



**A PROJECT REPORT ON IMPLEMENTATION OF**

**CALCULATING THE MAXIMUM POSSIBLE  
FREQUENCY FOR THE GIVEN CIRCUIT USING  
STATIC TIMING ANALYSIS**

**FOR**

**SUBJECT: TESTING AND VERIFICATION OF DIGITAL CIRCUITS**

**ON**

**6<sup>th</sup> April 2025**

**Submitted By:**

**Harshvardhan Singh (22BEC120)**

**Srushti Patel (22BEC124)**

**Guided By:**

**Prof. Purvi Patel**

## TABLE OF CONTENTS

<b>Sr. No.</b>	<b>Title</b>	<b>Page No.</b>
1.	Introduction	3
2.	Proposed Methodology	4
	2.1 Flowchart	
3.	Algorithm	6
	3.1 Input Netlist	
	3.2 Results of netlist	
4.	Static Timing Analysis Using OpenSTA (Additional Work)	13
	4.1 Introduction	
	4.2 Methodology and Flowchart	
	4.3 Toolchain and Inputs	
	4.4 TCL commands executed	
	4.5 Results	
5.	Conclusion	18

## INTRODUCTION

In digital circuit design, achieving optimal performance requires precise analysis of timing behavior to ensure that data signals propagate correctly through the logic elements within a single clock cycle. Static Timing Analysis (STA) is a widely used method that evaluates circuit timing without requiring input stimulus, offering a fast and accurate way to validate timing constraints. This report presents the development of an algorithm designed to compute the maximum allowable clock frequency of a digital design through the implementation of a Static Timing Analysis script.

The algorithm was developed with the objective of automating the timing analysis process, identifying the critical path in a given circuit, and determining the minimum clock period — and thereby the maximum achievable clock frequency. The script takes into account gate delays, interconnect delays, and setup times, systematically evaluating all possible paths to find the longest propagation delay. This delay ultimately gives the timing constraint for reliable operation.

As a point of comparison and validation, **OpenSTA**, an open-source static timing analysis tool, was also utilized to determine the maximum clock frequency of the design. OpenSTA provides industry-standard timing analysis capabilities, allowing users to load netlists, timing libraries (Liberty format), and constraints (SDC format) to perform comprehensive timing checks. By running OpenSTA on the same circuit, the calculated maximum frequency served as a benchmark to validate the accuracy and reliability of the custom-developed algorithm. This cross-verification helps ensure that the script aligns with established STA methodologies used in real-world digital design flows.

# PROPOSED METHODOLOGY

## 1. Netlist Parsing

The algorithm begins by reading a netlist file that defines components (e.g., flip-flops and logic gates) along with their associated delays, setup, and hold times. Connections between components are also parsed to build a directed graph representing the circuit's topology. This is done by using a *parse\_netlist()* function.

## 2. Path Extraction (Flip-Flop to Flip-Flop)

To perform timing analysis, the algorithm identifies all flip-flop to flip-flop (FF-to-FF) paths. These represent the valid timing paths that must be checked for setup and hold violations. A breadth-first search (BFS) is used to traverse the graph and extract these paths using the *find\_ff\_to\_ff\_paths()* function.

## 3. Path Delay Calculation

For each FF-to-FF path, the total combinational delay is computed by summing the delays of intermediate logic components. The total arrival time is calculated by adding the clock-to-Q delay of the source flip-flop to the combinational delay.

## 4. Setup and Hold Timing Analysis

The *analyze\_setup\_hold()* function performs setup and hold checks for each path:

- **Setup check** ensures that data arrives at the destination flip-flop within the given time frame, accounting for the setup time and clock skew.
- **Hold check** ensures that the data remains stable long enough after the clock edge, taking into consideration the hold time and skew.

The path with the **maximum total delay** (arrival time + setup) is identified as the **critical path**.

## 5. Critical Path Reporting

Once the critical path is determined, it is printed along with detailed breakdowns including clock-to-Q delay, combinational delay, and setup time. This helps designers identify and optimize the slowest path in the design.

