University of Alberta

TEST AND DIAGNOSTIC SCHEMES FOR LOCAL INTERCONNECTS OF XILINX VIRTEX FPGAs

by

Christian Giasson

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Electrical and Computer Engineering

Edmonton, Alberta
Fall 2004

Être homme, c'est précisément être responsable. C'est connaître la honte en face d'une misère qui ne semblait pas dépendre de soi. C'est être fier d'une victoire que les camarades ont remportée. C'est sentir, en posant sa pierre, que l'on contribue à bâtir le monde.

– Antoine de Saint-Exupéry, Terre des hommes

To be a man is, precisely, to be responsible. It is to feel shame at the sight of what seems unmerited misery. It is to take pride in a victory won by one's comrades. It is to feel, when setting one's stone, that one is contributing to the building of the world.

– Antoine de Saint-Exupéry in *Wind, Sand, and Stars*

À Eve-Lyne, pour son soutient quotidien.
À Diane et Charles-André, pour toute leur éducation.
À Marie-Hélène, pour être un bon modèle.

Acknowledgements

This work was partly supported by the Natural Sciences and Engineering Research Council of Canada, by the Informatics Circle of Research Excellence, by the Canadian Microelectronics Corporation, and by the University of Alberta. I would like to give special thanks to my supervisors Xiaoling Sun and Bruce Cockburn for their guidance and assistance. Finally, I would like to thank all my student colleagues from the VLSI design and test laboratory at the University of Alberta for all their help and support.

Table of Contents

List of Figures

List of Tables

List of Acronyms

Acronym	**Significance**
BIST	built-in self-test
CLB	configurable logic block
CUT	circuit under test
DFF	D-type flip-flop
FPGA	field-programmable gate array
GRM	general routing matrix
IC	integrated circuit
IMUX	input multiplexer
IOB	input/output block
IRM	input routing matrix
OL	output line
OMUX	output multiplexer
ORA	output response analyzer
ORM	output routing matrix
PS	programmable switch
RLE	run-length encoding
TC	test configuration
TPG	test pattern generator
WG	wire group
WUT	wire under test

Chapter 1

Introduction

SRAM-based *field-programmable gate arrays* (FPGAs) are programmable logic devices widely used in digital systems. An FPGA consists of a two-dimensional $m \times n$ array of *tiles* and *input/output blocks* (IOBs). Each tile contains *configurable logic blocks* (CLBs) and interconnects surrounding them. Interactions between CLBs and IOBs are possible with a system of programmable interconnect resources composed of *programmable switches* (PSs) and *wire segments.* Figure 1.1 shows a general representation of an FPGA structure, where $m = 3$ and $n = 4$.

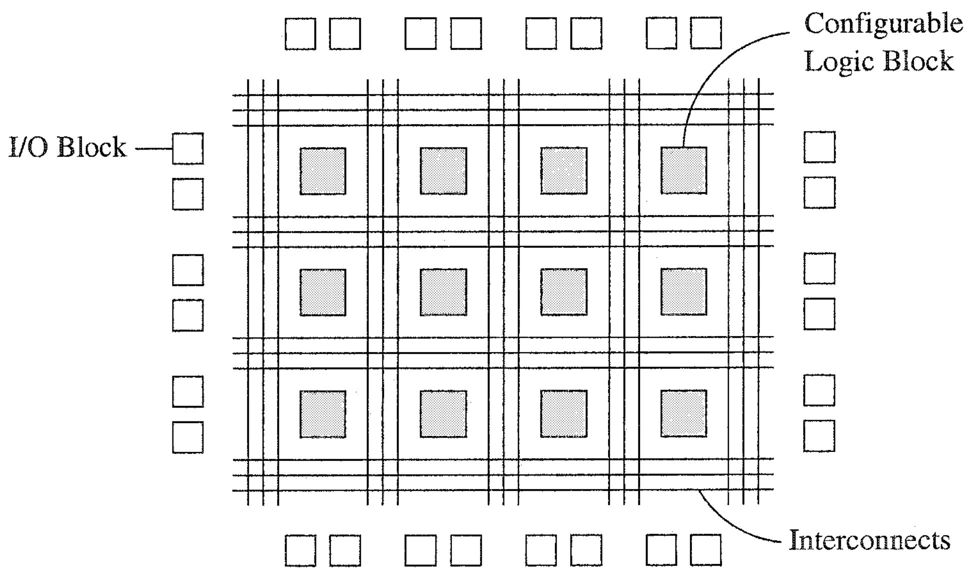


Figure 1.1: FPGA structure

FPGA interconnects can be divided into two categories: *global* and *local* *interconnects.* The global interconnect structure provides programmable resources to enable the routing of signals through the FPGA without entering the CLBs. On the other hand, the local interconnects provide possible routing between the global interconnects network and the CLBs.

The Xilinx Virtex group of FPGAs was the first million-gate FPGA device family. The interconnect structure of these devices is relatively irregular and more complex than earlier device families, such as the XC4000. Approximately 80% of all SRAM cells in a Xilinx Virtex FPGA are dedicated to interconnects, which consist of *programmable switches* (PSs) and wire segments. Local interconnects account for 65% of interconnect PSs.

Like other *integrated circuits* (ICs), the integrity of FPGAs needs to be verified during the manufacturing process and also when it is incorporated in an electronic design (in-system). Therefore test strategies should be developed to achieve appropriate testing of the device.

1.1 Thesis Scope

Testing is an important step for the design and manufacturing of integrated circuits. An FPGA will be tested several times in its life cycle (e.g.: manufacturing and in-system tests). In this thesis, we will focus on the local interconnects test of one specific familly of FPGA: the Xilinx Virtex. It is important to mention that this thesis work is applicable to the Xilinx Virtex-E and Virtex-EM family as well as the Xilinx Spartan-II and Spartan-IIE FPGAs since their structure is very similar to that of the Virtex family.

The *test strategy* should be able to verify the integrity of all local interconnects in the FPGA. The re-programmability of the FPGA allows a test strategy with more than one *testing configuration* (TC). The overall test that will be applied to the FPGA should be as fast as possible to minimize the test duration. We know, from experience, that the FPGA programming time is more time-consuming than the test performed by every TC. Therefore, an important goal to achieve is to obtain the minimum number of TCs.

Therefore an adequate interconnect model as well as an algorithm should be developed to generate a minimum set of TCs. The proposed interconnect modeling is based on graph theory. Basically, vertices model the wire segments and the edges of the graph are the PSs. The resulting graph is a k-partite graph that represents k wire segment groups. An edge coloring algorithm was developed to generate a set of TCs. Our experiments on Virtex FPGAs obtain a minimal set of 26 TCs.

The proposed testing scheme is applicable to both manufacturing tests and *built-in self tests* (BISTs). BIST is a popular test architecture which places all the test circuitry in the chip. The test scheme uses some new architectural features of the Virtex FPGA such as the RAM blocks.

When a circuit has failed a test, it is important to know the location of the faults detected in the integrated circuit. Diagnostic tests can identify faulty parts that should be replaced or avoided. Fault diagnosis is essential for fault tolerant systems and helpful for yield improvement in a manufacturing environment. Therefore, the proposed test scheme can be extended to support diagnostic test by saving test responses on the FPGA. When the diagnostic test is over, data are is sent out of the FPGA for further analysis.

1.2 Thesis Organization

This thesis is divided into six chapters organized as follows. Chapter 2 explains some important concepts related to the FPGA architecture. It also presents a literature review related to this thesis. This review shows where the work of this thesis can be included relative to previous work in the field. The BIST test scheme used in this project is presented in Chapter 3. It also presents the fault models, the test vectors, and proofs of testability. Chapter 4 introduces a graph model for the FPGA interconnects. It also explains the algorithm, based on the edge coloring of bipartite graphs, that finds the TCs. The modeling and the algorithm are then applied to a simplified FPGA resources example to demonstrate how they work to find the TCs. More details about the algorithm and experiments with the Virtex FPGA resources are also presented. Chapter 5 presents two other schemes that can perform diagnosis on the Virtex FPGA interconnects. These schemes are based the test scheme shown in chapter 3. The two diagnostic schemes are useful for two different applications: fault tolerance and manufacturing. Finally, Chapter 6 concludes with a brief review of the thesis, a summary of its important achievements, and possible further work.

Chapter 2

Background

This chapter first presents a background section on FPGA architecture. It includes the general architecture, the logic, and the interconnects of the Xilinx Virtex FPGA. A literature review is also presented. It presents different published strategies for testing FPGAs.

2.1 FPGA Architecture

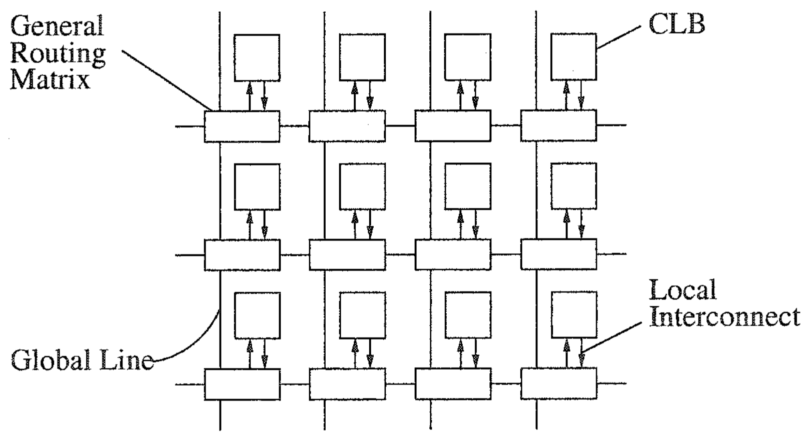


Figure 2.1: FPGA interconnect system

An FPGA typically consists of a two-dimensional array of uncommitted *configurable logic blocks* (CLBs) surrounded by *Input/Output* (I/O) blocks. A system of programmable interconnects is used to route signals between CLBs to create an electrical circuit. The programmable interconnect network can also connect the previously mentioned circuit to I/O blocks to allow external

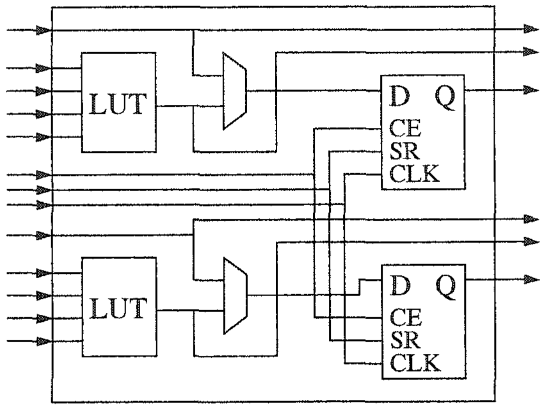


Figure 2.2: CLB logic

control and observation. Figure 2.1 shows a simplified diagram of the regular part of the FPGA made of CLBs and interconnects. This architecture, called cluster-based, is the one used in the Xilinx Virtex FPGA.

Every CLB includes several components such as *look-up tables* (LUTs), multiplexers, and storage elements. In the Virtex FPGA, each LUT can be configured as a 16×1-bit synchronous RAM or ROM as well as any asynchronous 4-input logic function. The storage elements can also be configured in two modes: *D-type flip-flops* (DFFs) or level-sensitive latches. Figure 2.2 shows a simplified representation of the Virtex CLB logic.

The interconnect network is made of *wire segments* and *programmable switches* (PSs). A PS is made of one SRAM cell and a pass transistor. PSs are mostly concentrated in some area in the FPGA called *switching matrices.* FPGA interconnects can be divided into two categories: *global* and *local interconnects.* The global interconnect structure provides programmable resources to enable the routing of signals through the FPGA without entering the CLBs. On the other hand, the local interconnects provide possible routing between the global interconnect network and the CLBs.

Figure 2.3 shows more details of Virtex local interconnects by presenting a simplified FPGA tile. The *input routing matrix* (IRM) links the *general routing matrix* (GRM) with the CLB's input lines. On the opposite side of the CLBs, the *output routing matrix* (ORM) helps to link output lines of the CLBs

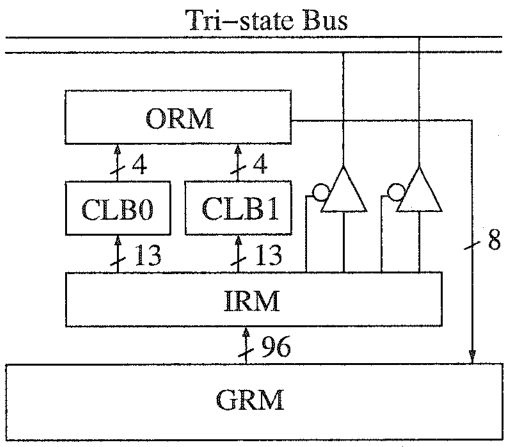


Figure 2.3: Local interconnects

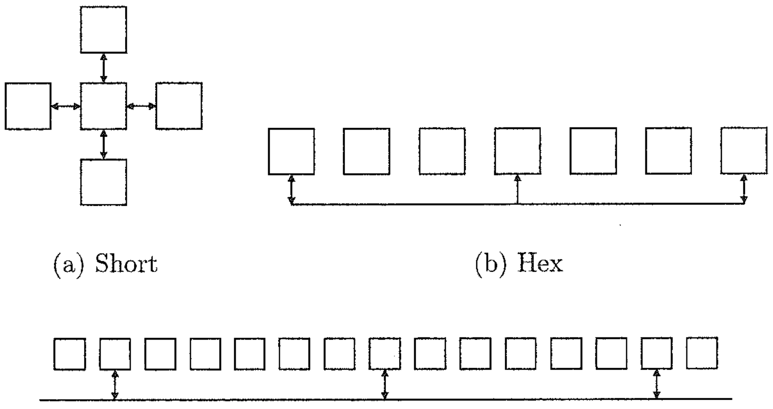


Figure 2.4: Global lines

with the GRM. Note that there are two tri-state buffers in every tile linked to a tri-state bus that crosses the FPGA horizontally. Every row of tiles has a tri-state bus made of four lines. Tri-state buses act like MUX buses where many tri-state buffers are allowed to be linked to a line but only one can put its value on the line at a time.

There are different groups of global interconnects classified by their lengths. Figure 2.4 shows the different kinds of global lines: *short* (a), *hex* (b), and *long* (c) lines. Short lines link the GRM with all its orthogonal GRM neighbors in the next tiles. Hex lines can connect two GRMs that are six tiles away horizontally or vertically. Hex lines can be driven only at their extremities but they can be read either at their extremities or their midpoints (three tiles from the source). The long lines can route signals vertically and horizontally and they span the entire height and width of the FPGA. Their connection points to GRMs are situated every six tiles as shown in Figure 2.4 (c).

Different techniques have been tried to test FPGAs. The following sections will present different test techniques and schemes related to this thesis.

2.2 Current Measurement Techniques

The most popular current measurement technique to test CMOS devices is I_{DDQ} testing [1]. This technique consists of monitoring the *steady state supply current* (I_{DDQ}) to detect abnormally high currents during the steady state in CMOS circuits. I_{DDQ} testing can detect low level faults such as transistor stuck-open, transistor stuck-closed, transistor gate oxide shorts, interconnect bridging shorts, and unpowered interconnect opens.

The I_{DDQ} testing technique requires a lot of equipment. Because of this disadvantage, that technique is mostly used in manufacturing test and not in field test when the integrated circuit is in a system. Furtermore, it takes a significant amount of settling time to get accurate current measurements.

Most of the work on FPGA I_{DDQ} test was done by Zhao *et al.*. They applied I_{DDQ} techniques to test different parts of the FPGA such as the Input/Output resources [2], the logic resources [3], and the interconnects [4].

2.3 Voltage Measurement Techniques

Most of the testing on integrated circuits uses voltage measurement techniques. These techniques measure logic responses at certain points in the circuit to detect potential faults. Voltage measurement techniques are widely used for FPGA testing.

Because of FPGA's programming flexibility, not all the FPGA resources can be programmed in one circuit configuration. The detectability of a fault therefore depends strongly on the circuit implemented in the FPGA. For example, a particular fault can be *redundant* in a particular FPGA configuration and consequently cannot be detected within that configuration. Therefore, most of the FPGA test schemes apply tests successively on a set of configurations to verify the integrity of all resources.

Presently, the two main voltage measurement strategies to test FPGAs are the device testing technique and the *built-in self-test* (BIST) technique.

2.3.1 Device Testing Techniques

The device testing techniques generally need full access to the I/O pins of the integrated circuit. In this kind of test, the test vectors are applied in parallel from outside the chip and the test results are read at other dedicated output pins. This kind of test is often used in manufacturing testing.

This technique can be used to test global and local interconnects in FPGAs. In [5, 6, 7, 8, 9, 10, 11, 12], device testing is used to test global interconnects. In the majority of these research papers, the basic idea is to configure the global interconnects to form *long buses* and then test them by applying classical bus testing vectors. The test vectors are applied to some chip I/Os and the test responses are subsequently read from other I/Os. Figure 2.5 shows an example of *test configuration* (TC) for the global interconnects, where multiple test buses are formed, and all the general routing matrices are configured in the same way to create diagonal routings in the FPGA. The ends of these buses are connected to I/O Blocks.

The general idea when testing local interconnects is to apply signals to the CLB inputs and propagate them to the outputs. The approaches in [13, 14] supply signals to the combinational and sequential logic programmed in the CLBs. Figure 2.6 shows the general idea of this methodology. In this kind of scheme, some CLB outputs are linked to their neighboring inputs to create a chain of logic functions. Testing signals are applied to all CLBs from an external source and test responses from each CLB logic are passed on to the next CLB on the right. The advantage is that the number of I/Os to observe

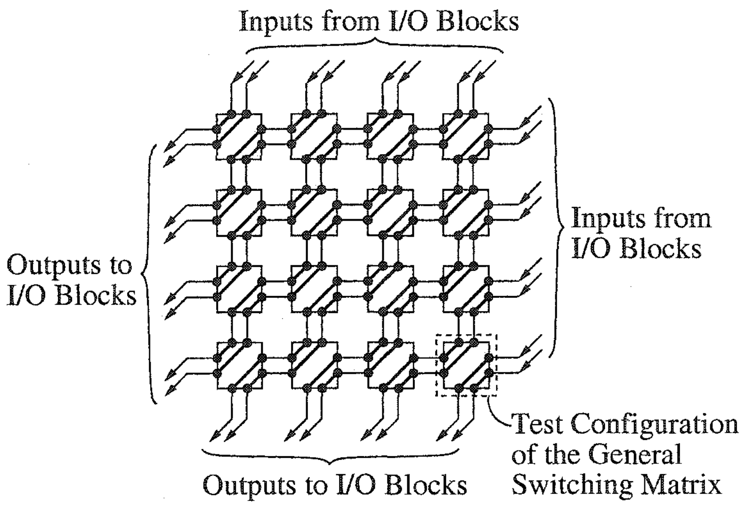


Figure 2.5: One global interconnect test configuration

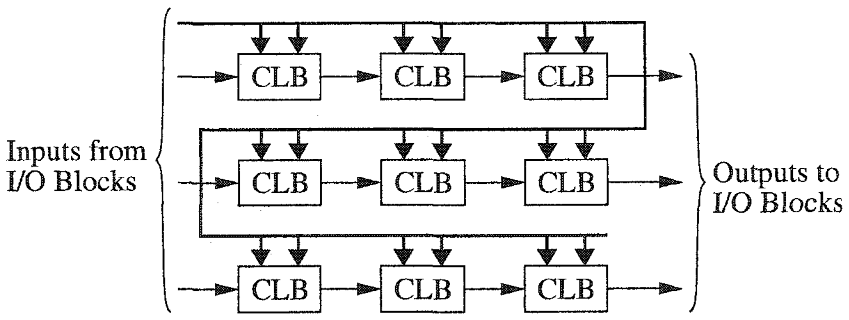


Figure 2.6: A local interconnect test strategy

is less than the number of test responses produced by all CLBs. Because of the intensive use of I/Os in these schemes, they are classified as device testing techniques.

Doumar *et al.* proposed a modified FPGA architecture in [15] to test FPGA logic cells and interconnects. The new design modified the SRAM part of the FPGA to allow faster loading of the test configuration data. The idea is to use the redundancy in a test configuration to program the FPGA in parallel. Since the research focus of this thesis is on a fixed design of FPGA, the technique presented by Doumar *et al.* is not applicable.

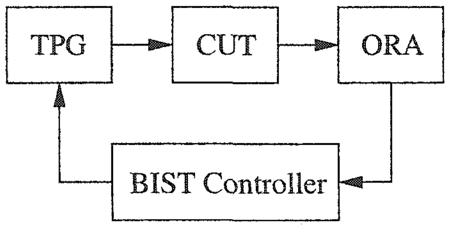


Figure 2.7: BIST architecture

2.3.2 Built-In Self-Test Technique

A *built-in self-test* (BIST) strategy is used when the test interface has a limited access to the I/O pins of an integrated circuit. For example a BIST scheme can be useful to test a chip already implemented on a *printed circuit board* (PCB) or in a bigger system. This kind of test is called a field-test. A BIST scheme is essentially composed of four parts: a BIST controller, a *test pattern generator* (TPG), an *output response analyzer* (ORA), and a *circuit under test* (CUT). Figure 2.7 shows the general architecture of a BIST scheme.

The first BIST approach to test FPGA interconnects was developed by Stroud *et al.* in [16]. This scheme tests the interconnects by configuring *programmable switches* (PSs) to form two groups of *wires under test* (WUTs). The two WUTs receive identical test vectors from the same group of CLBs configured as a TPG. On the other end of the WUTs an ORA compares the results from the two WUTs. This technique supports a good fault coverage for many kind of faults, but it fails to detect multiple faults that make the two WUTs behave in the same faulty way.

In [17], Abramovici *et al.* present an on-line FPGA testing and diagnosis method. Every test configuration (TC) is localized on only one column and one row of the FPGA. Since that represents only a small part of the overall FPGA area, the FPGA can still perform its function during the test. The entire FPGA can be tested by moving the test row and the test column.

In [18, 19, 20, 21, 22], Sun *et al.* use error control coding to detect faulty behavior in interconnects. In the overall scheme, the FPGA is split into two equal parts: one with the CUT and another one with the BIST controller, the

TPG, and the ORA. The idea is to configure interconnects into WUTs with a parity wire. These WUTs are configured in the CUT part of the FPGA. A TPG supplies test vectors as well as a parity bit to the WUTs. At the other end of the WUTs (or at some intermediate point on the WUTs), a parity code generator regenerates the parity and a XNOR gate compares the parity from the parity wire of the WUTs and the generated one. This scheme can achieve very high multiple fault coverage.

2.4 Interconnects Testing

2.4.1 Fault Models

Fault models are the abstract representation of physical defects that can appear in an electronic system, produced by process defects, material defects, age defects, and package defects. Having a fault model is necessary to do structural testing on any kind of integrated circuit.

For the interconnects testing, the typical fault models include wire segment stuck-at 0 or 1, segment stuck-open, and segment bridging. They also include programmable switch stuck-on (stuck-closed) and stuck-off (stuck-open). Many fault models also allow the multiple appearance of the faults (e.g. multiple stuck-at 0/1 faults). Fault models sometime include combinations of different kinds of faults which can be present at the same time.

Some research has been done on more exotic types of faults. Metra *et al.* investigated an FPGA test to detect *transient* and *crosstalk* faults affecting interconnects in [23]. Abramovici and Stroud, in [24], presented a method to detect *delay-faults* in FPGAs.

2.4.2 Testing

The testing of FPGA interconnects can be divided into global interconnect testing and local interconnect testing categories. Prior work on FPGA interconnect testing mainly focuses on global interconnect testing [5, 6, 7, 8, 9, 10, 11, 12, 18, 19]. In [14, 20, 22] a local interconnect test scheme is presented and in [13, 16, 17, 21, 25, 26] a combined local/global scheme is explained.

In the previous section, different test architectures were presented. Two other steps that rely on the test architecture are needed to derive a completely functional test. First, a modeling of the interconnects should be done to have a certain level of abstraction. This modeling allows the use of an algorithm that can configure the programmable interconnects in test configurations. This thesis focuses on the FPGA interconnect modeling and the aforementioned algorithm.

A majority of the test methodologies use reprogrammability of the FPGA to achieve high fault coverage with a set of *test configurations* (TCs). It is desirable to minimize the number of TCs since programming the FPGA is more time consuming than the test application time. Some research has been devoted to optimization problems aimed at deriving a minimal number of TCs. Heuristic approaches to these problems can be found in [12, 20, 21, 26].

A number of graph models have been used to model FPGA interconnects and derive interconnect TCs. In [20], an adjacency graph is employed to represent local interconnects, where a vertex denotes a PS and an edge represents a test requirement between two PSs. TCs are derived by solving a vertex coloring problem of the graph. A global interconnect bipartite graph model, where a vertex represents a wire segment and an edge represents a PS, is given in [26] for maximum fault coverage and diagnosability. In [12], both wire segments and PSs of global interconnects are modeled by edges. Source and drain vertices are introduced to the graph and a maximum flow algorithm is used to derive TCs. However, the choice of a routing path through the graph is random and, thus, constraints on using particular nodes for the source and destination cannot be accommodated. Conversely, the algorithm in [27] supports choices on source and destination nodes when forming routing paths, but does not guarantee complete coverage of edges. A graph model is required to represent the interconnect resources.

2.4.3 Diagnosis

In a test procedure, the *diagnosis* is the action of identifying where a detected fault is located.

The role of testing is to detect whether something went wrong. On the other hand, the role of diagnosis is to determine exactly what went wrong. Diagnosis is essential to fault tolerance techniques because it makes it possible to avoid faulty parts of an integrated circuit and thus improve the yield in the manufacturing process. Diagnosing a fault is usually more difficult and requires more programming steps than simply detecting the presence a fault.

Some research has been done on diagnosing faults in wiring interconnects on *printed circuit boards* (PCBs) by Park in [28], by Cheng *et al.* in [29], and by Hassan *et al.* in [30]. They used boundary scan design techniques to test the interconnects between chips on a PCB. Those techniques cannot be directly applied to the FPGA interconnect fault diagnosis because the architecture is different. However, the diagnosis test vector sets presented in these papers can be reused for diagnosing faults in an FPGA interconnect network.

Huang *et al.* in [5], Yu *et al.* in [6, 7], and Liu *et al.* in [31] worked on the diagnosis of FPGA interconnect faults. In these papers, *fine* resolution diagnoses are presented. Tahoori, in [32], presents two diagnostic test schemes for FPGA interconnects: a coarse-grain and a fine-grain diagnosis. The coarse-grain diagnosis is first applied to the interconnects. If a fault is found, a fine-grain diagnosis is applied to determine where the fault is with more precision.

Chen *et al.* worked on interconnects in general in [33]. They developed a test vector derivation based on graph coloring techniques. In this methodology, the adjacencies between nets are known.

Chapter 3

The Proposed Test Scheme

Built-in self-test (BIST) techniques add overhead to digital circuits but have great advantages such as improved testability compared to other testing techniques, support of at-clock-speed test of modules, and reduced need for expensive *automatic test equipment* (ATE). BISTs also allow for *field test*, where the chip under test is in an assembled system.

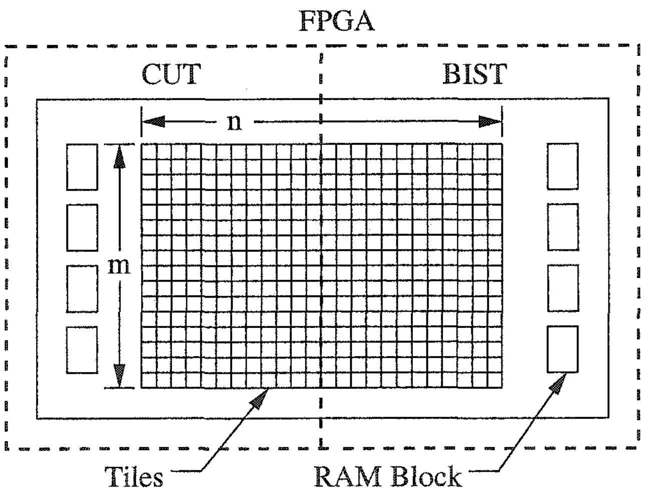


Figure 3.1: Simplified representation of the Virtex FPGA

The proposed test strategy for FPGA local interconnects is a BIST strategy. The overall BIST strategy uses the re-programmability of the FPGA to perform a full test on the local interconnects. The test strategy is composed of a set of test programs called *test configurations* (TCs). In every TC, the FPGA resource is divided into two parts: the *circuit under test* (CUT) and

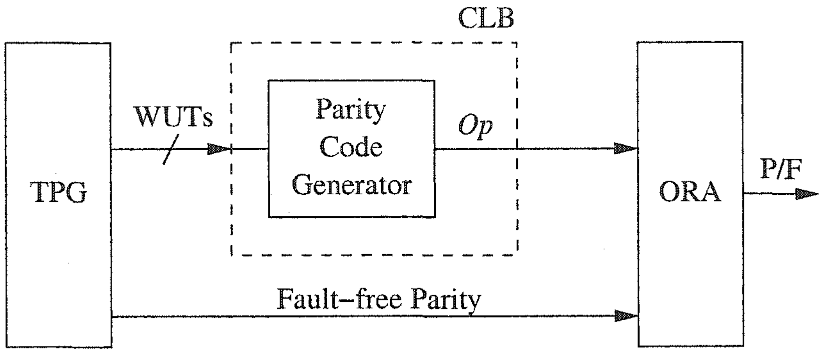


Figure 3.2: Conceptual block diagram

the BIST circuit.

Figure 3.1 shows a FPGA with $m \times n$ tiles. If a total of k TCs are needed to test the local interconnects of one half of the FPGA, the total number of TCs is $2k$ because all TCs need to be applied on both left and right parts.

Figure 3.2 shows the conceptual block diagram of the proposed BIST strategy. A portion of wire segments and programmable switches configured together is called a *wire group* (WG). The wire segments that are part of a WG in one TC are called *wires under test* (WUTs). One extremity of a WUT should be connected to a CLB. Because most of the local interconnects help to route signals in and out of the CLBs, the testing signals have to pass through the CLBs. This is why the testing scheme uses the CLB logic to perform test calculations and is able to get relatively parallel test on every tile in the CUT.

During the test sequence, the WUTs receive test vectors from the *test pattern generator* (TPG). The TPG also supplies a fault-free parity directly to the *output response analyzer* (ORA). Inside each CLB, the parity code generator calculates the *output parity* (O_p). The O_p is sent to the, ORA which compares the O_p with a fault-free parity and gives a pass/fail result.

The parity calculator test scheme, combined with carefully chosen test vectors, has a high fault coverage. The next section presents more details on the proposed BIST scheme. It also gives the fault models, assumptions, the proposed test sets, and the proof of fault detectability.

