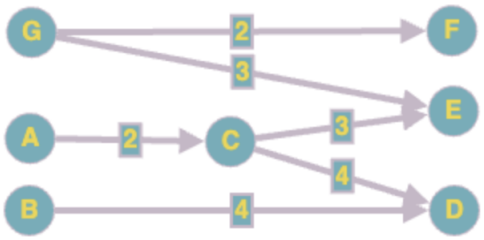
1 Circuits with D Flip Flops

1.1 Problem Description

In this assignment, you are tasked with extending your previous delay analysis program to handle circuits with D Flip Flops (DFFs). The DFFs introduce a new element into the analysis, where their inputs represent the end of one combinational path, and their outputs signify the start of a new combinational path. The goal is to determine the longest combinational delay in the given circuit. For the whole discussion, we denote n as the number of signals and m as the number of Gates.

1.2 Signal Representation

We represent and store each signal in a dictionary and store their delays in the same. For out algorithm we think of Signals as Vertex and Gates as Edges in our problem. The graph under consideration is a directed graph. Their is a directed edge from vertex u to v if there exists a Gate with u as one of the input and v as the output. The time complexity required to construct the graph is $\mathcal{O}(n+m)$ because we only need to read each Signal and Gate once. The circuit given as example in the Assignment description is represented below. Note that the DFFs outputs are considered as inputs only. **The basic idea is same as SW1 but here we "cut" the cycles in a circuit at DFFs.**



1.3 Auxiliary Functions

1.3.1 `read_circuit_file`

The function simply reads the Signals and the Gates and make a dictionary of predecessor and successor for each signal and set the default delay to each input and output DFFs signal to 0 and rest of the signal to the delay caused only by their corresponding gate only. The time complexity required for same is $\mathcal{O}(n+m)$.

1.3.2 `read_gate_delays`

Reads the smallest delay caused by gate of each type. The time complexity required for same is $\mathcal{O}(m)$

1.3.3 `Topological_Sort`

This gives a topologically sorted order of the gates. It uses a helper function `bfs` which does breadth for search at each of the input signals. The time complexity required for BFS is $\mathcal{O}(n+m)$

1.3.4 `maximum_delay`

It goes through the sorted signals and calculate the delay for the same. It also keeps the track of the maximum delay caused and returns the same. The time complexity required for BFS is $\mathcal{O}(m)$

1.4 Algorithm Description

The implementation is primarily composed of few functions which are as explained below:

1.5 `Topological_Sort`

This function sorts gates to align with their processing sequence. It addresses the potential disparity between input order and actual connections in `circuit.txt`. The output-linked gate, which might be first, is positioned appropriately. On each input signal the function BFS is performed. The description of BFS is as follows:-

1. The signal on which BFS is performed is inserted in the sorted list.
2. Each successor of the Signal, if absent is added to the end of the list or if present (if other input of the gate is processed earlier) is removed from the last location and added at the end of the list.
3. BFS is called on each of the successor of Signal.

1.6 `calculate_output_delay`

For each signal in the sorted list, the following operation is performed iteratively on each Signal. For each predecessor v of u, the delay of u is set to be

$$\texttt{delay}(u) = \texttt{max}(\texttt{delay}(u), \texttt{Signal}(u) + \texttt{delay}(v))$$

Remember that `Signal(u)` contains the delay caused by its corresponding gate. Thus the equation solves the problem. The time complexity for each graph G is going to be

$$T(G) = \sum_{u \in G} (\mathcal{O}(1 + \texttt{outdegree(u)}))$$

Thus using properties of the graph, we get

$$T(G) = O(n + m)$$

1.7 Time Complexity Analysis

The overall time complexity of the program for a graph G is

$$T(G) = \mathcal{O}(n + m) + \mathcal{O}(m) + \mathcal{O}(n + m) + \mathcal{O}(n) = \mathcal{O}(n + m)$$

Thus $T(G) = \mathcal{O}(n + m)$, in worst case scenario is when $m = n(n - 1)/2$ thus worst case complexity is $\mathcal{O}(n^2)$. For sparse graph, it is essentially linear and as the density increases it approaches quadratic in n.

2 Area Optimisation

2.1 Problem Description

Find the smallest digital circuit design that minimizes total gate area while respecting a maximum combinational path delay constraint. Gates have three implementation choices: fast, slow, and moderate, each with varying gate areas.

2.2 Signal Representation

We use the same representation as in part with subtle variation discussed below. It corresponds to which implementation of the gate of which it is output is considered. 0 being the fastest one and 2 representing the slowest one.