## 6. Maximum Clock Frequency Estimation

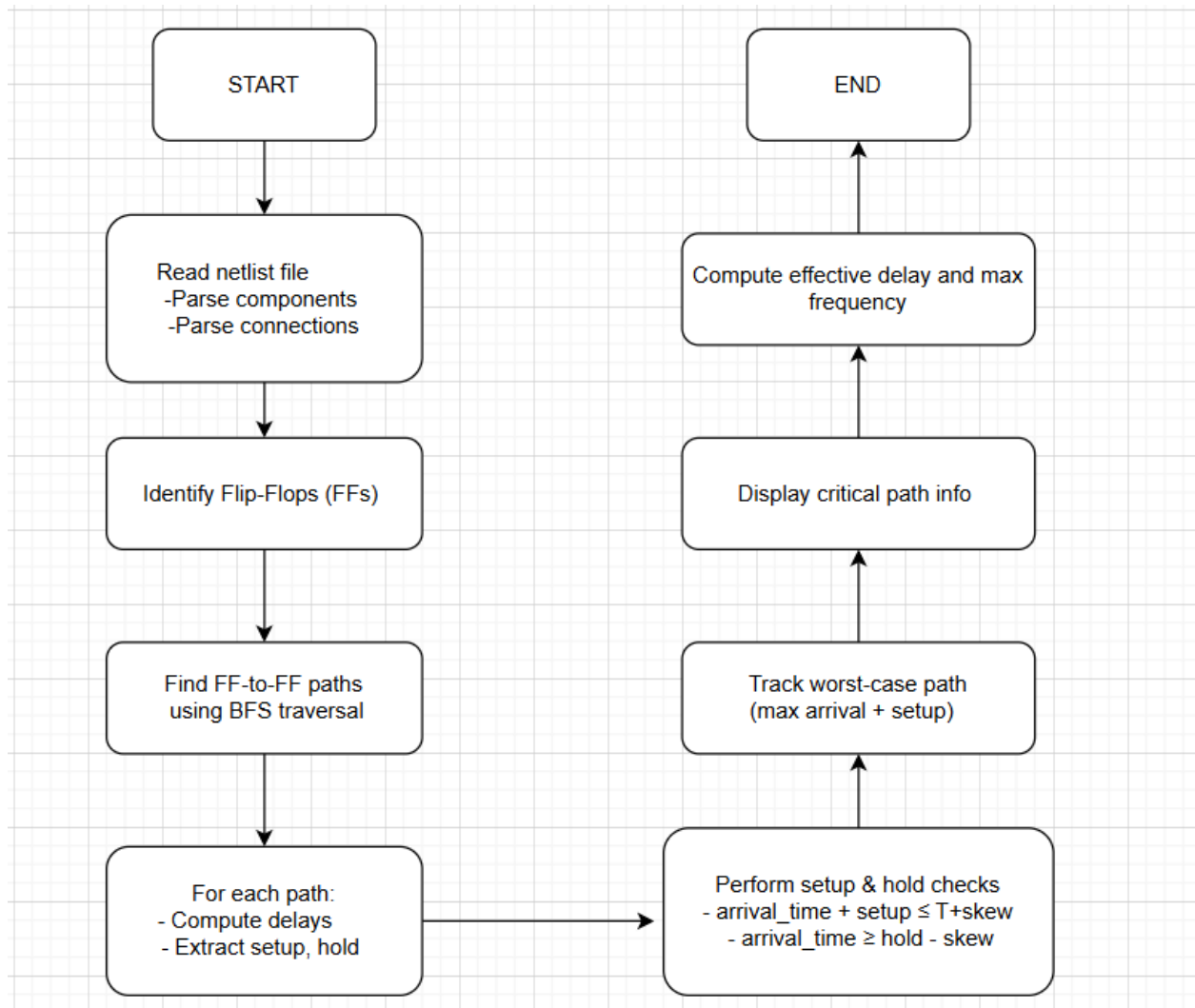
The effective delay of the critical path is computed by subtracting the clock skew. The **minimum clock period** is determined as:

$$T_{min} = (\text{Clock-to-Q} + \text{Combinational Delay} + \text{Setup Time} - \text{Skew})$$

And the corresponding **maximum clock frequency** is calculated as:

$$f_{\max} = 1 / T_{\min}$$

## FLOWCHART



## ALGORITHM

```
from collections import defaultdict, deque

# -----
# Parse netlist from file
# -----

def parse_netlist(filename):
    components = {}
    netlist = defaultdict(list)

    with open(filename, 'r') as f:
        for line in f:
            line = line.strip()
            if line.startswith("COMPONENT"):
                parts = line.split()
                name = parts[1]
                delay = float(parts[2])
                comp_type = "LOGIC"
                setup = hold = 0
                for part in parts[3:]:
                    if "TYPE=" in part:
                        comp_type = part.split('=')[1]
                    elif "SETUP=" in part:
                        setup = float(part.split('=')[1])
                    elif "HOLD=" in part:
                        hold = float(part.split('=')[1])
                components[name] = {
```

```

        'delay': delay,
        'type': comp_type,
        'setup': setup,
        'hold': hold
    }

    elif line.startswith("CONNECTION"):
        _, src, dst = line.split()
        netlist[src].append(dst)

    return components, netlist

# -----
# Find FF-to-FF paths
# -----

def find_ff_to_ff_paths(components, netlist):
    ff_nodes = [name for name, comp in components.items() if
comp['type'] == 'FF']

    ff_to_ff_paths = []

    for src_ff in ff_nodes:
        queue = deque([(src_ff, [src_ff])])
        while queue:
            current, path = queue.popleft()
            for neighbor in netlist.get(current, []):
                if components[neighbor]['type'] == 'LOGIC':
                    queue.append((neighbor, path + [neighbor]))
                elif components[neighbor]['type'] == 'FF' and
neighbor != src_ff:

```

```

        ff_to_ff_paths.append((src_ff, neighbor,
path + [neighbor]))

    return ff_to_ff_paths

# -----
# Compute delay of a path
# -----
def compute_path_delay(path, components):
    delay = 0
    for node in path[1:-1]: # exclude FFs at both ends
        delay += components[node]['delay']
    return delay

# -----
# Perform setup and hold analysis
# -----
def analyze_setup_hold(components, netlist, clock_period, skew):
    paths = find_ff_to_ff_paths(components, netlist)
    worst_arrival_plus_setup = 0
    critical_path_info = None

    for src, dst, path in paths:
        tclk_to_q = components[src]['delay']
        tcomb = compute_path_delay(path, components)
        tsetup = components[dst]['setup']
        thold = components[dst]['hold']

        arrival_time = tclk_to_q + tcomb

```



```

        # Setup and Hold Checks (not printed, just for
completeness)

        setup_ok = (arrival_time + tsetup) <= (clock_period +
skew)

        hold_ok = arrival_time >= (thold - skew)

        if arrival_time + tsetup > worst_arrival_plus_setup:
            worst_arrival_plus_setup = arrival_time + tsetup
            critical_path_info = {
                'path': path,
                'src': src,
                'dst': dst,
                'arrival_time': arrival_time,
                'tclk_to_q': tclk_to_q,
                'tcomb': tcomb,
                'tsetup': tsetup,
                'thold': thold
            }

        return worst_arrival_plus_setup, critical_path_info

# -----
# Main
# -----

filename = 'netlist2.txt' # Change to your file
clock_period = 10.0      # Clock period in ns
skew = 6.0               # Clock skew in ns

```

```

components, netlist = parse_netlist(filename)

worst_arrival_plus_setup, critical_path_info =
analyze_setup_hold(components, netlist, clock_period, skew)

# -----
# Highlight Critical Path Only
# -----

print("\n" + "="*40)
print("Critical Path (Maximum Delay)")
print("="*40)

if critical_path_info:
    print(f"Path: {' -> '.join(critical_path_info['path'])}")

    print(f"  Clock-to-Q (FF1):
{critical_path_info['tclk_to_q']} ns")

    print(f"  Combinational Delay: {critical_path_info['tcomb']}
ns")

    print(f"  Setup Time (FF2): {critical_path_info['tsetup']}
ns")

    print(f"  Total Delay (Clk-Q + Comb + Setup):
{critical_path_info['arrival_time'] +
critical_path_info['tsetup']:.2f} ns")

    print(f"  Skew: {skew} ns")
else:
    print("No valid FF-to-FF path found.")

# -----
# Max frequency calculation
# -----

effective_critical_path = worst_arrival_plus_setup - skew

```

```

if effective_critical_path <= 0:
    print("\nInvalid configuration: Effective path delay <= 0.
    Cannot compute frequency.")
else:
    max_freq = 1 / (effective_critical_path * 1e-9)  # ns to s
    print(f"\nCurrent Time period = {clock_period:.2f} ns")
    print(f"Minimum Time period required =
    {effective_critical_path:.2f} ns")
    print(f"Maximum Possible Frequency = {max_freq/1e6:.2f}
    MHz")