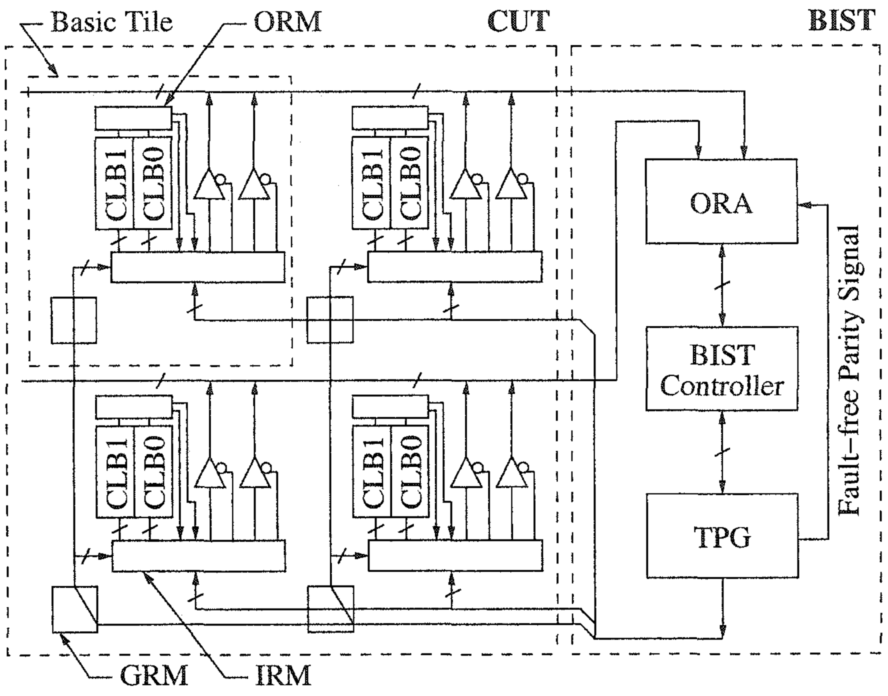


Figure 3.3: Proposed BIST architecture

3.1 Proposed BIST Architecture

Figure 3.3 shows a simplified representation of the proposed BIST architecture. As previously mentioned, the Virtex FPGA is split vertically in two parts: the CUT and the BIST. The CUT part is represented as a 2 x 2 array of *basic tiles* in Figure 3.3. Three major parts are in the BIST section: the BIST controller, the TPG, and the ORA. The BIST controller controls the test sequence. The TPG generates the vectors and supplies all the tiles in the CUT via the global interconnects. The TPG also generates the fault-free parity code and gives it directly to the ORA. The *output parity* (O_p) signals, calculated by the CLBs, are put on the tri-state buses. Since the tri-state buses are horizontal in the Virtex FPGA, and the scheme uses them to carry the test responses, the CUT and the BIST should be side by side horizontally. The ORA receives all the test responses from the tri-state buses and compares them to the fault-free parity code generated by the TPG. Depending on the result of the comparison, the ORA gives a passed or failed result to the BIST controller.

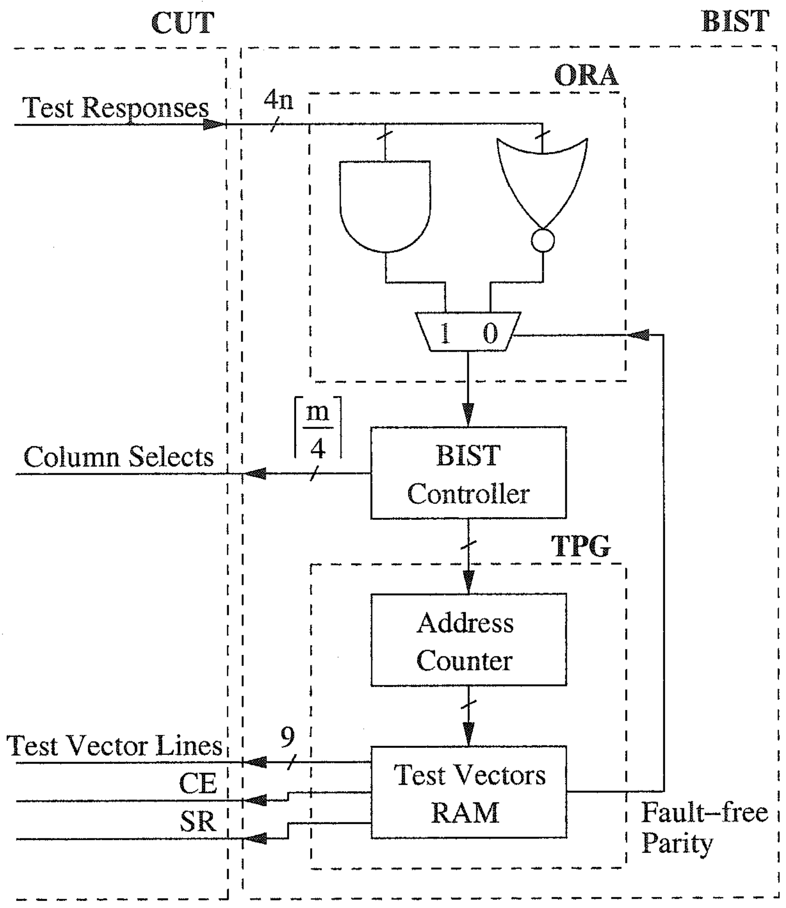


Figure 3.4: BIST block diagram

Figure 3.4 shows a block diagram of the BIST part of the test configuration. The central BIST controller is a state machine that controls the test sequence.

The implementation of a TPG is usually done using only the CLB memory resources. Since the Virtex FPGA family has some *random-access memory blocks* (RAM blocks), they can be used with a counter to create a TPG that will save some CLB programming resources for the rest of the BIST circuit. In the present BIST scheme, a binary counter is tied to the address of the RAM block to create the TPG. Test vectors, two control signals (CE and SR), and a fault-free parity signal are stored in the RAM. A new test value is applied every time the counter increments.

The ORA compares the test responses from the tri-state buffer with the fault-free parity code from the TPG. AND and NOR logic gates are used as

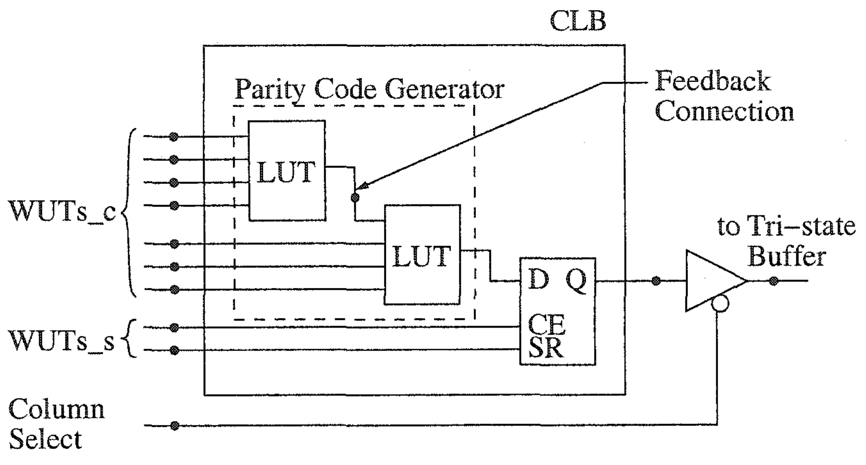


Figure 3.5: Test logic

a comparator. If the value of the fault-free parity is a logic *1*, the AND gate is selected to have a faulty output signal when at least one input is equal to *0*. If the results (parity responses) from the CUT are supposed to be *0*s, the NOR logic gate is used for the comparison. The ORA gives a passed or failed result to the BIST controller at the end of the process.

In the CUT part of the TC, the global interconnects help to route signals from the BIST controller and the TPG. Thus, the Long, Hex and Short lines, and the dedicated clock lines supply the test vectors and the control signals (column selects) to every tile. Figure 3.3 shows some characteristics of a basic tile. In every tile, the two CLBs receive the test vectors via the *general routing matrix* (GRM) and the *input routing matrix* (IRM). The CLBs perform some test logic and also latch the results. The result is then passed though the *output routing matrix* (ORM) and the tri-state buffers put the two results that come from both CLBs on the tri-state lines. The tri-state buffers are used to pass the test results back to the BIST for two main reasons. First, some local interconnects are connected to the tri-state buffer input and control, and the tri-state buffer should therefore be included in the local test logic. The second reason is that using the tri-state buffer buses allow good observability of the test responses without the problem of routing many test responses independently back to the BIST part of the FPGA.

The tri-state buses cannot propagate all the test responses in parallel because there is a limiting quantity of four tri-state lines per row of tiles. There is a need for control signals that can select which test responses are on the tri-state buses. That is why the BIST controller interacts directly with the CUT part with some column select signals. The number of column select signals is dependent on the size of the FPGA. Since the CUT takes half of the FPGA columns ($m/4$) and two columns of tiles can share the same select signal, the number of column select signals is $C = \lceil m/4 \rceil$. Note that the ceiling bracket in the function C come from the fact that $m/4$ can be a non-integer.

Figure 3.5 presents in more detail the testing circuit implemented in all the CLBs in every tile of the CUT part of the FPGA. It can be seen that the WUTs are subdivided in two categories. The WUTs that are connected to the combinational logic are the *combinational wire segments under test* WUTs_c. Similarly, the WUTs connected to the flip-flop are the *sequential wire segments under test*, WUTs_s. The WUTs_s supply only two signals: CE and SR. The column select is also part of the WUTs, but it supplies the tri-state buffers instead of the CLBs.

The parity code generator is implemented in the CLB with LUTs. The biggest parity code generator (XOR logic gate) that can be implemented in one CLB has seven inputs. The parity code generator is implemented with two LUTs and needs a feedback path via the ORM and the OUT lines. The parity signal is latched by a D flip-flop in the CLB to help the timing of the test.

3.2 Fault Models and Assumptions

To test the lines, typical fault models for interconnect networks are used [34]:

1. Multiple segment *stuck-at* 0/1 faults,
2. Multiple segment *stuck-open* faults,
3. Multiple segment/programmable switch *bridging* faults,

4. Multiple programmable switch *stuck on/off* faults,
5. Combinations of the above faults.

Some assumptions should be made in order to use the previous fault models:

Assumption 1 *Suppose there is a stuck-open fault on a wire. Applying a logic value of 0 or 1 at one end will generate a logic value of 0 at the other end.*

This assumption is realistic even if a stuck-open fault can generate a floating voltage at the *output* end of the fault because that voltage will be interpreted by the next testing logic block as a logic low or a logic high.

Assumption 2 *The floating voltage caused by a bridging fault will be either logic 0 or 1. These create a wired-AND or a wired-OR respectively.*

The behavior of a bridging fault is a function of the driving capacity of the wires involved. In this fault model we assume that this type is deterministic. We consider the bridging fault to be deterministic because we assume that it will react as either a wired-AND or a wired-OR function.

Assumption 3 *A switch stuck-on fault creates the same behavior as a bridging fault between two wire segments connected by the switch.*

The switch stuck-on fault will have the same behavior as a bridging fault. Therefore, if the test can detect the bridging faults it can also detect the switch stuck-on faults.

Assumption 4 *The components of the CLBs (LUTs, lines, MUX, memory elements, ...) are fault-free.*

Finally, we assume that the CLB logic is fault-free in order to target the faults on the interconnects.

Table 3.1: Test 1 and Test 2

		Test 1		**Test 2**		
		$v_1v_2v_3v_4$	$v_5v_6v_7v_8$	$v_1v_2v_3v_4$	...	$v_{125}v_{126}v_{127}v_{128}$
WUTs_c	e_1	0 0 0 0	1 1 1 1	0 1 0 1		0 1 0 1
	e_2	0 0 0 0	1 1 1 1	0 0 1 1		0 0 1 1
	e_3	0 0 0 0	1 1 1 1	0 0 0 0		1 1 1 1
	e_4	0 0 0 0	1 1 1 1	0 0 0 0	...	1 1 1 1
	e_5	0 0 0 0	1 1 1 1	0 0 0 0		1 1 1 1
	e_6	0 0 0 0	1 1 1 1	0 0 0 0		1 1 1 1
	e_7	0 0 0 0	1 1 1 1	0 0 0 0		1 1 1 1
WUTs_s	CE	0 0 1 1	0 0 1 1	1 1 1 1	...	1 1 1 1
	SR	0 1 0 1	0 1 0 1	0 0 0 0	...	0 0 0 0

Table 3.2: Test 3

		Test 3						
		v_1v_2	v_3v_4	v_5v_6	v_7v_8	v_9v_{10}	$v_{11}v_{12}$	$v_{13}v_{14}$
WUTs_c	e_1	1 1	0 0	0 0	0 0	0 0	0 0	0 0
	e_2	0 0	1 1	0 0	0 0	0 0	0 0	0 0
	e_3	0 0	0 0	1 1	0 0	0 0	0 0	0 0
	e_4	0 0	0 0	0 0	1 1	0 0	0 0	0 0
	e_5	0 0	0 0	0 0	0 0	1 1	0 0	0 0
	e_6	0 0	0 0	0 0	0 0	0 0	1 1	0 0
	e_7	0 0	0 0	0 0	0 0	0 0	0 0	1 1
WUTs_s	CE	0 1	0 1	0 1	0 1	0 1	0 1	0 1
	SR	1 0	1 0	1 0	1 0	1 0	1 0	1 0

3.3 Proposed Test Sets

In the BIST architecture presented in section 3.1, there are seven wires in the WUTs_c and the WUTs_s includes two signals: CE and SR. The test vector sequence proposed here is composed of tree tests. Table 3.1 and Table 3.2 show Tests 1 to 3.

1. Test 1 is composed of exhaustive logic vectors on CE and SR with all 0s and all 1s applied on the WUTs_c consecutively.

2. Test 2 is composed of exhaustive patterns on WUTs_c with CE at 1 and SR at 0.

3. Test 3 is a repeating sequence of two vectors (0,1) and (1,0) for (CE,SR) with *walking 1s* on the WUTs_c.

Each of the proposed tests is designed to detect a subset of the modeled faults described in subsection 3.2. Test 1 is capable of detecting multiple segment-stuck-at 0 (stuck-open) faults, segment-stuck-at 1 faults and switch-stuck-off faults in the WUTs_s, and the multiple-wire bridging faults (switch-stuck-on faults) within WUTs_s. Test 2 detects the same kind of faults but on WUTs_c. Test 3 detects the multiple-wire bridging faults between the WUTs_c and WUTs_s.

Prior to starting any of the previously mentioned test sequences, the *D-type flip-flop* (DFF) must be initialized. If DFFs are all in reset mode, then any test vector where CE = 0 and SR = 1 will reset them. It is assumed that all DFFs are initialized to a logic 0 before any set of test vectors.

3.4 Fault Detectability

To prove that Test 1 can detect all the faults mentioned earlier on CE and SR, we enumerated all the possible cases. Table 3.3 shows the Q response with all the possible cases that are parts of the fault models. Bold values in the table are faulty responses compared to the corresponding fault-free response.

Table 3.3: Q responses to Test 1 vectors

Condition of the Wire Group	Q_{v1}	Q_{v2}	Q_{v3}	Q_{v4}	Q_{v5}	Q_{v6}	Q_{v7}	Q_{v8}
Fault-free	0	0	0	0	0	0	1	0
CE s-a-0	0	0	0	0	0	0	**0**	0
CE s-a-1	0	0	0	0	**1**	0	1	0
SR s-a-0	**1**	**1**	0	0	0	0	1	**1**
SR s-a-1	0	0	0	0	0	0	**0**	0
CE s-a-0, SR s-a-0	0	0	0	0	0	0	**0**	0
CE s-a-0, SR s-a-1	0	0	0	0	0	0	**0**	0
CE s-a-1, SR s-a-0	0	0	0	0	**1**	**1**	1	**1**
CE s-a-1, SR s-a-1	0	0	0	0	0	0	**0**	0
CE bridges with SR (wires-AND)	0	0	0	0	0	0	**0**	0
CE bridges with SR (wires-OR)	0	0	0	0	0	0	**0**	0

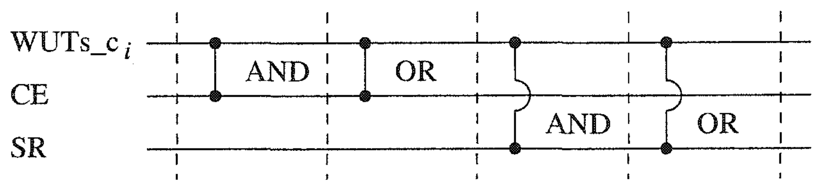


Figure 3.6: Single-bridge faults

Table 3.4: Response of Test 3 for single-bridge fault

Condition of the Circuit	Q_{2i-2}	Q_{2i-1}	Q_{2i}	Q_{2i+1}	Q_{2i+2}
Fault-free	1	0	1	0	1
$WUTs\_c_i$-CE AND	**0**	0	1	0	**0**
$WUTs\_c_i$-CE OR	**0**	0	1	0	**0**
$WUTs\_c_i$-SR AND	1	0	**0**	0	1
$WUTs\_c_i$-SR OR	1	0	**0**	0	1

We can see that, for every faulty case, there is always at least one faulty Q response.

Test vectors in the Test 2 sequence are exhaustive patterns on WUTs_c with CE = 1 and SR = 0. Sun *et al.* proved, in [18], that an exhaustive test pattern on the WUTs_c can detect all the assumed faults mentioned in our fault models.

It has to be proved that Test 3 can detect the multiple-wire bridges between the two wire groups WUTs_c and WUTs_s. Let's define WUTs_c as a set of

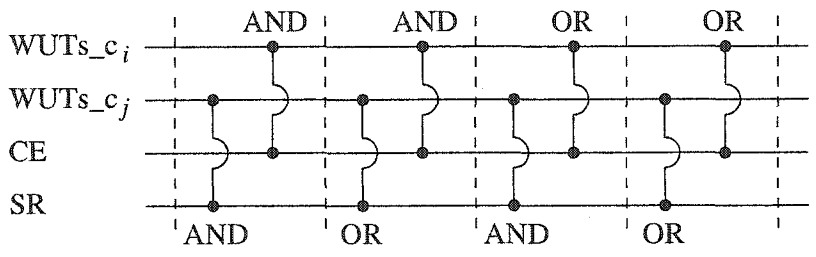


Figure 3.7: Double-bridge faults

Table 3.5: Response of Test 3 with two bridge faults

Condition of the Circuit		Q_{2i-1}	Q_{2i}	Q_{2j-1}	Q_{2j}
Fault-free		0	1	0	1
$WUTs\_c_i$-CE AND	$WUTs\_c_j$-SR AND	0	1	0	**0**
$WUTs\_c_i$-CE AND	$WUTs\_c_j$-SR OR	0	1	0	**0**
$WUTs\_c_i$-CE OR	$WUTs\_c_j$-SR AND	**1**	1	0	1
$WUTs\_c_i$-CE OR	$WUTs\_c_j$-SR OR	0	1	0	**0**

wires:

$$WUTs\_c = \{WUTs\_c_1, WUTs\_c_2, \ldots, WUTs\_c_n\}, \tag{3.1}$$

where n is the number of wires in WUTs_c.

First, consider the cases where there is only one bridge between the two groups of wires. Figure 3.6 shows all the possible single-bridge cases between one line in WUTs_c and CE and SR. Table 3.4 is a generic table that shows Q results of five testing vectors that are function of i, where $WUTs\_c_i$ is the line implied in the bridge. In this table, $i \in \mathbb{Z}$ and $1 \leq i \leq n$, where $\mathbb{Z}$ represents the set of integers. It can be seen that a test vector sequence built like Test 3 can detect any single-bridge fault between a group of n wires and CE or SR. It must be mentioned that when $i = 1$, the vector Q_0 does not exist because the vector v_0 does not exist. Similarly, responses Q_{2i+1} and Q_{2i+2}, when $i = n$, do not exist because the test vectors v_{2n+1} and v_{2n+2} do not exist.

Using Figure 3.7 and Table 3.5, it is possible to prove, in a similar way, that double-bridge fault cases between WUTs_c and WUTs_s groups can be detected. Again, Table 3.5 shows four test responses for all the possible double-

Table 3.6: Estimation of test time

Virtex FPGA Device	Number of Test Vectors	Number of Column Select Lines $\lceil m/4 \rceil$	Test Time (μs)
XCV50	150	6	12.0
XCV100	150	8	15.0
XCV150	150	9	16.5
XCV200	150	11	19.5
XCV300	150	12	21.0
XCV400	150	15	25.5
XCV600	150	18	30.0
XCV800	150	21	34.5
XCV1000	150	24	39.0

bridge faults that imply $WUTs\_c_i$ and $WUTs\_c_j$, where

$$\{(i,j) \mid i \in \mathbb{Z},\ 1 \leq i \leq n,\ j \in \mathbb{Z},\ 1 \leq j \leq n,\ i \neq j\}. \tag{3.2}$$

For any pair of wires in WUTs_c that can bridge with CE and SR, test vectors in Test 3 can detect these faults from a wrong response Q.

3.5 Fault Coverage

The fault coverage of a test is determined by the ratio of faults detected by the test divided by the total number of faults, defined by the fault model, present in the circuit. Since no fault simulation was performed on the circuit, it not possible to determine a precise fault coverage.

The previous section proved analytically that the three sets of test vectors can detect many kinds of faults describe in the fault model. Therefore, it proved that this test can have a high fault coverage. These proofs are valuable because they are general and can then be applicable to a different system of local interconnects.

3.6 Testing Time

Now that the test scheme has been presented in detail, an estimation of the testing time can be determined. The testing time will be estimated by mul-

```
1: for all Test Vectors do
2:     for all Column Selects do
3:         Compare test responses in parrallel with fault-free parity signal
4:     end for
5: end for
```

Figure 3.8: Testing sequence

tiplying the total number of clock cycle in the test by an estimation of the FPGA clock period. The Equation 3.3 shows how to obtain the number of clock cycles for the test as a function of the number of test vectors and the number of column selects.

$$Number\ of\ clock\ cycles = Test\ vectors\ (Column\ selects + 2) \tag{3.3}$$

The Figure 3.8 shows a simple algorithm of the test sequence. To derive Equation 3.3, it was assumed that the application of any test vector takes two clock cycles and one clock cycle is needed to select a column and to get a pass or fail result from the ORA. Table 3.6 shows the testing time for different FPGAs in the Virtex family. Here it was assumed that the clock frequency is 100 MHz. Since our design was not simulated and not implemented in an FPGA, it is not easy to estimate the maximum clock frequency of the design. Therefore, a realistic estimation of half the maximum clock frequency of the Virtex FPGA was taken.

blank page-no text

Chapter 4

Interconnects Model and Routing Algorithm

In the previous chapter, the overall test scheme was presented and the fault coverage of the scheme was analyzed. In order to get an FPGA testing program from the test scheme, the local routing matrices have to be programmed in combination with test logic in the *configurable logic blocks* (CLBs). The local routing matrices are *input routing matrix* (IRM), *output routing matrix* (ORM), and a part of the global routing matrix (GRM). In the present work, a divide-and-conquer approach is taken: *test configurations* (TCs) are generated for one tile and duplicated for all tiles under test in the FPGA.

Due to the structure of the IRM and ORM, made of MUX switches, it is not possible to configure all the PSs in one TC. Therefore, to test all the local interconnects, more than one TC is needed. The FPGA programming time is usually relatively long (10 - 100 ms [35]) compared to the time one test program takes to perform the actual test (see Section 3.6). Since one of the most important concerns in testing integrated circuits is to reduce test time, a heuristic way of minimizing the number of TC is needed.

This chapter first presents some constraints related to the derivation of TCs followed by a definition of the problem. It will be followed by a presentation of a local interconnects model based on graph theory. A TC derivation algorithm based on a sequence of bipartite graph edge coloring is also presented. Finally, the last two parts of this chapter present an analysis and experimental results

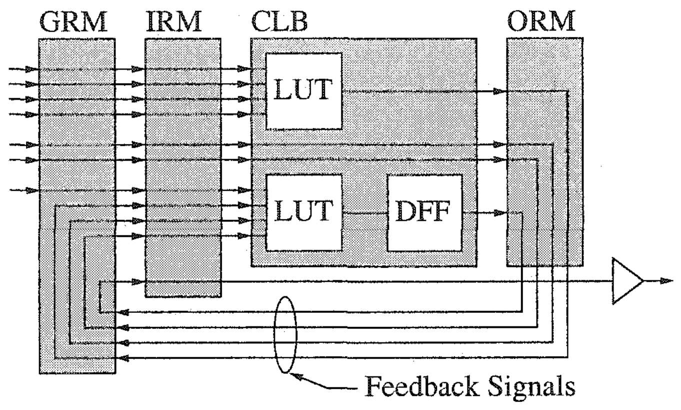


Figure 4.1: CLB configuration example

of the algorithm.

4.1 Test Logic and Feedback Routing

As mentioned in Chapter 3, the local test logic used to test local interconnects is a parity generator implemented in CLBs. In order to minimize the number of TCs, the algorithm has to maximize the number of PSs and wire segments used in these TCs. Therefore the parity generator implemented in the CLBs should be configured with the maximum number of CLB inputs and outputs.

In order to configure a parity generator with the maximum number of inputs using the two LUTs of the CLB, feedback must be used. The structure of the CLB allows signals to pass through it without entering into the logic. Consequently, another way to add more CLB inputs and outputs to the test logic is to use these kinds of signal combined with feedback signals. The other feedback signals are used to route the latched values of the logic to tri-state buffers. Therefore, one of the constraints that the algorithm has to deal with is building the TCs local routing according to the test logic.

Figure 4.1 shows an example of a CLB configuration with four feedback signals. In this example, it is possible to observe that feedback is needed to be able to configure the two LUTs as a parity generator. The parity output is latched by a *D type flip-flop* (DFF). Finally, the output needs another feedback signal to reach the tri-state buffer that will drive the test response on the tri-

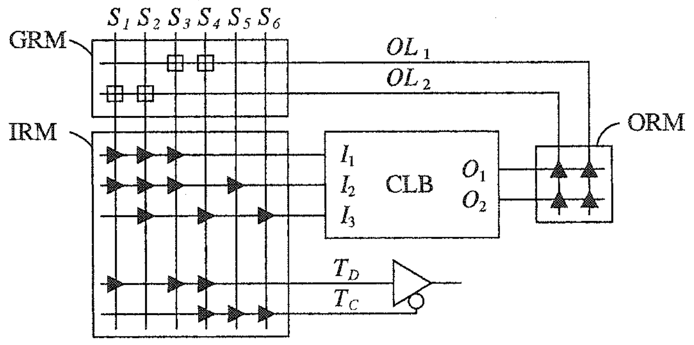


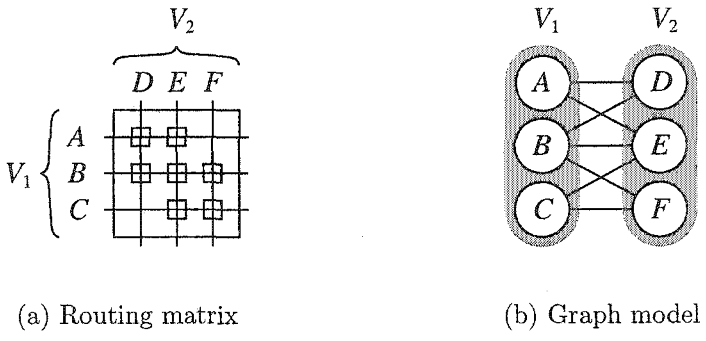
Figure 4.2: Local interconnect network

state bus.

4.2 Interconnect Model

A local interconnect model is necessary for Virtex FPGAs in order to support a TC derivation algorithm. Figure 4.2 shows a simple model of a CLB with the local interconnects, including the IRM, ORM, and the part of the GRM that connects the *output lines* (OLs) to the global network. The IRM and the ORM are made of *multiplexer PSs*, called *input multiplexers* (IMUXs) and *output multiplexers* (OMUXs) respectively. A multiplexer PS functions as a conventional many-to-1 MUX. The other kind of PS is called a *basic PS* denoted by squares in the GRM. A basic PS connects two wire segments when it is programmed on. The basic PSs in the GRM between the OL wire segments and the short wire segments $(S_1, S_2, .., S_6)$ are included in the local interconnects according to the definitions in chapter 2.

A routing matrix can be modeled as a bipartite graph $(G = \{V, E\})$, where wire segments and PSs are denoted by *vertices* (set V) and *edges* (set E), respectively. In a bipartite graph, V is split into two disjoint sets, V_1 and V_2, and an edge is a link between two vertices, u and v, where $u \in V_1$ and $v \in V_2$. The two disjoint vertex sets of a bipartite graph represent two sets of wire segments. Edges of the graph represent PSs that can be programmed in order to connect two wire segments. Figure 4.3 shows an example of routing matrix modeling. Two groups of lines, V_1 and V_2, are



(a) Routing matrix (b) Graph model

Figure 4.3: Routing matrix modeling example

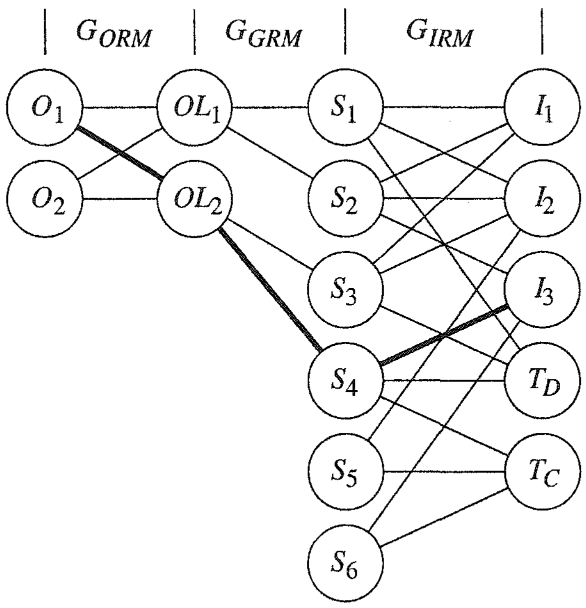


Figure 4.4: Modeled k-partite graph

modeled as two groups of vertices, where $V_1 = \{A, B, C\}$ and $V_2 = \{D, E, F\}$. All possible PS connections are also represented on the bipartite graph by edges.

Similarly, the IRM, the ORM, and the local interconnect part of the GRM from Figure 4.2 can be modeled by three bipartite graphs G_{IRM}, G_{ORM}, and G_{GRM}, respectively. Graphs are combined to form a k-partite graph, a graph with k disjoint sets of vertices, where $k = 4$, by merging the common vertices shared between bipartite graphs. Figure 4.4 depicts the k-partite graph. The formation of the k-partite graph is convenient to trace a routing path needed to

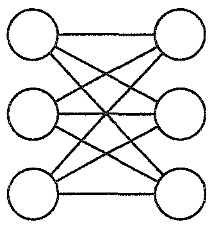
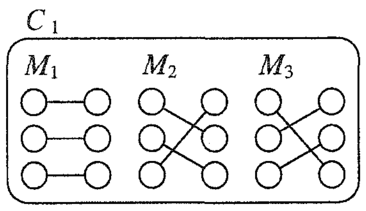
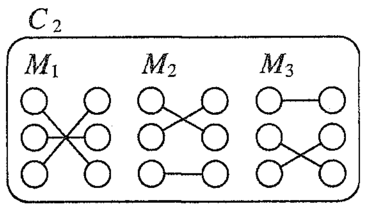


Figure 4.5: Graph G_1



(a) Coloring C_1 of G_1



(b) Coloring C_2 of G_1

Figure 4.6: Edge coloring examples

test local PSs and wire segments. For example, a routing path, $O_1 \rightarrow OL_2 \rightarrow S_4 \rightarrow I_3$, shown in bold lines, permits the test of one PS in the ORM, one PS in the IRM, one basic PS in the GRM, and corresponding wire segments.

4.3 Algorithm

The proposed algorithm is based on the graph modeling of FPGA interconnects shown in the previous section. It is the first to use bipartite graphs to model FPGA local interconnects, and to derive TCs from a k-partite graph.

4.3.1 Edge Coloring

Finding a minimal or near-minimal set of TCs, which verify the integrity of local interconnects, can be considered as an *edge coloring* problem of graphs. Edge coloring involves coloring all edges of a given graph $G = \{V, E\}$ with the minimum number of colors so that no two edges connected to the same vertex are assigned the same color (i.e., a coloring $C = \{M_1, M_2, ..., M_k\}$, where M_i is a set of edges of the same color). If each color represents a TC, then edges included in that TC are PSs that can be tested simultaneously.

Figure 4.5 shows a bipartite graph called G_1 and Figure 4.6 (a) shows an edge coloring of G_1 called C_1. The edge coloring of a bipartite graph is often not unique. For example, Figure 4.6 (b) shows another possible coloring C_2 of the graph G_1

The minimum number of colors, known as the *edge chromatic number*, is equal to the maximum vertex degree of the graph, $\Delta(G)$. Since each color of the graph G represents a TC, the minimum number of TCs is $\Delta(G)$. Maximum degrees of vertices for G_{IRM} and G_{ORM} are $\Delta(G_{IRM})$ and $\Delta(G_{ORM})$ respectively. For Virtex FPGAs $\Delta(G_{IRM}) > \Delta(G_{ORM})$. Therefore, the theoretical minimum number of TCs is determined by $\Delta(G_{IRM})$.