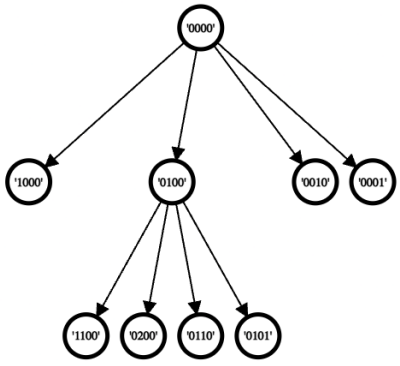
2.3 Auxiliary Functions

The functions `read_circuit_file`, `read_gate_delays and Topological_Sort` `does exactly the same work as in part 1.The "working function" of part 2 ie.calc_min_area calculates what the problem requires. It is described in Algorithm Description section.`

2.4 Algorithm Description

2.5 `calc_min_area`

Let n and m denote the number of signals and gates. Consider a mapping from gates to integers 0,1 or 2. Each gate is initialized to be 0. This mapping denotes the kind of implementation of the concerned gate, 0 being the fastest and so on. We now increase each gate number one by one and calculate the delay for the concerned. If the delay is withing bounds, we take the one with minimum area to be processed next. We repeat this on it again. Schematic diagram is shown below for a system with 4 gates.



We start from 0000 and process the "child" nodes. If say 0100 and 0010 satisfies the delay constraint then, we take the one with least area and repeat the same process with its children.

2.6 Time Complexity Analysis

For a particular mapping say 0000, to calculate the delay and area for this case requires $O(n)$ operations as we go from already sorted signals in the order. For each mapping we have m children and calculating and choosing the best if any required $O(mn + m)$ time complexity. After this we move 1 level down on the graph, ie exactly one of the mapping values increases by 1. In worst case situation we may need to increase all values by 2 (0–1–2), thus we need to go down $2m$ levels. Thus the worst case complexity is $O((2m)(mn + m))$ which is $O(nm^2)$

2.7 Proof of Correctness and Degree of Efficiency

We first discuss why the our algorithm decreases the area followed by why it might not always give the best solution.

We can consider our mapping as a point in m dimensions. The algorithm imitates Gradient Descent, where we change the co-ordinate in a dimension(thus corresponding mapping value) which minimizes the area. And thus repeat the process from this new location.

The reason why our algorithm is not full proof can be contributed to 2 reasons.

1. Our increments in the coordinate is "quantized" and fixed to one dimension at a time. Elaborating on the same, in gradient descent we can move our points by changing the co-ordinate in multiple dimensions depending on gradient. Here we are bound to change it in only 1. And the magnitude of change in gradient descent depends upon the Gradient of the function at the point of evaluation. In our algorithm the increment is always 1.

2. The gradient descent by its inherent nature finds the local extremum and not the global. This can also be seen in our program if for say global minima occurs at 0101 but on going down one level 0010 has the least area. Thus making us miss the global minima.

From the above 2 reason we can see why it is not guaranteed to achieve the best solution. But the trade-off between exponential time complexity to polynomial with accuracy is good because in real life we are dealing with circuits with number of gates in range of millions (Intel I7 chip has 1.4 billion), whereas brute-force algorithm takes ≥ 5 minutes to process even 20 gates circuit.

3 Testcases

We have tested our code on few of the testcases and measured the time for implementation which helped up in resolving any hitherto unforeseen problem in our program.

1. `Testcase_1:-`It has a cycle consisting of DFFs in the circuit with 20 signals. It help us check our code for part 1.
2. `Testcase_2:-`It has a sequence of 20 gates in series (output of one is input of next). It helped us check the running time of our circuit. For exponential implementation, it would take at-least 10 minutes.
3. `Testcase_3:-`The delay constraint is set very low that no implementation passes it.
4. `Testcase_4:-`The delay constraint is set very high that every implementation passes it.

Through the use of these test cases,we have verified the outcomes of the following processes and tabulated out findings as below-

Test-Case	Time taken Part A	Time taken Part B	Conclusion
Test-case 1	0.02ms	0.09 ms	Circuits with signals are computed correctly
Test-case 2	0.08ms	0.17ms	Time-Complexity is NOT exponential
Test-case 3	0.00ms (less than least count)	0.01ms	If no implementation gives correct output
Test-case 4	0.05ms	0.08ms	Time-taken to reach the worst case scenario is reasonably low

4 Submission Details

The submission consists of a `.zip` file which includes the testcase folders and `main.py` file. Each testcase folder contains the following items:

1. `main.py`
2. `Report`
3. `Testcase1`
4. `Testcase2`
5. `Testcase3`
6. `Testcase4`