```

## Input Netlist

```

# Primary Input
COMPONENT PI 0.0 TYPE=INPUT

# Flip-Flops
COMPONENT DFF1 3 TYPE=FF SETUP=2 HOLD=3
COMPONENT DFF2 4 TYPE=FF SETUP=4 HOLD=5

# Logic Gates
COMPONENT AND1 2 TYPE=LOGIC
COMPONENT OR1 3 TYPE=LOGIC
COMPONENT AND2 13 TYPE=LOGIC
COMPONENT AND3 2 TYPE=LOGIC

# Netlist Connections
CONNECTION PI DFF1

```

```
CONNECTION DFF1 AND1
CONNECTION AND1 OR1
CONNECTION AND1 AND2
CONNECTION OR1 AND3
CONNECTION AND2 AND3
CONNECTION AND3 DFF2
```

### **Result For Above Given Netlist:**

```
=====
Critical Path (Maximum Delay)
=====
Path: DFF1 -> AND1 -> AND2 -> AND3 -> DFF2
  Clock-to-Q (FF1): 3.0 ns
  Combinational Delay: 17.0 ns
  Setup Time (FF2): 4.0 ns
  Total Delay (Clk-Q + Comb + Setup): 24.00 ns
  Skew: 6.0 ns

Current Time period = 10.00 ns
Minimum Time period required = 18.00 ns
Maximum Possible Frequency = 55.56 MHz
```

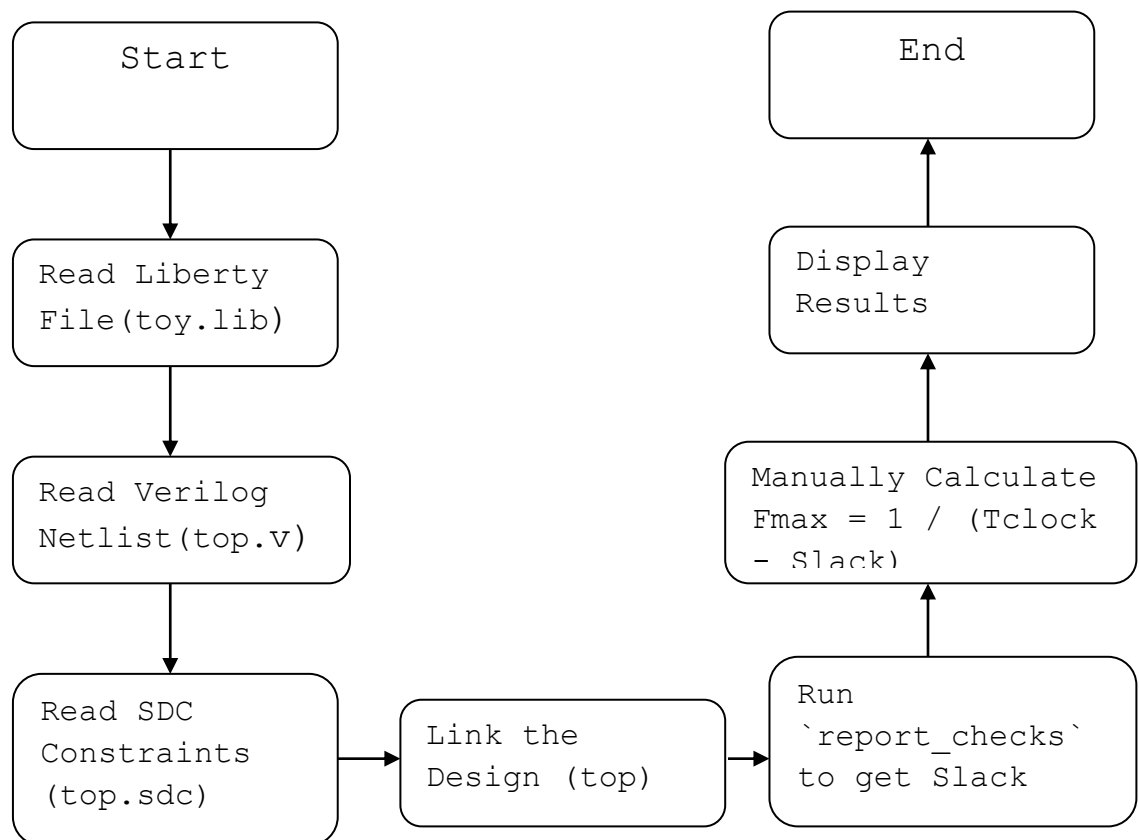
# STATIC TIMING ANALYSIS USING OPENSTA (ADDITIONAL WORK)

## Introduction

To supplement the core algorithmic work developed in Python, we extended our analysis by performing **Static Timing Analysis (STA)** using the **OpenSTA tool**. The purpose of this step was to estimate the **maximum operating frequency** of a digital circuit implementation by analyzing the timing delays and constraints in a synthesized design. This step bridges our algorithm with practical hardware realizability and timing closure.