4.3.2 Algorithm Overview

As stated in Chapter 2, no existing algorithms can be used to solve coloring problems of a k-partite graph for our test application mostly because of the constraints generated by the test logic. What is proposed in this algorithm is to first treat the k-partite graph coloring problem as coloring problems of two bipartite graphs, G_{ORM} and G_{IRM}, to obtain minimal solutions. The two colorings are consolidated to form a maximal or near-maximal number of routing paths, which in turn support the minimal or near-minimal coloring of the two bi-partite graphs (i.e., the minimal or near minimal number of TCs for the local interconnects).

The routing algorithm developed in the present research has three major parts:

Part1 Edge coloring of G_{ORM} + Completion,

Part2 Feedback routing + Completion,

Part3 Edge coloring rest of G_{IRM} + Completion.

All the PSs programmed for each TC in ORM will be part of the feedback. Since feedback is critical to the test logic, the algorithm takes care of them by

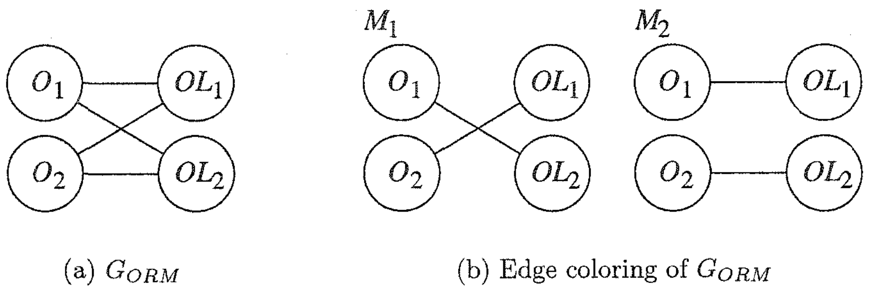


Figure 4.7: Edge coloring of G_{ORM}

configuring the ORM first. The first part of the algorithm is to configure PSs in the ORM by coloring its corresponding bipartite graph G_{ORM}. The coloring of G_{ORM} is followed by what is called a *completion* of the coloring. Even if edge coloring of a particular bipartite graph ensures that all the PSs are used in at least one TC, some TCs will need more PSs to be sure that the test logic is respected. Some edges, which were already present in other colors, are included in the specific TCs to complete then. The number of colors generated by this coloring is equal to $\Delta(G_{ORM})$.

The purpose of the second part of the algorithm is to create correct test logic by routing the feedback. Part two finds feedback signals that include all the edges of all the colors of the G_{ORM}. A completion of the TC is also necessary to route signals that are not part of the feedback signals. These signals are control signals (SR, CE, CLK) and logic inputs. After the second part is done, TCs numbered 1 to $\Delta(G_{ORM})$ are completed.

The last part of the algorithm colors the rest of the edges in G_{IRM} that were uncolored in the previous part. This coloring configures PSs in TCs numbered $\Delta(G_{ORM}) + 1$ to $\Delta(G_{IRM})$. The completion is done by adding some already tested PSs, used in previous TCs, to create test logic.

4.3.3 Example

This subsection presents an example of applying of the algorithm. The algorithm will be applied on the simple local interconnect modeling presented in Section 4.2. From that model, the theoretical number of TCs the algorithm

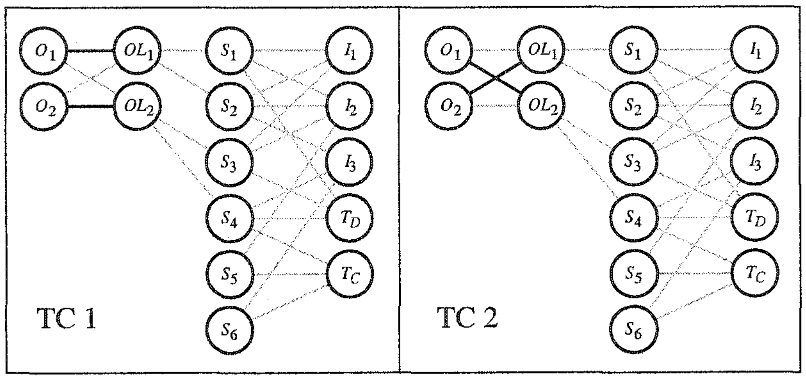


Figure 4.8: Coloring of part 1

should derive is equal to $\Delta(G_{IRM}) = 4$.

As described in the previous section, the first part of the algorithm is to color G_{ORM}. Figure 4.7 (a) and (b) show graph G_{ORM} and an edge coloring of it, respectively. Because all the vertices have the same degree in G_{ORM}, the completion phase of the first part does not have to be performed. These colors can be integrated into TC1 and TC2 as shown in Figure 4.8. At this stage, TC3 and TC4 are still empty.

The second part of the algorithm makes feedback paths from the CLB outputs to the CLB inputs that include the edges configured in part 1. When the feedback paths are routed, the algorithm completes TC1 and TC2 by adding edges in IRM. Added edges will be inputs of the local test configuration where the test vectors will be applied. Figure 4.9 shows TC1 and TC2 completed by the inclusion of the routed feedback paths and the edges added by the completion.

All PSs in IRM that were not configured in TC1 and TC2 need to be configured in TC3 or TC4. An edge coloring is performed on the rest of G_{IRM}. Figure 4.10 (a) and (b) shows the rest of G_{IRM} and its corresponding coloring. Again, the colors can be transfered to the TCs as shown in Figure 4.11.

Since the ORM has been already tested, it is now possible to use any PSs in it to route feedbacks for TC3 and TC4. These TCs also need to be completed because the coloring does not include all the CLB inputs necessary to have correctly configured test logic in all TCs. Figure 4.12 shows the completed

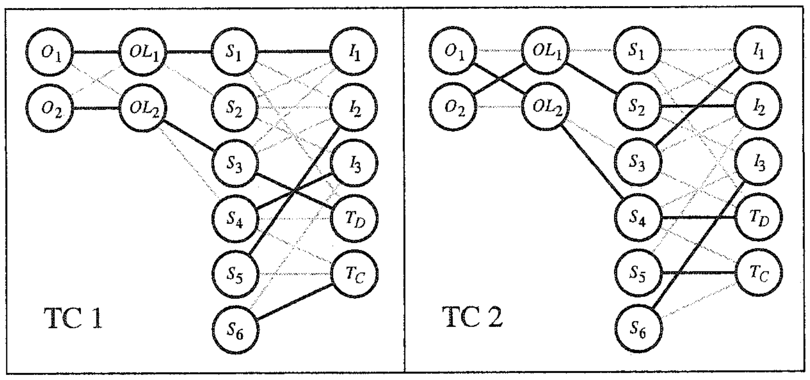


Figure 4.9: Feedback paths and completion of part 2

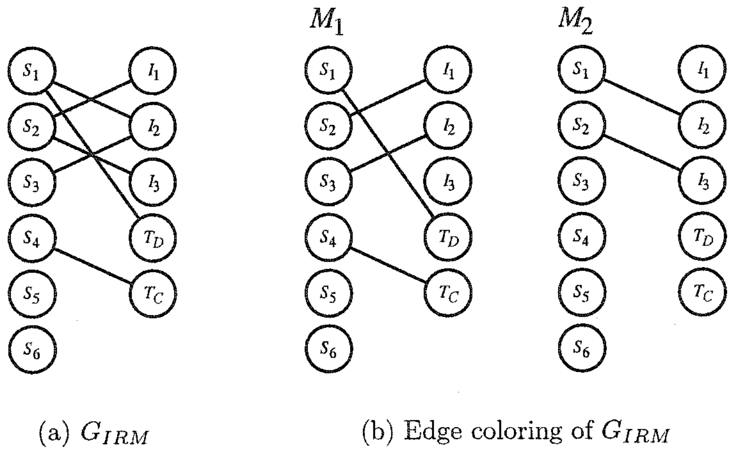


Figure 4.10: Edge coloring of G_{IRM}

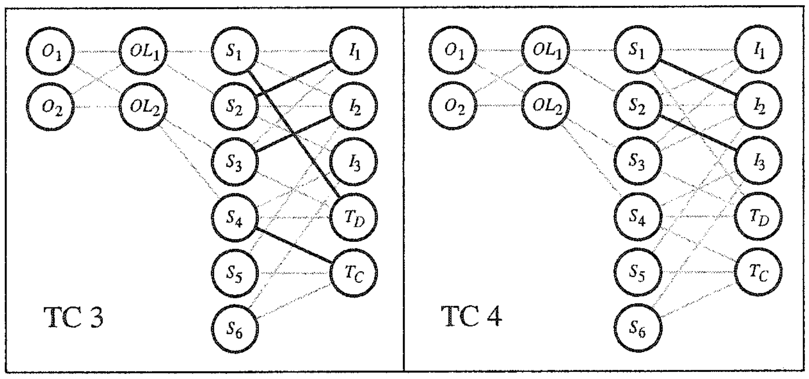


Figure 4.11: Coloring of part 3

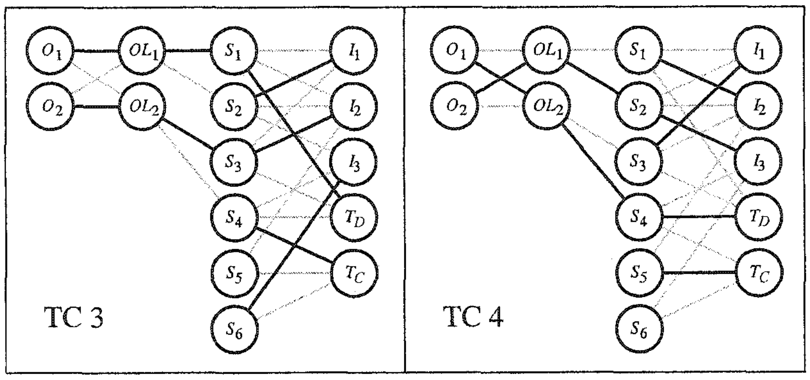


Figure 4.12: Coloring and completion of part 3

TC3 and TC4 after the inclusion feedback paths and the completion of G_{IRM} coloring.

4.3.4 Algorithm Details

To gain a more in-depth understanding of the algorithm, some parts require further explanation. Most algorithms used in this thesis are presented with pseudo-code in Appendix A. More details can be found in Appendix B where all the Perl-coded programs are included.

In the first part of the algorithm, an edge coloring of a bipartite graph is performed on G_{ORM}. In fact, this edge coloring of G_{ORM} is actually not a straight edge coloring of a bipartite graph. Some of the colors generated by an edge coloring of G_{ORM} are not compatible with any feedback combination. The result is that no test logic, as described in Section 4.1, can be used.

In order to get compatible colors, an algorithm which employs enumeration of edge coloring of bipartite graphs is used. As mentioned earlier, the edge coloring of a specific graph is often not unique. In this case, it is possible to use that property of the edge coloring to find a set of colors that can match a certain feedback routing. The procedure used is based on the work of Matsui and Uno in [36]. More details about this algorithm are presented by the Algorithm 2 in Appendix A.

The algorithm developed by Matsui and Uno in [36] takes a bipartite graph as input and basically builds a tree of colors in a *depth-first* manner. Figure

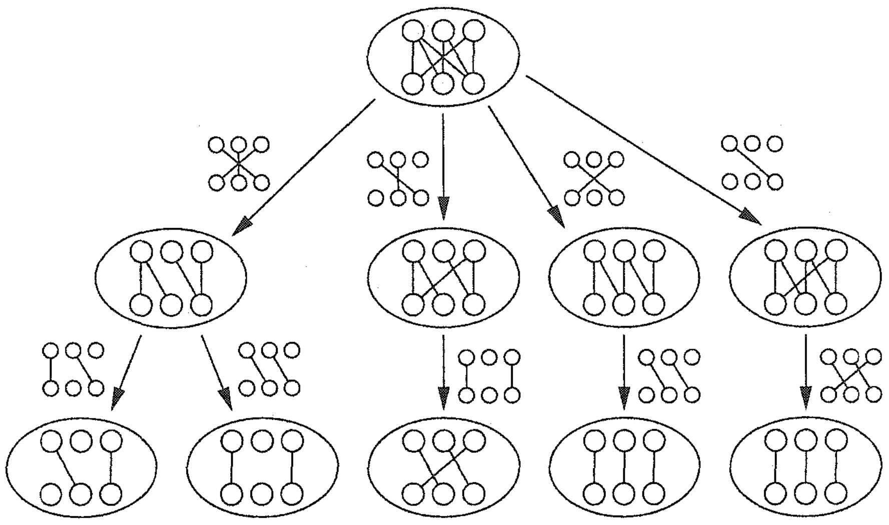


Figure 4.13: Example of edge coloring enumeration

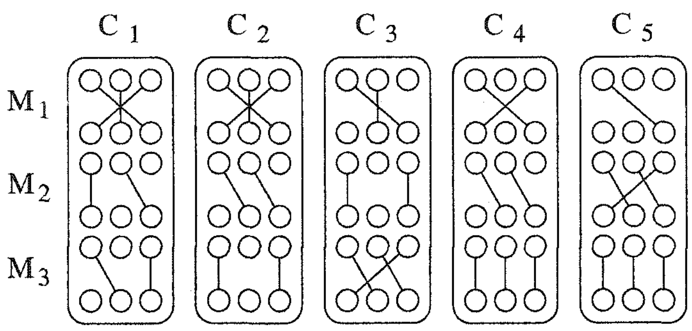


Figure 4.14: All edge coloring

4.13 presents an example of the enumeration of the colorings of a bipartite graph. Every path from the roots to the leaves is an edge coloring of the bipartite graph on the top of the figure. Figure 4.14 shows all the five possible edge colorings (C_1 to C_5) that can be derived from the tree of Figure 4.13. All edge colorings get the minimum number of three colors M_1, M_2, and M_3. Because the number of possible colorings can be very high in this case, the run time of this algorithm can be long.

Since the present application needs only one edge coloring that can be routed in the FPGA, it is possible to modify the algorithm and speed up the process. The modified algorithm checks if every new color found is able to generate at least one test logic configuration with feedback. If the color found cannot generate routable feedback paths, the algorithm does not continue further in that part of the tree. If the algorithm reaches the bottom of the tree (a leaf), it stops the search. The pseudocode of the modified algorithm is presented in Algorithm 3 in Appendix A.

The non-modified enumeration algorithm used in part one has a time complexity of $O(|V|N)$, where $|V|$ represents the number of vertices and N the number of edge coloring. This time complexity is from [36].

The modified algorithm checks the routablility of feedback paths for every new color generated according to the CLB test logic configurations. To find the routability of the feedback paths, the algorithm performs a linear search through the CLB configurations and checks if all paths in those configurations can be routed in the FPGA. The added procedure increases the worst-case time complexity that is now $O(|V|NBS)$, where B is the number of CLB configurations and S the number of feedback paths per CLB configuration. The time complexity seems very high, but in fact, the algorithm does not have to pass through all of the enumeration. The $O(BS)$ complexity procedure actually speeds up the process because only one solution is necessary. In fact, the best case is $O(\Delta BS)$ where Δ is the maximum degree of the bipartite graph. The average case is a lot faster than the worst case of $O(|V|NBS)$, but it is a function of the probability of finding a routable color.

The completion step of the first part of the algorithm is done in a worst time complexity of $O(\Delta IBS)$, where Δ represents the number of incomplete colors in the coloring, I is an over-bounded value for the maximum number of missing edges in an incomplete color, and B and S are the same as before. Algorithm 5 shows this completion pseudocode. Even if this procedure seems to be highly time consuming, the number of incomplete colors and their corresponding missing edges are low. In practice, the running time for this procedure is relatively low.

In the second part, the algorithm has to find feedback paths for all edges in colors that were generated by the coloring and completion in part one. The algorithm performs a sequential search for each color of the coloring. The worst-case time complexity for the sequential search is very high, $O(\Delta BS)$, but it will most likely not happen because the search stops when a combination of paths is found. It does not have to search through all possibilities. This procedure is shown in Algorithm 6.

The completion phase of part two treats each color separately and adds edges in color IRM. For one specific color, the vertices related to the edges already in the color are deleted. On the resulting G_{IRM}, a procedure tries to find a match (or a color). The edges that are part of the match are transferred from G_{IRM} to the color. Since finding a new match in a bipartite graph can be done in $O(max\{|E|, |V|\ log|V|\ log^2\Delta\})$ as described in [37], the present completion can be done in $O(\Delta(max\{|E|, |V|\ log|V|\ log^2\Delta\}))$, where Δ is the maximum degree of any vertex in G_{IRM}. Algorithm 7 shows the pseudocode of this part of the algorithm.

The last part of the algorithm first performs a simple edge coloring of a bipartite graph on the rest of G_{IRM} to build the last TCs. A simple implementation of the edge coloring of bipartite graphs can run in $O(|V||E|)$ where $|V|$ is the number of graph vertices and $|E|$ is the number of edges in G_{IRM}. Algorithm 1 shows the pseudo-code of a simple edge coloring that runs in $O(|V||E|)$. This algorithm is presented in [38].

The last completion is performed in $O(\Delta IBS)$, similar to the completion of the first coloring. Here, $\Delta = \Delta(G_{IRM}) - \Delta(G_{ORM})$ because it is part of the algorithm which colors the second part of G_{IRM}. Algorithm 8 presents more details about this procedure.

4.3.5 Experiments

The proposed algorithm was tested on Virtex FPGAs, whose $\Delta(G_{IRM}) = 26$. The coloring algorithm, coded in Perl, modeled the interconnects resource into bipartites graphs and successfully derived a minimal set of 26 TCs with the edge coloring sequence described earlier in this chapter. The program ran in under 6 minutes. The experiments were carried out on a 750 MHz UltraSPARC III microprocessor with 2 GB of RAM.

The programs, presented in Appendix B and C, derived the set of 26 TCs. These TCs are presented by 26 conceptual diagrams in Appendix D. Figure 4.15 shows the TC number 14. On this diagram, the routing matrices are identified in gray on Figure 4.15. The wire segments are represented as circles and the programmed PSs as straight lines in the routing matrices. Two CLBs and two tri-state buffers are identified by CLB0, CLB1, T0, and T1, respectively, on the diagram. The short lines are the only global lines shown on the periphery of the GRM. Some other global lines, like some hex lines, are connected to the IRM. The inputs and the outputs of the CLBs are below and on top of the CLBs, respectively. The out lines, are shown at two places for more clarity: on the side of ORM and also on the side of GRM.

It should be mentioned that the set of TCs presented in Appendix D is not unique. That is according to the fact that the edge coloring of a bipartite graph is not unique as well. For example, other sets of TCs can be found if the order of the PSs is changed in the inputs of the program.

Since a divide-and-conquer approach was used, and the TCs derived are duplicated to all remaining tiles to form a TC, the TC derivation algorithm is independent of the dimensions of Virtex FPGAs.

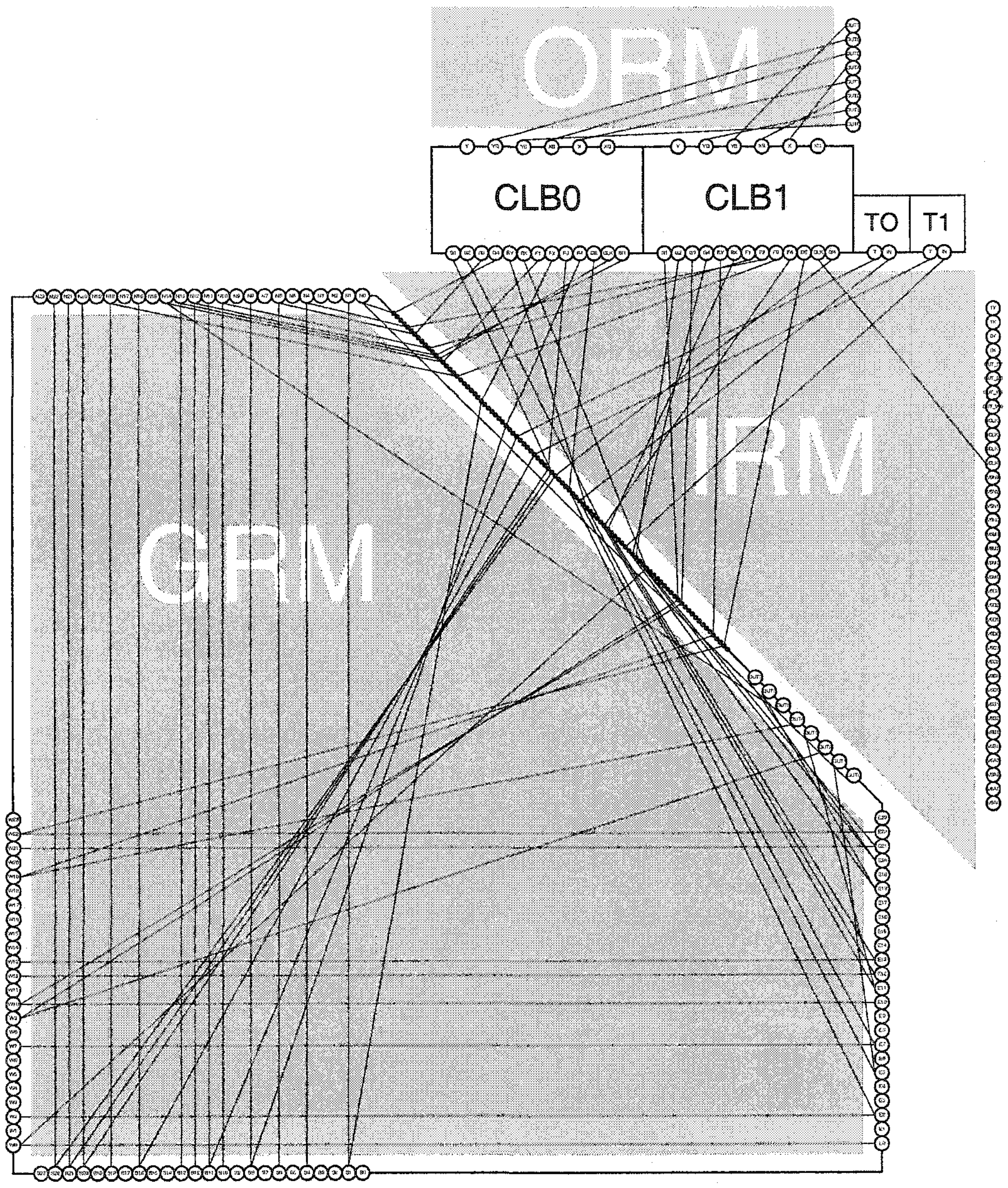


Figure 4.15: Test configuration 14

blank page - no text

Chapter 5

Proposed Diagnostic Schemes

Diagnostic tests are applied after a circuit has failed a *pass/fail* test. These tests have to identify a faulty part or area that should be replaced, modified, or avoided. Diagnostic tests, like normal pass/fail tests, can be applied to an integrated circuit in its design system, manufacturing system, and application system. A diagnostic test is characterized by its *diagnostic resolution*, defined as the smallest faulty feature that the diagnostic test can detect. The diagnostic resolution can be at a transistor level, where transistors and wire segments can be faulty. At higher levels, the diagnosis can declare a register, a chip, or even a complete printed circuit board to be faulty.

In the case of FPGA interconnects, the smallest resolution that can be achieved is at the *wire segment* and *programmable switch* (PS) level. Practically, it is not possible to have a diagnosis at that level since not all wire segments and PSs are tested in one *test configuration* (TC). Therefore, the diagnostic resolution will be bigger.

It is useful to develop FPGA diagnostic tests that can be embedded in the chip, like a diagnostic BIST scheme. For manufacturing diagnosis, this kind of embedded test avoids the use of expensive test machines such as *automatic test equipment* (ATE). When an FPGA is embedded in an electronic system, a diagnostic test can repair the FPGA. That is called a *fault tolerant* system. In order to repair an FPGA, the diagnostic test must let the FPGA programming device know which parts are defective. The programming device can then avoid the defective parts of the FPGA.

A certain diagnosis can operate at the tile level by keeping the same TCs as the previous section. In one TC, it is possible to select a specific tile because two test responses are put on a tri-state bus for each tile, and the column select signals can control one column at a time. Consequently, it is possible to have a diagnostic resolution with a size of half a tile according to the test scheme presented in Chapter 3. Of course, the BIST circuitry has to be modified, but the *circuit under test* (CUT) of the scheme presented in Chapter 3 can be preserved.

This chapter presents two diagnostic test schemes that are modified versions of the scheme presented in Chapter 3. The first section presents a general diagnostic scheme. The second and third sections present a fault tolerant scheme and a manufacturing scheme respectively.

5.1 General Diagnostic Scheme

Chapter 3 presented advantages of having a BIST scheme. The present diagnostic scheme is also a BIST scheme. Some modifications are needed in the BIST part to adapt it to diagnosis. One new consideration that the diagnostic scheme has to deal with is to determine where and how diagnostic result information will be stored in the FPGA. Since the new scheme uses the same CUT part, the scheme can keep the same test vectors described in Chapter 3. Since some RAM blocks are not used for test vector storage, they can be used to store diagnostic test responses. When the test is over, test responses stored in RAM blocks can be sent out of the FPGA.

The sequence of the diagnostic test is similar to the one presented in Chapter 3. First, test vectors are applied in parallel to the CUT part of the FPGA. Second, the column select signal from the BIST part selects two columns of tiles that will be under test. Four test responses per row are sent back to the BIST part in parallel via tri-state buses. Each test response received by the BIST part is compared with a fault-free signal. This process generates a sequence of *1*'s and *0*'s, meaning passed and failed, respectively.

One of the major differences between the diagnostic test scheme and the regular one presented in Chapter 3 is that test responses need to be stored during the diagnostic test. Test responses are not analyzed in parallel by combinational logic like in the test scheme in presented Chapter 3. Figure 3.4 shows the ORA that was composed of an AND, a NOR and a multiplexer. In the diagnostic scheme, test responses from the CUT are stored in a shift register and analyzed one by one. This is slower but it allows no loss of test response information.

Different FPGA applications may need different diagnostic tests and thus may need a different kind of test result data. In the next subsections, two diagnosis schemes will be presented. Their differences come from the way the data is stored for two different applications. The first diagnostic test is ideal when the FPGA is used in a fault-tolerant system. The other diagnostic test can be applied in a manufacturing environment where a fine diagnosis is needed.

5.2 Fault Tolerant Scheme

A *fault-tolerant* system has the capacity to work in the presence of faults. In the case of FPGAs, fault tolerance is achieved by avoiding faulty resources the FPGA and using only good ones when programmed, allowing the design to function properly. Of course, when there are too many faulty resources, the FPGA design can not be implemented properly.

A diagnostic test scheme, based on the Chapter 3 scheme, can be developed for fault-tolerant systems with FPGAs. Diagnostic tests embedded in an integrated circuit can perform, inside the integrated circuit, testing computations that is usually computed outside. In the present scheme, diagnostic test outputs can be coordinates of faulty FPGA tiles. As mentioned before, the smallest resolution that a diagnosis program can achieve is half a tile. Since it is easier for an FPGA programming device to avoid a faulty tile than avoid

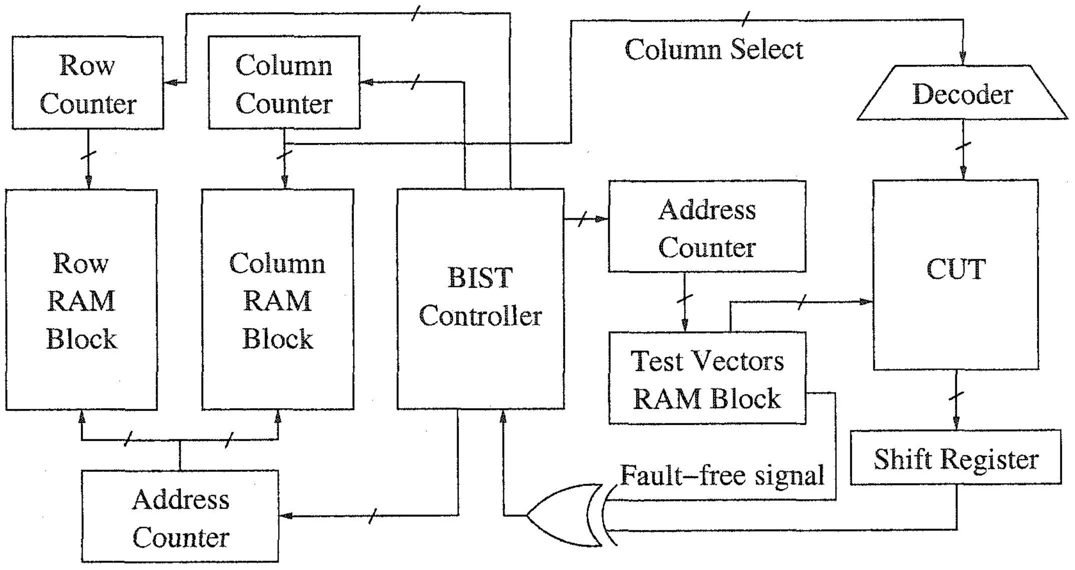


Figure 5.1: Fault-tolerant diagnosis test scheme

only one particular faulty element in this tile, outputting the coordinates of faulty tiles is a good choice of output data for a fault-tolerant scheme.

In this diagnostic scheme, the diagnostic results are stored in the available RAM blocks in the FPGA. In fact, not all the results need to be saved. To comply with a fault-tolerant scheme, only row and column numbers of all faulty tiles have to be saved. Figure 5.1 shows a diagram of the fault-tolerant scheme. When the BIST controller detects a faulty test response in a specific tile, its row and column number are saved in two separate RAM blocks. The address of both memories will be incremented every time a new faulty tile is found. The test does not have to pass through all test vectors: when a tile gets one faulty response, it is declared faulty and the test continues on to the next tile.

Because memory space is a concern in the present scheme, more investigation is needed to prove the feasibility of the scheme. Every RAM block in the FPGA has 1 kB of memory divided into two 512-Byte ports. These ports can be configured as shown in Table 5.1. Many memory widths can be used. Table 5.2 shows some characteristics for every FPGA in the Virtex family.

Practically, RAM ports need to be configured at least 8 bits wide to be able

Table 5.1: RAM port configurations

Memory Width	Memory Depth
1	4096
2	2048
4	1024
8	512
16	256

Table 5.2: Characteristics of Virtex FPGAs

Virtex FPGA Device	Number of Rows	Number of Columns	Number of RAM Blocks
XCV50	16	24	8
XCV100	20	30	10
XCV150	24	36	12
XCV200	28	42	14
XCV300	32	48	16
XCV400	40	60	20
XCV600	48	72	24
XCV800	56	84	28
XCV1000	64	96	32

to store the row number of the biggest FPGA. A 4-bit wide memory is not sufficient. Due to the vertical division between the BIST part and the CUT part in the test scheme (see Chapter 3), only half of the total RAM blocks are accessible to save test information. Even with this constraint, there is enough space to store all tile addresses of the biggest FPGA in the worst case where all tiles are faulty. Table 5.3 shows that the number of bits is enough to store the addresses of all the tiles in the CUT part of the FPGA.

When the test is over the RAM block information is read from outside the FPGA. The diagnostic results (column and row addresses of faulty tiles), for all the TCs, are accumulated outside the FPGA. These diagnostic results are used by the compiler tool to recompile the original design and then avoid the faulty resources.

Table 5.3: Number of tiles under test

Virtex FPGA Device	Number of Tiles Under Test	Bytes Available
XCV50	384	3584
XCV100	600	4608
XCV150	864	5632
XCV200	1176	6656
XCV300	1536	7680
XCV400	2400	9728
XCV600	3456	11776
XCV800	4704	13824
XCV1000	6144	15872

5.3 Manufacturing Diagnostic Scheme

In certain tests, further analysis on the diagnostic test results is needed outside the FPGA. Manufacturing diagnostic test falls into in this category. The collection of all test results for all TCs can help to perform a more accurate diagnosis outside the FPGA. In that kind of test, no loss in the test results is allowed.

Storing test responses on-chip instead of sending them one-by-one outside the chip can speed up the test process because the circuit outside does not have to be synchronized with the FPGA during the test. At the end of the test, the FPGA sends out stored values. With this kind of test scheme, many FPGAs can be tested in parallel, which further speeds up the testing process.

Because the available storage space is limited, a quick analysis is needed to see if there is enough space in the RAM block to save all of the test responses.

