Indian Institute of Technology Delhi

Department of Computer Science

COL-215
Software Assignment Report - 1

Dinu Goyal - 2022CS51647
Spandan Kukade - 2022CS51138

October 20, 2023

Contents

1 Introduction

In this project, we're going deeper into digital circuit analysis. We're adding a key component called D Flip Flops (DFF) to make things more complex. We'll be looking at circuits that have a sequence or order to their operations, focusing on how DFFs are involved. Then, we'll deal with the challenges of choosing the right gates and making design improvements to make sure everything works smoothly and quickly, meeting specific time constraints.

2 Problem Statement

This project looks into circuits that work step by step, focusing on how long it takes for them to give results once they have the necessary information. Each gate in these circuits has a mark indicating how long it takes to produce an output after receiving inputs. In the first part, we study this process using something called D Flip Flops. In the second part, we try to make these circuits work faster by designing the gates in the best way possible while also making sure they meet certain rules about how long they can take to process information. Finding a balance between making the circuits efficient and making sure they work well is really important for improving how we design digital systems.

3 Methodology

The program's approach comprises several vital steps.

- Input Processing:Input data is obtained from files, including the circuit structure (`circuit.txt`) , gate delay information (`gate_delays.txt`) , delay constraint file (`delay _constraint.txt`) , minimum area output(`minimum_area.txt`) and longest output timing delays (`longest_delays.txt`).

- Circuit Representation: We create a tree-like structure using adjacency lists, where each gate is represented as a node and edges denote interconnections.

- Calculating Delays We traverse the circuit graph bottom-up to determine the delay of primary outputs. We calculate the earliest time the output becomes ready at each gate node by considering the gate delays and input readiness times. For the converse problem, we assess the input readiness times needed to meet specific output timing requirements. This involves traversing the circuit graph top-down and considering the desired output timing constraints.

- Output Generation:The program generates output delay files (`longest_delays.txt`) and input delay files (`minimum_area.txt`), which contain computed delay and readiness time information, respectively.

4 Implementation

Similar to the previous project, for this project, we chose Python due to its simplicity and versatility in managing files and complex data structures. The implementation strategy is structured into key phases:

- Parsing and Data Storage: Input files are meticulously parsed, and the data is organized into dictionaries, optimizing data retrieval. This streamlined storage approach ensures quick access to circuit information.
- Constructing the Circuit: The circuit's architecture is constructed as a tree of interconnected gates, utilizing information extracted from the circuit representation file. This structured approach lays the foundation for subsequent traversal and calculation processes.
- Traversals and Calculations: The heart of the program lies in its traversal algorithms. These algorithms, designed with precision, are instrumental in computing output delays and input readiness times. Building upon the specifications provided, these traversals navigate the circuit, employing tailored formulas to calculate delays and readiness times accurately.
- Algorithm Selection: Depending on the size of the circuit (n), the program intelligently selects the algorithm to use. For circuits with $n < 11$, the 3^n algorithm is applied. For larger circuits, a dynamic programming approach is employed. This algorithm utilizes dynamic programming tables to calculate the minimum area while ensuring delays are less than the specified constraint. The formula

  $$dp(i, j) = \min(dp(i - 1, j - delay(i)) + area(i))$$

  is utilized, where i represents the gate index and j represents the delay constraint. Notably, the areas are discretized, enabling the application of dynamic programming to a problem that would otherwise be computationally infeasible. To prevent an exponential increase in the number of paths, a limit of 10^6 is set.
- Output Generation: The program generates output files containing computed delay and readiness time information, facilitating further analysis and decision-making.

Incorporating insights from previous project implementations, the Python implementation mirrors the C++ version's core logic. However, the flexibility of Python allows for intuitive adaptations and enhancements. The strategic choice of algorithms based on circuit size demonstrates a nuanced approach to balancing efficiency and accuracy.

5 Time Complexity Analysis

The algorithm's time complexity is a critical factor in assessing its efficiency. The time complexity of the implementation in this project can be broken down into several key stages:

- **Parsing and Data Storage:** Parsing input files and storing data in Python dictionaries involves iterating through the input data, resulting in a linear time complexity of $O(N)$, where N is the size of the input data.
- **Constructing the Circuit:** Building the circuit structure as a tree of interconnected gates from the circuit representation file involves iterating through the input data to establish gate connections. The time complexity for this step is $O(N)$, where N represents the number of gates in the circuit.
- **Algorithm Selection:** The selection of algorithms based on circuit size involves constant time operations ($O(1)$) for evaluating the circuit size and applying the appropriate algorithm (either a 3^n algorithm for smaller circuits or dynamic programming for larger circuits). The decision-making process has a time complexity of $O(1)$.
- **Dynamic Programming (DP) Approach:** In the dynamic programming step, the algorithm utilizes dynamic programming tables to calculate the minimum area while ensuring delays are less than the specified constraint. The time complexity for this step is $O(N^2 \times D)$, where N is the number of gates and D depends on the delay constraint. The discretization of areas contributes to the efficiency of the dynamic programming approach.
- **Output Generation:** Generating output files involves iterating through the computed delay and readiness time information to create the output files. The time complexity for this step is $O(N)$, where N represents the number of gates.

