ECE1378 - ASSIGNMENT 1

# Maze Routing with a History Based Search for FPGAs

October 13, 2017

Edward S. Rogers Sr. Department of Electrical and Computer Engineering
University of Toronto
Nicholas V. Giamblanco
Student ID: 1000324534

# Contents

1 Introduction . . . . . 2
2 Implementation . . . . . 3
  2.1 Data Structures . . . . . 3
  2.2 Main Methods . . . . . 4
  2.3 Implementation Challenges . . . . . 5
3 Results . . . . . 5
  3.1 Track Segments Used Versus Track Width . . . . . 5
  3.2 An Note On Switches . . . . . 6
  3.3 Parallelization . . . . . 7
4 Output . . . . . 9

## 1 Introduction

The concept of *routes* are of significant importance in many fields and aspects of modern life. Examples of such usages may present itself in transit systems, taxi-like services, cellular/mobile communication systems, and electronic devices. All of these services and systems aim to find an *optimal* route, where optimality is defined for the particular system or service. For example, a taxi service may aim to provide an optimal route from some point $a$ to $b$ by finding the shortest distance path between $a$ and $b$.

FPGA's require routes between different logical elements within the body of the FPGA, as this is the basis for any form of analog and digital communication. However, due to the user specified behaviour of the FPGA, routes are dynamic, and require adjustment during synthesis of the use specified operation of the FPGA. Therefore, many routes are required, introducing complexity in the synthesis of user specified designs.

In this report, we investigate and implement a routing algorithm, specifically *Maze Routing* [1]. Two switch styles were analyzed with Maze Routing, namely Wilton Style switches [2], and Fully-Connected Style switches. This report also explores an optimization for Maze Routing, which we define here as History Based Search. Our results indicate that the addition of this feature resulted in a lower possible track width.

## 2 Implementation

To implement this routing algorithm, we logically segmented the problem in the following way:

- There should be some concept of an integrated circuit: `circuit.cpp`
- Within the integrated circuit, there should be a grid of switches and logical elements: `sblck.cpp`, and `lblck.cpp`
- Between each switch, should be a track of wires which connects them `track.cpp`.
- Between orthogonal intersections, logical elements should be connected to switches through a track that is nearest that element.
- Routing is performed on the integrated circuit with a router: `router.cpp`
- Store a routed path: `path.cpp`
- Routing is a subset of synthesis operations required for integrated circuit construction, and is considered as a utility: `circutil.cpp`

### 2.1 Data Structures

To formally describe our implementation, we must identify the *main* data structures used:

- Elementary data structures such as queues, lists and look-up tables were elementary structures of the following object definitions [1].
- `Track` Object: an object that represents a track of wire segments (Using a `vector` <> template from `C++11`. Each wire segment is represented with a integer value. These wires are directly connected to pins of two `Switch` Objects and one or two `Logic` Objects.
- `Switch` Object: an object that maintains the functionality of either a Fully-Connected or Wilton Style switch (through a look-up like implementation) and holds references of up to 4 `Track` objects (with a minimum of 2). These objects also contain methods applicable to updating connected track segments, connectivity checks, etc.
- `Logic` Object: an object that holds reference to 4 `Track` objects, and maintains state on 4 pins (used to identify source and target pins). Objects are able to set track segments.
- `Circuit` Object: holds a grid of $l \times l$ size of `Logic` objects, a grid of $(l+1) \times (l+1)$ size of `Switch` objects, and creates and holds the `Track` objects to which `Logic` grid and `Switch` grid refer to.
- `Router` object: an object that holds methods that enable maze routing to take place on a `Circuit` object, and return back a routable path, held in a `Path` object.
- `Path` object: an object which holds a list of `Switch` objects used in a route for a particular net in a netlist.

[1] The way we have defined these objects are for readability of this report. Specifically, the implementation named `Switch` as `Sblck` and `Logic` as `Lblck`

Several other data structures exist, but are not "crucial" for primary operation of Maze Routing. Please refer to the source code for additional information if necessary. It is important to note the heavy use of the word reference. This entire implementation is heavily memory dependent, such that a `Circuit` object references **and** contains a grid of `Switch` objects and `Logic` objects, which all refer to some `Track` object that any one of the `Switch` objects and `Logic` objects are connected to. This heavy reference scheme alleviates several computational costs (less conditional checks per requested resource, etc.), but introduces complexity when separating tasks (may not be parallelizable).

## 2.2 Main Methods

- `Sblck` :: `get_pin`(), `Sblck` :: `set_pin`(), `Sblck` :: `set_switch`(): These methods are member of the `Switch` object, and enable read and write to the referenced track, and corresponding track segments that the switch is connected to.
- `Router` : `begin_search`(): This method (which is a member of the `Router` object), references the freshly constructed integrated circuit for routing, and identifies the `Switch` objects to begin the search process from. By using a relative coordinate system from the `Switch` object grid to the `Logic` object, we obtain the nearest `Switch` objects to the source pin for search.
- `Router` :: `search`(): This method places the initial `Switch` objects found in `Router` : `begin_search`(), and begins a nearest neighbour search, and updates the neighbours track segments. That is, one `Switch` object is inspected at a time, first checking the connect track segments for the Target pin, or it searches for any available neighbouring `Switch` objects connected to the `Switch` object in question and appends a cost to connected segments. The available neighbours will be added to a `queue` of search candidates, and this search process will iterate over these candidates until one of the connected track segments is found. To improve the efficiency of the run-time of this method, any `Switch` object that was inspected in the queue can never be reused within the same route search. Another optimization used to reduce the track width implemented a history based search, forcing the router to search differently based off a history of past directions taken during the search.
- `Router` :: `check_for_target`(): To logically segment the problem of searching and checking if the target has been reached, this method was created. It aided in the inspection of track segments, allowing either the traceback to begin to execute (if a target pin was identified), or return to searching for the target.
- `Router` :: `traceback`(): Once a target pin is reached, we must route one path from target to source. This method accomplishes this task, discovering the cost to arrive at the target pin, iteratively finding the path to a switch that is one less the cost of the previously found switch in the path until we arrive at a cost of `0` (indicating we have arrived at the source). This particular method required two logical cases for the two switch types, as the Fully-Connected switch should be exploited for its incredible flexibility, and the Wilton switch should be handled carefully (due to the set of rules a Wilton Switch must abide by). If there are multiple track segments which meet the cost requirement, this router randomly select one of the paths to continue along.[2]

[2]This makes this router ultimately non-deterministic for results, but will always route.