The length of the binary test vector sequence applied to a tile is the same for every device in the Virtex family. On the other hand, FPGAs in the Virtex family get different numbers of test responses due to their different sizes. Equation 5.1 describes the total number of test responses applied during one TC.

$$\begin{aligned} Test\ response\ total &= \frac{Columns}{2} \times 2 \times Rows \times Number\ of\ test\ vectors \\ &= Columns \times Rows \times Number\ of\ test\ vectors \end{aligned} \tag{5.1}$$

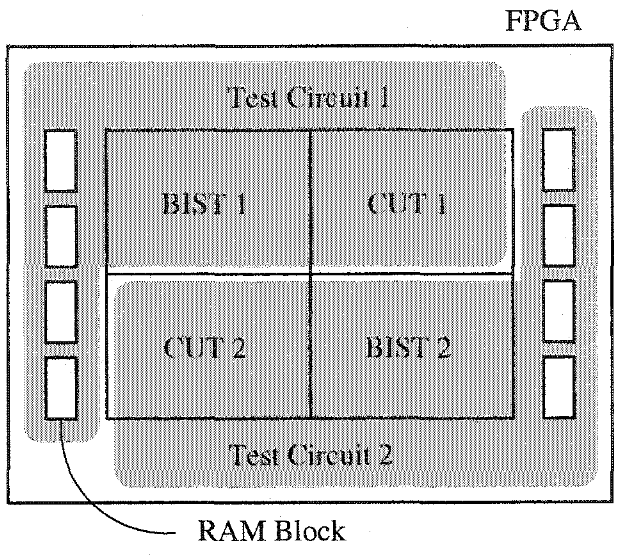


Figure 5.2: A test configuration with two BISTs

The *Number of test vectors* variable in this equation is equal to 150 from test vector sets derived in Chapter 3. The number of *Columns* is divided by two because the CUT takes only half the space divided vertically. It is also multiplied by two because there are two test responses per tile.

In the regular BIST scheme described in section 3, only half of the RAM blocks are accessible from the BIST because the CUT block the accesses the one adjacent to it. Since the BIST part of the FPGA is essentially the same for all TCs for all the FPGAs in the family, there are available logic resources in the BIST part of the TC for the biggest FPGAs. Therefore, to be able to reach the RAM blocks on both the right and left sides of the FPGA, a new TC partition of the FPGA can be used. Figure 5.2 shows the proposed solution where the logic resources are split into four parts. Two test circuits are present in this TC. The BIST 1 tests the CUT 1 and uses the column of RAM Blocks on the left side. Similarly, BIST 2 tests CUT 2 and uses right side RAM blocks. Therefore the number of test responses will be divided by two for FPGAs that are twice as big as the smallest FPGA in the Virtex family, the XCV50. Some interactions are needed between the two BISTs because the values stored in the RAM blocks must be read only when both test circuits have finished there tests. Having more than two test circuits in one TC will not give access to

Table 5.4: Compression needed for all Virtex FPGAs

Virtex FPGA Device	No. of BIST Used	No. of Test Responses	No. of RAM Ports Available	Available Memory (Bits)	Compression Ratio Needed
XCV50	1	57600	7	28672	2.01
XCV100	1	90000	9	36863	2.44
XCV150	2	64800	11	45056	1.44
XCV200	2	88200	13	53248	1.66
XCV300	2	115200	15	61440	1.88
XCV400	2	180000	19	77824	2.31
XCV600	2	259200	23	94208	2.75
XCV800	2	352800	27	110592	3.19
XCV1000	2	460800	31	126976	3.63

more memory, but it can speed up test application.

Table 5.4 shows the number of test responses for all the FPGAs in the Virtex family. It is possible to see that the number of test responses is influenced by the number of BISTs used in the TCs. It also shows the number of RAM ports available for each test circuit to store test responses. The available memory space is derived from the number of RAM ports available by multiplying it by 512, the number of bytes per port, and by 8, the number of bits in one byte.

Since the number of test responses is more than the number of bits available in RAM blocks, the test responses need to be encoded. Therefore, the minimum compression ratio needed is defined by the number of test responses divided by the number of bits available in the RAM blocks. The compression ratios needed are presented in Table 5.4. The biggest compression ratio needed is for the XCV1000, the biggest FPGA in the Virtex family.

Figure 5.3 shows a diagram that presents the manufacturing diagnostic test scheme. The major difference between Figure 5.3 and Figure 5.1, which shows the fault-tolerant scheme, is the encoder.

5.3.1 Run-Length Encoding

Run-length encoding (RLE) is a technique used to reduce the size of a sequence

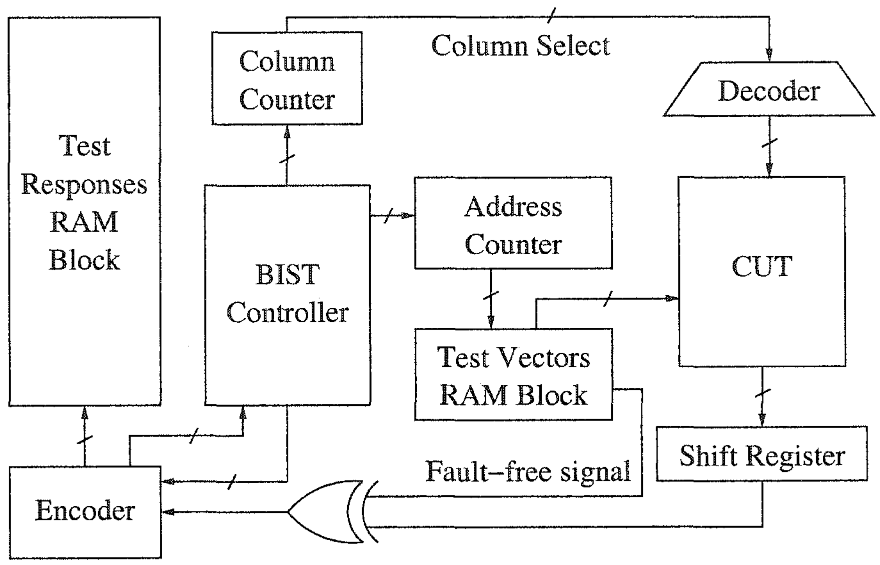


Figure 5.3: Manufacturing diagnosis test scheme

of characters by exploiting their repetitions. A repeating sequence of characters is called a *run*. RLE encodes a run into two values: a *count* and a *symbol*. For example, the binary sequence *111100111* can be encoded in three runs as {{4,1}, {2,0}, {3,1}} with a RLE encoder, where the first number in each couple is the count and the other is the symbol. In that way, RLE can encode any kind of data, but its compression ratio varies with the data content to be compressed. RLE is easy to implement in FPGAs with a few resources and is also quick to execute due to its simplicity. Therefore, RLE is a good solution for the compression of diagnostic test results in the manufacturing diagnostic scheme because the space to program the BIST is limited. Furthermore, encoding part of the scheme must be performed as fast as possible because time is always a major concern in testing.

Practically, the FPGA implementation of the RLE encoder is not complicated and can be implemented using a few FPGA logic resources. In the diagnostic test scheme, binary test results are stored in RAM blocks. The present work uses a simple and common way of storing the data in the FPGA.

The same memory is used to store the count and symbol. Therefore, the count and symbol data have different bits assigned in the memory word. In

Table 5.5: Example RLE in RAM port

Memory	Memory Content	
Address	Symbol	Count
$000	1	000 1001
$001	0	000 0001
$002	1	000 0100
$003	0	000 0010
...	...	...

the case of binary symbols, one bit in the word has to be assigned for the symbols. Table 5.5 shows a RAM port with encoded data of the stream of data *1111111110111100* into it.

In this example, the RAM port is configured to be 8 bits wide. The most significant bit of the word is assigned to the symbol and the rest to the count. In this particular case, the length of the counts can be up to 255. When a run has a count value greater than 255, the encoder resets the count and starts counting at the next memory space. When a RAM port is full, the encoder continues its encoding in the next port. With this kind of compression, all test responses can be recovered by a decoder outside the FPGA.

There is also no compression achieved in this example encoding because the number of bits in the input data (16 bits) is less than the number stored by the encoder (32 bits). That is due to the short runs compared to the capacity of each memory space. However, if the runs are much longer, some compression will be achieved.

5.3.2 Feasibility of the Scheme

A quick explanation of the test sequence is necessary to understand the feasibility analysis of the manufacturing diagnostic scheme. First, the BIST applies a test vector to all tiles in the CUT part of the FPGA. Next, column select signals select a column that becomes the *column under test.* All test results from the tiles in the column under test are compared one by one with a fault-free signal in the BIST part of the FPGA. Finally, a 1 or a 0, meaning passed and

```
1: for all Test Vectors do
2:     for all Column $\in$ CUT do
3:         for all Tile $\in$ Column do
4:             Compare the test response of the Tile with fault-free signal
5:         end for
6:     end for
7: end for
```

Figure 5.4: Test sequence

failed respectively, is sent to the encoder. Figure 5.4 shows a small pseudocode that presents the test sequence. In the present subsection, the sequence is simplified to make the concept easier to understand. But results on the feasibility of the scheme are according with the real scheme.

Since the feasibility of the scheme is related to the capacity of the encoder to compress the sequence of test results, the analysis will present the best and the worst cases from the point of view of the encoder.

There are two cases where the encoder will perform the best compression. One of these cases is when the CUT is all fault-free. When this occurs, the sequence of test results has identical symbols of logic 1s. In the case of a fault-free FPGA, all of the test responses can be stored if the RAM port width is set to 8 bits. The compression ratio can be easily calculated for the best case. Remember that the maximum number of test responses that can be stored in a 8-bit wide memory by the RLE is $2^8 - 1 = 255$. Therefore the compression ratio achieved in this case is represented by the following equation:

$$\begin{aligned} Compression\,ratio &= \frac{Number\ of\ test\ responses}{Memory\ available} \\ &= \frac{Capacity\ of\ a\ cell \times Number\ of\ memory\ cells}{Width\ of\ memory \times Number\ of\ memory\ cells} \\ &= \frac{Capacity\ of\ a\ cell}{Width\ of\ memory} \\ &= \frac{(Width\ of\ memory)^2 - 1}{Width\ of\ memory}. \end{aligned} \tag{5.2}$$

From Equation 5.2 it is possible to see that the compression ratio depends only of the *Width of memory* and it is independent of the *Number of memory*

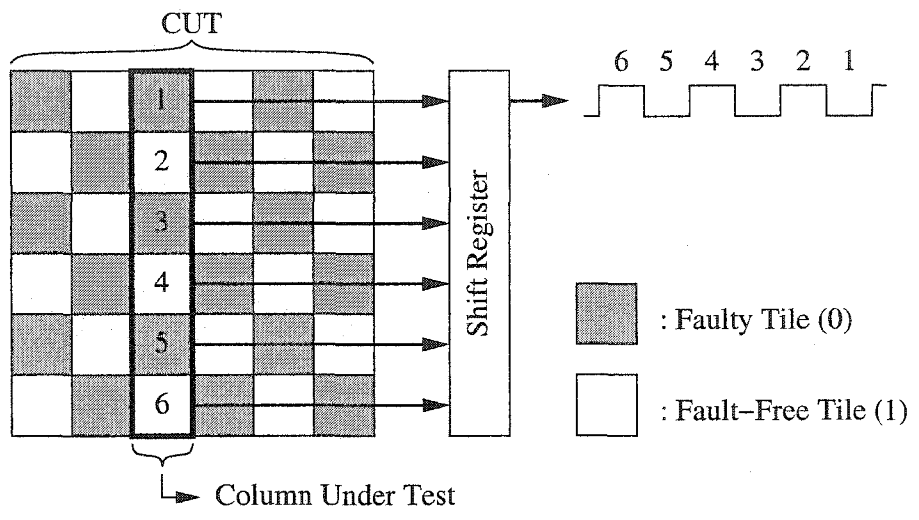


Figure 5.5: Worst case circuit under test

cells. Therefore, a compression ratio of $255/8 = 31.875$ can be found for an 8-bit wide memory. For a 16-bit wide RAM block, the compression ratio is even better. It is equal to $(16^2 - 1)/16 = 4095.9$. Since these compression ratios are larger than the one in Table 5.4, all the test responses can be stored in the RAM ports. With a similar calculation, it is possible to demonstrate that a 4 bit wide memory cannot store all the values even for the best case. That means that the RAM ports have to be configured either 8 bits or 16 bits wide.

A similar scenario occurs when the entire FPGA is faulty. In this case, the test responses will be all faulty and will generate a sequence of logic 0s. The compression ratio will be the same as before. Even if this case is very unlikely to happen, it is interesting to observe that all these values can be stored in the RAM blocks.

The worst case scenario for the encoder occurs when symbols are alternated every test response. In that case, there is not enough space in the RAM Blocks and the test must stop when the RAM blocks are full.

Let's assume that a tile gets a fault that always returns faulty responses to the BIST. The worst case happens when faulty and fault-free tiles are alternated in the CUT part of the FPGA. Figure 5.5 shows this scenario in

Table 5.6: Number of test vectors saved in the worst case

Virtex FPGA	Number of Test Vectors	
Device	8-bit Wide	16-bit Wide
XCV50	9.33	4.67
XCV100	7.68	3.84
XCV150	13.04	6.52
XCV200	11.32	5.66
XCV300	10.00	5.00
XCV400	8.11	4.05
XCV600	6.81	3.41
XCV800	5.88	2.94
XCV1000	5.17	2.58

the CUT part of the TC. This generates a checker board pattern in the CUT. In this case, the compression ratio is very low because only one value can be stored in one word in the memory regardless of the width of the memory.

To characterize the performance of the encoder in the worst case, the number of test vectors applied to the CUT that will fill the memories has been counted for each FPGA in the Virtex family. Table 5.6 shows these results. The numbers of test vectors are not integers because a fraction of a test vector can be stored in the RAM. Table 5.6 shows that for the worst case, it is better to have a memory with narrower words and more memory cells. Therefore, 8-bit wide RAM ports are better than 16-bit wide RAM ports in this case.

5.3.3 Postprocessing of Diagnostic Results

Similarly to the fault-tolerant diagnostic scheme, the information stored in the RAM blocks is read from outside the FPGA when the test is over. The diagnostic results from all the TCs can be analyzed outside the FPGA and can possibly lead to an accurate diagnostic resolution. The post processing should analyze diagnostic results of all TCs simultaneously and consequently determined faulty relations between results from different TCs to identify faulty resources as fine as an individual wire segment or PS. Therefore, the most accurate diagnostic resolution can be achieved. This analysis is outside the scope of this thesis, but it is mentioned as further work in the conclusion chapter.

blank page - no text

Chapter 6

Conclusion

State-of-the-art FPGAs, such as Xilinx Virtex FPGAs, are composed of tens of millions of *programmable switches* (PSs) and wire segments, which are accessible either through the programmable interconnect network or require the use of programmable logic blocks. The main challenge for interconnects testing is to verify the integrity of the tens of millions of programmable elements in the shortest time possible.

The way FPGA manufacturers test their FPGAs is proprietary information. Since FPGAs need to be tested by users after manufacturing, much research has been done to create FPGA test strategies for that case. One common approach employs a divide-and-conquer technique that tests some of these elements at a time, and uses a set of tests to complete the verification. Testing local interconnects is an especially hard task because test signals have to get in and out of *configurable logic blocks* (CLBs). Previous research is mostly on the XC4000 or similar architectures that have simpler local interconnect networks than the Virtex FPGA. This thesis presents the first work that heuristically derives *test configurations* (TCs) for the Virtex FPGAs.

It should be mentioned that the *built-in self-test* (BIST) test scheme presented in this thesis can be applied in manufacturing test as well. Since the test processing is inside the FPGA, the test can be performed in parallel on many FPGAs. The cost of testing can be considerably reduced if enough tests are done in parallel to beat the performance of *automatic test equipment.*

6.1 Main Contributions

In Chapter 3, a new BIST scheme for the Virtex FPGA is presented. This BIST scheme is different than previous schemes because test responses are carried by the tri-state buses. The local interconnects related to the tri-state buffer can then be tested. Another important feature of the BIST scheme is that test vectors are stored in the RAM blocks of the FPGA. Three sets of test vectors are created to detect common types of interconnects faults such as segment *stuck-at* 0/1, segment *stuck-open*, segment/*programmable switch* (PS) *bridge*, and PS *stuck on/off* faults.

One critical aspect of the test scheme is the local routing itself. Chapter 4 described an algorithm that can derive the minimum necessary number of TCs. The algorithm is based on a graph model. In the model, vertices and edges of the graph represent wire segments and PSs, respectively. The models of the three local interconnects are *bipartite* graphs. The algorithm uses graph *coloring* of bipartite graphs to build the TCs. The experiments on the Virtex resources show that the theoretical minimum number of TCs is equal to 26. The algorithm can derive that minimum number of TCs. The total time complexity for the algorithm is $O(|V|NBS)$. Though this time complexity may appear to be slow, the algorithm derives the Virtex FPGA local TCs in approximately six minutes.

Two diagnostic test schemes were presented in Chapter 5. These two tests were created for two different applications: fault-tolerant diagnosis and manufacturing diagnosis. In both schemes, test results are stored on chip in the RAM blocks, but they are stored differently. Because of the nature of a fault-tolerant system, it is only necessary to store the addresses (rows and columns) of the faulty FPGA tiles. In the case of the manufacturing diagnostic scheme, all the test results need to be stored. Since the space in the RAM blocks is not sufficient to store all of the test responses, an encoder was needed. A *run-length encoder* (RLE) was chosen for its simplicity, which makes it easy to implement and fast to execute in an FPGA. A two-BIST manufacturing scheme, that allows the access of the entire RAM block in the bigger Virtex FPGAs (XCV150 and over), was also presented. An analysis of the feasibility of this scheme with a RLE showed that all the test results can be stored when the FPGA is fault-free. The analysis also shows that, for the worst-case scenario, the scheme can save the test responses of 5 complete test vectors over a total of 150 test vectors.

6.2 Further Work

Though feasibility studies were performed on all the schemes presented in this thesis, none of the schemes were physically implemented in a real FPGA. Therefore, an implementation could be useful to see if the BIST can be implemented in half of the FPGA for the smallest FPGA. Simulations and tests could then be performed on the different test schemes to see how fast the tests can be loaded and applied.

For the manufacturing diagnostic test scheme, it could be interesting to perform further investigation and have finer test resolution. That could be achieved by comparing test responses between TCs. New sets of test vectors could be developed to have better diagnostic resolution. Therefore, test vectors could be different for every TC. They could also be adapted from the test results from the previously applied TCs.

More investigation into how TCs can be programmed in FPGAs and how data in the RAM blocks can be read should be part of the further work of this project. One interesting property of the Virtex/Spartan-II is that it is possible to use run-time configuration to partially reconfigure some parts of the FPGA when other parts stay the same. In the present case, the BIST can stay the same and the CUT can be reprogrammed. This can speed up the programming of the FPGA and also reduce the size of the TC outside the FPGA. It seems also possible to read only the content of the RAM blocks from outside the FPGA. This could speed up the reading of the test results in the diagnostic schemes. The modularity of local TCs can also be used to save memory space in the system outside the FPGA.

Bibliography

[1] M. L. Bushnell and V. D. Agrawal, *Essentials of Electronic testing for Digital, Memory and Mixed-signal VLSI Circuits*, Kluwer Academic Publishers, 2000.

[2] L. Zhao, D.M.H. Walker and F. Lombardi, "I_{DDQ} testing of input/output resources of SRAM-based FPGAs," in *Proc. of Asian Test Symposium*, 1999, pp. 375–380.

[3] L. Zhao, D. Walker and F. Lombardi, "I_{DDQ} testing of bridging faults in logic resources of reconfigurable field programmable gate arrays," *IEEE Transactions on Computers*, pp. 1136–1152, 1998.

[4] L. Zhao, D.M.H. Walker and F. Lombardi, "Bridging fault detection in FPGA interconnects using I_{DDQ}," in *Proc. of International Symposium on Field Programmable Gate Arrays*, 1998, pp. 95–104.

[5] W.K. Huang, X. Cheng and F. Lombardi, "On the diagnosis of programmable interconnect systems: theory and application," in *Proc. of IEEE VLSI Test Symposium*, 1996, pp. 204–209.

[6] Y. Yu, J. Xu, W.K. Huang and F. Lombardi, "A diagnosis method for interconnects in SRAM-based FPGAs," in *Proc. IEEE Asian Test Symposium*, 1998, pp. 278–282.

[7] Y. Yu, J. Xu, W.K. Huang and F. Lombardi, "Minimizing the number of programming steps for diagnosis of interconnects faults in FPGAs," in *Proc. of Asian Test Symposium*, 1999, pp. 357–362.

[8] M. Renovell, J. Figueras and Y. Zorian, "Test of RAM-based FPGA: Methodology and application to the interconnect," in *Proc. of IEEE VLSI Test Symposium*, 1997, pp. 230–237.

[9] M. Renovell, J.M. Portal, J. Figueras, and Y. Zorian, "Testing the interconnect of RAM-based FPGAs," *IEEE Design and Test of Computers*, vol. 15, no. 1, pp. 45–50, 1998.

[10] M. Renovell and Y. Zorian, "Different experiments in test generation for XILINX FPGAs," in *Proc. IEEE International Test Conference*, 2000, pp. 854–862.

[11] M.B. Tahoori, S. Mitra, S. Toutounchi and E.J. McCluskey, "Fault grading FPGA interconnect test configurations," in *Proc. of International Test Conference*, 2002, pp. 608–617.

[12] M.B. Tahoori and S. Mitra, "Automatic configuration generation for FPGA interconnect testing," in *Proc. VLSI Test Symposium*, 2003, pp. 134–139.

[13] H. Michinishi, T. Yokohira, T. Okamoto, T. Inoue and H. Fujiwara, "A test methodology for interconnect structures of LUT-based FPGAs," in *Proc. of Asian Test Symposium*, 1996, pp. 68–74.

[14] M. Renovell, J.M. Portal, J. Figueras, and Y. Zorian, "Testing the local interconnect resources of SRAM-based FPGAs," *Journal of Electronic Testing: Theory and Applications*, vol. 16, no. 5, pp. 513–520, 2000.

[15] A. Doumar and H. Ito, "Testing the logic cells and interconnect resources for FPGAs," *IEEE Asian Test Symposium*, pp. 369–374, 1999.

[16] C. Stroud, S. Wijesuriya, C. Hamilton and M. Abramovici, "Built-in self-test of FPGA interconnect," in *Proc. of IEEE International Test Conference*, 1998, pp. 404–411.

[17] M. Abramovici, C. Stroud, C. Hamilton, S. Wijesuriya and V. Verma, "Using roving STARs for on-line testing and diagnosis of FPGAs in fault-tolerant applications," 1999, pp. 973–982.

[18] X. Sun, J. Xu, B. Chan and P. Trouborst, "Novel technique for built-in self-test of FPGA interconnects," in *Proc. of IEEE International Test Conference*, 2000, pp. 795–803.

[19] X. Sun, S. Xu, J. Xu and P. Troubst, "Design and implementation of a parity-based BIST scheme for FPGA global interconnects," in *Proc. of IEEE Canadian Conference on Electrical and Computer Engineering*, 2001, vol. 2, pp. 1251–1257.

[20] X. Sun, J. Xu, A. Alimohammad and P. Trouborst, "Minimal test configurations for FPGA local interconnects," in *Proc. IEEE Canadian Conference on Electrical and Computer Engineering*, 2002, vol. 1, pp. 427–432.

[21] X. Sun, A. Alimohammad and P. Trouborst, "Modeling of FPGA local/global interconnect resources and derivation of minimal test configurations," in *Proc. IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, 2002, pp. 284–292.

[22] X. Sun, K. Ogden, H. Chan and P. Trouborst, "A novel FPGA local interconnect test scheme and automatic tc derivation/generation," *Journal of Systems Architecture (in press)*.

[23] C. Metra, A. Pagano and B. Ricco, "On-line testing of transient and crosstalk faults affecting interconnects of FPGA-implemented systems," in *IEEE International Test Conference*, 2001, pp. 939–947.

[24] M. Abramovici and C. Stroud, "BIST-based delay-fault testing in FPGAs," *Jounal of Electronic Testing: Theory and Application*, pp. 549–558, 2003.

[25] I.G. Harris and R. Tessier, "Interconnect testing in cluster-based FPGA architectures," in *Proc. of ACM/IEEE-CAS/EDAC Design Automation Conference*, 2000, pp. 49–54.

[26] I.G. Harris and R. Tessier, "Testing and diagnosis of interconnect faults in cluster-based FPGA architectures," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 21, no. 11, pp. 1337–1343, 2002.

[27] C. Nomikos, A. Pagourtzis and S. Zachos, "Routing and path multicoloring," *Information Processing Letters*, vol. 80, no. 5, pp. 249–256, 2001.

[28] S. Park, "A new complete diagnosis patterns for wiring interconnects," in *Proc. Design Automation Conference*, 1996, pp. 203–208.

[29] W. Cheng, J.L. Lewandowshi and E. Wu, "Diagnosis for wiring interconnects," in *Proc. of IEEE International Test Conference*, 1990, pp. 565–571.

[30] A. Hassan, J. Rajski and V.K. Agrawal, "Testing and diagnosis of interconnects using boundary scan architecture," in *IEEE International Test Conference*, 1988, pp. 126–137.

[31] T. Liu, F. Lombardi and J. Salinas, "Diagnosis of interconnects and FPICs using a structured walking-1 approach," in *Proc. of IEEE VLSI Test Symposium*, 1995, pp. 256–261.

[32] M. B. Tahoori, "Diagnosis of open defects in FPGA interconnect," in *Proc. of Conference on Field-Programmable Technology*, 2002, pp. 328–331.

[33] X.T. Chen, F.J. Meyer and F. Lombardi, "Structural diagnosis of interconnects by coloring," *ACM Transactions on Design Automation of Electronic Systems*, vol. 3, no. 2, pp. 249–271, 1998.

[34] M. Abramovici, M. A. Breuer and A. D. Friedman, *Digital Systems Testing and Testable Design*, IEEE Press, revised printing edition, 1990.

[35] "The programmable logic data book," Xilinx Inc., 2002.

[36] Y. Matsui and T. Uno, "A fast enumerating algorithm for edge colorings of bipartite graphs," *Submitted to Algorithmica.*

[37] R. Cole and J. Hopcroft, "On edge coloring bipartite graphs," *SIAM Journal on Computing*, vol. 11, no. 3, pp. 540–546, 1982.

[38] O. Ore, *The Four-Color Problem*, New York: Academic Press, 1967.

blank page - no text

Appendix A

Algorithm's Pseudocode

Algorithm 1 Edge Coloring of Bipartite Graph

```
 1: procedure BIPARTITE_GRAPH_EDGE_COLORING($G$)
 2:     $V_s \leftarrow$ Choose one set of vertices between $V_1$ and $V_2$
 3:     for all $V_{source} \in V_s$ do
 4:         for all $V_{drain} \in$ of adjacency list of $V_{source}$ do
 5:             $\alpha \leftarrow$ lower missing color at edges adjacent to $V_{source}$
 6:             $\beta \leftarrow$ lower missing color at edges adjacent to $V_{drain}$
 7:             if $\alpha \neq \beta$ then
 8:                 $P \leftarrow$ path where colors $\alpha$ and $\beta$ alternate (start with $\beta$ at
                        $V_{source}$)
 9:                 Interchange colors $\alpha$ and $\beta$ in $P$
10:             end if
11:             assign color $\beta$ to edge $V_{source}$,$V_{drain}$
12:         end for
13:     end for
14: end procedure
```

Algorithm 2 Enumaration of Edge Coloring

```
procedure ENUM_EDGE_COLORING($G$)
    if G is a matching then
        print ADD last color $G$
    else
        $V' \leftarrow$ max_degree_vertices($G$)
        $e \leftarrow$ edge incident to a vertex in $V'$
        $M \leftarrow$ find_covering_matching($G$)
        enum_covering_matching($G^+(e), V', M$)
    end if
end procedure

procedure ENUM_COVERING_MATCHING($G, V', M$)
    if a new match $M' \neq M$ can be found in $G$ then
        $e \leftarrow$ edge in $M \backslash M'$
        enum_covering_matching($G^+(e), V', M$)
        enum_covering_matching($G \backslash e, V', M'$)
    else
        print ADD color $M$
        enum_edge_coloring($G \backslash M$)
        print DELETE color $M$
    end if
end procedure
```

Algorithm 3 Enumaration of Edge Coloring (Modified)

```
procedure ENUM_EDGE_COLORING($G$)
    if G is a matching then
        if match_routable($G$) then                                ▷ NEW
            print ADD last color $G$
            exit program                                           ▷ NEW
        end if
    else
        $V' \leftarrow$ max_degree_vertices($G$)
        $e \leftarrow$ edge incident to a vertex in $V'$
        $M \leftarrow$ find_covering_matching($G$)
        enum_covering_matching($G^+(e), V', M$)
    end if
end procedure

procedure ENUM_COVERING_MATCHING($G, V', M$)
    if a new match $M' \neq M$ can be found in $G$ then
        $e \leftarrow$ edge in $M \backslash M'$
        enum_covering_matching($G^+(e), V', M$)
        enum_covering_matching($G \backslash e, V', M'$)
    else
        if match_routable($G$) then                                ▷ NEW
            print ADD color $M$
            enum_edge_coloring($G \backslash M$)
            print DELETE color $M$
        end if
    end if
end procedure
```

Algorithm 4 Verification of Color Routability

```
1: procedure COLOR_ROUTABLE($M$,CLB_conf,OUT_adj_list)
2:     for all Conf ∈ CLB_conf do
3:         conf_ok ← 1
4:         for all Request ∈ Conf do
5:             if not possible to route the Request with the corresponding edge
                   in $M$ according to the OUT_adj_list then
6:                 conf_ok ← 0
7:             end if
8:         end for
9:         if conf_ok = 1 then
10:            Return 1
11:        end if
12:    end for
13:    Return 0
14: end procedure
```

Algorithm 5 Completion of G_{ORM} Coloring

```
1: procedure G_ORM_COLORING_COMPLETION($C$,CLB_conf,
       OUT_adj_list, ORM_adj)
2:     for all $M$ ∈ Incomplete colors in $C$ do
3:         repeat
4:             CLB_conf_tmp ← CLB_conf
5:             Conf ← push(CLB_conf_tmp)
6:             repeat
7:                 $M^*$ ← new_missing_edge_combination($M$,$M^*$,ORM_adj)
8:                 conf_ok ← 1
9:                 for all Request ∈ Conf do
10:                    if not possible to route the Request with the correspond-
                           ing edge in $M^*$ according to the OUT_adj_list then
11:                        conf_ok ← 0
12:                    end if
13:                end for
14:            until No more M* or conf_ok = 1
15:        until CLB_conf_tmp is empty or conf_ok = 1
16:        if conf_of = 1 then
17:            Replace $M$ by $M^*$
18:        end if
19:    end for
20: end procedure
```

Algorithm 6 Feedback Routing

```
1: procedure FEEDBACK_ROUTING($C$,CLB_conf,OUT_adj,IRM_adj)
2:     for all $M \in C$ do
3:         repeat
4:             CLB_conf_tmp $\leftarrow$ CLB_conf
5:             Conf $\leftarrow$ push(CLB_conf_tmp)
6:             conf_ok $\leftarrow$ 1
7:             for all $E \in M$ do
8:                 if path_is_found($E$, Conf, OUT_adj, IRM_adj) = 0 then
9:                     conf_ok $\leftarrow$ 0
10:                end if
11:            end for
12:        until CLB_conf_tmp is empty or conf_ok = 1
13:    end for
14: end procedure
```

Algorithm 7 Completing Test Configurations in IRM

```
1: procedure COMPLETE_COLORING_1(Feedbacks, $G_{IRM}$)
2:     for all $C \in$ colors do
3:         $G_{tmp} \leftarrow G\_IRM$
4:         for all Feedback Edge E=$\{u, v\}$ in $G_{IRM}$ do
5:             Delete $u$ and $v$ in $G_{tmp}$
6:         end for
7:         $M \leftarrow$ find a Match in $G_{tmp}$
8:         Put $M$ in $C$
9:     end for
10: end procedure
```