5.1 Code Snippets:

The overall time complexity of the implementation is dominated by the dynamic programming step, resulting in $O(N^2 \times D)$ complexity. However, the strategic selection of algorithms based on circuit size and the efficient use of dynamic programming contribute to the program's ability to handle circuits of various sizes and complexities. The implementation's time complexity ensures practical applicability in real-world digital circuit analysis tasks while optimizing both efficiency and accuracy.

6 Testing Strategy and Test Cases

To validate the correctness and effectiveness of the program in this extended project, a meticulous testing strategy was implemented, covering a wide array of scenarios and

```
backtrack = []
def process(path):
    global backtrack, a
    n = len(path)
    print("n is : ",n)
    # Initialize data structures for DP and backtracking
    dp = [[INF] * (a + 1) for _ in range(n + 1)]
    selected_values = [[-1] * (a + 1) for _ in range(n + 1)]

    dp[0][0] = 0
    print(path)
    for i in range(0, n + 1):
        for j in range(a + 1):
            continized_j = continize(j)
            if i == 0 :
                dp[i][j] = 0
            else :
                for k in range(3):
                    if continized_j - ndaa[path[i - 1]][k][0] >= 0:
                        temp = ndaa[path[i - 1]][k][1] + dp[i - 1][descretize(continized_j - ndaa[path[i - 1]][k][0])]
                        if temp < dp[i][j]:
                            dp[i][j] = temp
                            selected_values[i][j] = k
            print(dp[i][j]," ",end="")
        print("")
    print(selected_values)
    print("delay constraint is : ",(delayconstraint))
    print("Final value returned : ",dp[n][descretize(delayconstraint)])
    # Backtrack to find which values were assigned
    assigned_values = []
    j = descretize(delayconstraint)
    for i in range(n, 0, -1):
```

Figure 1: Code Snippet 1

```
for k in range(3):
    if continized_j - ndaa[path[i - 1]][k][0] >= 0:
        temp = ndaa[path[i - 1]][k][1] + dp[i - 1][descretize(continized_j - ndaa[path[i - 1]][k][0])]
        if temp < dp[i][j]:
            dp[i][j] = temp
            selected_values[i][j] = k
int(dp[i][j]," ",end="")
"")
```

Figure 2: Code Snippet 2

edge cases. The following test cases were meticulously designed and executed to assess the program's behaviour:

6.0.1 DFF in Loops:

Testing the program's ability to handle circuits with D Flip Flops (DFF) within loops, ensuring accurate computation of output delays in complex loop structures.

6.0.2 Multiple Loops:

Assessing the program's capability to navigate intricate circuits containing multiple loops, validating its accuracy in determining output latencies in such convoluted configurations.

6.0.3 Multiple Outputs:

Testing circuits with multiple outputs to evaluate the program's proficiency in computing distinct output delays simultaneously, accounting for diverse paths within the circuit.

6.0.4 High Gate Delays:

Evaluating the program's efficiency in managing circuits with gates featuring high propagation delays, ensuring accurate computation of output latencies despite prolonged gate response times

6.0.5 Low Gate Delays:

Examining the program's agility in circuits where gates exhibit minimal propagation delays, affirming its precision in determining swift output responses in low-delay scenarios.

7 Results and conclusions

In this project, our team successfully analyzed the complexities of sequential circuits, focusing on D Flip Flops (DFF) dynamics. We accurately determined computing latency for primary outputs post-input availability and optimized gate implementations to meet stringent path delay constraints.
Results:

1. **In-Depth Analysis of Sequential Circuits:** Our team conducted a meticulous analysis of sequential circuits, focusing on D Flip Flops (DFF). This examination provided valuable insights into computing latency for primary outputs after the availability of input values, deepening our understanding of sequential circuit behaviors.
2. **Optimized Gate Implementations:** In the second phase, we optimized gate implementations to meet stringent constraints on the longest combinational path delay. Through careful selection and refinement, we identified optimal gate configurations that ensured efficiency while adhering to the specified constraints.

.

Comprehensive testing established the program's correctness and reliability.