FPGA Maze Router - Input Pin Commutativity and its Impact on Routability

ECE1387 : CAD for Digital Circuit Synthesis and Layout

By

Aditya Srichandan

1009210113

Department of Electrical & Computer Engineering

October, 2024

OVERVIEW

This assignment 1 implementation involved the FPGA maze routing algorithm, aiming to test and optimize the routing of test circuits across different configurations of FPGA architectures with varying channel tracks, switch blocks, and logic blocks (CLBs). The goal was to investigate two routing styles: baseline (non-swappable pins) swappable inputs and assess the impact of pin commutativity on the overall routability of the design.

The FPGA routing architecture was designed to follow the pin numbering and connectivity constraints specified in the assignment description, with each logic blocks having pin connection (North, East, South West) directions and odd number sink pins connected to even track ids while even numbered sink pins to odd track ids. The routing logic allowed routing connections between source and sink pins through planar switch blocks, with the challenge being to find the minimum number of tracks per channel (W) that could successfully route all test circuits. Two styles of routing were explored: a baseline approach that rigidly followed the input pin assignments and a swappable inputs approach that provided flexibility by allowing connections to any available input pin on the destination block.

The routing algorithm was implemented using a combination of the Lee-Moore maze routing technique, with visual feedback provided through EZGL graphics. This allowed real-time display of routing progress and final routing results for debugging and visualization. The router was tested on several circuits, varying the number of routing tracks and recording the number of wire segments used. Additionally, the program supported automatic testing of multiple circuits and performed binary search optimizations to find the minimum viable track count (W) required to route each test case.

MAZE ROUTER GUI

Maze router is implemented with swappable and non-swappable pins on load as options. Below is paper plot for cct1,cct2,cct3 and cct4 test circuits for swappable and non-swappable settings. Source (Load) CLB is represented by Red color, Destination (sink) CLB is colored by green and CYAN is used for unused logic blocks.

CCT1

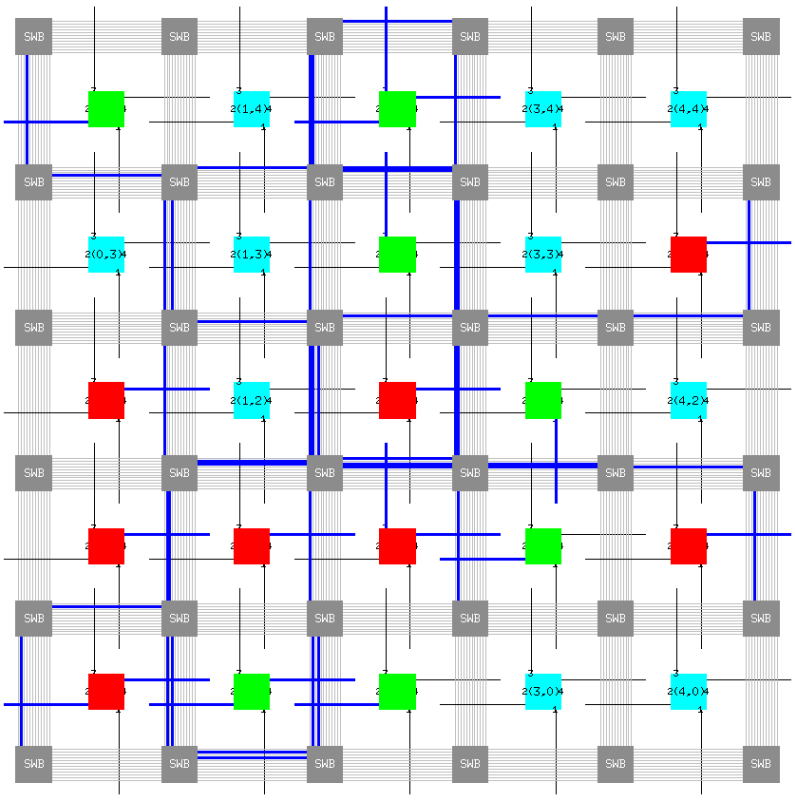

Figure 1: Non swappable 5x5 grid and 12 track per channel

```
FPGA Gridsize:5
FPGA Number of Tracks in each channel:12
Total Number of Test Cases:10
Routing Complete, Total Wiresegments for test case 0 : 5
Routing Complete, Total Wiresegments for test case 1 : 4
Routing Complete, Total Wiresegments for test case 2 : 3
Routing Complete, Total Wiresegments for test case 3 : 9
Routing Complete, Total Wiresegments for test case 4 : 4
Routing Complete, Total Wiresegments for test case 5 : 3
Routing Complete, Total Wiresegments for test case 6 : 4
Routing Complete, Total Wiresegments for test case 7 : 4
Routing Complete, Total Wiresegments for test case 8 : 9
Routing Complete, Total Wiresegments for test case 9 : 1
Total number of routing segments used: 46
Test Case : 1
Test Case : 2
Test Case : 3
Test Case : 4
Test Case : 5
Test Case : 6
Test Case : 7
```

Figure 2: console output for non swappable settings with 46 used routing segments