Algorithm 8 Completing Test Configurations in IRM

```
1: procedure COMPLETE_COLORING_2(coloring, $G_{IRM}$, OUT_line_adj)
2:     for all $C \in$ coloring do
3:         repeat
4:             CLB_conf_tmp $\leftarrow$ CLB_conf
5:             Conf $\leftarrow$ push(CLB_conf_tmp)
6:             conf_ok $\leftarrow 1$
7:             for all Request $\in$ Conf do
8:                 if Request cannot be route in IRM then
9:                     conf_ok $\leftarrow 0$
10:                end if
11:            end for
12:        until CLB_conf_tmp is empty or conf_ok $= 1$
13:        Add the routing in the present color according to the configuration
14:        Add edges in IRM to complete the coloring
15:    end for
16: end procedure
```

Appendix B

Perl Programs

B.1 Edge Coloring of G_{ORM}

```
#!/usr/local/bin/perl
#################################################################################
=head1 NAME

good_coloring

=head1 AUTHOR (DATE)

Christian Giasson (June 18th, 2003)

=head1 SYNOPSIS

enum_edge_coloring  -a  {name of the file that contains the ORMG}
            -b  {name of the file that contains the CLB_conf}
            -c  {name of the file that contains the OUT_adj_list}

=head1 DESCRIPTION

This program find a edge coloring of a particular bipartite graph (if one
exist) that is agree with the CLB_configuration and the adj list of the OUT
lines.

This program is based on the paper:

A Fast Enumerating Algorithm for Edge Coloring in Bipartite Graphs
by Yasuko MATSUI and UNO

=over 4

=item INPUT

ORMG
A graph on an adjacency list format. This program is only for Bipartite
Graphs.

Note: the edges of the graph are defined only for the first group of
vertices (V1). Then the adjacent list for the second group {V2} is empty

CLB_conf

OUT_adj_list

=item OUTPUT

Outputs is a coloring and is passed as a standard output. The output format
can seems strange. It is composed of ADD and DELETE of matches. The ouput
generate a tree of coloring.

=item DEPENDENCIES

This program has some dependencies on some libraries:

Graph;
Graph::Directed;
Graph::Undirected;
Graph::Writer::Dot;
Graph::DFS;
Set::Array;
```

```
note: those libraries are not part of the standard Perl libraries. You can
find these libraries www.cpan.org

=cut
##############################################################################
use warnings;
use strict;
use Graph;
use Graph::Directed;
use Graph::Undirected;
use Graph::Writer::Dot;
use Graph::DFS;
use Set::Array;
use Getopt::Std;


##############################################################################
#Get the inputs of the program
##############################################################################

getopts('ha:b:c:');
our($opt_h, $opt_a, $opt_b, $opt_c);

if ($opt_h)
{
    print"\n\tOption_of_this_program:\n
________\t-h:_\t\t\thelp
________\t-a_[file]:\t\tadj_list_of_the_ORMG
________\t-b_[file]:\t\tlist_of_the_CLB_configuration
________\t-c_[file]:\t\tadj_list_of_the_OUT_lines\n\n";
        exit;
}

my @adj_G_input = &read_file($opt_a) if (($opt_a)&&(!$opt_h));
my @CLB_conf_input = &read_file($opt_b) if (($opt_b)&&(!$opt_h));
my @OUT_adj_input = &read_file($opt_c) if (($opt_c)&&(!$opt_h));


##############################################################################
#build the ORM graph
##############################################################################
my $global_graph = Graph->new();
#put edges in the Graph
my @graph_edges = &adj2edge(\@adj_G_input);
$global_graph->add_edges(@graph_edges);

#global count to test the program
my $level = 1;
my $count_coloring = 0;

##############################################################################
#build the data structure of the CLB_conf and the OUT_adj_list
##############################################################################

#data structure for CLB_conf is a hash of arrays
my %CLB_conf;
foreach my $conf (@CLB_conf_input)
{
    if( $conf =~ /(.*): (.*)/)
    {
        my $conf_name = $1;
        my $fb = $2;
        @{$CLB_conf{$conf_name}} = split(/ /,$fb);
    }
}

#data structure for OUT_adj is a hash of arrays
my %OUT_adj_list;
my %OUT_adj_list_0;
my %OUT_adj_list_1;
foreach my $OUT_line (@OUT_adj_input)
{
    if( $OUT_line =~ /(.*): (.*)/)
    {
        my $OUT = $1;
        @{$OUT_adj_list{$1}} = split(/ /,$2);
        #print "@{$OUT_adj_list{$1}}\n";
        ######################################################################
        #   Split the OUT lines adj list into two list of connections
        #   OUT_adj_list_0 : edges related to CLB0
        #   OUT_adj_list_1 : edges related to CLB1
        #   note1:  The CLB inputs are in the same format than the CLB
        #       configuration.
        #   note2:  The T in OUT_adj_list_0 is T0 and
        #       The T in OUT_adj_list_1 is T1.
        ######################################################################

        foreach (@{$OUT_adj_list{$OUT}})
        {
            if ($_ =~ /S(\d)_(.*)_B(.*)/)
            {
                my $num = $3;
                if ($1 == 0)
                {
                    if ($num eq '')
                    {push (@{$OUT_adj_list_0{$OUT}},"$2");}
                    else
                    {push (@{$OUT_adj_list_0{$OUT}},"$2$num");}
                }
                else
                {
                    if ($3 eq '')
                    {push (@{$OUT_adj_list_1{$OUT}},"$2");}
                    else
                    {push (@{$OUT_adj_list_1{$OUT}},"$2$3");}
                }
            }
            #Modifications to allow the CLBO to connect to the
            # tri-state buffer 1 and vice versa

            elsif ($_ =~ /T_IN(\d)/)
            {
                if ($1 == 0)
                {
                    push (@{$OUT_adj_list_0{$OUT}},"T0");
                    push (@{$OUT_adj_list_1{$OUT}},"T0");
                }
                else
                {
                    push (@{$OUT_adj_list_1{$OUT}},"T1");
                    push (@{$OUT_adj_list_0{$OUT}},"T1");
                }
            }
        }
        #test
        #print "$OUT CLB0: @{$OUT_adj_list_0{$OUT}}\n";
        #print "$OUT CLB1: @{$OUT_adj_list_1{$OUT}}\n";
    }
}

##########################################################################
# Main part: call of the principal recursive function enum_edge_coloring
##########################################################################

&enum_edge_coloring(\@graph_edges);

##########################################################################
#output the graph in a .dot format (graphviz)
##########################################################################
#my $dot_writer = Graph::Writer::Dot->new();
#$dot_writer->write_graph($graph, 'mygraph.dot');


#######################################################################
#######################################################################
#######################################################################
#            Functions
#######################################################################
#######################################################################
#######################################################################

#####################################################################
#Function Name: enum_edge_coloring
#Inputs: A list of edges that belongs to G
#Output:
#note:
#####################################################################
sub enum_edge_coloring
{
    #get the input data
    my @edges = @{(shift(@_))};
    #build the graph G
    my $G = Graph->new;
    $G->add_edges(@edges);
    #####################################################################
    # 1. if all edges are incident to an edge or
    #if G is a matching then output "edge coloring by adding"
    #an the unique edge coloring of G.
    #
    #modification to that part of the algorithm:
    #check if the graph is a matching
    #if it is a matching then Output the match
    #
    # probably that part is there to avoid the generation of empty problems
    #####################################################################

    #Call the check_if_match
    my $if_match = &check_if_matching(\@edges);
    if ($if_match == 1)
    {
        my @M_copy = @edges;
        my $OK = &match_ok(\@M_copy,\%CLB_conf,\%OUT_adj_list_0,\%OUT_adj_list_1);
        if($OK == 1)
        {
            print "ADD_the_MATCH: @edges\n";
            #exit the program when the match of a coloring is found
            exit;
        }
        else
        {
            return;
        }
        $count_coloring++;
        #print "Coloring number $count_coloring\n";
        return;
    }

    ##########################################################################
    # 2. find a covering matching M of G
    #(A match that cover every vertex in V_max)
    ##########################################################################

    #indenfify the set V_max
    my @V_max = &find_V_max(\@edges);
    #find a first covering matching
    my @M = &C1(\@edges);
    #note: the founctions find_covering matching and &MD2 are old functions
    #       that did not work.  You can find a copy of those functions in the
    #       last version of enum_edge_coloring. C1 is the good function to
    #       find the first covering matching.
    #my @M = &find_first_covering_matching(\@edges,\@V_max);
    #test
    #print "the graph : @edges\n";
    #my @M = &MD2(\@edges);

    ##########################################################################
    # 3. Choose (can be random pick) an edge "e" in M
    ##########################################################################

    #let's choose the first one
    my @M_copy = @M;
    my @e;
    push(@e, shift(@M_copy));
    push(@e, shift(@M_copy));

#   print for test
#   print "***************** enum_edge_coloring ***********************\n
#   print "Edges of the graph = @edges\n";
#   print "Maximum degree vertices = @V_max\n";
#   print "First match = @M\n";
#   print "Global e = @e\n";

    ##########################################################################
    # 4. Call enum_covering_matching(G+(e), V_max, M)
    ##########################################################################

    #G+(e) is the subgraph of G obtained by removing "e" and all edges
    #adjacent to "e".
    #create G_plus_e
    my $G_plus_e = Graph->new;
    $G_plus_e->add_edges(@edges);

    #print "e = @e\n";
    #create the graph G+(e)
    foreach (@e)
    {
        $G_plus_e->delete_vertex($_);
    }

    #list of edges of G+(e)
    my @G_plus_e_edges = $G_plus_e->edges;
    #print "@G_plus_e_edges\n";

    &enum_covering_matching(\@G_plus_e_edges, \@V_max, \@M);
}
##########################################################################
#Function Name: enum_covering_matching
#Inputs:
#Output:
#note:
##########################################################################
sub enum_covering_matching
{
    #get the input data
    my @edges = @{(shift(@_))};
    my @V_max = @{(shift(@_))};
```

```
        my @M = @{(shift(@_))};
        #build the G passed to the function
        my $G = Graph->new;
        $G->add_edges(@edges);

        ##########################################################################
        # Check if the graph is empty (part added to the Algorithm)
        ##########################################################################

        #my $edge_count = @edges;
        #if ($edge_count == 0)
        #{return 0;}

        ##########################################################################
        # Check if there is a cycle or not in G
        # Call the function find_covering_matching()
        ##########################################################################

        #build the list @V1 and @V2 they are the partition of the graph
        my @V1 = &find_V1(\@edges);
        #print "V1 = @V1\n";
        #substract @V1 to all the vertices to get @V2
        #Set for all the verices and for V1
        my $all_V = Set::Array->new;
        my $V1 = Set::Array->new;
        #buid $all_V
        foreach (@edges) {$all_V->push($_);}
        #build $V1
        foreach (@V1) {$V1->push($_);}
        #$V2 = $all_V \ $V1
        my @V2 = $all_V->difference($V1);
        #print "V2 = @V2\n";


        #(\@{list_of_edges_G},\@{V1},\@{V2},\@{V_max},\@{edges_match})
        my @M_prime = &find_covering_matching(\@edges,\@V1,\@V2,\@V_max,\@M);

        ##########################################################################
        # if there is no cycle in D then
        ##########################################################################
        my $M_prime_count = @M_prime;

        #test
        my $M_prime_text = "@M_prime";
#       print for debugging
#       print "***************** enum_covering_matching ***********************\n";
#       print "edge of G = @edges\n";
#       print "V1 = @V1\n";
#       print "V2 = @V2\n";
#       print "V_max = @V_max\n";
#       print "count = $M_prime_count\n";
#       print "M = @M\n";
#       print "M_prime = $M_prime_text\n";

        if ($M_prime_count == 1) #if there is only a 0 in the match_prime
        {
            #NEW : verify if the match can be route
            my @M_copy = @M;
            my $OK = &match_ok(\@M_copy,\%CLB_conf,\%OUT_adj_list_0,\%OUT_adj_list_1);
            if ($OK == 1)
            {
                #Output : "Add" and the edges of M
                print "ADD_the_match:@M\n";

                #Substact the Match M from the global_graph
                $global_graph->delete_edges(@M);

                #test
                #$level++;
                #if ($level < 7)
                #{print "level: $level  num coloring done: $count_coloring\n";}
                #print "level:$level\n";

                #Call the function enum_edge_coloring(G\M)
                #This recursive call defines a new problem with the graph G
                #minus the covering matching M
                my @graph_edges = $global_graph->edges;
                &enum_edge_coloring(\@graph_edges);

                #Output : "Delete" and the edges of M
                print "DELETE_the_match:@M\n";

                #put back the match M in the global_graph
                $global_graph->add_edges(@M);

                #test
                $level--;
            }
        }
```

```
##################################################################
# if there is a cycle in D then
##################################################################
else
{
    #1. Choose an edge "e" in M\M'
    my $s_M = Set::Array->new;
    my $s_M_prime = Set::Array->new;
    #build the set of edges in M
    my $index = 1;
    my $u;
    my $v;
    foreach my $vertex (@M)
    {
        if ($index == 1)
        {
            $u = $vertex;
            $index = 0;
        }
        else
        {
            $v = $vertex;
            $s_M->push("$u-$v");
            $index = 1;
        }
    }
    #build the set of edges in M_prime
    $index = 1;
    foreach my $vertex (@M_prime)
    {
        if ($index == 1)
        {
            $u = $vertex;
            $index = 0;
        }
        else
        {
            $v = $vertex;
            $s_M_prime->push("$u-$v");
            $index = 1;
        }
    }

    ##################################################
    # new thing added to the algorithm
    # Check if M' include M if true then swap M<->M'
    ##################################################
    my $not_include = 0; #assume that is include
    $s_M->foreach(sub{
        if(!($s_M_prime->exists($_)))
        {
            $not_include = 1;
        }
    });

    my @diff_Ms;
    if ($not_include == 1)
    {
        #M\M'
        @diff_Ms = $s_M->difference($s_M_prime);
    }
    else
    {
        #M\M'
        @diff_Ms = $s_M_prime->difference($s_M);
    }

    if(scalar(@diff_Ms) == 0)
    {
        print "ERROR:_M_same_as_M'\n";
        exit;
    }


    #get a e from the difference
    my $e_temp = shift(@diff_Ms);
    my @e = split(/-/,$e_temp);

    #print for test
    #print "e = @e\n";


    #2. Build the graph G+(e)
    my $G_plus_e = Graph->new;
    $G_plus_e->add_edges(@edges);
    #create the graph G+(e)
    foreach (@e)
    {
        $G_plus_e->delete_vertex($_);
```

```
            }
            #list of edges of G+(e)
            my @G_plus_e_edges = $G_plus_e->edges;
            #print "@G_plus_e_edges\n";

            #3. Build the graph G\e
            my $G_diff_e = Graph->new;
            $G_diff_e->add_edges(@edges);
            $G_diff_e->delete_edge(@e);
            my @G_diff_e_edges = $G_diff_e->edges;
            #print "@G_diff_e_edges\n";

            #4. call enum_covering_matching(G+(e), V_max, M)
            &enum_covering_matching(\@G_plus_e_edges, \@V_max, \@M);

            #5. call enum_covering_matching(G\e, V_max, M')
            &enum_covering_matching(\@G_diff_e_edges, \@V_max, \@M_prime);
        }
}
##########################################################################
#Function Name: find_V1
#Inputs: A list of graph edges
#Output: A list of vertices that composed V1
#note:
##########################################################################
sub find_V1
{
    #get the input data
    my @edges = @{(shift(@_))};
    #build the graph G
    my $G = Graph->new();
    $G->add_edges(@edges);

    #define a set for V1
    my $V1 = Set::Array->new;

    #for all the edges
    my $half_index = @edges;
    $half_index /= 2;
    for (1..$half_index)
    {
        my $u = shift(@edges);
        my $v = shift(@edges);
        $V1->push($u);
    }
    $V1->unique();

    my @V1;
    $V1->foreach(sub{push(@V1,$_)});
    return @V1;
}
##########################################################################
#Function Name: check_if_matching
#Inputs: A list of graph edges
#Output: 1 if it is a Match and 0 if it's not a Match
#note:
##########################################################################
sub check_if_matching
{
    #get the input data
    my @edges = @{(shift(@_))};
    #build the graph G
    my $G = Graph->new();
    $G->add_edges(@edges);
    my @vertices = $G->vertices;
    #check if it is a matching
    foreach my $vertex (@vertices)
    {
        my @adj = $G->neighbors("$vertex");
        my $adj_size = @adj;
        if($adj_size > 1)
        {
            #print "graph not a match\n";
            return 0;
        }
    }

    #print "graph is a match\n";
    return 1;
}
##########################################################################
#Function Name: find_V_max
#Inputs: A list of graph edges
#Output: A list (array) of all the max degree vertices
#note:
##########################################################################
sub find_V_max
{
    #get the input data
    my @edges = @{(shift(@_))};
    #build the graph G
    my $G = Graph->new();
    $G->add_edges(@edges);
    my @vertices = $G->vertices;

    #find a set of the maximum degree vertices
    my @V_max;
    my $max_degree = 0;
    #find the maximum degree
    foreach (@vertices)
    {
        my @adj = $G->neighbors($_);
        my $adj_size = @adj;
        if ($adj_size > $max_degree)
        {
            $max_degree = $adj_size;
        }
    }
    #find all the vertices with a degree of $max_degree
    foreach (@vertices)
    {
        my @adj = $G->neighbors($_);
        my $adj_size = @adj;
        if ($adj_size == $max_degree)
        {
            push(@V_max,$_);
        }
    }
    return @V_max;
}
##################################################################
#Function Name: adj2edge
#Inputs: &adj2edge(@{adj_list})
#Output: return a list with all pair of vertices (edges)
#note: The element of the adj should be on that format
#    A:B,C,D
##################################################################
sub adj2edge
{
    my @adj_line = @{(shift(@_))};
    my @return_list;

    foreach my $line (@adj_line)
    {
        my $f_vertex;
        my @s_vertex;

        if($line =~ /^(.*):(.*)$/)
        {
            $f_vertex = $1;
            @s_vertex = split(",",$2);
            foreach (@s_vertex)
            {
                push(@return_list,"$f_vertex");
                push(@return_list,"$_");
            }
        }
    }
    return @return_list;
}
##################################################################
#Function Name: dot_the_graph
#Inputs: &dot_the_graph(${name},@{list_of_edges})
#Output: a ${name}.dot file in the current directory
#note: this function create a directed .dot file no mater what
##################################################################

sub dot_the_graph
{
    my $local_graph_name = shift;
    my @local_edges = @_;
    my $local_graph = new Graph::Undirected;
    $local_graph->add_edges(@local_edges);

    #output the graph in a .dot format (graphviz)
    my $dot_writer = Graph::Writer::Dot->new();
    $dot_writer->write_graph($local_graph, "$local_graph_name.dot");
}
##################################################################
#Function Name: find_covering_matching
#Inputs:
#(\@{list_of_edges_G},\@{V1},\@{V2},\@{V_max},\@{edges_match})
#Output: return a list(an array) of edges that represent the match
#    or 0 if there is no match.
##################################################################
```

```
sub find_covering_matching
{
    ##################################################################
    # put all the input of the function in arrays
    ##################################################################

    my @graph_edge = @{(shift(@_))};
    my @graph_V1 = @{(shift(@_))};
    my @graph_V2 = @{(shift(@_))};
    my @graph_V_max = @{(shift(@_))};
    my @graph_edge_match = @{(shift(@_))};

    #test
    #print "graph = @graph_edge\n";
    #print "V1 = @graph_V1\n";
    #print "V2 = @graph_V2\n";


    ##################################################################
    # create the sets of vertices that are covered by the graph
    ##################################################################

    my $s_g_covered = Set::Array->new(@graph_edge);
    $s_g_covered->unique;

    ##################################################################
    # create the sets V1, V2, V_max, and m_covered
    ##################################################################

    my $s_V1 = Set::Array->new(@graph_V1);
    my $s_V2 = Set::Array->new(@graph_V2);
    #create V_max (set of maximum degree vertices)
    my $s_V_max = Set::Array->new(@graph_V_max);
    #create the set of vertices covered by the input matching
    my $s_m_covered = Set::Array->new(@graph_edge_match);

    ##################################################################
    # create the set $s_V_bar
    ##################################################################

    #create V_bar (a special set in the algorithm)
    my $s_V_bar = Set::Array->new();
    my $s_temp1 = Set::Array->new();
    my $s_temp2 = Set::Array->new();

    #(vertices covered by M in V1)that are not in V_max in $s_temp1
    $s_m_covered->foreach(sub{$s_temp1->push($_)});
    my @temp1 = $s_temp1->intersection($s_V1)->difference($s_V_max);
    $s_temp1->clear;
    foreach (@temp1) {$s_temp1->push($_);}

    #(vertices uncovered by M in V2)that are not in V_max in $s_temp2
    $s_V2->foreach(sub{$s_temp2->push($_)});
    my @temp2 = $s_temp2->difference($s_m_covered)->difference($s_V_max);
    $s_temp2->clear;
    foreach (@temp2) {$s_temp2->push($_);}

    #Union sets $s_temp1 an $s_temp2 in $s_V_bar
    @temp1 = $s_temp1->union($s_temp2);
    foreach (@temp1) {$s_V_bar->push($_);}
    #$s_V_bar->print(1);

    ##################################################################
    # create the set $s_V_rest
    ##################################################################

    #create $s_V_rest (It's a special set in the algorithm)
    #$s_V_rest = $s_g_covered \ ( ($s_V_max) U ($s_V_bar) )
    my $s_V_rest = Set::Array->new();
    @temp1 = $s_V_max->union($s_V_bar);
    $s_temp1->clear;
    foreach (@temp1) {$s_temp1->push($_);}
    @temp1 = $s_g_covered->difference($s_temp1);
    foreach (@temp1) {$s_V_rest->push($_);}
    #$s_V_rest->print(1);

    ##################################################################
    # create the new directed graph D
    ##################################################################

    my $graph_D = new Graph::Directed;
#   ---------first: Oriente edges of M from V1 to V2---------
#   my @edge_temp = @graph_edge_match;
#   my $for_var = @edge_temp;
#   $for_var = $for_var/2;
#   for (1..($for_var))
#   {
#   my $u = shift(@edge_temp);
#     my $v = shift(@edge_temp);
#     if(!($s_V1->exists("$u")) && !($s_V2->exists("$v")))
#     {
#         $graph_D->add_edge($v,$u);
#     }
#     else
#     {
#         $graph_D->add_edge($u,$v);
#     }
#     }

      #test
      my @edge_temp = @graph_edge_match;
      for(1 .. (scalar(@edge_temp)/2))
      {
          my $u = shift(@edge_temp);
          my $v = shift(@edge_temp);
          $graph_D->add_edge($u,$v);
      }

      #-------- second: Oriente edges of E\M from V2 to V1------
      #create a set E (edges in the input graph)

      my @g_edge;
      @edge_temp = @graph_edge;
      my $for_var = @edge_temp;
      $for_var = $for_var/2;
      for (1..($for_var))
      {
          my $u = shift(@edge_temp);
          my $v = shift(@edge_temp);
          if(($s_V1->exists("$u")) && ($s_V2->exists("$v")))
          {
              push(@g_edge,"$v->$u");
          }
          else
          {
              push(@g_edge,"$u->$v");
          }
      }

      my $s_g_E = Set::Array->new(@g_edge);
      #create set M (edges in the match)

      #test
      my @m_edge;
      @edge_temp = @graph_edge_match;
      for(1 .. (scalar(@edge_temp)/2))
      {
          my $u = shift(@edge_temp);
          my $v = shift(@edge_temp);
          push (@m_edge,"$v->$u");
      }

      my $s_m_E = Set::Array->new(@m_edge);
      #E\M
      my @edge_V2_to_V1 = $s_g_E->difference($s_m_E);

      foreach my $local_edge (@edge_V2_to_V1)
      {
          if($local_edge =~ /(.*)->(.*)/)
          {
              $graph_D->add_edge($1,$2);
          }
      }

      #-------- third: create a vertex S in the graph------

      $graph_D->add_vertex("S_ADD");

      #---------Add edges from S to all the element in V_bar------

      $s_V_bar->foreach(sub{$graph_D->add_edge('S_ADD',"$_");});

      #---------Add edges from V_rest to V------

      $s_V_rest->foreach(sub{$graph_D->add_edge("$_",'S_ADD');});

      ##############################################################
      # Find a cycle in the graph D
      ##############################################################

      #---------Put the graph in a adjacency list form---------
      my $graph_adj = '';
      foreach my $vertice ($graph_D->vertices)
      {
          my @adj = $graph_D->successors($vertice);
          my $temp = '';
          foreach (@adj) {$temp = "$temp$_,";}
```

```
    if($temp =~ /(.*),$/)
    {$temp = $1;}
    $graph_adj = "$graph_adj$vertice:$temp\n";
}

#--------- Call the graph_cycle.pl program to find a cycle---------
#--------- Call the function graph_cycle to find a cycle---------

my @graph_D_edges = $graph_D->edges;

my @cycle = &graph_cycle(\@graph_D_edges);
my $cycle_le = @cycle;

if ($cycle_le < 2)
{
    return 0;
}

#--------- Build a graph with the path------------------

#need repeat in the middle
my $cycle_length = @cycle;
$cycle_length -= 2;
push(@cycle,(shift(@cycle)));
for (1..$cycle_length)
{
    my $cycle_temp = shift(@cycle);
    push(@cycle,$cycle_temp);
    push(@cycle,$cycle_temp);
}
push(@cycle,(shift(@cycle)));
#print "@cycle\n";
my $cycle_l = @cycle;
$cycle_l /= 2;


my $graph_cycle = Graph->new();
for (1..$cycle_l)
{
    my $u = shift(@cycle);
    my $v = shift(@cycle);
    $graph_cycle->add_edge($u,$v);
}

#--------- Delete the vertex S-------------------------

$graph_cycle->delete_vertex('S_ADD');

##################################################################
# create the other match
##################################################################

#------------ 1. Build a Set of edges from the input match-----

my $s_m_edges = Set::Array->new();
my @graph_edge_match_copy = @graph_edge_match;
my $half_index = @graph_edge_match_copy;
$half_index /= 2;
for (1..$half_index)
{
    my $u = shift(@graph_edge_match_copy);
    my $v = shift(@graph_edge_match_copy);
    if ($s_V1->exists("$v"))
    {
        $s_m_edges->push("$v-$u");
    }
    else
    {
        $s_m_edges->push("$u-$v");
    }
}

#----------- 2. Build a Set of edge from the cycle-----------

my $s_cycle = Set::Array->new();
my @cycle_edges = $graph_cycle->edges;
$half_index = @cycle_edges;
$half_index /= 2;
for (1..$half_index)
{
    my $u = shift(@cycle_edges);
    my $v = shift(@cycle_edges);
    if ($s_V1->exists("$v"))
    {
        $s_cycle->push("$v-$u");
    }
    else
    {
        $s_cycle->push("$u-$v");
        }
    }

    #------------ 3. Perform the simetric difference M'=M delta E'

    my @new_match = $s_m_edges->symmetric_difference($s_cycle);
    @new_match = split(/[\s-]/,"@new_match");
    return @new_match;
    #print "@new_match\n";
}
#######################################################################
#Function Name: graph_cycle
#Inputs:
#(\@{list_of_edges_G})
#Output:
#The ouput format is a string with all the sequence of vertices in the
#cycle separated with space.
#note: that function has been cleened. previous version of that function can
#   be find in the latest version of "enum_edge_coloring".
#######################################################################
sub graph_cycle
{
    my @graph_edge = @{(shift(@_))};
    #build the graph
    my $graph = Graph->new();

    $graph->add_edges(@graph_edge);

    my @l_vertices = $graph->vertices;

    #do a DFS on the graph
    my %param;
    my $dfs = Graph::DFS->new($graph,%param);
    my @dfs_edges = $dfs->edges;

    #put the DFS result in an array of edges
    my $ed;
    my @dfs_edge_seq;
    my $toggle=1;
    foreach my $ve (@dfs_edges)
    {
        if($toggle == 1)
        {
            $ed = $ve;
            $toggle = 2;
        }
        else
        {
            $ed = "$ed-$ve";
            push (@dfs_edge_seq,$ed);
            $toggle = 1;
        }
    }
    #find a cycle in the DFS edges sequence
    #curent edge
    my $s_cur;
    my $d_cur;
    #previous edge
    my $s_prev;
    my $d_prev;
    #Set of the elements of the cycle
    my @set_cycle;
    my $set_cycle;
    #Flag that tell that a path was found
    my $flag_cycle = 0;
    #shift the first edge
    my $first_edge = shift(@dfs_edge_seq);
    if ($first_edge =~ /(.*)-(.*)/)
    {
        $s_cur = $1;
        $d_cur = $2;
        push(@set_cycle,$s_cur);
        push(@set_cycle,$d_cur);
    }

    foreach my $cur_edge (@dfs_edge_seq)
    {
        $s_prev = $s_cur;
        $d_prev = $d_cur;
        if ($cur_edge =~ /(.*)-(.*)/)
        {
            $s_cur = $1;
            $d_cur = $2;
        }
        $set_cycle = "@set_cycle";

        #Check if the path continue
        if ($d_prev eq $s_cur)
```

```
            {
                #check if a cycle is found
                #check if the element $d_cur is in the list
                foreach my $vertex (@set_cycle)
                {
                    if ($vertex eq $d_cur)
                    {
                        #need to suppress the edges of the path before the cycle
                        push(@set_cycle,$d_cur);
                        my $shift_val='';
                        while($shift_val ne $d_cur)
                        {
                            $shift_val = shift(@set_cycle);
                        }
                        unshift(@set_cycle,$shift_val);
                        #return the cycle
                        return @set_cycle;
                    }
                }
                push(@set_cycle,$d_cur);

            }
            #check if the cycle is broken
            else
            {
                #check if the source of the curent edge is in the path
                my $s_in_the_path = 0;
                foreach my $vertex (@set_cycle)
                {
                    if($vertex eq $s_cur)
                    {
                        $s_in_the_path = 1;
                        last;
                    }
                }
                #check if the drain of the curent edge is in the path

                if ($s_in_the_path == 1)
                {
                    #remove edges in the path until the value $s_cur
                    #and put both $s_cur and $d_cur
                    my $dump = '';
                    while ($dump ne $s_cur)
                    {
                        $dump = pop(@set_cycle);
                    }
                    push(@set_cycle,$s_cur);

                    #check if there is a cycle
                    foreach my $vertex (@set_cycle)
                    {
                        if($vertex eq $d_cur)
                        {
                            push(@set_cycle,$d_cur);
                            return @set_cycle;
                        }
                    }
                    push(@set_cycle,$d_cur);
                }
                else
                {
                    #elsif the source of the current edge is not in the
                    #path then clear the entire path and push the curent
                    #edge in the cycle.
                    my @temp_array;
                    push(@temp_array,$s_cur);
                    push(@temp_array,$d_cur);
                    @set_cycle = @temp_array;
                }
            }
        }
        return 0;
    }

    ######################################################################
    #Function Name: C1
    #Inputs:
    #(\@{list_of_edges_G})
    #Output:     the color number 1 of the coloring
    #        (the fonction find an entire coloring but return only)
    #Dependencies: functions:
    #            lower_color
    #            alt_path
    #            path
    #            inv_path_color
    #            check_color
    ######################################################################
```

```
sub C1
{
    ################################################################
    # put all the input of the function in arrays
    ################################################################

    my @graph_edge = @{(shift(@_))};
    my @graph_edge_copy = @graph_edge;

    ################################################################
    # Build the graph
    ################################################################

    my $G = Graph::Undirected->new;
    $G->add_edges(@graph_edge);


    ################################################################
    # Set all the edges color to 0
    ################################################################

    for (1 .. (scalar(@graph_edge)/2))
    {
        my $color = 0;
        my $vs = shift(@graph_edge);
        my $vd = shift(@graph_edge);
        push @graph_edge,$vs;
        push @graph_edge,$vd;

        $G->set_attribute('color',$vs,$vd,$color);
        $G->set_attribute('color',$vd,$vs,$color);
    }

    ################################################################
    # Star of the main loop
    ################################################################

    my $alpha;
    my $beta;

    for(1 .. (scalar(@graph_edge)/2))
    {
        my $vs = shift(@graph_edge);
        my $vd = shift(@graph_edge);
        push @graph_edge,$vs;
        push @graph_edge,$vd;

        ################################################################
        # find the lower color missing at $vs
        ################################################################

        $alpha = &lower_color(\$G,$vs);

        ################################################################
        #find the lower color missing at $vd
        ################################################################

        $beta = &lower_color(\$G,$vd);

        ################################################################
        # Check if alpha not equal beta
        ################################################################
        if ($alpha != $beta)
        {
            ################################################################
            #find an alternative path and swap color
            ################################################################
            &alt_path(\$G,$vs,$vd,$alpha,$beta);
        }
        ################################################################
        #color the edge (vs,vd) with beta
        ################################################################
        $G->set_attribute('color',$vs,$vd,$beta);
        $G->set_attribute('color',$vd,$vs,$beta);
    }
    ################################################################
    # return color 1
    ################################################################

    my @color1_edges;

    for(1 .. (scalar(@graph_edge)/2))
    {
        my $vs = shift(@graph_edge);
        my $vd = shift(@graph_edge);
        push @graph_edge,$vs;
        push @graph_edge,$vd;

        if($G->get_attribute('color',$vs,$vd) == 1)
```

```
            {
                push @color1_edges ,$vs ,$vd ;
            }

            #test
            my $test = $G->get_attribute('color',$vs ,$vd );
            #print "color $vs-$vd = $test\n";
        }
        return @color1_edges ;
}

#Return the lower missing color at the vertex
sub lower_color
{
    my $G = shift ;
    my $vertex = shift ;
    my $color = 1;
    my @color_list ;

    my @adj_list = $$G->neighbors($vertex );

    #build a list of all the color
    foreach my $adj_vertex (@adj_list)
    {
        my $temp = $$G->get_attribute('color',$vertex ,$adj_vertex );
        push @color_list , $temp;
    }

    #find the smallest missing
    @color_list = sort(@color_list );
    my $prev_c = shift @color_list ;
    foreach my $c (sort(@color_list))
    {
        if($prev_c == $c)
        {next;}
        elsif($c != $color)
        {last;}
        else
        {$color++;}
        $prev_c = $c;
    }
    return $color ;
}

sub alt_path
{
    my $G = shift ;
    my $vs = shift ;
    my $vd = shift ;
    my $alpha = shift ;
    my $beta = shift ;

    my @alternate_path = &path(\$G,$vs ,$alpha ,$beta );

    if(@alternate_path)
    {
        &inv_path_color(\$G,\@alternate_path ,$alpha ,$beta );
    }
    else
    {
        @alternate_path = &path(\$G,$vd ,$alpha ,$beta );
        if(@alternate_path)
        {
            &inv_path_color(\$G,\@alternate_path ,$alpha ,$beta );
        }
        else
        {return;}
    }
}

sub path
{
    my $G = shift ;
    my $vertex = shift ;
    my $alpha = shift ;
    my $beta = shift ;

    my $temp_color = $beta ;
    my @path_list ;
    my $next_vertex ;

    $next_vertex = &check_color(\$G, $vertex ,$temp_color );
    if($next_vertex eq 0)
    {return @path_list ;}

    while($next_vertex ne 0)
    {
        push (@path_list ,$next_vertex );
        push (@path_list ,$next_vertex );
        if($temp_color == $alpha)
        {$temp_color = $beta;}
        else
        {$temp_color = $alpha;}

        $next_vertex = &check_color(\$G,$next_vertex,$temp_color);
    }
    my $extra = pop @path_list;
    unshift (@path_list,$vertex);
    return @path_list;
}

sub inv_path_color
{
    my $G = shift;
    my @path = @{(shift(@_))};
    my $alpha = shift;
    my $beta = shift;

    #test
    #print "@path\n";

    for(1..(scalar(@path)/2))
    {
        my $start = shift @path;
        my $vertex = shift @path;

        if($$$G->get_attribute('color',$start,$vertex) == $beta)
        {
            $$$G->set_attribute('color',$start,$vertex,$alpha);
            $$$G->set_attribute('color',$vertex,$start,$alpha);
        }
        elsif($$$G->get_attribute('color',$start,$vertex) == $alpha)
        {
            $$$G->set_attribute('color',$start,$vertex,$beta);
            $$$G->set_attribute('color',$vertex,$start,$beta);
        }
    }
}

sub check_color
{
    my $G = shift;
    my $vertex = shift;
    my $color = shift;

    my $edge_color;

    my @vertex_adj = $$$$G->neighbors($vertex);
    foreach my $v_end (@vertex_adj)
    {
        $edge_color = $$$$G->get_attribute('color',$vertex,$v_end);
        if($edge_color == $color)
        {
            return $v_end;
        }
    }
    return 0;
}

##################################################################
#Function Name: match_ok
#Inputs:
#    M a match (a list)
#    CLB_CONF (a hash table)
#    OUT_adj_list (a hash table)
#Output:
#    1 if it is OK
#    0 if it is not OK
##################################################################
sub match_ok
{
    ##############################################################
    #    Get the input parameters
    ##############################################################
    #the match
    my $ref_M = shift;
    #the CLB_CONF
    my $ref_CLB_CONF = shift;
    #the OUT_adj_list 0 and 1
    my $ref_adj_list_0 = shift;
    my $ref_adj_list_1 = shift;

#    test print
#    foreach (@{$ref_M})
#    {
#    print "$_\n";
#    }
#   foreach my $hash_key (keys%{$ref_CLB_CONF})
#   {
#   print "$hash_key: @{$$ref_CLB_CONF{$hash_key}}\n";
#   }
#   foreach my $hash_key (sort keys%{$ref_adj_list_0})
#   {
#   print "$hash_key: @{$$ref_adj_list_0{$hash_key}}\n";
#   }
#   foreach my $hash_key (sort keys%{$ref_adj_list_1})
#   {
#   print "$hash_key: @{$$ref_adj_list_1{$hash_key}}\n";
#   }

    #################################################################
    # Split the M in M0 and M1
    #################################################################
    my @M0;
    my @M1;
    my @M_t = @{$ref_M};
    for(1 .. (scalar(@M_t)/2))
    {
        my $u = shift @M_t;
        my $v = shift @M_t;

        if ($u =~ /S(\d)_(.*)/)
        {
            if ($1 == 0)
            {
                push @M0,$2;
                push @M0,$v;
            }
            else
            {
                push @M1,$2;
            push @M1,$v;
            }
        }
    }
    #################################################################
    # Start the checking loop
    #################################################################


    my $find_one_CLB0 = 0;
    my $find_one_CLB1 = 0;
    my $first_T_IN = 3;

    foreach my $CLB_conf_name (sort keys%{$ref_CLB_CONF})
    {
        #print "----------CLB0-------------\n";
        #print "\nChange in the CLB conf\n";
        #print "@M0\n\n";


        #Start the check for connections related to CLB0
        my $check_CLB_0 = 0; #false first
        foreach my $conf_conn (@{$$ref_CLB_CONF{$CLB_conf_name}})
        {

            #print "configuration $CLB_conf_name: $conf_conn\n";

            #find the connection in M0 that got the same source as
            #$conf_conn.
            #print "conf_conn_before : $conf_conn\n";

            my $conf_conn_source;
            my $conf_conn_drain;
            if($conf_conn =~ /(.*)->(.*)/)
            {
                $conf_conn_source = $1;
                $conf_conn_drain = $2;
            }
            my $out_line = 0;
            #print "M0: @M0\n";
            for (1 .. (scalar(@M0)/2))
            {
                my $source = shift(@M0);
                my $drain = shift(@M0);

                if($conf_conn_source eq $source)
                {$out_line = $drain;}

                push (@M0,$source);
                push (@M0,$drain);
            }
            #print "OUT_line: $out_line\n";
            #if that connection does not exist in M0 then go to
            # the next CLB connection (in the same configuration)
            # Because sometime the configurations are not complete
            if ($out_line eq 0)
            #{last;} #next CLB configuration
            {next;} #same CLB configuration

            #print "conf: @{$$ref_CLB_CONF{$CLB_conf_name}}\n";
            #print "We looking for a connection at : $conf_conn_drain at $out_line\n";
            #print "conf_conn : $conf_conn\n";
            #check in the OUT line adj list if the input can be reach
            #if the loop did not find any path the stop the function
            #and return false.
            my $local_check = 0;
            foreach my $adj_out_line (@{$$ref_adj_list_0{$out_line}})
            {
                #test
                #print "$adj_out_line  $conf_conn_drain\n";

                #print "adj_out_line $out_line : $adj_out_line\n";
                if ($adj_out_line eq $conf_conn_drain)
                {
                    #print "First loop drain: $conf_conn_drain\n";
                    $local_check = 1;
                    last;
                }
                elsif($adj_out_line =~ /(T)(\d)/)
                {
                    #print "Enter in the loop with $1$2 and conf_drain: $conf_conn_drain\n";
                    if($conf_conn_drain eq 'T')
                    {
                        $local_check = 1;
                        $first_T_IN = $2;
                        #print "F_T_IN : $2\n";
                        last;
                    }
                }
            }
            #if there is no connection possible for that connection
            #then go out of the loop and change the CLB conf
            if ($local_check == 0)
            {
                $check_CLB_0 = 0;
                last;
            }
            #print "conf OK\n";
            $check_CLB_0 = 1;
        }
        #next CBL conf if it is not good
        if ($check_CLB_0 == 0)
        {
            next;
        }
        else
        {
            #got out of the loop if one CLB configuration works with the match
            $find_one_CLB0 = 1;
            last;
        }
    }
    #print "find CLB0 conf = $find_one_CLB0\n";
    if($find_one_CLB0 == 0)
    {
        return 0;
    }

    foreach my $CLB_conf_name (sort keys%{$ref_CLB_CONF})
    {
        #print "——————CLB1————————\n";
        #Start the check for connections related to CLB1
        my $check_CLB_1 = 0;
        foreach my $conf_conn (@{$$ref_CLB_CONF{$CLB_conf_name}})
        {
            #print "configuration $conf_conn\n";
            #find the connection in M0 that got the same source as
            #$conf_conn.
            my $conf_conn_source;
            my $conf_conn_drain;
            if($conf_conn =~ /(.*)->(.*)/)
            {
                $conf_conn_source = $1;
                $conf_conn_drain = $2;
            }
            #print "M1: @M1\n";
            my $out_line = 0;
            for (1 .. (scalar(@M1)/2))
            {
                my $source = shift(@M1);
                my $drain = shift(@M1);

                if($conf_conn_source eq $source)
                {$out_line = $drain;}

                push (@M1,$source);
                push (@M1,$drain);
            }
            #print "OUT_line: $out_line\n";
            if ($out_line eq 0)
            {next;}

            #check in the OUT line adj list if the input can be reach
            #if the loop did not find any path the stop the function
            #and return false.
            my $local_check = 0;
            foreach my $adj_out_line (@{$$ref_adj_list_1{$out_line}})
            {
                #test
                #print "$adj_out_line   $conf_conn_drain\n";

                if ($adj_out_line eq $conf_conn_drain)
                {
                    $local_check = 1;
                    last;
                }
                elsif($adj_out_line =~ /T(\d)/)
                {
                    if($conf_conn_drain eq 'T')
                    {
                        #print "T$1 first_T_IN:$first_T_IN\n";
                        if($1 ne $first_T_IN)
                        {
                            $local_check = 1;
                            last;
                        }
                    }
                }
            }
            if ($local_check == 0)
            {
                $check_CLB_1 = 0;
                last;
            }
            $check_CLB_1 = 1;
        }
        if ($check_CLB_1 == 0)
        {
            next;
        }
        else
        {
            $find_one_CLB1 = 1;
            last;
        }
    }
    #print "find CLB1 conf = $find_one_CLB1\n";
    if($find_one_CLB1 == 0)
    {
        return 0;
    }
    return 1;
}
############################################################################
#Function Name: read_file
#Inputs:
#   name of the file you want to open
#Output:
#   the array that contain all the line of the file
############################################################################
sub read_file
{
    my ($filename) = @_;
    open(FILE,"$filename")||die("could_not_open_the_file");
    my @file = <FILE>;
    close(FILE);
    chomp(@file);
    return @file;
}
```