- `Circuit:reset()`: resets the state of any track segments that are not used for some path to an available state.

### 2.3 Implementation Challenges

As noted before, many of these data structures are passed around the program by reference. This idea was prioritized over a pass by value implementation due to superiority in reflecting changes in shared objects. That is, a change in a referenced data structure is reflected for all objects referencing that structure. This also presented drawbacks (what if one objects writes to a referenced object, and then another object sequentially writes to the same referenced object?). Also, it was encountered that boundary conditions are an essential check (due to the grid structure of this system). Hence, several checking mechanisms were implemented to formally prohibit an out of bounds error.

## 3 Results

### 3.1 Track Segments Used Versus Track Width

As specified in this assignment, the software implementation of this routing algorithm allowed for variable track width, $W$. The following data was collected from varying $W$ to an un-route-able state for each test circuit. The number of used tracks $\tau$ was also recorded. We denote $\tau_F$ and $W_F$ for the number of used tracks per width for the Fully Connected Style Switch, and $\tau_W$, $W_W$ for the Wilton Style switch. The results are listed in Table 1.

To further reduce the width, a modification was applied to the router in this assignment. This modification was a well known probabilistic mechanism which tracks historical patterns of certain events occurring over time. This *history* based tracking is accomplished through counting specific events and is title as Hysteresis. A basic history pattern tracking implementation can operate in the following way:

Suppose this an event $S$, which can either occur, or not. Therefore, $S$ is a random variable which can take on two values $S = 1$ or $S = 0$. As we conduct experiments on the event $S$, we are interested to predict the value of $S$. In order to predict that either $S = 0$ or $S = 1$, then we count the number of times the event occurred previously. For example, if we draw $S = 0$ three times, we record this historical frequency $f$. However, by recording the entire history of the experiments drawn on $S$ may be cumbersome, and may hold no additional information. Therefore, we place a limit on the accumulation of historical data. For example, we will only ever record a maximum of $X$ historical points for each time value of $S$ took on 1. However, if we did not encounter 1, we decrement the historical frequency, until it has reached zero. This provides us with the phenomenon of hysteresis, where a threshold can be placed on $f$, such that at some value $T_h$, where $0 \leq T_h \leq X$, if $f \geq T_h$, then we can predict $S = 1$, otherwise we predict $S = 0$.

By using this mechanism, we can apply this to the searching operating of the maze router, such that as we continue to search in the same direction for more than $T_h$ times, do not explore this path. Otherwise, if we search in some direction less than $T_h$ times, explore this path. This mechanism is useful if there are several long routes which utilize a long narrow path within the switch grid. By reserving this long path, other potential routes will take on extra difficulty discovering a route. By using the concept of history of searches, other potential routes will not encounter as much difficulty.

**Table 1:** Track Segments Used Versus Track Width (Minimum Achievable Value)

| *Circuit Files* | $\tau_F$ | $W_F$ | $\tau_W$ | $W_W$ |
|---|---|---|---|---|
| CCT1 | 32 | 3 | 32 | 3 |
| CCT2 | 65 | 5 | 67 | 6 |
| CCT3 | 263 | 8 | 316 | 8 |
| CCT4 | 2998 | 15 | 2376 | 21 |

## 3.2 An Note On Switches

In this assignment, two different switch types were explored for their effect on minimum track width, and minimum track segments used. Specifically, we explored Wilton Style and Fully Connected switches. For the remainder of our discussion of switches, assume each switch has a geometry of a square, where each side is labeled as $X, Y, Q, R$ respectively and each side has $n$ pins. We must now define the properties of the switches, and how each side can connect to each other.

For the Fully Connected switch, any pin $p_i$ on any side, can connect to any other pin on any of the other sides. More formally, if $p_i$ is on the side $X$, and there are four available sides $D = \{X, Y, Q, R\}$, then $p_i$ can connect to any other pin in $D - X$. Hence Fully connected switches provide superior flexibility to any other switch implementation. Therefore, Fully Connected switches should be able to maintain rout-ability with a *reduced track width and reduced track segment usage, $\zeta$*. We can also describe the flexibility of the Fully Connected Switch. Flexibility of a switch, $F_s$, is defined as the internal number of connections pin $p_i$ on side $X$ has with other pins within the switch. For the Fully-Connected block, $F_s = 3n$.