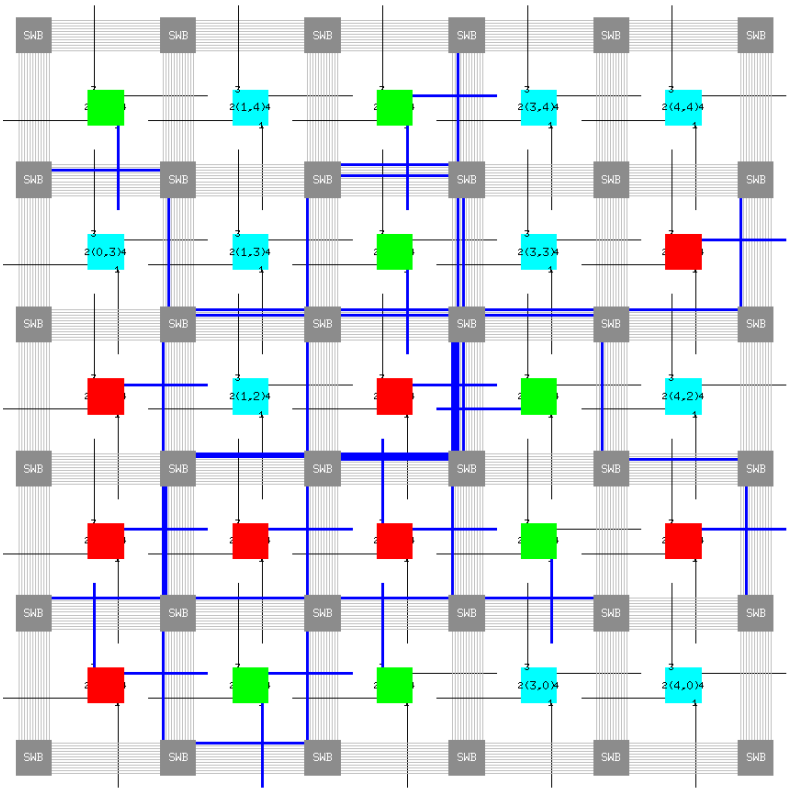

Figure 3: swappable input pins 5x5 grid and 12 track per channel

```
FPGA Gridsize:5
FPGA Number of Tracks in each channel:12
Total Number of Test Cases:10
Routing Complete, Total Wiresegments for test case 0 : 4
Routing Complete, Total Wiresegments for test case 1 : 3
Routing Complete, Total Wiresegments for test case 2 : 2
Routing Complete, Total Wiresegments for test case 3 : 8
Routing Complete, Total Wiresegments for test case 4 : 4
Routing Complete, Total Wiresegments for test case 5 : 3
Routing Complete, Total Wiresegments for test case 6 : 4
Routing Complete, Total Wiresegments for test case 7 : 3
Routing Complete, Total Wiresegments for test case 8 : 8
Routing Complete, Total Wiresegments for test case 9 : 2
Total number of routing segments used: 41
Test Case : 1
Test Case : 2
```