B.2 Completion of G_{ORM} Edge Coloring

```
#!/usr/bin/env perl

############################################################################
# File name: complete_ORM_coloring.pl
# Author: Christian Giasson
# Date : ???
# Comments: This script complete ORM coloring
############################################################################
```

```
use warnings;
use strict;
use Getopt::Std;

##########################################################################
#Get the inputs of the program
##########################################################################

getopts('ha:b:c:');
our($opt_h,$opt_a,$opt_b,$opt_c);

if ($opt_h)
{
    print"\n\tOption␣of␣this␣program:\n
␣␣␣␣␣␣␣␣\t-h:␣\t\t\thelp
␣␣␣␣␣␣␣␣\t-a␣[file]:\t\tcoloring␣of␣the␣ORMG
␣␣␣␣␣␣␣␣\t-b␣[file]:\t\tlist␣of␣the␣CLB␣configuration
␣␣␣␣␣␣␣␣\t-c␣[file]:\t\tadj_list␣of␣the␣OUT␣lines\n\n";
        exit;
}

my @lines = &read_file($opt_a) if (($opt_a)&&(!$opt_h));
my @clb_conf_2 = &read_file($opt_b) if (($opt_b)&&(!$opt_h));
my @out_adj_list = &read_file($opt_c) if (($opt_c)&&(!$opt_h));

#get the color 1 to 8 without the "ADD" lable
#replace the YA by Y
my @color_1_8;
foreach my $line (@lines)
{
    if($line =~ /:(.*)/)
    {
        my $tmp = $1;
        $tmp =~ s/S0_YA/S0_Y/g;
        $tmp =~ s/S1_YA/S1_Y/g;
        push @color_1_8,$tmp;
        #print "$tmp\n";
    }
}

#my @out_adj_list = &read_file('OUT_adj_list');
#my @clb_conf_2 = &read_file('CLB_CONF_2');
##########################################################################
#build the data structure of the CLB_conf and the OUT_adj_list
##########################################################################

#data structure for CLB_conf is a hash of arrays
my %CLB_conf;
foreach my $conf (@clb_conf_2)
{
    #if( $conf =~ /(.*): (.*) (.*) (.*) (.*)/)
    if( $conf =~ /(.*): (.*)/)
    {
        my $conf_name = $1;
        my $fb = $2;
        @{$CLB_conf{$conf_name}} = split(/ /,$fb);
    }
}

#data structure for OUT_adj is a hash of arrays
my %OUT_adj_list;
my %OUT_adj_list_0;
my %OUT_adj_list_1;
foreach my $OUT_line (@out_adj_list)
{
    if( $OUT_line =~ /(.*): (.*)/)
    {
        my $OUT = $1;
        @{$OUT_adj_list{$1}} = split(/ /,$2);
        #print "@{$OUT_adj_list{$1}}\n";
        ################################################################
        #   Split the OUT lines adj list into two list of connections
        #   OUT_adj_list_0 : edges related to CLB0
        #   OUT_adj_list_1 : edges related to CLB1
        #   note1: The CLB inputs are in the same format than the CLB
        #          configuration.
        #   note2: The T in OUT_adj_list_0 is T0 and
        #          The T in OUT_adj_list_1 is T1.
        ################################################################

        foreach (@{$OUT_adj_list{$OUT}})
        {
            if ($_ =~ /S(\d)_(.*)_B(.*)/)
            {
                my $num = $3;
                if ($1 == 0)
                {
                    if ($num eq '')
                    {push (@{$OUT_adj_list_0{$OUT}},"$2");}
                    else
                    {push (@{$OUT_adj_list_0{$OUT}},"$2$num");}
                }
                else
                {
                    if ($3 eq '')
                    {push (@{$OUT_adj_list_1{$OUT}},"$2");}
                    else
                    {push (@{$OUT_adj_list_1{$OUT}},"$2$3");}
                }
            }
            elsif ($_ =~ /T_IN(\d)/)
            {
                if ($1 == 0)
                {
                    push (@{$OUT_adj_list_0{$OUT}},"T0");
                    push (@{$OUT_adj_list_1{$OUT}},"T0");
                }
                else
                {
                    push (@{$OUT_adj_list_1{$OUT}},"T1");
                    push (@{$OUT_adj_list_0{$OUT}},"T1");
                }
            }
        }
        #print "$OUT CLB0: @{$OUT_adj_list_0{$OUT}}\n";
        #print "$OUT CLB1: @{$OUT_adj_list_1{$OUT}}\n";
    }
}

#A hash of arrays that include all the coloring
my %all_coloring;
#The completion of the matches.
my @clb_list = qw(S0_XB S0_XQ S0_Y S0_YB S1_XB S1_XQ S1_Y S1_YB);
my @out_list = qw(OUT0 OUT1 OUT2 OUT3 OUT4 OUT5 OUT6 OUT7);
my @clb_out_missing;
my @out_missing;
my $color_count = 0;
foreach my $color (@color_1_8)
{
    $color_count++;
    #check if there is some connection missing
    my @count = split (/ /,$color);
    my $count = @count;
    if ($count == 16)
    {
        #print "@count\n";
        @{$all_coloring{"$color_count"}} = @count;
        next;
    }

    #checking the missing CLB output
    @clb_out_missing = @clb_list;
    for (1 .. scalar(@clb_out_missing))
    {
        my $match = 0;
        my $pin = pop @clb_out_missing;
        foreach my $color_vertex (@count)
        {
            #print "$pin -- $color_vertex\n";
            if ($pin eq $color_vertex)
            {
                $match = 1;
                #print "match\n";
                last;
            }
        }
        if ($match == 0)
        {
            #print "no match\n";
            unshift @clb_out_missing, $pin;
        }
    }

    #checking the missing OUT lines
    @out_missing = @out_list;
    for (1 .. scalar(@out_missing))
    {
        my $match = 0;
        my $pin = pop @out_missing;
        foreach my $color_vertex (@count)
        {
            #print "$pin -- $color_vertex\n";
            if ($pin eq $color_vertex)
            {
                $match = 1;
                #print "match\n";
                last;
                }
            }
            if ($match == 0)
            {
                #print "no match\n";
                unshift @out_missing, $pin;
            }
        }

        if(scalar(@out_missing) == 1)
        {
            push @count, pop @clb_out_missing;
            push @count, pop @out_missing;
            @{$all_coloring{"$color_count"}} = @count;
            #print "@count\n";
            next;
        }

        #print "COLOR$color_count : @clb_out_missing ---- @out_missing\n";
        #print "new color : @count\n";
        push @count, pop @clb_out_missing;
        push @count, pop @out_missing;
        push @count, pop @clb_out_missing;
        push @count, pop @out_missing;
        push @count, pop @clb_out_missing;
        push @count, pop @out_missing;
        #my $color_ok = &match_ok(\@count,\%CLB_conf,\%OUT_adj_list_0,\%OUT_adj_list_1);
        #print "color_ok = $color_ok\n";

        @{$all_coloring{"$color_count"}} = @count;
        #print "@count\n";
}


#duplicate the color 1 to 8 to get color 9 to 16
# with subtitutions.
my @color_9_16;
foreach my $color (sort (keys(%all_coloring)))
{
    my $line = "@{$all_coloring{$color}}";
    $color_count++;
    $line =~ s/S0_Y /S0_X /g;
    $line =~ s/S0_XQ/S0_YQ/g;
    $line =~ s/S1_Y /S1_X /g;
    $line =~ s/S1_XQ/S1_YQ/g;
    @{$all_coloring{"$color_count"}} = split (/ /,$line);
    #push @color_9_16, $line;
}

foreach my $color (sort {$a <=> $b} keys(%all_coloring))
{
    my @color_copy = @{$all_coloring{$color}};
    #print "@color_copy\n";
    for (1 .. scalar(@{$all_coloring{$color}})/2)
    {
        my $s = shift @color_copy;
        my $d = shift @color_copy;
        print "COLOR$color:$s_$d\n";
    }
    #print "$color:@{$all_coloring{$color}}\n";
}

foreach my $color (@color_1_8)
{
    # print "$color\n";
}
foreach my $color (@color_9_16)
{
    # print "$color\n";
}
##################################################################
#Function Name: match_ok
#Inputs:
#    M a match (a list)
#    CLB_CONF (a hash table)
#    OUT_adj_list (a hash table)
#Output:
#    1 if it is OK
#    0 if it is not OK
##################################################################
sub match_ok
{
    ##############################################################
    #    Get the input parameters
    ##############################################################
    #the match
    my $ref_M = shift;
    #the CLB_CONF
    my $ref_CLB_CONF = shift;
    #the OUT_adj_list 0 and 1
    my $ref_adj_list_0 = shift;
    my $ref_adj_list_1 = shift;

#   test print
#   foreach (@{$ref_M})
#   {
#   print "$_\n";
#   }
#   foreach my $hash_key (keys%{$ref_CLB_CONF})
#   {
#   print "$hash_key: @{$$ref_CLB_CONF{$hash_key}}\n";
#   }
#   foreach my $hash_key (sort keys%{$ref_adj_list_0})
#   {
#   print "$hash_key: @{$$ref_adj_list_0{$hash_key}}\n";
#   }
#   foreach my $hash_key (sort keys%{$ref_adj_list_1})
#   {
#   print "$hash_key: @{$$ref_adj_list_1{$hash_key}}\n";
#   }
    #####################################################################
    # Split the M in M0 and M1
    #####################################################################
    my @M0;
    my @M1;
    my @M_t = @{$ref_M};
    for(1 .. (scalar(@M_t)/2))
    {
        my $u = shift @M_t;
        my $v = shift @M_t;

        if ($u =~ /S(\d)_(.*)/)
        {
            if ($1 == 0)
            {
                push @M0, $2;
                push @M0, $v;
            }
            else
            {
                push @M1, $2;
                push @M1, $v;
            }
        }
    }

    #####################################################################
    # Start the checking loop
    #####################################################################

    my $find_one_CLB0 = 0;
    my $find_one_CLB1 = 0;
    my $first_T_IN;

    foreach my $CLB_conf_name (sort keys%{$ref_CLB_CONF})
    {
        #print "----------CLB0---------------\n";

        #Start the check for connections related to CLB0
        my $check_CLB_0 = 0; #false first
        foreach my $conf_conn (@{$$ref_CLB_CONF{$CLB_conf_name}})
        {
            #print "configuration $conf_conn\n";

            #find the connection in M0 that got the same source as
            #$conf_conn.
            my $conf_conn_source;
            my $conf_conn_drain;
            if($conf_conn =~ /(.*)->(.*)/)
            {
                $conf_conn_source = $1;
                $conf_conn_drain = $2;
            }
            my $out_line = 0;
            #print "M0: @M0\n";
            for (1 .. (scalar(@M0)/2))
            {
                my $source = shift(@M0);
                my $drain = shift(@M0);

                if($conf_conn_source eq $source)
                {$out_line = $drain;}

                push (@M0, $source);
                push (@M0, $drain);
            }
            #print "OUT_line: $out_line\n";
            if ($out_line eq 0)
            {next;}
            #check in the OUT line adj list if the input can be reach
            #if the loop did not find any path then stop the function
            #and return false.


            my $local_check = 0;
            foreach my $adj_out_line (@{$$ref_adj_list_0{$out_line}})
            {
                #test
                #print "$adj_out_line  $conf_conn_drain\n";

                if ($adj_out_line eq $conf_conn_drain)
                {
                    $local_check = 1;
                    last;
                }
                elsif($adj_out_line =~ /T(\d)/)
                {
                    if($conf_conn_drain eq 'T')
                    {
                        $local_check = 1;
                        $first_T_IN = $1;
                        last;
                    }
                }
            }
            if ($local_check == 0)
            {
                $check_CLB_0 = 0;
                last;
            }
            $check_CLB_0 = 1;
        }
        if ($check_CLB_0 == 0)
        {
            #return 0;
            next;
        }
        else
        {
            $find_one_CLB0 = 1;
            #print "$CLB_conf_name(0) ";
            #last;
        }
    }

    if($find_one_CLB0 == 0)
    {
        return 0;
    }

    foreach my $CLB_conf_name (sort keys%{$ref_CLB_CONF})
    {
        #print "----------CLB1--------------\n";
        #Start the check for connections related to CLB1
        my $check_CLB_1 = 0;
        foreach my $conf_conn (@{$$ref_CLB_CONF{$CLB_conf_name}})
        {
            #print "configuration $conf_conn\n";
            #find the connection in M0 that got the same source as
            #$conf_conn.
            my $conf_conn_source;
            my $conf_conn_drain;
            if($conf_conn =~ /(.*)->(.*)/)
            {
                $conf_conn_source = $1;
                $conf_conn_drain = $2;
            }
            #print "M1: @M1\n";
            my $out_line = 0;
            for (1 .. (scalar(@M1)/2))
            {
                my $source = shift(@M1);
                my $drain = shift(@M1);

                if($conf_conn_source eq $source)
                {$out_line = $drain;}

                push (@M1,$source);
                push (@M1,$drain);
            }
            #print "OUT_line: $out_line\n";
            if ($out_line eq 0)
            {next;}

            #check in the OUT line adj list if the input can be reach
            #if the loop did not find any path the stop the function
            #and return false.
            my $local_check = 0;
            foreach my $adj_out_line (@{$$ref_adj_list_1{$out_line}})
            {
                #test
                #print "$adj_out_line  $conf_conn_drain\n";

                if ($adj_out_line eq $conf_conn_drain)
                {
                    $local_check = 1;
                    last;
                }
                elsif($adj_out_line =~ /T(\d)/)
                {
                    if($conf_conn_drain eq 'T')
                    {
                        if($1 ne $first_T_IN)
                        {
                            $local_check = 1;
                            last;
                        }
                    }
                }
            }
            if ($local_check == 0)
            {
                $check_CLB_1 = 0;
                last;
            }
            $check_CLB_1 = 1;
        }
        if ($check_CLB_1 == 0)
        {
            #return 0;
            next;
        }
        else
        {
            $find_one_CLB1 = 1;
            #print "$CLB_conf_name(1) ";
            #last;
        }
    }
    if($find_one_CLB1 ==0)
    {
        return 0;
    }
    return 1;
}
##########################################################################
#Function Name: read_file
#Inputs:
#   name of the file you want to open
#Output:
#   the array that contain all the line of the file
##########################################################################
sub read_file
{
    my ($filename) = @_;
    open(FILE,"$filename")||die("could_not_open_the_file");
    my @file = <FILE>;
    close(FILE);
    chomp(@file);
    return @file;
}
```