For the Wilton Switch, any pin $p_l$ on any side can only connect to a subset of pins on each of the other sides. Specifically, there are a set of rules that are applied to the Wilton Block that denote switching behaviour. If $\{X, Y, Q, R\} = \{\texttt{NORTH}, \texttt{EAST}, \texttt{SOUTH}, \texttt{WEST}\}$, then the following rules apply:

```
if entering on pin i from NORTH:
    goto: pin (i+1)mod(W) on EAST;

if entering on pin i from EAST:
    goto: pin (2W-2-i)mod(W) on SOUTH;

if entering on pin i from SOUTH:
    goto: pin (i+1)mod(W) on WEST;

if entering on pin i from WEST:
    goto: pin (W-i)mod(W) on NORTH;
```

By investigating these rules, the Wilton switch only has a flexibility $F_s = 3$, as entering from any pin from any given side, can only provide connections with most three other pins from all other sides. This limited flexibility will have an affect on the track width $W$, such that $W$ can not be reduced to the same state as a Fully Connected Switch. Since $W$ increases, the required track segments should also increase. However, the Wilton has some interesting properties: (1) It seems to reduce congestion, (2) it achieves similar amounts of required routing segments as a Fully Connected switch, (3) switch blocks are less complicated to manufacture. The Wilton switch, Although it is not able to reduce $W$ to the same extent as the Fully Connected switch, the Wilton Switch provides 'pretty' good reduction in $W$. Therefore, in general $W_{fc} \leq W_W$.

## 3.3 Parallelization

Parallelization is a concept which allows for a similar task to be completed simultaneously in time. Serialized instructions are executed within fractions of a second with a modern computer. Hence, there should be some notion of speed-up when a task can be ran in parallel in a modern computer. However, certain tasks cannot be parallelized totally, and can reduce performance in certain instances. The router implemented in this assignment was explored for parallelize-able sets of instructions. The implementation of this router primarily used `C` pointer arithmetic and referencing for many of the components required for route computation. This presents a large (near total) dependence on read/writes to exact memory addresses. Specifically, during any call to the `Router`::`search()` method required various `reads` and `writes` to a referenced memory location (recall that (1) the `Circuit` object is passed by reference, (2) all members of the `Circuit` object are passed by reference, (3) the `Router`::`search()` performs many `reads` and `writes` to the members of the `Circuit` object in one iteration of the search. This exact problem also arises in `Router`::`traceback()`. Hence, several synchronization techniques were explored at an attempt at parallelization.

1. Adding Mutexes to sequential data access: by adding a mutex to every object access of the `Circuit` object seemed like an ideal solution (a fine grained synchronization), however due to `C++`'s environment, mutexes cannot be added to copyable/movable elements (which all of `Circuit`'s members were). A redefinition of the inherit `copy` and `move` operations would have bee required, however due to the complexity of each object, this would have lead to buggy code.

2. Using Lambda Function Thread Calls within `C++`: in `C++11`, thread definitions can be applied to "chunks" of code. Hence, several points of introduction of these threading calls were investigated, yet due to the nature of the memory referenced code, it mirrored the lack of synchronization, and overwritten memory, segmentation faults, etc. and could not be applied.

3. Adding Mutexes to method calls: this sort of coarse grained approach to synchronization "locks" out threads from accessing a method in use by another thread. This usage is somewhat useless in the parallelization attempt, but posses an interesting take on `Router`::`search()`. By defining a new "threaded-lockout-friendly" `search()` method, it may be worth while of locking the search process for the routing the entire net, as threads generally run together at random. This is the approach that was implemented as our parallel implementation. By defining two threads which both race to lock our newly defined `search()` method, with each thread inspecting a different block to search from (only considering one block instead of two), the threads will randomly lock the method, and only one thread will be allowed to execute. *This random locking feature* makes this parallelization *non-deterministic*.

As noted, several design approaches were investigated to parallelize this implementation, but due to the time constraints of this project and the heavily memory dependent implementation, parallelization of this router did not improve performance. For many cases, it remained at par with execution times from the serial implementation. It also performed slower than the serial case with circuit `CCT4`. The results are tabulated in Table 2. We refer to the Wilton Switch with "WS", and the Fully Connected Switch as "FC".

**Table 2:** Comparison of Execution Times with a Parallel and Serial Implementation

| Circuit Files | Parallel (seconds) | Serial (seconds) | Switch |
|---|---|---|---|
| CCT1 | 0.0101 | 0.0012 | FC |
| CCT1 | 0.0023 | 0.0012 | WS |
| CCT2 | 0.0244 | 0.0169 | FC |
| CCT2 | 0.0244 | 0.0107 | WS |
| CCT3 | 0.0671 | 0.0426 | FC |
| CCT3 | 0.0509 | 0.0359 | WS |
| CCT4 | 1.4231 | 0.8494 | FC |
| CCT4 | 1.3662 | 0.9517 | WS |

## 4 Output

This section holds the outputs from the router implemented in this assignment.