OpenSTA is an open-source, industry-grade STA tool that enables designers to analyze the setup and hold timing paths in a digital design by interpreting netlist files, standard cell libraries (.lib), and constraint files (.sdc). Our objective was to use OpenSTA to identify the **worst-case slack**, from which we can calculate the **maximum achievable clock frequency** for our design.

## Methodology & Flowchart



## Toolchain & Inputs

- **Tool Used:** OpenSTA (v2.6.2)
- **Input Files:**
  - toy.lib: Standard cell library in Liberty format
  - top.v: Synthesized Verilog netlist of the circuit

```
module top(a, b, clk, reset, out);  
  
    input a, b, clk, reset;  
  
    output out;  
  
    wire y;  
  
    wire y1;  
  
    wire y2;  
  
    INV I1 ( .I(b), .ZN(y1));  
  
    DFFRNQ F1 ( .CLK(clk), .D(y1), .Q(y2), .RN(reset));  
  
    INV I2 ( .I(y2), .ZN(y3));  
  
    BUF B1 ( .I(y3), .Z(y4));  
  
    NAND2 N1( .A1(a), .A2(y4), .ZN(y5));  
  
    INV I3 ( .I(y5), .ZN(y6));  
  
    DFFRNQ F2 ( .CLK(clk), .D(y6), .Q(y7), .RN(reset));  
  
    DFFRNQ F3 ( .CLK(clk), .D(y7), .Q(y8), .RN(reset));  
  
    BUF B2 ( .I(y8), .Z(out));  
  
endmodule
```

- top.sdc: Constraints file specifying clock information

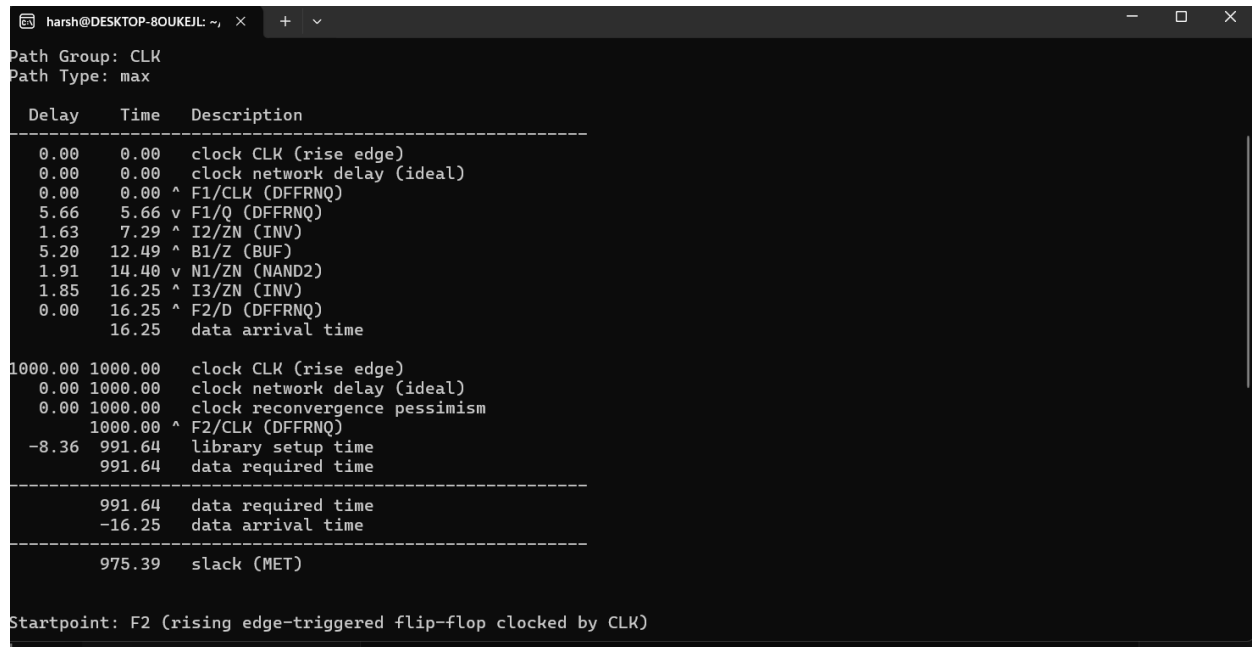
```
create_clock -name CLK -period 1000 [get_ports clk]
set_input_delay 5 -clock CLK [get_ports a]
set_input_delay 5 -clock CLK [get_ports b]
set_output_delay 5 -clock CLK [get_ports out]
```