Figure 4: swappable input pins settings with 41 used routing segments

CCT2

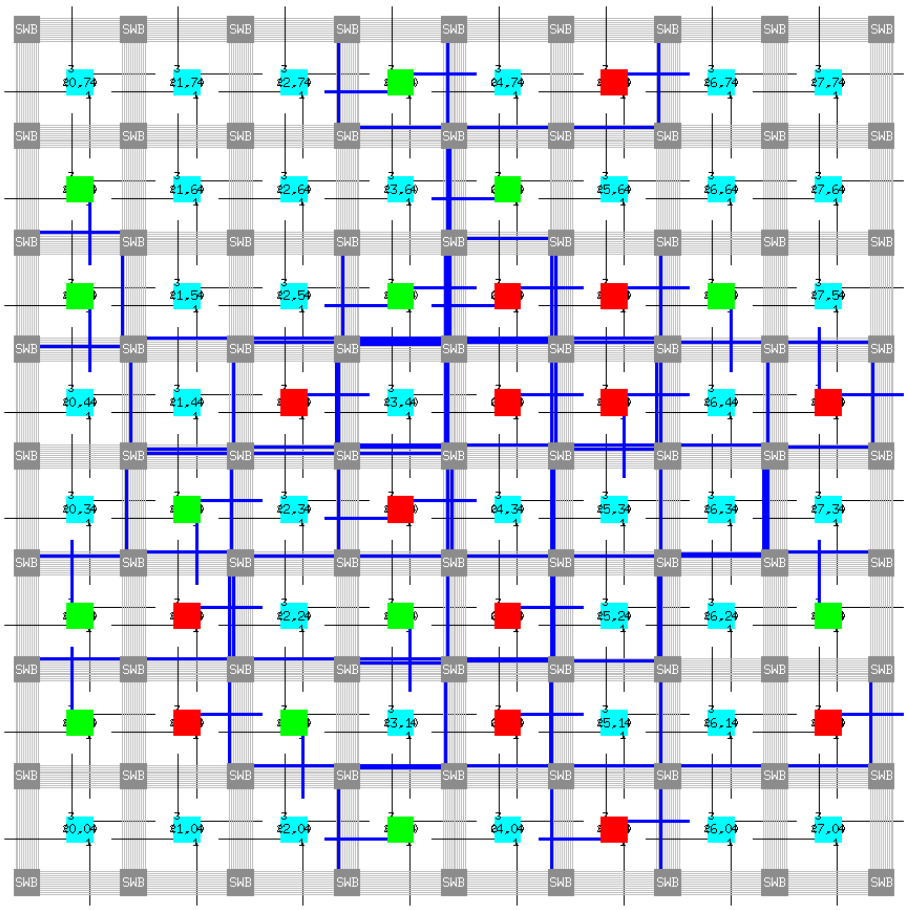

Figure 5: Non swappable 8x8 grid and 12 track per channel

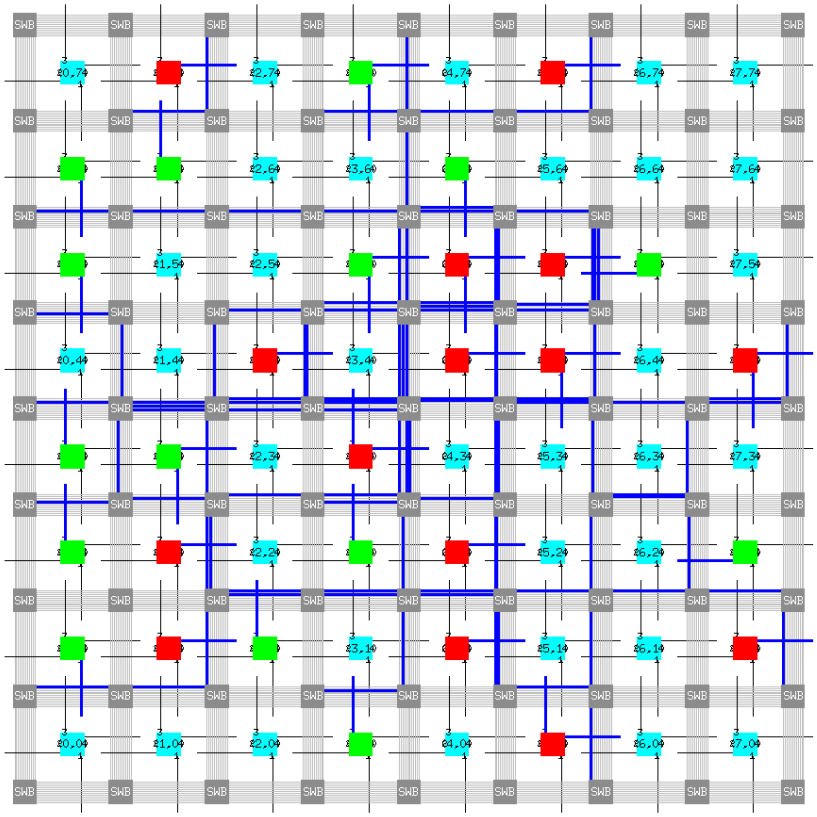

Figure 6: swappable input pins 8x8 grid and 12 track per channel

CCT3

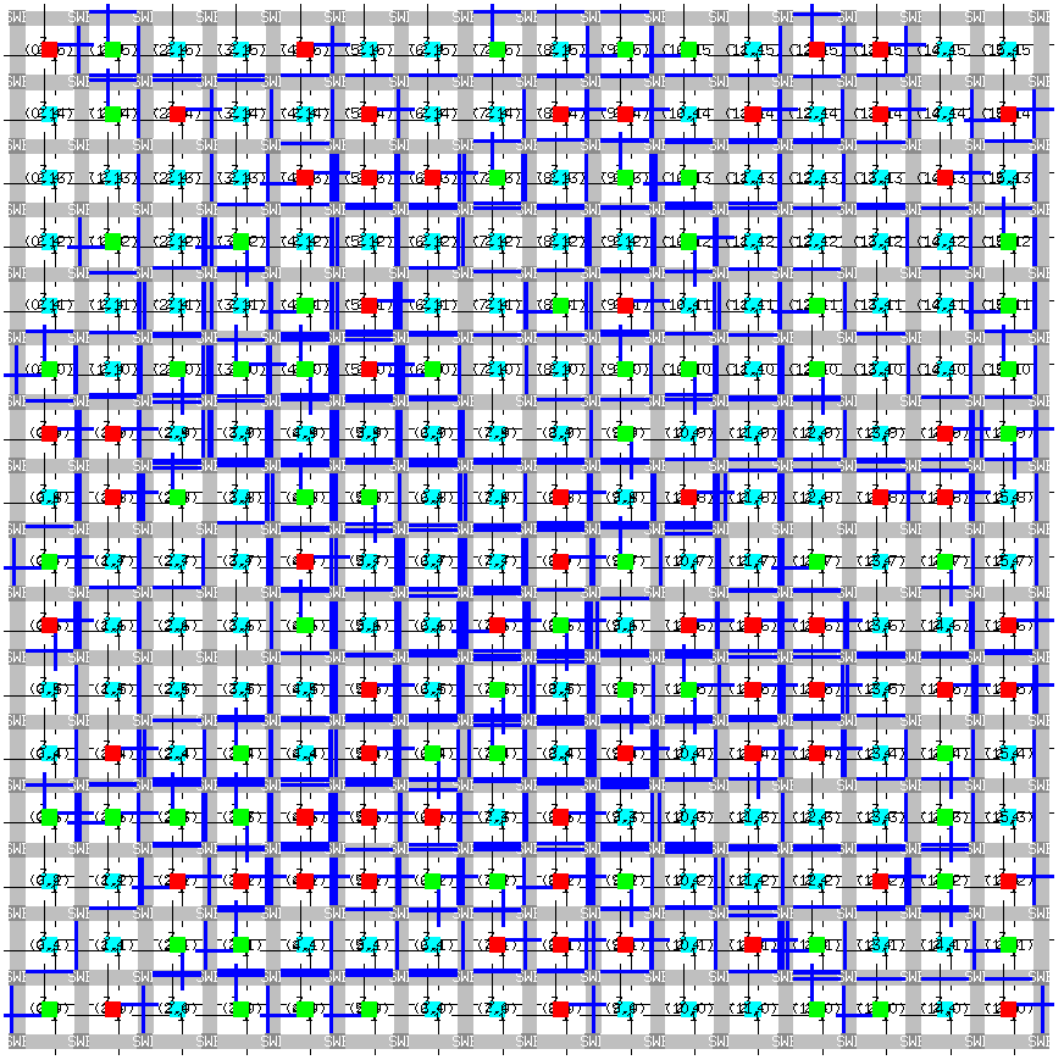

Figure 7: Non swappable 16x16 grid and 14 track per channel

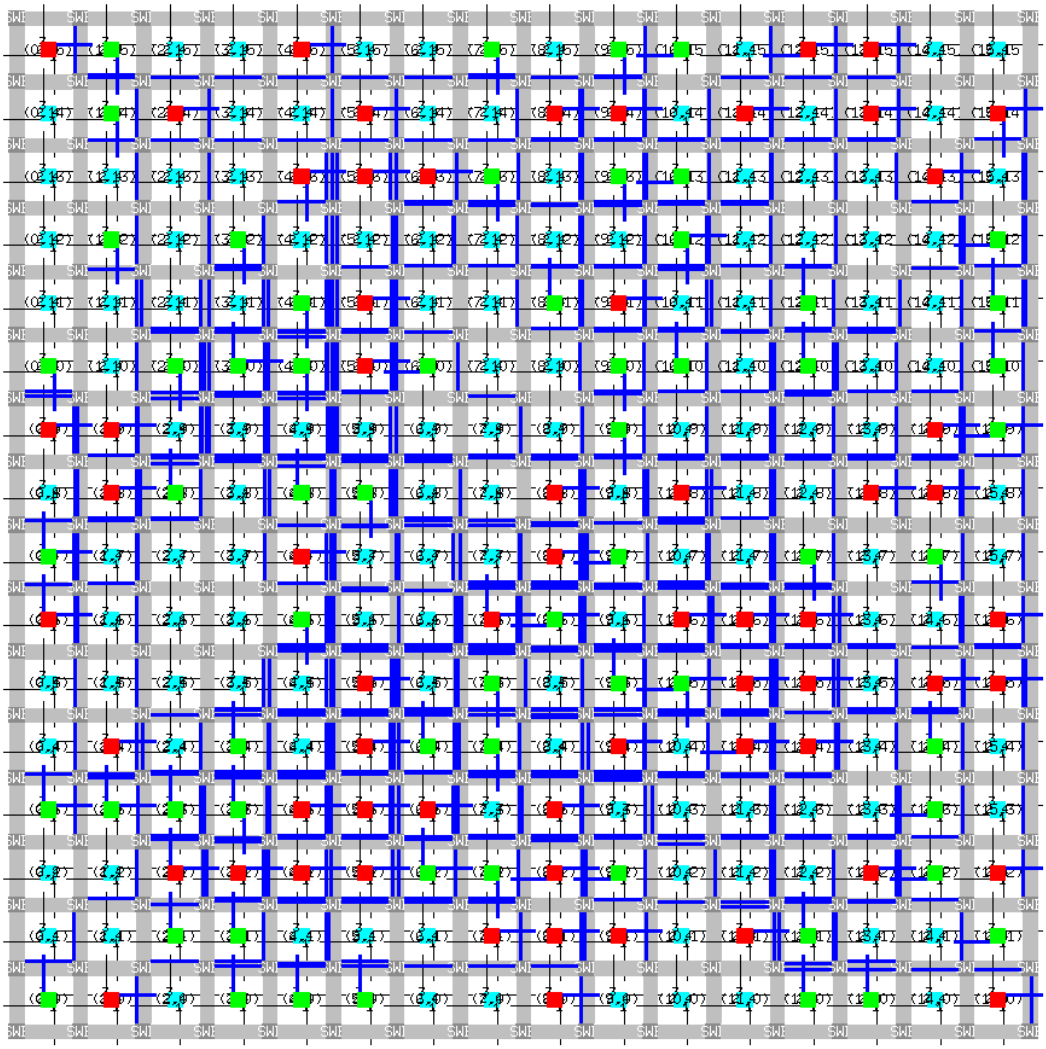

Figure 8: swappable input pins 16x16 grid and 14 track per channel

CCT4

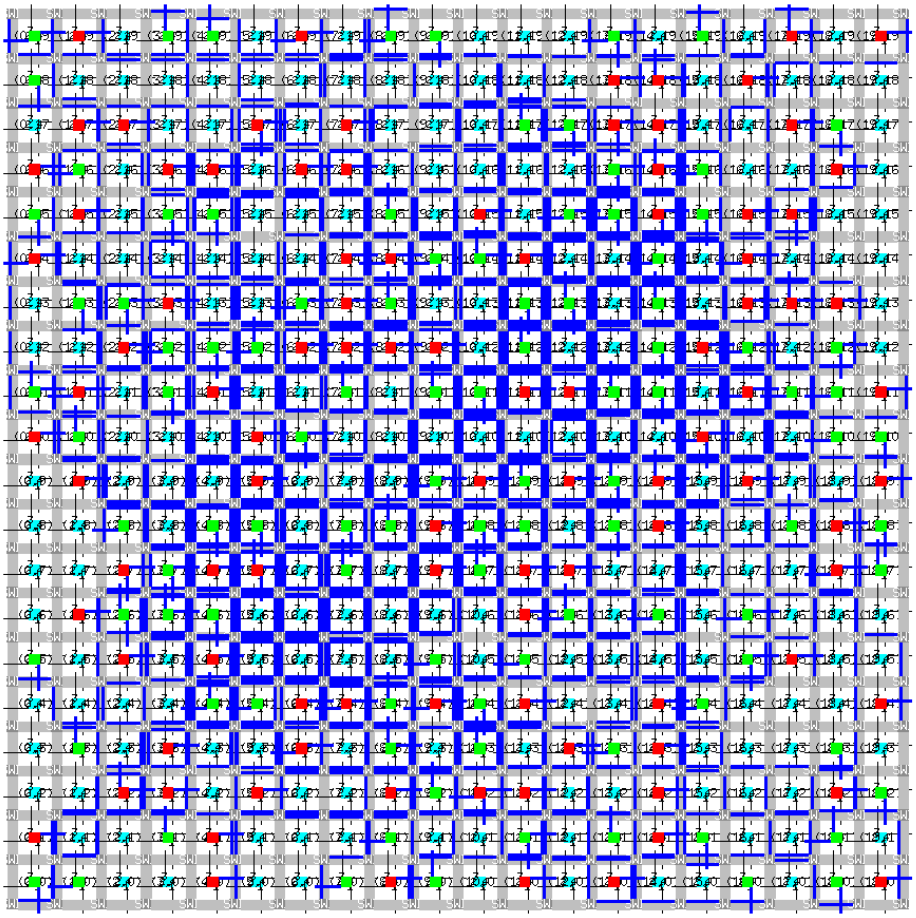

Figure 9: Non swappable 20x20 grid and 14 track per channel

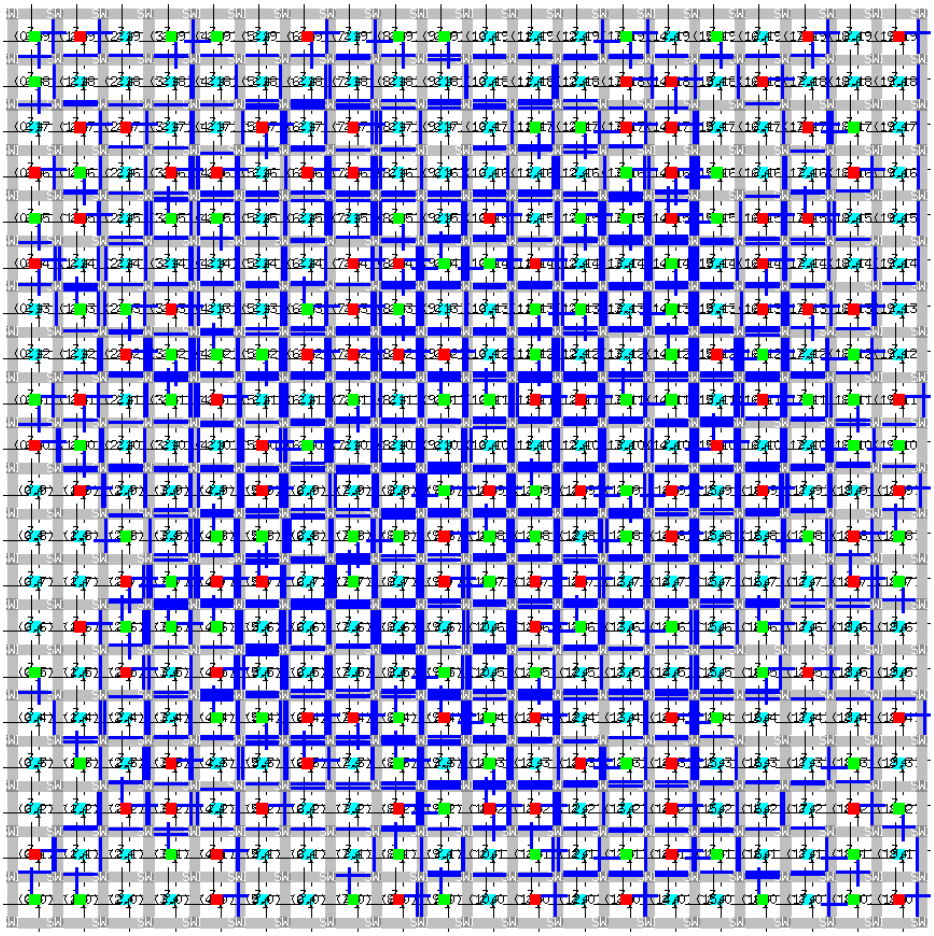

Figure 10: swappable input pins 20x20 grid and 14 track per channel

STATISTICAL RESULTS