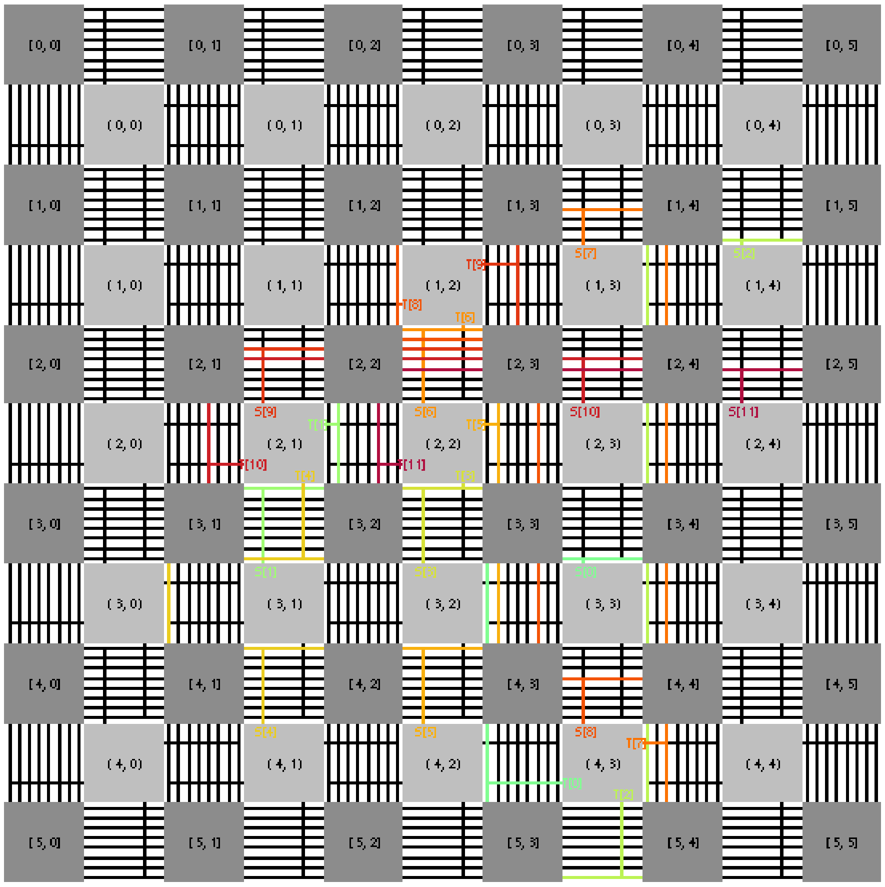

**Figure 1:** Output of CCT1 With Wilton Style Switch, **Segments used: 30**

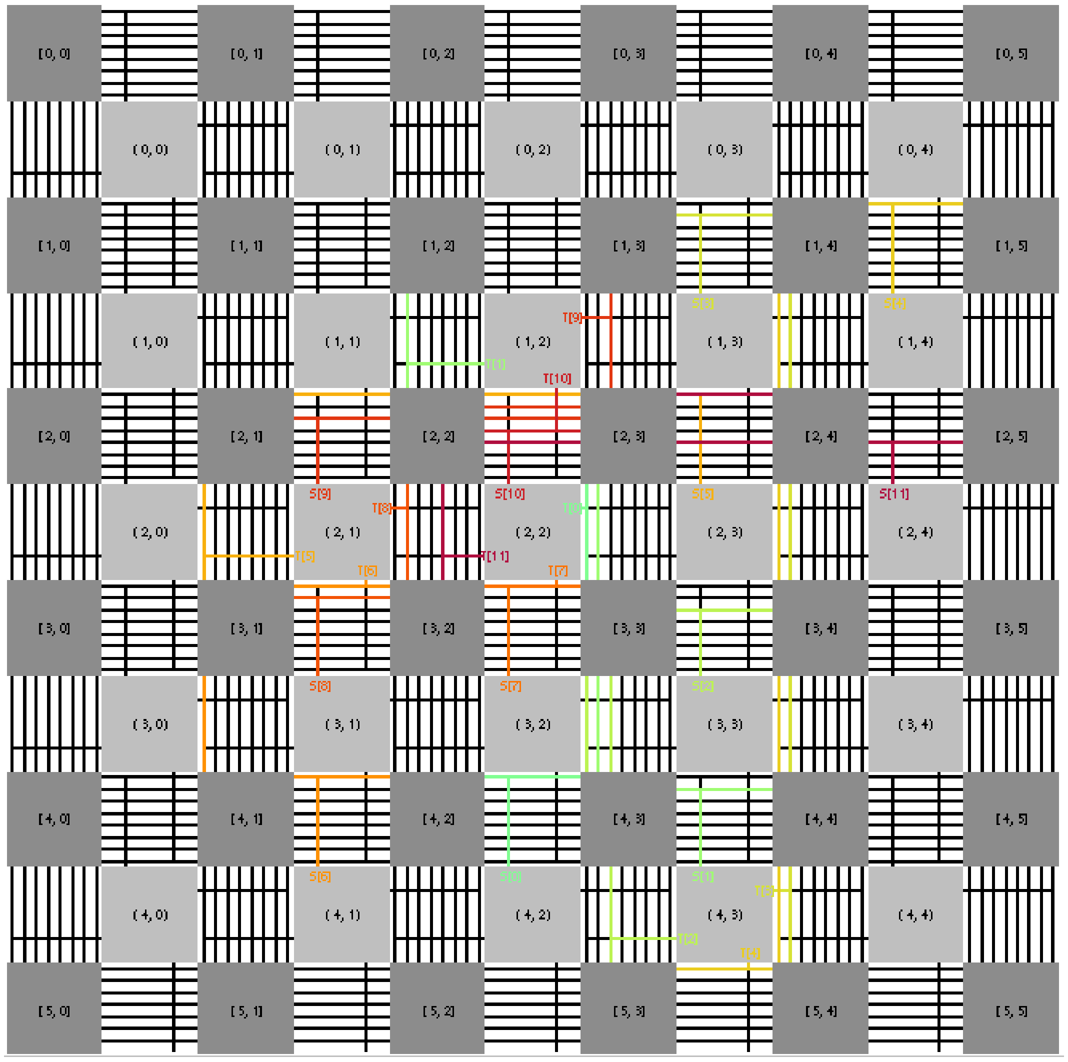

**Figure 2:** Output of CCT1 With Fully Connected Style Switch, **Segments used: 30**

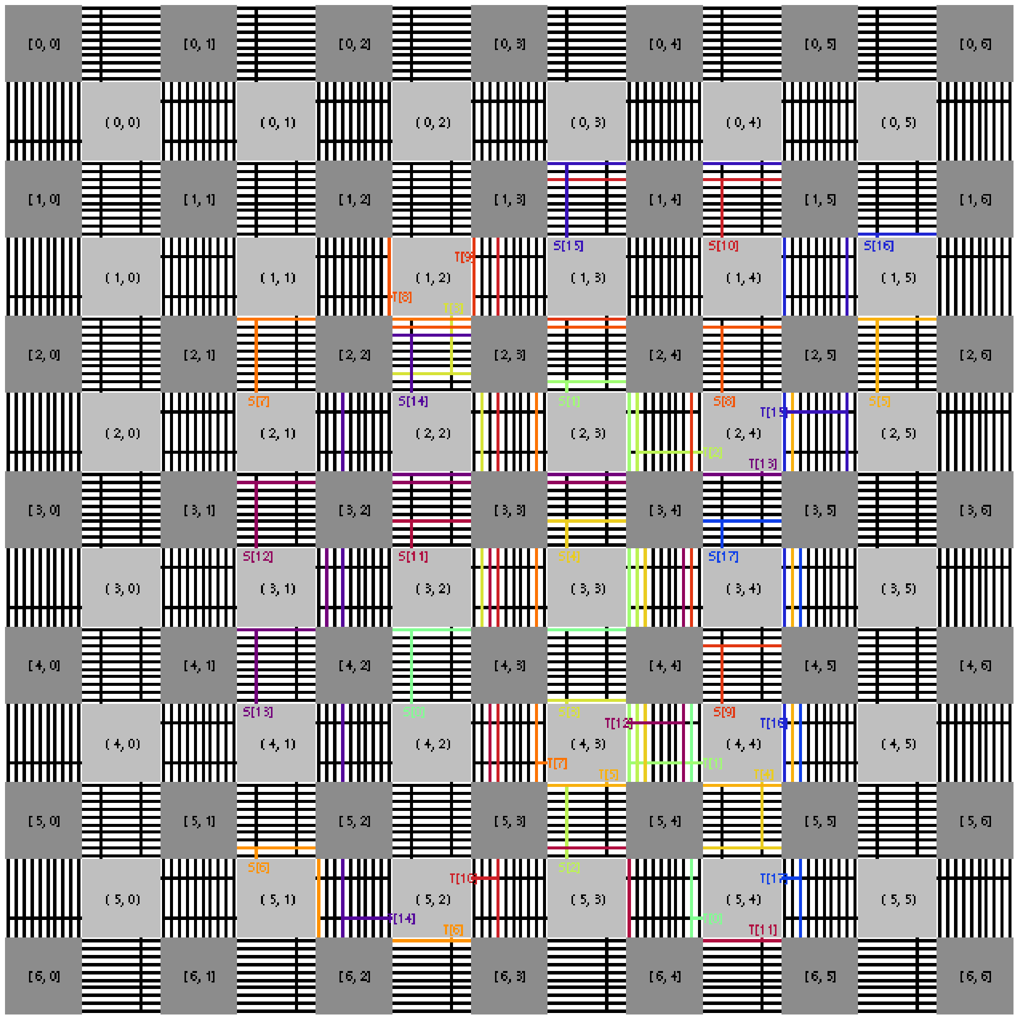