B.3 Feedback Routing

```
#!/usr/local/bin/perl
##########################################################################
=head1 NAME

color_path_1_16.pl

=head1 AUTHOR (DATE)

Christian Giasson (July 23rd, 2003)

=head1 SYNOPSIS

color_path_1_16.pl -a {16 colors of the ORMG}
             -b {CLB configuration 2 and 4}
             -c {Configuration for each colors}
             -d {List of single paths}
             -e {List of double paths}
```

```
            flags
            -h  help

=head1 DESCRIPTION

This program generate routable paths in the Virtex FPGA from each of the
colors (test configurations) generated by the program enum_edge_coloring.pl.

=over 4

=item INPUT

There is a list of all the file needed by the this program:

The coloring of the ORMG. That file includes complete (8 paths by color)
colors for the first 16 colors.

CLB configuration list. conf2 and conf4 are needed in the same file.

The list of all the CLB_configuration of each COLOR and both CLB
configurations generated by the program get_CLB_conf.pl.

A list of all the possible path for a routing between the the OUT lines and
the CLB inputs( actually two files single and double path).

=item OUTPUT

That program will generate a detail list of the routing for each color.  Also
the CLB0 and CLB1 confiration will be print for each color (or testing
configuration).

=item DEPENDENCIES

No dependencies on other programs or none built in functions

=cut
##########################################################################
use warnings;
use strict;
use Getopt::Std;


##########################################################################
#Get the inputs of the program
##########################################################################

getopts('ha:b:c:d:e:');
our($opt_h,$opt_a,$opt_b,$opt_c,$opt_d,$opt_e);

if ($opt_h)
{
    print"\n\tOption_of_this_program:\n
________\t-h:_\t\thelp
________\t-a_[file]:_\t16_colors_of_the_ORMG\t\t
________\t-b_[file]:_\tCLB_configuration_2_and_4\t\t
________\t-c_[file]:_\tCLB_configurations_for_each_colors\t\t
________\t-d_[file]:_\tList_of_single_paths\t\t
________\t-e_[file]:_\tList_of_double_paths\t\t\n\n";
        exit;
}

#push every lines of each file inside an array
my @ORM_color_l = &read_file($opt_a) if (($opt_a)&&(!$opt_h));
my @CLB_conf_2_4_l = &read_file($opt_b) if (($opt_b)&&(!$opt_h));
my @CLB_conf_color_l = &read_file($opt_c) if (($opt_c)&&(!$opt_h));
my @path_single_list = &read_file($opt_d) if (($opt_d)&&(!$opt_h));
my @path_double_list = &read_file($opt_e) if (($opt_e)&&(!$opt_h));


##########################################################################
#   New
#   Suppress all the orthogonal double paths
##########################################################################

for (1 .. scalar(@path_double_list))
{
    my $path = shift @path_double_list;
    if ($path =~ /^\w+->([NSEW])(\d+)->([NSEW])(\d+)/)
    {
        if ($2 eq $4)
        {
            next if (($1 eq 'N')&&($3 eq 'S'));
            next if (($1 eq 'S')&&($3 eq 'N'));
            next if (($1 eq 'E')&&($3 eq 'W'));
            next if (($1 eq 'W')&&($3 eq 'E'));

#           push @path_double_list, $path if (($1 eq 'N')&&($3 eq 'S'));
#           push @path_double_list, $path if (($1 eq 'S')&&($3 eq 'N'));
#           push @path_double_list, $path if (($1 eq 'E')&&($3 eq 'W'));
#     push @path_double_list , $path if (($1 eq 'W')&&($3 eq 'E'));
        }
    }
    push @path_double_list , $path;
}

##########################################################################
#   New test
#   Check How many connections can connect to the CLB inputs
##########################################################################
#my %occurence;
#foreach (@path_double_list)
#{
#     if (/^(\w+)->.*->(\w+)$/)
#     {
#    if(exists $occurence{"$1$2"})
#    {
#        $occurence{"$1$2"}++;
#    }
#    else
#    {
#        $occurence{"$1$2"} = 1;
#    }
#    }
#}
#print
#foreach (sort keys(%occurence))
#{
#     print "$_ : $occurence{$_}\n";
#}

#exit;

#data structure variables
my %ORM_color;
my %CLB_conf_2_4;
my %CLB_conf_color;
my %path_single;
my %path_double;
#build the data structure
&data_struct(   \%ORM_color,     \@ORM_color_l,
         \%CLB_conf_2_4 , \@CLB_conf_2_4_l ,
         \%CLB_conf_color ,\ @CLB_conf_color_l ,
         \%path_single , \ @path_single_list ,
         \%path_double , \ @path_double_list );


##########################################################################
##########################################################################
##          MAIN part of the program                     ##
##########################################################################
##########################################################################

##########################################################################
#   New
#   Create an array of all the short lines to check if the othogonal short
#   line are not taken in the same time.
##########################################################################
my @short_list;
for (0..23)
{
    push @short_list , ("N$_","S$_","E$_","W$_");
}


##########################################################################
#Check if each color can be match with a CLB configuration with
#SINGLE PATHs
##########################################################################

#new
my @color_sequence = (  'COLOR1', 'COLOR2', 'COLOR3', 'COLOR4',
              'COLOR5', 'COLOR6', 'COLOR7', 'COLOR8',
              'COLOR9', 'COLOR10', 'COLOR11', 'COLOR12',
              'COLOR13', 'COLOR14', 'COLOR15', 'COLOR16');

#COLOR_PATH is a hash of a hash of arrays {color}{path num} = array of
#connections that make the path.
my %COLOR_PATH;
#for all the COLOR
#foreach my $color (sort keys(%ORM_color))
foreach my $color (@color_sequence)
{
    my $found_CLB_match = 0;
    #for all the CLB0 configurations
    foreach my $CLB0_conf(sort keys(%{$CLB_conf_color{$color}{'CLB0'}}))
    {
        #for all the CLB1 configurations
        foreach my $CLB1_conf(sort keys(%{$CLB_conf_color{$color}{'CLB1'}}))
```

```
{
    #One pass for the (CLB0->T0 CLB1->T1)
    #and another one with (CLB0->T1 CLB1->T0)
    my $CLB0_tri = 'T_IN0';
    my $CLB1_tri = 'T_IN1';
    foreach my $order ('sta','rev')
    {
        if($order eq 'rev')
        {
            $CLB0_tri = 'T_IN1';
            $CLB1_tri = 'T_IN0';
        }

        #for all the conections in the color
        #try to connect to the input of the conf
        $found_CLB_match = 1;
        foreach my $CLB_out (sort keys(%{$ORM_color{$color}}))
        {
            #OUT line
            my $OUT_line = $ORM_color{$color}{$CLB_out};

            #CLB conf_0 and conf1 array
            my @conf0 = @{$CLB_conf_2_4{$CLB0_conf}};
            my @conf1 = @{$CLB_conf_2_4{$CLB1_conf}};

            my $CLB;
            my $short_name_CLB_out;
            if ($CLB_out =~ /S([01])_(\w+)/)
            {
                $CLB = $1;
                $short_name_CLB_out = $2;
            }

            #find the $CLB_in
            my $short_name_CLB_in;
            if ($CLB == 0)
            {
                foreach (@conf0)
                {
                    if ($_ =~ /$short_name_CLB_out->(\w+)/)
                    {
                        $short_name_CLB_in = $1;
                        last;
                    }
                }
            }
            else
            {
                foreach (@conf1)
                {
                    if ($_ =~ /$short_name_CLB_out->(\w+)/)
                    {
                        $short_name_CLB_in = $1;
                        last;
                    }
                }
            }
            #Put the CLB input name in long format
            my $CLB_in;
            if ($short_name_CLB_in eq 'T')
            {
                if ($CLB == 0)
                {
                    $CLB_in = "$CLB0_tri";
                }
                else
                {
                    $CLB_in = "$CLB1_tri";
                }
            }
            elsif ($short_name_CLB_in =~ /([GF])(\d)/)
            {
                $CLB_in = "S$CLB"."_$1"."_B$2";
            }
            elsif ($short_name_CLB_in =~ /(B[XY])/)
            {
                $CLB_in = "S$CLB"."_$1"."_B";
            }

            ##################################################
            #    Find a path
            ##################################################
            #find a path = 1, no path = 0
            my $find_a_path;
            my @connection_of_path;
            #call the function to find a path
            $find_a_path = &find_path(\@connection_of_path,
                                      \%path_single,
                                        $OUT_line,
                                        $CLB_in);
                #look for path in the single feedback
                if ($find_a_path == 1)
                {
                    #save the path in the COLOR_PATH hash
                    unshift @connection_of_path,$CLB_out;
                    push @{$COLOR_PATH{$color}{"S:_$CLB_out->"."$CLB_in"}} ,
                        @connection_of_path;
                }
                #look for a path in the double feedback
                elsif($find_a_path == 0)
                {
                    my @connection_of_path_2;
                    #check for the double path
                    $find_a_path = &find_path_2(\@connection_of_path_2,
                                               \%path_double,
                                               $OUT_line,
                                               $CLB_in);
                    if ($find_a_path == 1)
                    {
                        unshift @connection_of_path_2,$CLB_out;
                        push @{$COLOR_PATH{$color}{"D:_$CLB_out->"."$CLB_in"}},
                            @connection_of_path_2;
                    }
                }

                ##################################################
                #   if there is no path for one color connection
                ##################################################
                if($find_a_path == 0)
                {
                    $found_CLB_match = 0;
                    #delete every paths inside the COLOR_PATH hash
                    foreach my $path_name (sort keys(%{$COLOR_PATH{$color}}))
                    {
                        #put back all the path in the path_single hash
                        my $a = shift @{$COLOR_PATH{$color}{$path_name}};
                        my $b = shift @{$COLOR_PATH{$color}{$path_name}};
                        my $c = shift @{$COLOR_PATH{$color}{$path_name}};
                        my $d = shift @{$COLOR_PATH{$color}{$path_name}};

                        if ($path_name =~ /^S/)
                        {
                            $path_single{$a}{$b}{$c}{$d} = 1;
                        }
                        elsif($path_name =~ /^D/)
                        {
                            my $e = shift @{$COLOR_PATH{$color}{$path_name}};
                            $path_double{$a}{$b}{$c}{$d}{$e} = 1;
                        }

                        delete $COLOR_PATH{$color}{$path_name};
                    }
                    last;
                }
            }
            #check if the color is good
            if ($found_CLB_match == 1)
            {
                my $good_color = &check_color(\%{$COLOR_PATH{$color}});
                #print "good = $good_color\n";
                #last if (($found_CLB_match == 1)&&($good_color == 0));
                last if ($good_color == 1);
                $found_CLB_match = 0;
                #have to put back all the path in path_singe and
                #path_double
            }
        }
        last if $found_CLB_match == 1;
    }
    last if $found_CLB_match == 1;
}

foreach (sort keys(%{$COLOR_PATH{$color}}))
{
    print "$color:_@{$COLOR_PATH{$color}{$_}}\n";

    #delete all the path related to the path in IRM
    &delete_path(\@{$COLOR_PATH{$color}{$_}}, \%path_single, \%path_double);
}
}

######################################################################
#Function Name: find_path
#Inputs:
#Output:
######################################################################
```

```
sub find_path
{
    #input variables
    my $connection_of_path_ref = shift @_;
    my $path_single_ref = shift @_;
    my $OUT_line_var = shift @_;
    my $CLB_in_var = shift @_;

    #flag variable
    my $got_a_path = 0;

    foreach my $short_line (sort keys(%{$$path_single_ref{$OUT_line_var}}))
    {
        foreach my $node_line (sort keys(%{$$path_single_ref{$OUT_line_var}{$short_line}}))
        {
            foreach my $CLB_in_path (sort keys(%{$$path_single_ref{$OUT_line_var}
                {$short_line}{$node_line}}))
            {
                if ($CLB_in_path eq $CLB_in_var)
                {
                    $got_a_path = 1;
                    push @{$connection_of_path_ref},($OUT_line_var,
                                                     $short_line,
                                                     $node_line,
                                                     $CLB_in_var);

                    #new
                    #delete the path from the hash if it is a CLB input
                    #if ($CLB_in_path =~ /S[01]_(.*)_B/)
                    if ($CLB_in_path =~ /S[01]_[FG]_B/)
                    {
                        #delete $$path_single_ref{$OUT_line_var}{$short_line}
                        #{$node_line}{$CLB_in_path};
                    }
                    last;
                }
            }
            last if $got_a_path == 1;
        }
        last if $got_a_path == 1;
    }

    if ($got_a_path == 0)
    {return 0;}
    else
    {return 1;}
}

######################################################################
#Function Name: find_path_2
#Inputs:
#Output:
######################################################################
sub find_path_2
{
    #input variables
    my $connection_of_path_ref = shift @_;
    #my $deleted_path_ref = shift @_;
    my $path_double_ref = shift @_;
    my $OUT_line_var = shift @_;
    my $CLB_in_var = shift @_;

    #flag variable
    my $got_a_path = 0;

    foreach my $short_line1 (sort keys(%{$$path_double_ref{$OUT_line_var}}))
    {
        foreach my $short_line2 (sort keys(%{$$path_double_ref{$OUT_line_var}
        {$short_line1}}))
        {
            foreach my $node_line (sort keys(%{$$path_double_ref{$OUT_line_var}
            {$short_line1}{$short_line2}}))
            {
                foreach my $CLB_in_path (sort keys(%{$$path_double_ref{$OUT_line_var}
                {$short_line1}{$short_line2}{$node_line}}))
                {
                    if ($CLB_in_path eq $CLB_in_var)
                    {
                        $got_a_path = 1;
                        push @{$connection_of_path_ref},($OUT_line_var,
                                                         $short_line1,
                                                         $short_line2,
                                                         $node_line,
                                                         $CLB_in_var);

                        #new
                        #delete the path from the hash if it is a CLB input
                        #if ($CLB_in_path =~ /S[01]_(.*)_B/)
                        if ($CLB_in_path =~ /S[01]_[FG]_B/)
                        {
                            #delete $$path_double_ref{$OUT_line_var}{$short_line1}
                            #{$short_line2}{$node_line}{$CLB_in_path};
                        }

                        last;
                    }
                }
                last if $got_a_path == 1;
            }
            last if $got_a_path == 1;
        }
        last if $got_a_path == 1;
    }

    if ($got_a_path == 0)
    {return 0;}
    else
    #delete the path from the hash
    {return 1;}
}
##########################################################################
#Function Name: check_color
#Inputs:
#Output:
##########################################################################
sub check_color
{
    my $color_path_ref = shift;
    my %color_path_local = %{$color_path_ref};

    ####################################################
    #   verify if the same short line appear two times
    ####################################################
    foreach my $path_name (keys(%color_path_local))
    {
        my $array_print = "@{$color_path_local{$path_name}}";
        my @short;
        if ($array_print =~ /OUT\d ([NSEW]\d+) [NSEW]_/)
        {
            push @short,$1;
        }
        elsif ($array_print =~ /OUT\d ([NSEW]\d+) ([NSEW]\d+) [NSEW]_/)
        {
            push @short, ($1,$2);
        }
        #find if there is another same sort line
        foreach my $short_conn (@short)
        {
            foreach my $path_name2 (keys(%color_path_local))
            {
                next if ($path_name2 eq $path_name);
                for (@{$color_path_local{$path_name2}})
                {
                    return 0 if ($_ eq $short_conn);
                }
            }
        }
    }

    ####################################################
    #   verify if no orthogonal short lines are used
    ####################################################
    foreach my $path_name (keys(%color_path_local))
    {
        my $array_print = "@{$color_path_local{$path_name}}";
        my @short;
        if ($array_print =~ /OUT\d ([NSEW]\d+) [NSEW]_/)
        {
            push @short,$1;
        }
        elsif ($array_print =~ /OUT\d ([NSEW]\d+) ([NSEW]\d+) [NSEW]_/)
        {
            push @short, ($1,$2);
        }
        #find if the opposite line is used in the
        foreach my $short_conn (@short)
        {
            #find the opposite
            if($short_conn =~ /N(\d+)/)
            {
                $short_conn = "S$1"
            }
            elsif($short_conn =~ /S(\d+)/)
            {
                $short_conn = "N$1";
            }
            elsif($short_conn =~ /E(\d+)/)
            {
                $short_conn = "W$1";
            }
            elsif($short_conn =~ /W(\d+)/)
            {
                $short_conn = "E$1";
            }

            foreach my $path_name2 (keys(%color_path_local))
            {
                next if ($path_name2 eq $path_name);
                for (@{$color_path_local{$path_name2}})
                {
                    return 0 if ($_ eq $short_conn);
                }
            }
        }
    }
    return 1;
}
###########################################################################
#Function Name: delete_path
#Inputs:
#Output:
###########################################################################
sub delete_path
{
    my $color_path_ref = shift @_;
    my $path_single_ref = shift @_;
    my $path_double_ref = shift @_;

    my $CLB_in_line = pop @{$color_path_ref};
    my $node_line = pop @{$color_path_ref};
    push @{$color_path_ref},$node_line;
    push @{$color_path_ref},$CLB_in_line;


    #delete the path from the hash if it is a CLB input
    #if ($CLB_in_line =~ /S[01]_(.*)_B/)
    if ($CLB_in_line =~ /S[01]_[FG]_B/)
    {
        #delete related paths in single
        foreach my $OUT_line_var (sort keys(%{$path_single_ref}))
        {
            foreach my $short_line1 (sort keys(%{$$path_single_ref{$OUT_line_var}}))
            {
                if(exists($$path_single_ref{$OUT_line_var}{$short_line1}{$node_line}
                {$CLB_in_line}))
                {
                    #print ".";
                    delete $$path_single_ref{$OUT_line_var}{$short_line1}{$node_line}
                    {$CLB_in_line};
                }
            }
        }

        #delete related paths in double
        foreach my $OUT_line_var (sort keys(%{$path_double_ref}))
        {
            foreach my $short_line1 (sort keys(%{$$path_double_ref{$OUT_line_var}}))
            {
                foreach my $short_line2 (sort keys(%{$$path_double_ref{$OUT_line_var}
                {$short_line1}}))
                {
                    if(exists($$path_double_ref{$OUT_line_var}{$short_line1}
                    {$short_line2}{$node_line}{$CLB_in_line}))
                    {
                        #print ".";
                        delete $$path_double_ref{$OUT_line_var}{$short_line1}
                        {$short_line2}{$node_line}{$CLB_in_line};
                    }
                }
            }
        }
    }
}
###########################################################################
#Function Name: data_struct
#Inputs:
#Output:
###########################################################################
sub data_struct
{
    #input variables
    my $ORM_color_ref = shift @_;
    my $ORM_color_l_ref = shift @_;
    my $CLB_conf_2_4_ref = shift @_;
    my $CLB_conf_2_4_l_ref = shift @_;
    my $CLB_conf_color_ref = shift @_;
    my $CLB_conf_color_l_ref = shift @_;
    my $path_single_ref = shift @_;
    my $path_single_l_ref = shift @_;
    my $path_double_ref = shift @_;
    my $path_double_l_ref = shift @_;

    #data structure for the 16 colors
    foreach (@{$ORM_color_l_ref})
    {
        if($_ =~ /(COLOR\d+):(\w+) (\w+)/)
        {
            $$ORM_color_ref{$1}{$2} = $3;
        }
    }

    #data structure for the list of configurations
    foreach (@{$CLB_conf_2_4_l_ref})
    {
        if($_ =~ /(\w+): (.*)/)
        {
            my $temp = $2;
            chomp $temp;
            push (@{$$CLB_conf_2_4_ref{$1}},split(/ /,$2));
        }
    }

    #data structure for the list of configurations that fit with the colors
    foreach (@{$CLB_conf_color_l_ref})
    {
        if($_ =~ /(\w+):(\w+):(\w+)/)
        {
            $$CLB_conf_color_ref{$1}{$2}{$3} = 1;
        }
    }

    #data structure for the list of single paths
    foreach (@{$path_single_l_ref})
    {
        if($_ =~ /(\w+)->(\w+)->(\w+)->(\w+)/)
        {
            $$path_single_ref{$1}{$2}{$3}{$4} = 1;
        }
    }

    foreach (@{$path_double_l_ref})
    {
        if($_ =~ /(\w+)->(\w+)->(\w+)->(\w+)->(\w+)/)
        {
            my $m1 = $1;
            my $m2 = $2;
            my $m3 = $3;
            my $m4 = $4;
            my $m5 = $5;
            # to avoid overlap between the single and double path
            foreach (keys(%{$path_single_ref}))
            {
                if (exists $$path_single_ref{$_}{$m3}{$m4}{$m5})
                {}
                else
                {
                    $$path_double_ref{$m1}{$m2}{$m3}{$m4}{$m5} = 1;
                }
            }
        }
    }
}
#########################################################################
#Function Name: read_file
#Inputs:
#   name of the file you want to open
#Output:
#   the array that contain all the line of the file
#########################################################################
sub read_file
{
    my ($filename) = @_;
    open(FILE,"$filename")||die("could_not_open_the_file");
    my @file = <FILE>;
    close(FILE);
    chomp(@file);
    return @file;
}
```

B.4 Completion of Feedback Routing

```
#!/usr/local/bin/perl
##########################################################################
```

```
=head1 NAME

color_IRM.pl

=head1 AUTHOR (DATE)

Christian Giasson (August 7th, 2003)

=head1 SYNOPSIS

color_IRM

=head1 DESCRIPTION

This program build (or color) the configurations in the IRM

The coloring part of the program is based on the Vizing algorithm for
edge-color bipartite graphs.

=over 4

=item INPUT

The list of the connections (reduced) in the IRM.

The feedback paths associate with all the Testing configurations 1 to 16.

=item OUTPUT

The output the connections of the testing configurations 1 to 24.

=cut
#########################################################################
use warnings;
use strict;
use Graph;
use Graph::Directed;
use Graph::Undirected;
use Graph::Writer::Dot;
use Set::Array;
use Getopt::Std;


#########################################################################
#Get the inputs of the program
#########################################################################

getopts('ha:b:');
our($opt_h,$opt_a,$opt_b);

if ($opt_h)
{
    print"\n\tOption_of_this_program:\n
_____________\t-h:_\t\t\thelp
_____________\t-a_[file]:_IRM_Color\t\t
_____________\t-b_[file]:_Node_adj_list\t\t\n\n";
            exit;
}


my @graph_edge_l = &read_file($opt_a) if (($opt_a)&&(!$opt_h));
my @feedback_color_l = &read_file($opt_b) if (($opt_b)&&(!$opt_h));

$|=1;

##################################
#    Get the IRM connections
##################################

#Get the graph edges by the standard input
my @G_edges;
foreach (@graph_edge_l)
{
    if (/(\w+) -> (\w+)/)
    {
        push @G_edges, ($1, $2);
    }
}

########################################
#    Get the feedback paths 1 to 16
########################################

my %fb_IRM;
my $cn = 0;
foreach (@feedback_color_l)
{
    if ($_ =~ /(COLOR\d+): .* (\w+) (\w+)$/)
    {
        push @{$fb_IRM{$1}}, "$2->$3";
        #Test
        $cn++;
        #print "$1 $2->$3\n";
    }
}
##########################################
#   Get the opposit short line of the double path for color 1 to 16
##########################################

my %short_opposit;
foreach (@feedback_color_l)
{
    if ($_ =~ /(COLOR\d+): \w+ OUT\d (\w+) (\w+) \w+ \w+$/)
    {
        $short_opposit{$1}{$3} = "$2";
    }
}


##################################################
##################################################
#   Main program
##################################################
##################################################


##########################################
#   build the graph G
##########################################

my $G = Graph::Undirected->new;
$G->add_edges(@G_edges);

##########################################
#   Remove all the edges the graph G that are already used by the feedback
#   paths
##########################################

foreach my $color (sort keys(%fb_IRM))
{
    foreach my $conn (@{$fb_IRM{$color}})
    {
        if ($conn =~ /(\w+)->(\w+)/)
        {
            #delete that edge in the graph G
            $G->delete_edge($1, $2);
            #print "$color : $1 -- $2\n";
        }
    }
}

#test
#my @test_e = $G->edges();
#my $num = @test_e;
#$num = $num / 2;
#print "$num\n";

##########################################
#   for the color 1 to 16
##########################################
my $color_index = 1;
my %coloring;
foreach my $color (sort keys(%fb_IRM))
{
    my @edge_g = $G->edges();
    my @color_conn = @{$fb_IRM{$color}};
    #get the match(color)
    my @IRM_color_edge = &get_IRM_match(\@edge_g, \@color_conn, $color);

    # put the edges in the coloring hash
    for(1 .. (scalar(@IRM_color_edge)/2))
    {
        my $vs = shift(@IRM_color_edge);
        push @IRM_color_edge, $vs;
        my $vd = shift(@IRM_color_edge);
        push @IRM_color_edge, $vd;

        push @{$coloring{"$color-$color_index"}},"$vs_->_$vd";
    }

    # delete the edges in the Graph "G"
    for(1 .. (scalar(@IRM_color_edge)/2))
    {
        my $vs = shift(@IRM_color_edge);
        my $vd = shift(@IRM_color_edge);

        $G->delete_edge($vs, $vd);
    }

    #increment the color index
    $color_index++;
}

########################################
#   print the result
########################################
my $c = 1;
foreach my $color (sort keys(%coloring))
{
    foreach (@{$coloring{$color}})
    {
        print "$color_:_$_\n";
        #$c++;
    }
}
#print "$c\n";

#test
my @test_edges = $G->edges();
#my $d = 0;
for (1..(scalar(@test_edges)/2))
{
    my $vs = shift @test_edges;
    my $vd = shift @test_edges;

    #print "$vs -> $vd\n";
    #     $d++;
}
#print "number of connections left is $d\n";

#$d = $d/2;
#$c = $c + $d;
#print "$c\n";


#test for find_covering_matching function
#my @G_edges = qw(A D A E A F B E B F C D C F);
#my @G_edges = qw(A D A E A F B D B E B F C D C E C F);
#my @G_edges = qw(A C A D A E A F B C B D B E B F);
#my @G_edges = qw(A E A F A G B E B F B H C G D G D H);

#my @M = &C1(\@G_edges);
#print "@M\n";

######################################################################
#          Functions
######################################################################

######################################################################
#Function Name: get_IRM_match
# Inputs:
#   The remaining edges in G (IRM connections)
#   The connection of the colors in IRM
# Output:
#   A set of edges in IRM that match with the color edges
######################################################################
sub get_IRM_match
{
    #get the inputs of the function
    my $graph_edge_ref = shift;
    my $color_conn_ref = shift;
    my $color = shift;

    my @graph_edge = @{$graph_edge_ref};
    my @color_conn = @{$color_conn_ref};

    #build the local graph G
    my $G = Graph::Undirected->new;
    $G->add_edges(@graph_edge);

    ############################
    #   Remove all the vertices in the graph G that are related to the
    #   feedbacks connection in the IRM
    ############################

    foreach my $edge (@color_conn)
    {
        if ($edge =~ /(\w+)->(\w+)/)
        {
            $G->delete_vertex($1);
            $G->delete_vertex($2);
        }
    }

    ####################################
    # NEW
    #   Remove the edges relates to the opposite short line in GRM
    #   note: modified for diagonal short connections (not orthogonal)
    ####################################
    foreach my $edge (@color_conn)
    {
        if ($edge =~ /(\w+)->(\w+)/)
        {
            my $node_line = $1;
            my $op = 0;
            if ($node_line =~ /([NSEW])_P(\d+)/)
            {
                if (exists($short_opposit{$color}{"$1$2"}))
                {
                    $op = $short_opposit{$color}{"$1$2"};
                }
                if($op =~ /([NSEW])(\d+)/)
                {
                    #remove the first short edge in the double path
                    $G->delete_vertex("$1"."_P$2");
                    #remove the oposite of that edge too
                    if($1 eq 'N')
                    {
                        $G->delete_vertex("S_P$2");
                    }
                    elsif($1 eq 'S')
                    {
                        $G->delete_vertex("N_P$2");
                    }
                    elsif($1 eq 'W')
                    {
                        $G->delete_vertex("E_P$2");
                    }
                    elsif($1 eq 'E')
                    {
                        $G->delete_vertex("W_P$2");
                    }
                }
            }

            if($node_line =~ /S_P(\d+)/)
            {
                $G->delete_vertex("N_P$1");
            }
            elsif($node_line =~ /N_P(\d+)/)
            {
                $G->delete_vertex("S_P$1");
            }
            elsif($node_line =~ /E_P(\d+)/)
            {
                $G->delete_vertex("W_P$1");
            }
            elsif($node_line =~ /W_P(\d+)/)
            {
                $G->delete_vertex("E_P$1");
            }
        }
    }

    ###########################
    #   Color the remaining graph and get the color 1
    ###########################

    my @after_del_edge = $G->edges(); my @color1 =
    &C1(\@after_del_edge);

    ###########################
    #   Return the connections in the IRM
    ###########################

    return @color1;
}

##################################################################
#Function Name: C1
#Inputs:
#(\@{list_of_edges_G})
#Output:     the color number 1 of the coloring
#       (the fonction find an entire coloring but return only)
##################################################################

sub C1
{


    ############################################################
    # put all the input of the function in arrays
    ############################################################
```

```
my @graph_edge = @{(shift(@_))};
my @graph_edge_copy = @graph_edge;

#################################################################
# Build the graph
#################################################################

my $G = Graph::Undirected->new;
$G->add_edges(@graph_edge);

#################################################################
# Set all the edges color to 0
#################################################################

for (1 .. (scalar(@graph_edge)/2))
{
    my $color = 0;
    my $vs = shift(@graph_edge);
    my $vd = shift(@graph_edge);
    push @graph_edge,$vs;
    push @graph_edge,$vd;

    $G->set_attribute('color',$vs,$vd,$color);
    $G->set_attribute('color',$vd,$vs,$color);
}
#################################################################
# Star of the main loop
#################################################################

my $alpha;
my $beta;

#test variable
my $index = 1;
for(1 .. (scalar(@graph_edge)/2))
{

    #test
    #print "$index ";
    #$index++;

    my $vs = shift(@graph_edge);
    my $vd = shift(@graph_edge);
    push @graph_edge,$vs;
    push @graph_edge,$vd;


    #################################################################
    # find the lower color missing at $vs
    #################################################################

    $alpha = &lower_color(\$G,$vs);
    #test
    #print "$alpha ";


    #################################################################
    #find the lower color missing at $vd
    #################################################################

    $beta = &lower_color(\$G,$vd);
    #test
    #print "$beta ";

    #################################################################
    # Check if alpha not equal beta
    #################################################################
    if ($alpha != $beta)
    {
        #################################################################
        #find an alternative path and swap color
        #################################################################
        &alt_path(\$G,$vs,$vd,$alpha,$beta);
    }
    #################################################################
    #color the edge (vs,vd) with beta
    #################################################################
    $G->set_attribute('color',$vs,$vd,$beta);
    $G->set_attribute('color',$vd,$vs,$beta);

}
#################################################################
# return color 1
#################################################################

my @color1_edges;
```

```
    #my $count = 1;
    #foreach my $color (1..25)
    #{
    my $color = 1;
    for(1 .. (scalar(@graph_edge)/2))
    {
        my $vs = shift(@graph_edge);
        my $vd = shift(@graph_edge);
        push @graph_edge,$vs;
        push @graph_edge,$vd;

        #if($G->get_attribute('color',$vs,$vd) == 1)
        if($G->get_attribute('color',$vs,$vd) == $color)
        {
            push @color1_edges,$vs,$vd;
            #print "COLOR$color:$vs - $vd\n";
        }

        #test
        #my $test = $G->get_attribute('color',$vs,$vd);
        #print "color $vs-$vd = $test\n";
    }
    #}
    return @color1_edges;
}

#Return the lower missing color at the vertex
sub lower_color
{
    my $G = shift;
    my $vertex = shift;
    my $color = 1;
    my @color_list;

    my @adj_list = $$G->neighbors($vertex);


    #build a list of all the color
    foreach my $adj_vertex (@adj_list)
    {
        my $temp = $$G->get_attribute('color',$vertex,$adj_vertex);
        if ($temp != 0)
        {
            push @color_list, $temp;
        }
    }


    #find the smallest missing
    #@color_list = sort(@color_list);

    #test
    #print "@color_list: lowest missing ";
    my $got_it;
    for my $local_color (1..50)
    {
        $got_it = 1;
        foreach my $c (@color_list)
        {
            if ($c == $local_color)
            {
                $got_it = 0;
                last;
            }
        }
        if ($got_it == 1)
        {
            $color = $local_color;
            last;
        }
    }
    #my $prev_c = shift @color_list;
    #foreach my $c (sort(@color_list))
    #{
    #    if($prev_c == $c)
    #    {next;}
    #    elsif($c != $color)
    #    {last;}
    #    else
    #    {$color++;}
    #    $prev_c = $c;
    #}

    #test
    #print "$color\n";

    return $color;
}
```

```
sub alt_path
{
    my $G = shift;
    my $vs = shift;
    my $vd = shift;
    my $alpha = shift;
    my $beta = shift;

    #test
    #print "$G $vs $alpha $beta\n";


    my @alternate_path = &path(\$G,$vs,$alpha,$beta);


    if(@alternate_path)
    {
        &inv_path_color(\$G,\@alternate_path,$alpha,$beta);
    }
    else
    {
        @alternate_path = &path(\$G,$vd,$alpha,$beta);
        if(@alternate_path)
        {
            &inv_path_color(\$G,\@alternate_path,$alpha,$beta);
        }
        else
        {
            return;
        }
    }
}

sub path
{
    my $G = shift;
    my $vertex = shift;
    my $alpha = shift;
    my $beta = shift;

    my $temp_color = $beta;
    my @path_list;
    my $next_vertex;

    $next_vertex = &check_color(\$G,$vertex,$temp_color);
    if($next_vertex eq 0)
    {return @path_list;}

    while($next_vertex ne 0)
    {
        push (@path_list,$next_vertex);
        push (@path_list,$next_vertex);

        if($temp_color == $alpha)
        {$temp_color = $beta;}
        else
        {$temp_color = $alpha;}

        $next_vertex = &check_color(\$G,$next_vertex,$temp_color);


        # have to break the path when it is a loop because the program enter in
        # an infinite loop.

        # if the next_vertex is equal to vertex
        if ($next_vertex eq $vertex)
        {last;}

        my $flag = 0;
        foreach (@path_list)
        {
            if ($_ eq $next_vertex)
            {
                $flag = 1;
            }
        }
        last if $flag == 1;
        #test
        #print "$vertex:$next_vertex ";
        #print ".";
    }
    my $extra = pop @path_list;
    unshift (@path_list,$vertex);
    return @path_list;
}

sub inv_path_color
{
    my $G = shift;
    my @path = @{(shift(@_))};
    my $alpha = shift;
    my $beta = shift;

    #test
    #print "@path\n";

    for(1..(scalar(@path)/2))
    {
        my $start = shift @path;
        my $vertex = shift @path;

        if($$$G->get_attribute('color',$start,$vertex) == $beta)
        {
            $$$G->set_attribute('color',$start,$vertex,$alpha);
            $$$G->set_attribute('color',$vertex,$start,$alpha);
        }
        elsif($$$G->get_attribute('color',$start,$vertex) == $alpha)
        {
            $$$G->set_attribute('color',$start,$vertex,$beta);
            $$$G->set_attribute('color',$vertex,$start,$beta);
        }
    }
}

sub check_color
{
    my $G = shift;
    my $vertex = shift;
    my $color = shift;

    my $edge_color;

    my @vertex_adj = $$$$G->neighbors($vertex);
    foreach my $v_end (@vertex_adj)
    {
        $edge_color = $$$$G->get_attribute('color',$vertex,$v_end);
        if($edge_color == $color)
        {
            return $v_end;
        }
    }
    return 0;
}
##################################################################
#Function Name: read_file
#Inputs:
#   name of the file you want to open
#Output:
#   the array that contain all the line of the file
##################################################################
sub read_file
{
    my ($filename) = @_;
    open(FILE,"$filename")||die("could not open the file");
    my @file = <FILE>;
    close(FILE);
    chomp(@file);
    return @file;
}
```