### **TCL Commands Executed:**

```
read_liberty toy.lib
read_verilog top.v
link_design top
read_sdc top.sdc
report_checks -path_delay max -format full
```

## Results

From the `report_checks` command, we obtained the **worst-case setup slack** for the circuit. Using this value and the defined clock period in `top.sdc`, we manually computed the maximum operating frequency.



```
harsh@DESKTOP-80UKEJL: ~ \x + \v
Path Group: CLK
Path Type: max

-----
Delay    Time    Description
-----
0.00     0.00    clock CLK (rise edge)
0.00     0.00    clock network delay (ideal)
0.00     0.00    ^ F1/CLK (DFFRNQ)
5.66     5.66    v F1/Q (DFFRNQ)
1.63     7.29    ^ I2/ZN (INV)
5.20     12.49   ^ B1/Z (BUF)
1.91     14.40   v N1/ZN (NAND2)
1.85     16.25   ^ I3/ZN (INV)
0.00     16.25   ^ F2/D (DFFRNQ)
          16.25    data arrival time

1000.00  1000.00  clock CLK (rise edge)
0.00     1000.00  clock network delay (ideal)
0.00     1000.00  clock reconvergence pessimism
          1000.00  ^ F2/CLK (DFFRNQ)
-8.36    991.64  library setup time
          991.64  data required time
-----
          991.64  data required time
          -16.25  data arrival time
-----
          975.39  slack (MET)

Startpoint: F2 (rising edge-triggered flip-flop clocked by CLK)
```

### OpenSTA Slack Report

**Note:** We used a standalone TCL script (`calc_freq.tcl`) to take the clock period and slack as inputs and compute the maximum frequency in MHz.

```
# calc_freq.tcl

# Standalone script to calculate maximum frequency

puts "Enter Clock Period (Tclk) in ns:"

gets stdin Tclk
```



```

puts "Enter Worst Slack in ns:"
gets stdin Slack

# Convert to floating-point
set Tclk [expr {double($Tclk)}]
set Slack [expr {double($Slack)}]

# Calculate required period
set Trequired [expr {$Tclk - $Slack}]

if { $Trequired <= 0 } {
    puts "Error: Required clock period is non-positive. Check
your inputs."
    exit
}

# Calculate Fmax in MHz
set Fmax [expr {1000.0 / $Trequired}]

puts "\n----- Maximum Frequency Report -----"
puts " Given Clock Period      = $Tclk ns"
puts " Worst Setup Slack       = $Slack ns"
puts " Required Period          = $Trequired ns"
puts " Maximum Frequency        = $Fmax MHz"
puts "-----"

```

```

% exit
harsh@DESKTOP-80UKEJL:~/sta_project/project$ tclsh max_freq.tcl
Enter Clock Period (Tclk) in ns:
1000
Enter Worst Slack in ns:
975.39

----- Maximum Frequency Report -----
Given Clock Period    = 1000.0 ns
Worst Setup Slack     = 975.39 ns
Required Period       = 24.610000000000014 ns
Maximum Frequency     = 40.6338866314504 MHz
-----

```

### Frequency Calculation using tcl script

## CONCLUSION

This project explored two independent approaches to understanding and analyzing the timing behavior of digital circuits: a **custom algorithmic method using Python script**, and a **tool-based approach using OpenSTA**..

### Algorithm Development (Python script)

We developed a Python-based script to perform **Static Timing Analysis (STA)** on a digital circuit represented by a netlist. The algorithm computes the arrival times, required times, and slack values for each gate and pin in the design. This custom STA tool provides a foundational understanding of how timing paths are evaluated and how critical paths are determined. It also allows for extensibility into more complex features like clock skew, setup/hold checks, and path-based optimization in future iterations.

### Tool-Based Analysis (OpenSTA)

As an extension, we used the **OpenSTA tool** to perform STA on the same design, using standard input formats (.lib, .v, .sdc). By analyzing the worst-case slack reported by OpenSTA, we calculated the **maximum possible frequency** the circuit can run at. This step validated our algorithmic results with an industry-standard tool and provided real-world relevance to our design.