**Figure 3:** Output of CCT2 With Wilton Style Switch, **Segments used: 66**

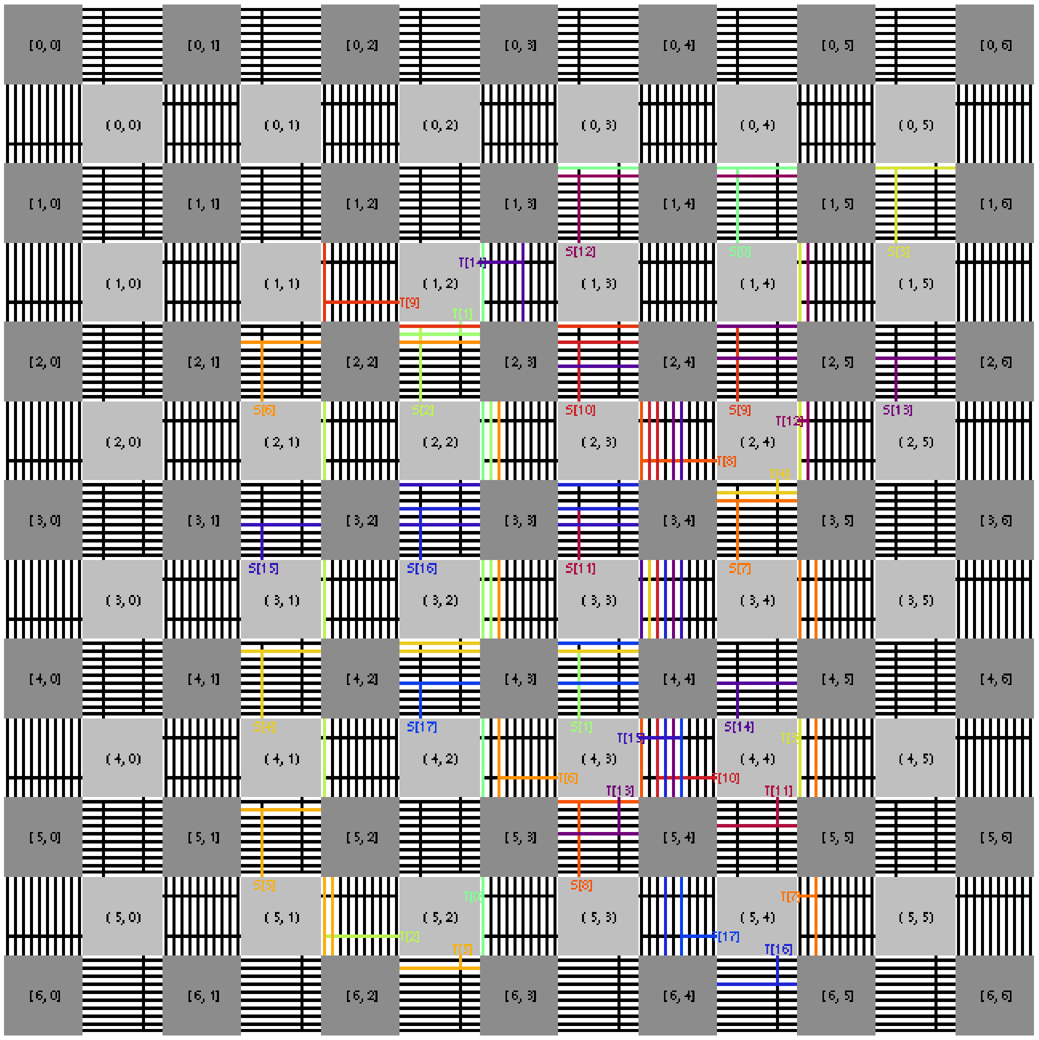

**Figure 4:** Output of CCT2 With Fully Connected Style Switch, **Segments used: 62**

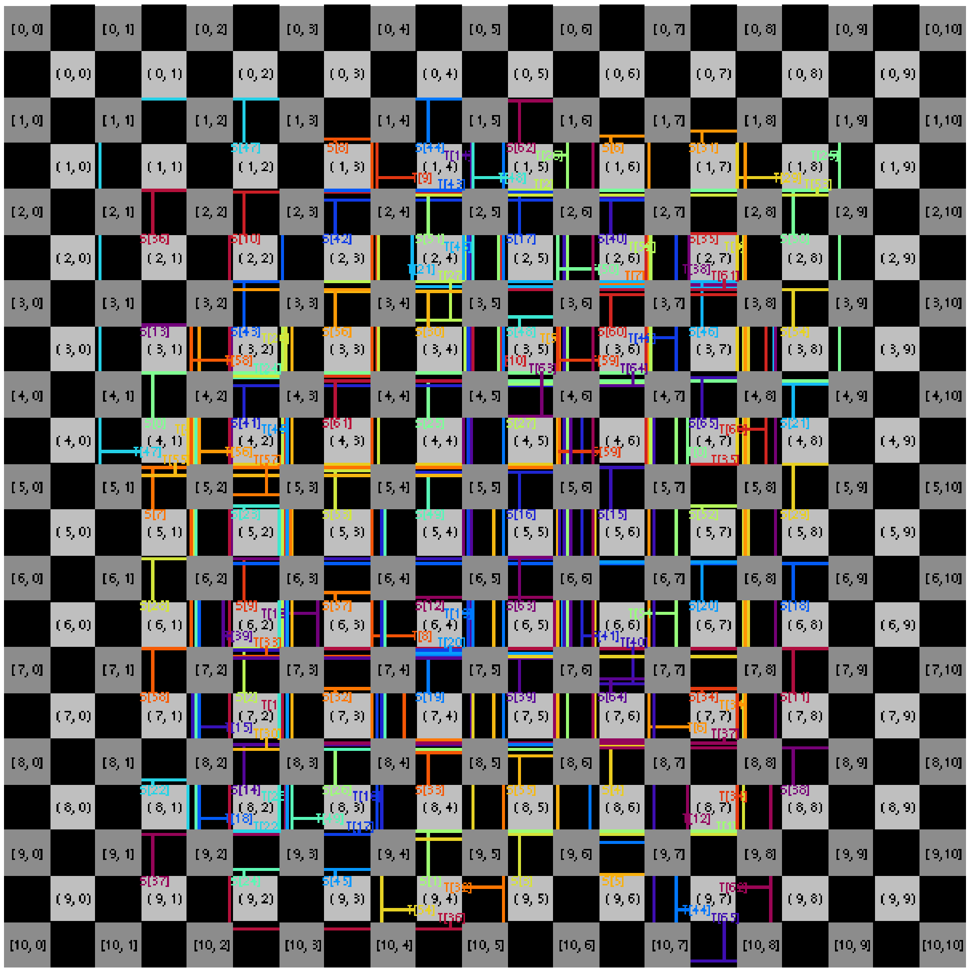

**Figure 5:** Output of CCT3 Wilton Style Switch, **Segments used: 363**

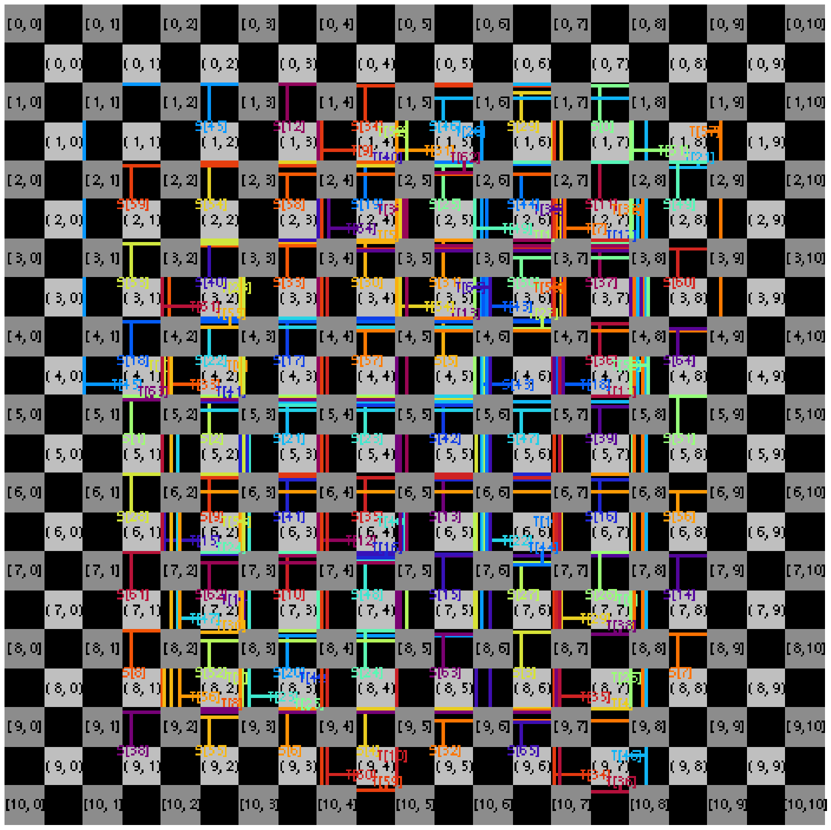