From the tables 1, 2 and figure 11, we can observe, less number of routing wires in total is required to route all the test cases compared to baseline, ofcourse this is due to the flexibility of choosing sink pin once the maze router algorithm reaches near the sink CLB location. In next section we discuss the swappable inputs impact on runtime.

test file	FPGA Grid	W	Total Wires
cct1	5x5	12	46
cct2	8x8	12	114
cct3	16x16	14	882
cct4	16x16	14	2050

Table 1: Non Swappable Inputs (Baseline)

test file	FPGA Grid	W	Total Wires
cct1	5x5	12	41
cct2	8x8	12	106
cct3	16x16	14	775
cct4	16x16	14	1908

Table 2: Swappable Inputs

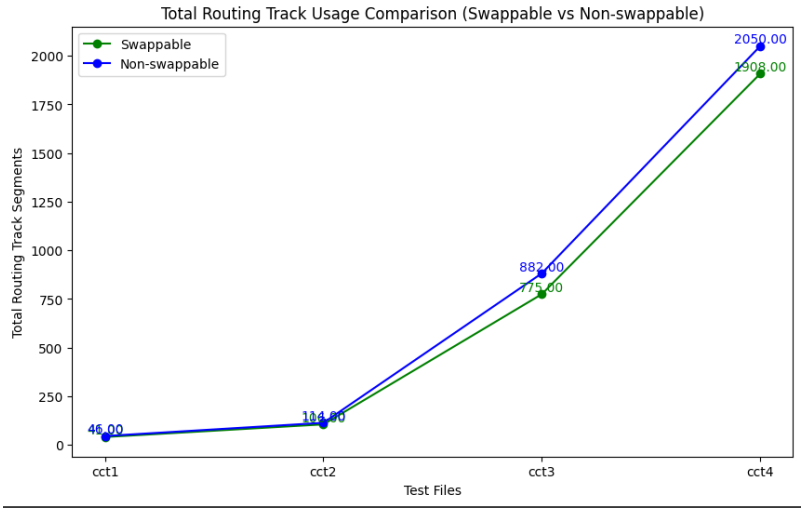

Figure 11: Plot comparision of swappable vs non swappable track segments used by all test cases

IMPACT OF SWAPPABLE INPUTS

Swappable inputs have a significant impact on reducing both the number of tracks per channel (W) and the total number of routing segments used. In my implementation, I observed