**Figure 6:** Output of CCT3 With Fully Connected Style Switch, **Segments used: 327**

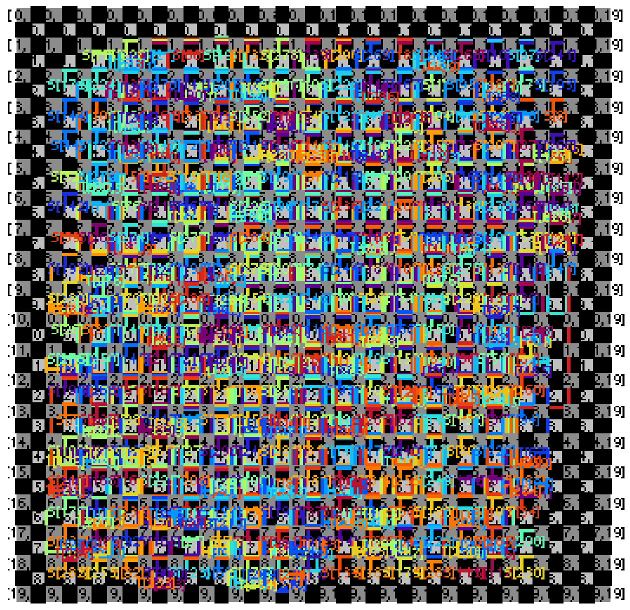

**Figure 7:** Output of CCT4 With Wilton Style Switch, total view, **Segments used: 3354**

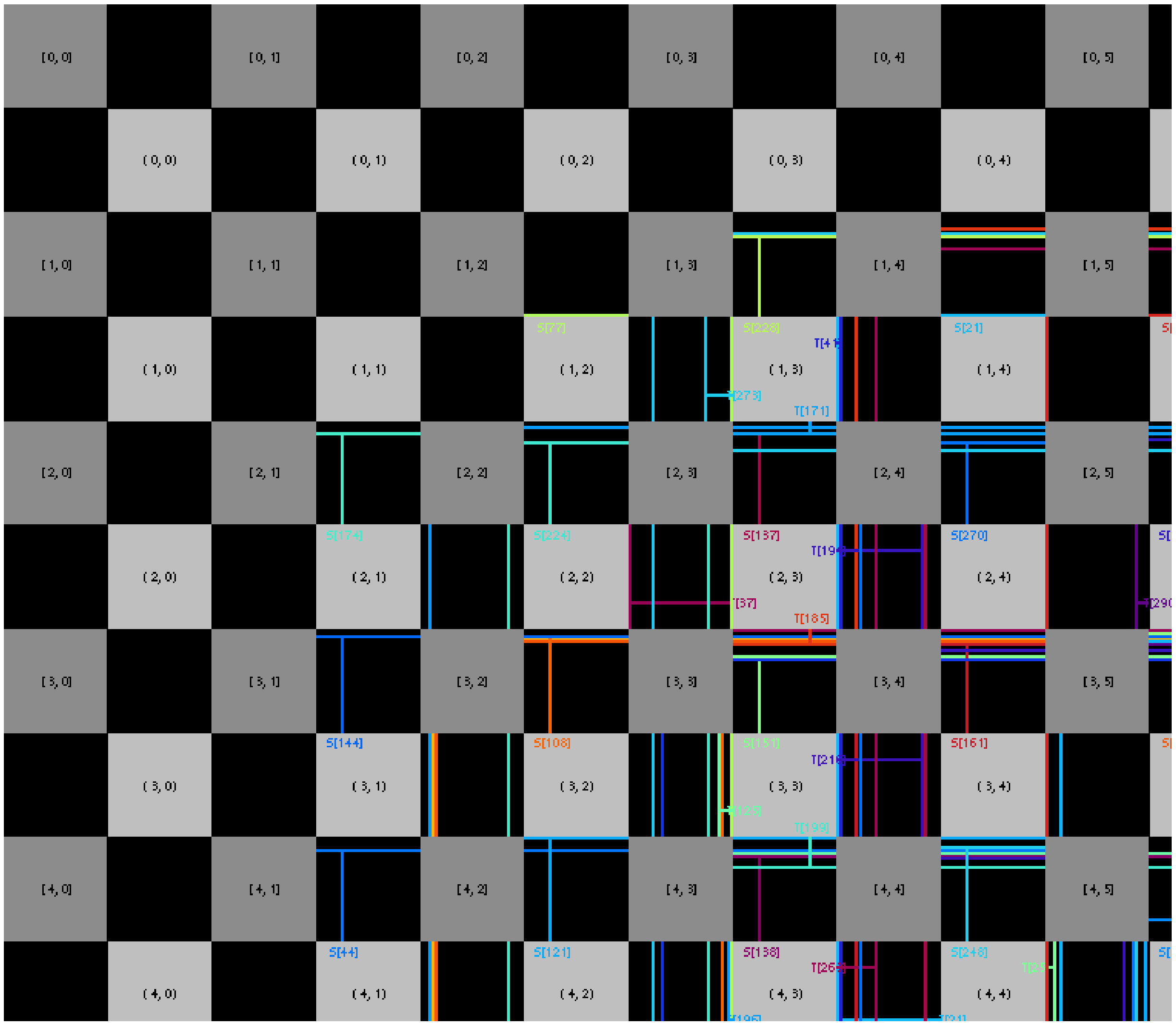

**Figure 8:** Output of CCT4 With Wilton Style Switch, zoomed on top left hand, **Segments used: 3354**

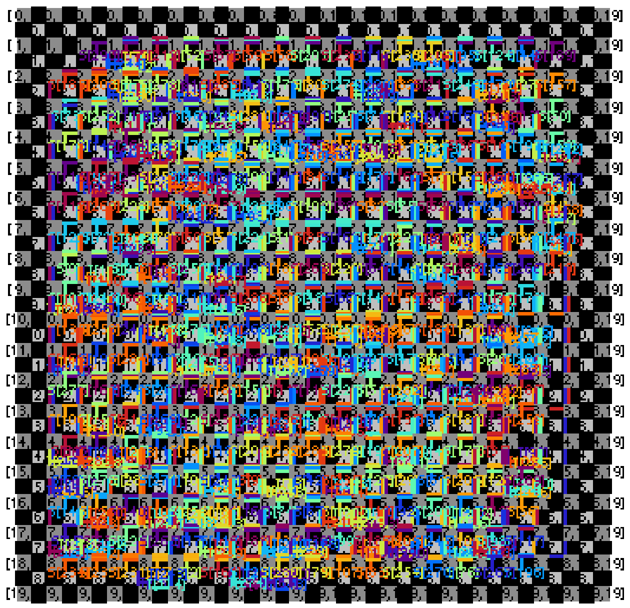

**Figure 9:** Output of CCT4 With Fully Connected Style Switch, total view, **Segments used: 3060**

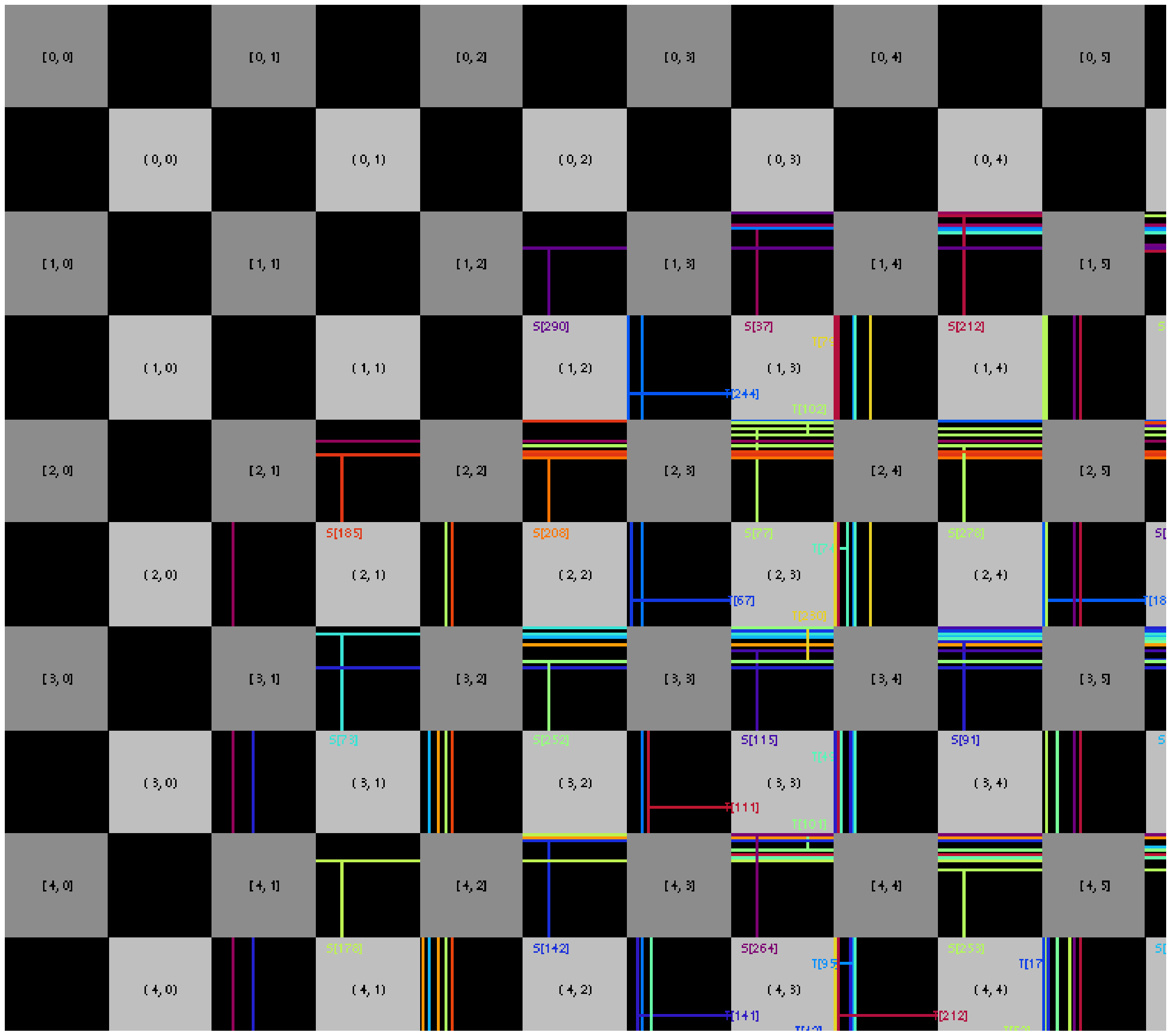

**Figure 10:** Output of CCT4 With Fully Connected Style Switch, zoomed on top left hand, **Segments used: 3060**

# Bibliography

[1] C. Y. Lee. An algorithm for path connections and its applications. *IRE Transactions on Electronic Computers*, EC-10(3):346–365, Sept 1961.

[2] M Imran Masud and Steven JE Wilton. A new switch block for segmented fpgas. In *International Workshop on Field Programmable Logic and Applications*, pages 274–281. Springer, 1999.