that enabling swappable inputs consistently reduced the total number of routing segments across all test circuits compared to the non-swappable case. This reduction is primarily due to the increased flexibility in selecting input pins, allowing the router to find more efficient paths with fewer track segments. Additionally, this flexibility leads to a more optimized routing process, which directly decreases the runtime, as fewer segments need to be searched and routed. The swappable inputs avoid congestion by redistributing the routing load across multiple pins, ultimately resulting in a more efficient routing strategy. Figure 12, shows runtime performance and comparision of the same.

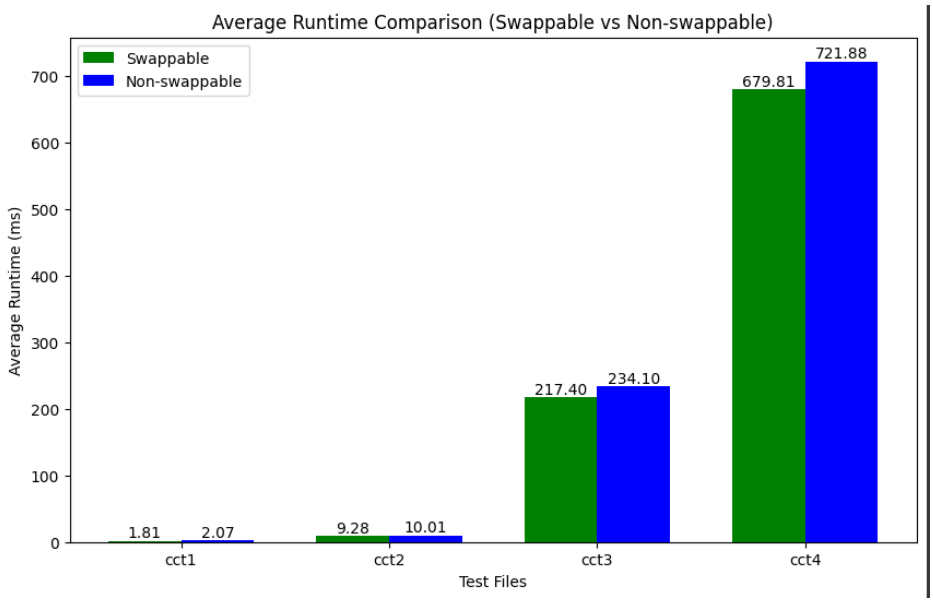

Figure 12: Bar chart comparision of swappable vs non swappable runtime

The speedup is calculated using the formula:

$$\text{Speedup} = \frac{\text{Non-swappable runtime}}{\text{Swappable runtime}}$$

Based on the runtimes obtained from the test cases, the speedup for each test case is as follows:

- **Test case 1 (cct1)**:
$$\text{Speedup} = \frac{2.074}{1.806} = 1.1486$$
- **Test case 2 (cct2)**:
$$\text{Speedup} = \frac{10.008}{9.281} = 1.0783$$
- **Test case 3 (cct3)**:
$$\text{Speedup} = \frac{234.101}{217.402} = 1.0769$$
- **Test case 4 (cct4)**:
$$\text{Speedup} = \frac{721.88}{679.815} = 1.0619$$

Speedup Summary:

- Test case 1 (cct1): 1.15x speedup
- Test case 2 (cct2): 1.08x speedup
- Test case 3 (cct3): 1.08x speedup
- Test case 4 (cct4): 1.06x speedup

The results indicate that the swappable inputs consistently provide a speedup across all test cases, ranging from approximately 1.06x to 1.15x. This demonstrates that the flexibility introduced by swappable pins leads to reduced routing times, with the most significant improvements observed in smaller test cases.

MINIMUM TRACKS TO ROUTE

In this section, we will discuss the results of number of minimum tracks per channel needed to route each test circuit cases as well as come up with number of minimum tracks per channel needed to route all 4 circuits, which by definition would be given by:

$$\textbf{minimum\_tracks\_needed} = \max\left(\min(\textbf{cct1}), \min(\textbf{cct2}), \min(\textbf{cct3}), \min(\textbf{cct4})\right)$$

Figure 13 shows an example cct1 case for minum W = 4 routed. From table 3 and 4, for cases non-swappable, we see the required number of minimum W is equal to or greater than that of swappable. hence minumum overall W for each test cases in non-swappable is 11, while the minimum W overall for each test cases in swappable case is 8. If we combine both analysis, for all circuits to route on all configurations like swappable or non-swappable styel, the minimum **W** = 11. However you may still observe, total wirelength for swappable is still smaller than non-swappable.

test file	FPGA Grid	Minimum W	Total Wirelength
cct1	5x5	4	46
cct2	8x8	4	121
cct3	16x16	7	880
cct4	16x16	11	2107

Table 3: Non swappable inputs (Baseline) minimum tracks per channel & total wirelenth

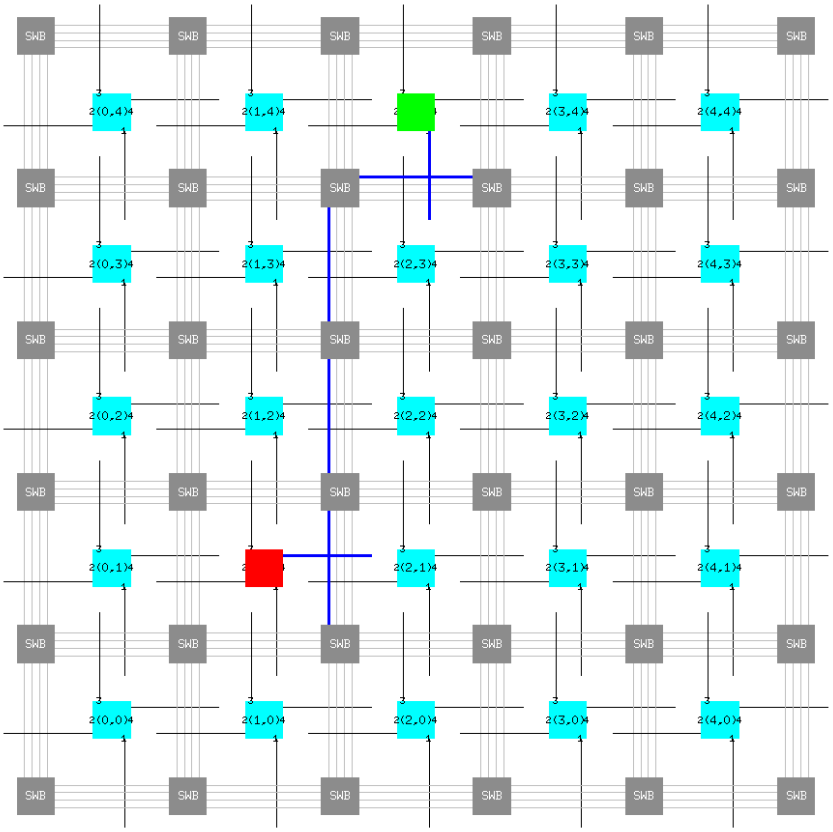

Figure 13: non-swappable cct1 exampmle on minimum W = 4

test file	FPGA Grid	Minimum W	Total Wirelength
cct1	5x5	4	42
cct2	8x8	3	110
cct3	16x16	5	794
cct4	16x16	8	1964

Table 4: Swappable inputs minimum tracks per channel & total wirelenth

OPTIMIZATIONS

To find minimum W across test cases, binary search or linear search can be incorporated, where the task would be to find next minimum value of W such that after which routing fails for at least one of the cases.

The following algorithm performs a binary search to find the minimum number of tracks per channel (W) that can successfully route all the test cases. The search explores values of W between a specified lower bound (W_{min}) and upper bound (W_{max}):

Algorithm 1: Binary Search for Minimum W

[1] Initialize $W_{\min} \leftarrow 1$ Initialize $W_{\max} \leftarrow$ maximum value of tracks per channel Initialize min_W $\leftarrow W_{\max}$ **while** $W_{min} \leq W_{max}$ **do**

Set $W_{\text{mid}} \leftarrow \frac{W_{\min}+W_{\max}}{2}$ Calculate the midpoint Create a router instance with W_{mid} Parse test cases from input file Route all test cases with W_{mid} **if** *routing is successful* **then**

Set min_W $\leftarrow W_{\text{mid}}$ Update $W_{\max} \leftarrow W_{\text{mid}} - 1$ Search in the lower half **else**

Update $W_{\min} \leftarrow W_{\text{mid}} + 1$ Search in the upper half Return min_W as the minimum number of tracks per channel for successful routing

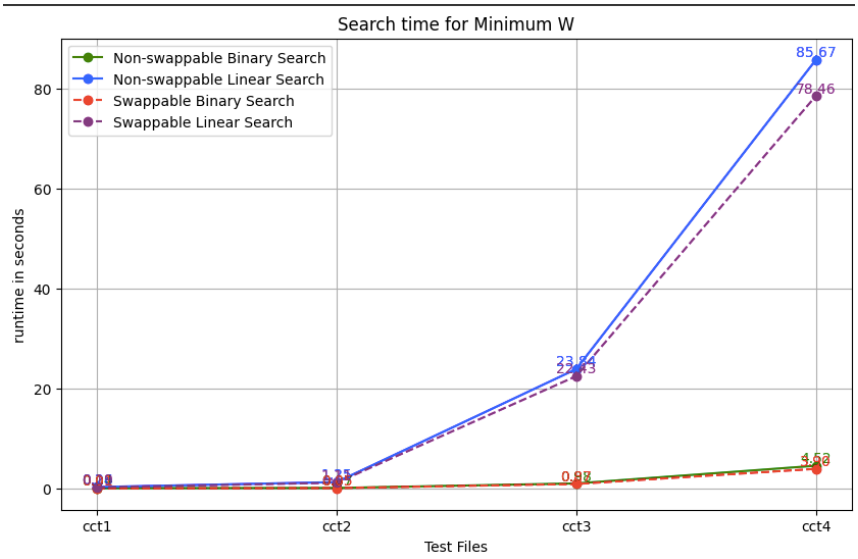

Figure 14: runtime required for linear vs binary search on finding minimum W

SPEEDUP ANALYSIS FOR BINARY VS LINEAR SEARCH

The speedup for both non-swappable and swappable routing approaches with respect to binary and linear search methods is calculated using the formula:

$$\text{Speedup} = \frac{\text{Linear Search Time}}{\text{Binary Search Time}}$$

The following tables provide the speedup values for each test case (cct1, cct2, cct3, cct4) for both non-swappable and swappable routing cases.

Test Case	Linear Time (s)	Binary Time (s)	Speedup (Non-Swappable)
cct1	0.2618	0.0398	6.58
cct2	1.2475	0.075	16.63
cct3	23.845	0.982	24.29
cct4	85.67	4.52	18.95

Table 5: Speedup for Non-swappable Routing (Binary vs Linear Search)

Test Case	Linear Time (s)	Binary Time (s)	Speedup (Swappable)
cct1	0.1914	0.00874	21.90
cct2	1.1121	0.0521	21.35
cct3	22.43	0.871	25.75
cct4	78.4645	3.89885	20.13

Table 6: Speedup for Swappable Routing (Binary vs Linear Search)

NON-SWAPPABLE

SWAPPABLE

SUMMARY OF SPEEDUPS

- **Non-swappable:** Speedups range from approximately 6.58 to 24.29 times faster for binary search compared to linear search.
- **Swappable:** Speedups range from approximately 21.90 to 25.75 times faster for binary search compared to linear search.

This demonstrates that **binary search** consistently provides significant speedups over **linear search**, with the swappable input case generally exhibiting higher speedups than the non-swappable case due to more flexibility in routing options.

FLOW OF THE SOFTWARE AND DATA STRUCTURES

This project implements an FPGA maze router, utilizing a visualization component and focusing on both **baseline** and **swappable input** routing styles. The project was structured with multiple core routines and data structures that interact to achieve efficient routing on an FPGA grid. The router optimizes for the minimum number of tracks per channel needed to complete the routing, using both linear and binary search approaches. The visualization is built using the EZGL graphics library, displaying the routing on a GUI. This description explains the flow of the software, major routines, and key data structures, along with decisions made during the development process. This section provides a detailed overview of the major components, the rationale behind their design, and challenges faced during the implementation.

0.1 MAIN COMPONENTS OVERVIEW

The key components of the router include:

- **Main.cpp**: Handles the user input for selecting test files, routing modes, and visualization settings. It calls appropriate methods for routing and outputs runtime and total track usage.
- **FPGAVisualizeEZGL.cpp/hpp**: Handles all aspects of visualizing the FPGA grid and routes. This component creates a graphical interface using EZGL and supports real-time updates as the routes are created.
- **MazeRoutingFPGA.cpp/hpp**: Implements the main maze routing algorithm, defining the grid, tracks, and paths. It also includes key functionalities like searching

for valid paths, resetting tracks, and labeling grid nodes.

Grid Representation

The FPGA grid is represented using a **3D vector**:

```
vector<vector<vector<FPGARouting::Track*>>> FPGAGrid;
```

- The first dimension of the grid represents the **x-coordinates**.
- The second dimension represents the **y-coordinates**.
- The third dimension holds **tracks** within a specific grid location.

This 3D vector structure was chosen for its simplicity and efficiency in managing the 2D layout of logic blocks, with the third dimension providing multiple tracks between blocks. This allowed for efficient storage and quick access to the track information required for routing.

Track Representation

Each individual track within the grid is modeled by a `Track` class. This class includes properties like the track's coordinates (`locationTrackX`, `locationTrackY`), track ID, and whether it's available for routing:

```
class Track {
    int locationTrackX;
    int locationTrackY;
    int trackID;
    int Tracklabel;
    bool usable;
};
```

By encapsulating each track in a class, the routing algorithm can label tracks, mark them as used or available, and assign source/destination pins efficiently. This was crucial for dynamically routing connections between FPGA logic blocks.

Path Representation

Each route between pins is represented by the `Path` class. This class stores the coordinates of the source and sink pins, as well as a vector of tracks used in the path:

```
class Path {
    int loadX;
    int loadY;
    int loadP;
    int sinkX;
    int sinkY;
    int sinkP;
    vector<FPGARouting::Track*> usedTracks;
};
```

This structure makes it straightforward to track the source and sink of each connection, as well as store the path's used tracks. The use of vectors allows for flexibility in how many tracks can be utilized, making it easy to add or remove tracks as the algorithm proceeds with routing and backtracking.

Challenges with Graph-Based Approach

In the initial phase of the project, the FPGA routing problem was modeled as a **graph**, where:

- **Nodes** represented tracks.
- **Edges** represented possible connections between tracks.

While this graph-based approach was conceptually appealing, it introduced several challenges:

- **Graph Construction Overhead**: Representing each track as a node and each connection as an edge significantly increased memory consumption. The need to

dynamically update the graph as routing proceeded added both computational and memory overhead.

- **Complex Path Expansion**: Expanding routes by traversing graph nodes (tracks) and updating edges (connections) made the pathfinding process slow and difficult to manage. For larger FPGA grids, this complexity compounded, leading to inefficient performance.

Due to these issues, the project shifted from a graph-based approach to using **3D vectors** for grid representation and **vector-based track/path structures**. This reversion to simpler data structures yielded significant benefits:

- **Direct Access**: The use of vectors allowed direct access to tracks based on grid coordinates, eliminating the need for costly graph traversals.
- **Efficient Memory Usage**: Vectors are more memory-efficient than graph structures when the routing problem primarily involves grid-based traversal.
- **Improved Performance**: By avoiding graph-based overhead, the routing process became faster and easier to implement, particularly for larger grids and more complex routing cases.

MAZE ROUTING ALGORITHM

The core routing logic is housed in `MazeRoutingFPGA.cpp`. It relies on several data structures and functions to model the FPGA grid, identify routes, and manage track availability. Here's an overview of the main classes and methods:

Track Class

Represents individual routing tracks. Each track stores information about its availability, the coordinates it covers, and its label (used in the expansion process).

- `resetTrack()`: Resets the track to its initial state (available and unlabeled).

- `setTrackLabel()`: Sets the label of the track, which determines if it's part of a route or an unused track.

Path Class

Represents routing test cases, including source and destination pin information. Stores a list of used tracks to backtrack the route.

- `useTrack()`: Marks a track as used by a path.
- `resetUsedTracks()`: Clears all tracks used by the path for rerouting purposes.

MazeRoute Class

The main controller for the routing process. It initializes the grid, parses input test files, and executes the routing search algorithm. Key methods include:

- `createFPGAGrid()`: Initializes the FPGA grid with vertical and horizontal tracks.
- `searchRoute()`: Implements the maze expansion routing algorithm. It labels tracks using a breadth-first search (BFS) and backtracks once a destination is found.
- `lookAround()`: Finds neighboring tracks for expansion during the routing process.
- `labelFPGAGrid()`: Marks the grid before each routing test case to reset the routing labels.

BINARY AND LINEAR SEARCH FOR MINIMUM TRACKS

Two search methods are implemented to find the minimum number of tracks per channel (W):

- **Binary Search**: In `binary_search_minimum_tracks()`, the search space between the lower bound and upper bound is split in half. The middle value is tested, and if routing is successful, the upper bound is reduced; otherwise, the lower bound is increased. This approach efficiently converges to the minimum W that allows all test cases to route successfully.

- **Linear Search**: In `linear_search_minimum_tracks()`, the algorithm tests each possible track count starting from the upper bound until the minimum successful W is found. This approach is simpler but slower compared to binary search.

GRAPHICS AND VISUALIZATION

The visualization component (`FPGAVisualizeEZGL.cpp`) is responsible for drawing the FPGA grid and routes in real-time:

- `FPGAVisualizeInitVisualization()`: Initializes the graphical window, sets up buttons for user interaction, and draws the initial grid layout.
- `FPGAVisualizedrawGrid()`: Draws the grid structure, including switch blocks, logic blocks, and routing tracks.
- `FPGAVisualizedrawIncrementalTestCases()`: Dynamically draws paths as the routing algorithm progresses, highlighting the source and destination CLBs and the wire segments used.

The visualization is interactive, allowing users to step through each routing process using "Next Path" and "Previous Path" buttons, which call `FPGAVisualizeNextPath()` and `FPGAVisualizePreviousPath()` respectively. These functions update the grid display based on the current routing state.

KEY DECISIONS IN THE ALGORITHM

- **Swappable Inputs**: A major feature is the option for swappable inputs. The software supports two modes:
  - **Baseline Mode**: The router must connect exactly to the specified input pins.
  - **Swappable Mode**: The router can choose any input pin on the destination block. This flexibility typically reduces the number of used routing segments and improves runtime, as demonstrated in test results.

- **Routing Failures**: In the event of routing failures, the algorithm backtracks and marks paths as failed for future attempts. The decision to use BFS ensures that the shortest paths are found first, optimizing both runtime and routing length.
- **Track Assignment**: The choice to divide tracks into horizontal and vertical sets simplifies the routing expansion process and reduces conflicts between competing paths. Each track has a specific ID, and even/odd track assignments are used to handle pin-to-track connections effectively.

RUNTIME ANALYSIS AND OPTIMIZATION

To evaluate the runtime of the routing process, a Python script runs the routing algorithm multiple times and records the total runtime. The script uses the `subprocess` module to execute the router with different parameters and collects the total runtime from the output. The results show a clear improvement in runtime when swappable inputs are enabled and when using binary search compared to linear search for finding the minimum tracks.