B.5 Intermediate Code

```
#!/usr/local/bin/perl
#####################################################################
=head1 NAME

color_IRM_rest.pl

=head1 AUTHOR (DATE)

Christian Giasson (August 7th, 2003)

=head1 SYNOPSIS

color_IRM_rest

=head1 DESCRIPTION

This program output the rest of edges in IRM that were not color yet to
the standard output.

This code is really similar to color_IRM.pl

=over 4
```

```
=item INPUT

The list of the connections (reduced) in the IRM.

The feedback paths associate with all the Testing configurations 1 to 16.

=item OUTPUT

Rest of IRM to the STDOUT.

=cut
##############################################################################
use warnings;
use strict;
use Graph;
use Graph::Directed;
use Graph::Undirected;
use Graph::Writer::Dot;
use Set::Array;
use Getopt::Std;


##############################################################################
#Get the inputs of the program
##############################################################################

getopts('ha:b:');
our($opt_h,$opt_a,$opt_b);

if ($opt_h)
{
    print"\n\tOption␣of␣this␣program:\n
␣␣␣␣␣␣␣␣␣␣␣␣␣\t-h:␣\t\thelp
␣␣␣␣␣␣␣␣␣␣␣␣␣\t-a␣[file]:␣IRM␣Color\t\t
␣␣␣␣␣␣␣␣␣␣␣␣␣\t-b␣[file]:␣Node␣adj␣list\t\t\n\n";
            exit;
}


my @graph_edge_l = &read_file($opt_a) if (($opt_a)&&(!$opt_h));
my @feedback_color_l = &read_file($opt_b) if (($opt_b)&&(!$opt_h));

$|=1;

####################################
#    Get the IRM connections
####################################

#Get the graph edges by the standard input
my @G_edges;
foreach (@graph_edge_l)
{
    if (/(\w+) -> (\w+)/)
    {
        push @G_edges, ($1, $2);
    }
}

##########################################
#    Get the feedback paths 1 to 16
##########################################

my %fb_IRM;
my $cn = 0;
foreach (@feedback_color_l)
{
    if ($_ =~ /(COLOR\d+): .* (\w+) (\w+)$/)
    {
        push @{$fb_IRM{$1}}, "$2->$3";
        #Test
        $cn++;
        #print "$1 $2->$3\n";
    }
}
###################################################
###################################################
#    Main program
###################################################
###################################################


##########################################
#    build the graph G
##########################################

my $G = Graph::Undirected->new;
$G->add_edges(@G_edges);

##########################################
```

```
#   Remove all the edges the graph G that are already used by the feedback
#   paths
##########################################

foreach my $color (sort keys(%fb_IRM))
{
    foreach my $conn (@{$fb_IRM{$color}})
    {
        if ($conn =~ /(\w+)->(\w+)/)
        {
            #delete that edge in the graph G
            $G->delete_edge($1, $2);
            #print "$color : $1 -- $2\n";
        }
    }
}

#test
#my @test_e = $G->edges();
#my $num = @test_e;
#$num = $num / 2;
#print "$num\n";

##########################################
#   for the color 1 to 16
##########################################
my $color_index = 1;
my %coloring;
foreach my $color (sort keys(%fb_IRM))
{
    my @edge_g = $G->edges();
    my @color_conn = @{$fb_IRM{$color}};
    #get the match(color)
    my @IRM_color_edge = &get_IRM_match(\@edge_g, \@color_conn);

    # put the edges in the coloring hash
    for(1 .. (scalar(@IRM_color_edge)/2))
    {
        my $vs = shift(@IRM_color_edge);
        push @IRM_color_edge,$vs;
        my $vd = shift(@IRM_color_edge);
        push @IRM_color_edge,$vd;

        push @{$coloring{"$color-$color_index"}},"$vs_->_$vd";
    }

    # delete the edges in the Graph "G"
    for(1 .. (scalar(@IRM_color_edge)/2))
    {
        my $vs = shift(@IRM_color_edge);
        my $vd = shift(@IRM_color_edge);

        $G->delete_edge($vs, $vd);
    }

    #increment the color index
    $color_index++;
}

##########################################
#   print the result
##########################################
my $c = 1;
foreach my $color (sort keys(%coloring))
{
    foreach (@{$coloring{$color}})
    {
        #print "$color : $_\n";
        #$c++;
    }
}
#print "$c\n";

#test
my @test_edges = $G->edges();
#my $d = 0;
for (1..(scalar(@test_edges)/2))
{
    my $vs = shift @test_edges;
    my $vd = shift @test_edges;

    print "$vs_->_$vd\n";
    #     $d++;
}
#print "number of connections left is $d\n";

#$d = $d/2;
#$c = $c + $d;
#print "$c\n";
```

```
#test for find_covering_matching function
#my @G_edges = qw(A D A E A F B E B F C D C F);
#my @G_edges = qw(A D A E A F B D B E B F C D C E C F);
#my @G_edges = qw(A C A D A E A F B C B D B E B F);
#my @G_edges = qw(A E A F A G B E B F B H C G D G D H);

#my @M = &C1(\ @G_edges);
#print "@M\n";

##########################################################################
#           Functions
##########################################################################

##########################################################################
#Function Name: get_IRM_match
# Inputs:
#   The remaining edges in G (IRM connections)
#   The connection of the colors in IRM
# Output:
#   A set of edges in IRM that match with the color edges
##########################################################################
sub get_IRM_match
{
    #get the inputs of the function
    my $graph_edge_ref = shift;
    my $color_conn_ref = shift;

    my @graph_edge = @{$graph_edge_ref};
    my @color_conn = @{$color_conn_ref};

    #build the local graph G
    my $G = Graph::Undirected->new;
    $G->add_edges(@graph_edge);

    ##############################
    #   Remove all the vertices in the graph G that are related to the
    #   feedbacks connection in the IRM
    ##############################

    foreach my $edge (@color_conn)
    {
        if ($edge =~ /(\w+)->(\w+)/)
        {
            $G->delete_vertex($1);
            $G->delete_vertex($2);
        }
    }

    ######################################
    # NEW
    #   Remove the edges relates to the opposite short line in GRM
    ######################################
    foreach my $edge (@color_conn)
    {
        if ($edge =~ /(\w+)->(\w+)/)
        {
            my $node_line = $1;
            if ($node_line =~ /N_P(\d+)/)
            {
                $G->delete_vertex("S_P$1");
            }
            elsif($node_line =~ /S_P(\d+)/)
            {
                $G->delete_vertex("N_P$1");
            }
            elsif($node_line =~ /E_P(\d+)/)
            {
                $G->delete_vertex("W_P$1");
            }
            elsif($node_line =~ /W_P(\d+)/)
            {
                $G->delete_vertex("E_P$1");
            }
        }
    }

    ##############################
    #   Color the remaining graph and get the color 1
    ##############################

    my @after_del_edge = $G->edges(); my @color1 =
    &C1(\ @after_del_edge);

    ##############################
    #   Return the connections in the IRM
    ##############################
    return @color1;
}

##########################################################################
#Function Name: C1
#Inputs:
#(\@{list_of_edges_G})
#Output:     the color number 1 of the coloring
#        (the fonction find an entire coloring but return only)
##########################################################################

sub C1
{

    ##################################################################
    # put all the input of the function in arrays
    ##################################################################

    my @graph_edge = @{(shift(@_))};
    my @graph_edge_copy = @graph_edge;

    ##################################################################
    # Build the graph
    ##################################################################


    my $G = Graph::Undirected->new;
    $G->add_edges(@graph_edge);

    ##################################################################
    # Set all the edges color to 0
    ##################################################################

    for (1 .. (scalar(@graph_edge)/2))
    {
        my $color = 0;
        my $vs = shift(@graph_edge);
        my $vd = shift(@graph_edge);
        push @graph_edge, $vs;
        push @graph_edge, $vd;

        $G->set_attribute('color',$vs,$vd,$color);
        $G->set_attribute('color',$vd,$vs,$color);
    }

    ##################################################################
    # Star of the main loop
    ##################################################################

    my $alpha;
    my $beta;

    #test variable
    my $index = 1;
    for(1 .. (scalar(@graph_edge)/2))
    {

        #test
        #print "$index ";
        #$index++;

        my $vs = shift(@graph_edge);
        my $vd = shift(@graph_edge);
        push @graph_edge, $vs;
        push @graph_edge, $vd;


        ##################################################################
        # find the lower color missing at $vs
        ##################################################################

        $alpha = &lower_color(\$G,$vs);
        #test
        #print "$alpha ";


        ##################################################################
        #find the lower color missing at $vd
        ##################################################################

        $beta = &lower_color(\$G,$vd);
        #test
        #print "$beta ";

        ##################################################################
        # Check if alpha not equal beta
        ##################################################################
        if ($alpha != $beta)
        {
            ################################################################
            #find an alternative path and swap color
            ################################################################
            &alt_path(\$G,$vs,$vd,$alpha,$beta);
        }
        ################################################################
        #color the edge (vs,vd) with beta
        ################################################################
        $G->set_attribute('color',$vs,$vd,$beta);
        $G->set_attribute('color',$vd,$vs,$beta);

    }
    ################################################################
    # return color 1
    ################################################################

    my @color1_edges;

    #my $count = 1;
    #foreach my $color (1..25)
    #{
    my $color = 1;
    for(1 .. (scalar(@graph_edge)/2))
    {
        my $vs = shift(@graph_edge);
        my $vd = shift(@graph_edge);
        push @graph_edge,$vs;
        push @graph_edge,$vd;

        #if($G->get_attribute('color',$vs,$vd) == 1)
        if($G->get_attribute('color',$vs,$vd) == $color)
        {
            push @color1_edges,$vs,$vd;
            #print "COLOR$color:$vs - $vd\n";
        }

        #test
        #my $test = $G->get_attribute('color',$vs,$vd);
        #print "color $vs-$vd = $test\n";
    }
    #}
    return @color1_edges;
}

#Return the lower missing color at the vertex
sub lower_color
{
    my $G = shift;
    my $vertex = shift;
    my $color = 1;
    my @color_list;

    my @adj_list = $$G->neighbors($vertex);


    #build a list of all the color
    foreach my $adj_vertex (@adj_list)
    {
        my $temp = $$G->get_attribute('color',$vertex,$adj_vertex);
        if ($temp != 0)
        {
            push @color_list, $temp;
        }
    }


    #find the smallest missing
    #@color_list = sort(@color_list);

    #test
    #print "@color_list: lowest missing ";
    my $got_it;
    for my $local_color (1..50)
    {
        $got_it = 1;
        foreach my $c (@color_list)
        {
            if ($c == $local_color)
            {
                $got_it = 0;
                last;
            }
        }
        if ($got_it == 1)
        {
            $color = $local_color;
```

```
                last;
            }
        }
        #my $prev_c = shift @color_list;
        #foreach my $c (sort(@color_list))
        #{
        #     if($prev_c == $c)
        #     {next;}
        #     elsif($c != $color)
        #     {last;}
        #     else
        #     {$color++;}
        #     $prev_c = $c;
        #}

        #test
        #print "$color\n";

        return $color;
}

sub alt_path
{
    my $G = shift;
    my $vs = shift;
    my $vd = shift;
    my $alpha = shift;
    my $beta = shift;

    #test
    #print "$G $vs $alpha $beta\n";


    my @alternate_path = &path(\$G,$vs,$alpha,$beta);


    if(@alternate_path)
    {
        &inv_path_color(\$G,\@alternate_path,$alpha,$beta);
    }
    else
    {
        @alternate_path = &path(\$G,$vd,$alpha,$beta);
        if(@alternate_path)
        {
            &inv_path_color(\$G,\@alternate_path,$alpha,$beta);
        }
        else
        {
            return;
        }
    }
}

sub path
{
    my $G = shift;
    my $vertex = shift;
    my $alpha = shift;
    my $beta = shift;

    my $temp_color = $beta;
    my @path_list;
    my $next_vertex;

    $next_vertex = &check_color(\$G,$vertex,$temp_color);
    if($next_vertex eq 0)
    {return @path_list;}

    while($next_vertex ne 0)
    {
        push (@path_list,$next_vertex);
        push (@path_list,$next_vertex);

        if($temp_color == $alpha)
        {$temp_color = $beta;}
        else
        {$temp_color = $alpha;}

        $next_vertex = &check_color(\$G,$next_vertex,$temp_color);


        # have to break the path when it is a loop because the program enter in
        # an infinite loop.

        # if the next_vertex is equal to vertex
        if ($next_vertex eq $vertex)
        {last;}
        my $flag = 0;
        foreach (@path_list)
        {
            if ($_ eq $next_vertex)
            {
                $flag = 1;
            }
        }
        last if $flag == 1;
        #test
        #print "$vertex:$next_vertex ";
        #print ".";
    }
    my $extra = pop @path_list;
    unshift (@path_list,$vertex);
    return @path_list;
}

sub inv_path_color
{
    my $G = shift;
    my @path = @{(shift(@_))};
    my $alpha = shift;
    my $beta = shift;

    #test
    #print "@path\n";

    for(1..(scalar(@path)/2))
    {
        my $start = shift @path;
        my $vertex = shift @path;

        if($$$G->get_attribute('color',$start,$vertex) == $beta)
        {
            $$$G->set_attribute('color',$start,$vertex,$alpha);
            $$$G->set_attribute('color',$vertex,$start,$alpha);
        }
        elsif($$$G->get_attribute('color',$start,$vertex) == $alpha)
        {
            $$$G->set_attribute('color',$start,$vertex,$beta);
            $$$G->set_attribute('color',$vertex,$start,$beta);
        }
    }
}

sub check_color
{
    my $G = shift;
    my $vertex = shift;
    my $color = shift;

    my $edge_color;

    my @vertex_adj = $$$$G->neighbors($vertex);
    foreach my $v_end (@vertex_adj)
    {
        $edge_color = $$$$G->get_attribute('color',$vertex,$v_end);
        if($edge_color == $color)
        {
            return $v_end;
        }
    }
    return 0;
}
####################################################################
#Function Name: read_file
#Inputs:
#   name of the file you want to open
#Output:
#   the array that contain all the line of the file
####################################################################
sub read_file
{
    my ($filename) = @_;
    open(FILE,"$filename")||die("could_not_open_the_file");
    my @file = <FILE>;
    close(FILE);
    chomp(@file);
    return @file;
}

#!/usr/bin/perl
############################################################################
use warnings;
use strict;

# Author:    Christian Giasson
# Date:      August 10th 2003
# Comment:   This is a small program that pars the entire CLB tile to find all
#        the connections in the IRM that are not connect the node lines and the CLBs
#        and the tri-state buffers inputs.

my @file_lines = <>;

my %adj_list;
foreach my $line (@file_lines)
{
    chomp($line);

    #get the connection to the CLBs
    if($line =~ /(\w+ -> S[01]_\w+)\)/)
    {
        my $connection = $1;
        if ($connection !~ /[NSEW]_P\d+ ->/)
        {
            print "$connection\n";
        }
    }
    #get the tri-state buffer connections
    elsif($line =~ /(\w+ -> T\w+)\)/)
    {
        #print "$1\n";
        my $connection = $1;
        if (($connection !~ /[NSEW]_P\d+ ->/)&&($connection !~ /TBUF/))
        {
            print "$connection\n";
        }
    }
}
```

B.6 Edge Coloring of G_{IRM}

```
#!/usr/local/bin/perl
###########################################################################
=head1 NAME

color_IRM_part2.pl

=head1 AUTHOR (DATE)

Christian Giasson (August 10th, 2003)

=head1 SYNOPSIS

color_IRM_part2.pl

=head1 DESCRIPTION

This program build (or color) the testing configurations 17 to 26 in the IRM.

The coloring part of the program is based on the Vizing algorithm for
edge-color bipartite graphs.

=over 4

=item INPUT

The list of the connections left by the previous coloring [1..16] in the IRM.
plus other one that dont come from the node lines.

=item OUTPUT

The output the connections of the testing configurations 17 to 26.

=cut
###########################################################################
use warnings;
use strict;
use Graph;
use Graph::Directed;
use Graph::Undirected;
use Graph::Writer::Dot;
use Set::Array;
use Getopt::Std;


###########################################################################
#Get the inputs of the program
###########################################################################

getopts('ha:b:');
our($opt_h,$opt_a,$opt_b);

if ($opt_h)
```

```
{
    print"\n\tOption_of_this_program:\n
_____________\t-h:_\t\t\thelp
_____________\t-a_[file]:_IRM_edges
_____________\t-b_[file]:_IRM_edges\n\n";
            exit;
}


my @IRM_edges = &read_file($opt_a) if (($opt_a)&&(!$opt_h));
my @feedback_17_22_l = &read_file($opt_b) if (($opt_b)&&(!$opt_h));

#################################
#   Get the IRM connections
#################################

#Get the graph edges by the standard input
my @G_edges;
foreach (@IRM_edges)
{
    if (/(\w+) -> (\w+)/)
    {
        push @G_edges, ($1, $2);
    }
}

#######################################################
#   Get the feedback connections for the color [17..22]
#######################################################
my %feedback_17_22;
foreach (@feedback_17_22_l)
{
    if (/(\w+):(.*)$/)
    {
        push @{$feedback_17_22{$1}}, $2;
    }
}


#################################################
#################################################
#   Main program
#################################################
#################################################


#########################################
#   build the graph G
#########################################

my $G = Graph::Undirected->new;
$G->add_edges(@G_edges);


#########################################
#   Color graph G
#########################################
#BC1(\@G_edges);

#########################################
#   Remove all the edges the graph G that are already used by the feedback
#   paths
#########################################

foreach my $color (sort keys(%feedback_17_22))
{
    foreach my $conn (@{$feedback_17_22{$color}})
    {
        if ($conn =~ /(\w+) -> (\w+)/)
        {
            #delete that edge in the graph G
            $G->delete_edge($1, $2);
            #print "$color : $1 -- $2\n";
        }
    }
}

#########################################
#   for the color 17 to 22
#########################################
my %coloring;
foreach my $color (sort keys(%feedback_17_22))
{
    my @edge_g = $G->edges();
    my @color_conn = @{$feedback_17_22{$color}};
    my @IRM_color_edge = &get_IRM_match(\@edge_g, \@color_conn);

    # put the edges in the coloring hash
    for(1 .. (scalar(@IRM_color_edge)/2))
    {
        my $vs = shift(@IRM_color_edge);
        push @IRM_color_edge,$vs;
        my $vd = shift(@IRM_color_edge);
        push @IRM_color_edge,$vd;

        push @{$coloring{"$color"}},"$vs_->_$vd";
    }

    # delete the edges in the Graph "G"
    for(1 .. (scalar(@IRM_color_edge)/2))
    {
        my $vs = shift(@IRM_color_edge);
        my $vd = shift(@IRM_color_edge);

        $G->delete_edge($vs, $vd);
    }
}
########################################
#   for the color 23 to 26
########################################
for my $color (23..26)
{
    my @edge_g = $G->edges();
    my @IRM_color_edge = &C1(\@edge_g);

    # put the edges in the coloring hash
    for(1 .. (scalar(@IRM_color_edge)/2))
    {
        my $vs = shift(@IRM_color_edge);
        push @IRM_color_edge,$vs;
        my $vd = shift(@IRM_color_edge);
        push @IRM_color_edge,$vd;

        push @{$coloring{"COLOR$color"}},"$vs_->_$vd";
    }

    # delete the edges in the Graph "G"
    for(1 .. (scalar(@IRM_color_edge)/2))
    {
        my $vs = shift(@IRM_color_edge);
        my $vd = shift(@IRM_color_edge);

        $G->delete_edge($vs, $vd);
    }
}


########################################
#   print the result
########################################
#my $c = 1;
foreach my $color (sort keys(%coloring))
{
    foreach (@{$coloring{$color}})
    {
        print "$color_:_$_\n";
        #$c++;
    }
}
####################################################################
#           Functions
####################################################################

####################################################################
#Function Name: get_IRM_match
# Inputs:
#   The remaining edges in G (IRM connections)
#   The connection of the colors in IRM
# Output:
#   A set of edges in IRM that match with the color edges
####################################################################
sub get_IRM_match
{
    #get the inputs of the function
    my $graph_edge_ref = shift;
    my $color_conn_ref = shift;

    my @graph_edge = @{$graph_edge_ref};
    my @color_conn = @{$color_conn_ref};

    #build the local graph G
    my $G = Graph::Undirected->new;
    $G->add_edges(@graph_edge);

    ###########################
    #   Remove all the vertices in the graph G that are related to the
    #   feedbacks connection in the IRM
    ############################

    foreach my $edge (@color_conn)
    {
        if ($edge =~ /(\w+) -> (\w+)/)
        {
            $G->delete_vertex($1);
            $G->delete_vertex($2);
        }
    }

    ######################################
    # NEW
    #   Remove the edges relates to the opposite short line in GRM
    ######################################
    foreach my $edge (@color_conn)
    {
        if ($edge =~ /(\w+) -> (\w+)/)
        {
            my $node_line = $1;
            if ($node_line =~ /N_P(\d+)/)
            {
                $G->delete_vertex("S_P$1");
            }
            elsif($node_line =~ /S_P(\d+)/)
            {
                $G->delete_vertex("N_P$1");
            }
            elsif($node_line =~ /E_P(\d+)/)
            {
                $G->delete_vertex("W_P$1");
            }
            elsif($node_line =~ /W_P(\d+)/)
            {
                $G->delete_vertex("E_P$1");
            }
        }
    }

    ############################
    #   Color the remaining graph and get the color 1
    ############################

    my @after_del_edge = $G->edges();

    my @color1 = &C1(\@after_del_edge);

    ############################
    #   Return the connections in the IRM
    ############################

    return @color1;
}

######################################################################
#Function Name: C1
#Inputs:
#(\@{list_of_edges_G})
#Output:     the color number 1 of the coloring
#        (the fonction find an entire coloring but return only)
######################################################################

sub C1
{

    ##############################################################
    # put all the input of the function in arrays
    ##############################################################

    my @graph_edge = @{(shift(@_))};
    my @graph_edge_copy = @graph_edge;

    ##############################################################
    # Build the graph
    ##############################################################

    my $G = Graph::Undirected->new;
    $G->add_edges(@graph_edge);

    ##############################################################
    # Set all the edges color to 0
    ##############################################################

    for (1 .. (scalar(@graph_edge)/2))
```

```
{
    my $color = 0;
    my $vs = shift(@graph_edge);
    my $vd = shift(@graph_edge);
    push @graph_edge,$vs;
    push @graph_edge,$vd;

    $G->set_attribute('color',$vs,$vd,$color);
    $G->set_attribute('color',$vd,$vs,$color);
}
################################################################
# Star of the main loop
################################################################

my $alpha;
my $beta;

#test variable
my $index = 1;
for(1 .. (scalar(@graph_edge)/2))
{

    #test
    #print "$index ";
    #$index++;

    my $vs = shift(@graph_edge);
    my $vd = shift(@graph_edge);
    push @graph_edge,$vs;
    push @graph_edge,$vd;


    ################################################################
    # find the lower color missing at $vs
    ################################################################

    $alpha = &lower_color(\$G,$vs);
    #test
    #print "$alpha ";


    ################################################################
    #find the lower color missing at $vd
    ################################################################

    $beta = &lower_color(\$G,$vd);
    #test
    #print "$beta ";

    ################################################################
    # Check if alpha not equal beta
    ################################################################
    if ($alpha != $beta)
    {
        ################################################################
        #find an alternative path and swap color
        ################################################################
        &alt_path(\$G,$vs,$vd,$alpha,$beta);
    }
    ################################################################
    #color the edge (vs,vd) with beta
    ################################################################
    $G->set_attribute('color',$vs,$vd,$beta);
    $G->set_attribute('color',$vd,$vs,$beta);

}
################################################################
# return color 1
################################################################

my @color1_edges;

#my $count = 17;
#foreach my $color (1..50)
#{
my $color = 1;
for(1 .. (scalar(@graph_edge)/2))
{
    my $vs = shift(@graph_edge);
    my $vd = shift(@graph_edge);
    push @graph_edge,$vs;
    push @graph_edge,$vd;

    #if($G->get_attribute('color',$vs,$vd) == 1)
    if($G->get_attribute('color',$vs,$vd) == $color)
    {
        push @color1_edges,$vs,$vd;
        #print "COLOR$color:$vs - $vd\n";
```

```
            }

            #test
            #my $test = $G->get_attribute('color',$vs,$vd);
            #print "color $vs-$vd = $test\n";
        }
        #$count++;
        #}
        return @color1_edges;

}

#Return the lower missing color at the vertex
sub lower_color
{
    my $G = shift;
    my $vertex = shift;
    my $color = 1;
    my @color_list;

    my @adj_list = $$G->neighbors($vertex);


    #build a list of all the color
    foreach my $adj_vertex (@adj_list)
    {
        my $temp = $$G->get_attribute('color',$vertex,$adj_vertex);
         if ($temp != 0)
         {
              push @color_list, $temp;
         }
    }


    #find the smallest missing
    #@color_list = sort(@color_list);

    #test
    #print "@color_list: lowest missing ";
    my $got_it;
    for my $local_color (1..50)
    {
         $got_it = 1;
         foreach my $c (@color_list)
         {
              if ($c == $local_color)
              {
                   $got_it = 0;
              last;
              }
         }
         if ($got_it == 1)
         {
              $color = $local_color;
              last;
         }
    }
    #my $prev_c = shift @color_list;
    #foreach my $c (sort(@color_list))
    #{
    #     if($prev_c == $c)
    #     {next;}
    #     elsif($c != $color)
    #     {last;}
    #     else
    #     {$color++;}
    #     $prev_c = $c;
    #}

    #test
    #print "$color\n";

    return $color;
}

sub alt_path
{
    my $G = shift;
    my $vs = shift;
    my $vd = shift;
    my $alpha = shift;
    my $beta = shift;

    #test
    #print "$G $vs $alpha $beta\n";


    my @alternate_path = &path(\$G,$vs,$alpha,$beta);
    if(@alternate_path)
    {
        &inv_path_color(\$G,\@alternate_path,$alpha,$beta);
    }
    else
    {
        @alternate_path = &path(\$G,$vd,$alpha,$beta);
        if(@alternate_path)
        {
            &inv_path_color(\$G,\@alternate_path,$alpha,$beta);
        }
        else
        {
            return;
        }
    }
}

sub path
{
    my $G = shift;
    my $vertex = shift;
    my $alpha = shift;
    my $beta = shift;

    my $temp_color = $beta;
    my @path_list;
    my $next_vertex;

    $next_vertex = &check_color(\$G,$vertex,$temp_color);
    if($next_vertex eq 0)
    {return @path_list;}

    while($next_vertex ne 0)
    {
        push (@path_list,$next_vertex);
        push (@path_list,$next_vertex);

        if($temp_color == $alpha)
        {$temp_color = $beta;}
        else
        {$temp_color = $alpha;}

        $next_vertex = &check_color(\$G,$next_vertex,$temp_color);


        # have to break the path when it is a loop because the program enter in
        # an infinite loop.

        # if the next_vertex is equal to vertex
        if ($next_vertex eq $vertex)
        {last;}

        my $flag = 0;
        foreach (@path_list)
        {
            if ($_ eq $next_vertex)
            {
                $flag = 1;
            }
        }
        last if $flag == 1;
        #test
        #print "$vertex:$next_vertex ";
        #print ".";
    }
    my $extra = pop @path_list;
    unshift (@path_list,$vertex);
    return @path_list;
}

sub inv_path_color
{
    my $G = shift;
    my @path = @{(shift(@_))};
    my $alpha = shift;
    my $beta = shift;

    #test
    #print "@path\n";

    for(1..(scalar(@path)/2))
    {
        my $start = shift @path;
        my $vertex = shift @path;

        if($$$G->get_attribute('color',$start,$vertex) == $beta)
        {
            $$$G->set_attribute('color',$start,$vertex,$alpha);
            $$$G->set_attribute('color',$vertex,$start,$alpha);
        }
        elsif($$$G->get_attribute('color',$start,$vertex) == $alpha)
        {
            $$$G->set_attribute('color',$start,$vertex,$beta);
            $$$G->set_attribute('color',$vertex,$start,$beta);
        }
    }
}

sub check_color
{
    my $G = shift;
    my $vertex = shift;
    my $color = shift;

    my $edge_color;

    my @vertex_adj = $$$$G->neighbors($vertex);
    foreach my $v_end (@vertex_adj)
    {
        $edge_color = $$$$G->get_attribute('color',$vertex,$v_end);
        if($edge_color == $color)
        {
            return $v_end;
        }
    }
    return 0;
}
######################################################################
#Function Name: read_file
#Inputs:
#   name of the file you want to open
#Output:
#   the array that contain all the line of the file
######################################################################
sub read_file
{
    my ($filename) = @_;
    open(FILE,"$filename")||die("could_not_open_the_file");
    my @file = <FILE>;
    close(FILE);
    chomp(@file);
    return @file;
}
```

B.7 Completion of G_{IRM} Edge Coloring

```
#!/usr/local/bin/perl
#########################################################################
=head1 NAME

completion.pl

=head1 AUTHOR (DATE)

Christian Giasson (August 21st, 2003)

=head1 SYNOPSIS

completion

=head1 DESCRIPTION

This program complete the coloring made by color_IRM_v2.pl and
color_IRM_part2_v2.pl

If there is one:
The coloring part of the program is based on the Vizing algorithm for
edge-color bipartite graphs.

=over 4

=item INPUT

All the edges in IRM (a complete one)

coloring 1 to 16 : result from color_IRM_v2.pl (result_1_16)
coloring 17 to 26 : result from color_IRM_part2_v2.pl (result_17_26)

feedback path for the color 1 to 16 : feedback_path_v2
feedback for color 17 to 22 : feedback_17_22

=item OUTPUT
```

```
The output all the completed TESTING CONFIGURATION all in the same file with
all the full paths in the tile.

=cut
################################################################################
use warnings;
use strict;
use Graph;
use Graph::Directed;
use Graph::Undirected;
use Getopt::Std;


################################################################################
#Get the inputs of the program
################################################################################

getopts('ha:b:c:d:e:f:');
our($opt_h,$opt_a,$opt_b,$opt_c,$opt_d,$opt_e,$opt_f);

if ($opt_h)
{
    print"\n\tOption_of_this_program:\n
_____________\t-h:_\t\t\thelp
_____________\t-a_[file]:_edges_of_IRM\t\t
_____________\t-b_[file]:_colored_IRM_1_to_16\t\t
_____________\t-c_[file]:_colored_IRM_17_to_26\t\t
_____________\t-d_[file]:_feedback_for_1_to_16\t\t
_____________\t-e_[file]:_feedback_for_17_to_22\t\t
_____________\t-f_[file]:_all_the_tile\t\t\n\n";
                exit;
}


my @IRM_edge_line = &read_file($opt_a) if (($opt_a)&&(!$opt_h));
my @colored_1_16_line = &read_file($opt_b) if (($opt_b)&&(!$opt_h));
my @colored_17_26_line = &read_file($opt_c) if (($opt_c)&&(!$opt_h));
my @feedback_1_16_line = &read_file($opt_d) if (($opt_d)&&(!$opt_h));
my @feedback_17_22_line = &read_file($opt_e) if (($opt_e)&&(!$opt_h));
my @all_tile = &read_file($opt_f) if (($opt_f)&&(!$opt_h));


####################################
#    Build the IRM graph edges array
####################################
my @IRM_edges;
foreach (@IRM_edge_line)
{
    if (/(\w+) -> (\w+)/)
    {
        push @IRM_edges, ($1, $2);
    }
}

####################################
#    Build a hash with the color key and a list of connections that include all
#    the one of the feedback and the coloring itself for the color 1 to 26.
####################################
my %coloring;
#put colored 1 to 16 in
foreach(@colored_1_16_line)
{
    if(/(COLOR\d+)-\d+ : (.*)/)
    {
        push @{$coloring{$1}}, "$2";
    }
}
#put colored 17 to 26 in
foreach(@colored_17_26_line)
{
    if(/(COLOR\d+) : (.*)/)
    {
        push @{$coloring{$1}}, "$2";
    }
}
#put feedback 1 to 16 in
foreach(@feedback_1_16_line)
{
    if(/(COLOR\d+): .* (\w+) (\w+)$/)
    {
        push @{$coloring{$1}}, "$2_->_$3";
    }
}
#put feedback 17 to 22 in
foreach(@feedback_17_22_line)
{
    if(/(COLOR\d+):(.*)/)
    {
        push @{$coloring{$1}}, "$2";
    }
}

#Build a feedback GRM ressources
my %feedback_GRM_res;
foreach(@feedback_1_16_line)
{
    if(/(COLOR\d+): \w+ \w+ ([NSEW])(\d+) \w+ \w+ \w+$/)
    {
        push @{$feedback_GRM_res{$1}}, "$2"."_P$3";
    }
}

#build feedback paths
my %fb_rest;
foreach(@feedback_1_16_line)
{
    if(/(COLOR\d+): (\w+) (\w+) (\w+) (\w+) (\w+)$/)
    {
        push @{$fb_rest{$1}}, "$2_->_$3_->_$4_->_$5";
    }
    elsif(/(COLOR\d+): (\w+) (\w+) (\w+) (\w+) (\w+) (\w+)$/)
    {
        push @{$fb_rest{$1}}, "$2_->_$3_->_$4_==_$5_->_$6";
    }
}

#outputs of the IRM
my @IRM_output_list = (
    'S0_F_B1',
    'S0_F_B2',
    'S0_F_B3',
    'S0_F_B4',
    'S0_G_B1',
    'S0_G_B2',
    'S0_G_B3',
    'S0_G_B4',
    'S0_G_B4',
    'S0_BX_B',
    'S0_BY_B',
    'S0_CLK_B',
    'S0_CE_B',
    'S0_SR_B',
    'S1_F_B1',
    'S1_F_B2',
    'S1_F_B3',
    'S1_F_B4',
    'S1_G_B1',
    'S1_G_B2',
    'S1_G_B3',
    'S1_G_B4',
    'S1_G_B4',
    'S1_BX_B',
    'S1_BY_B',
    'S1_CLK_B',
    'S1_CE_B',
    'S1_SR_B',
    'T_IN0',
    'T_IN1',
    'TS_B0',
    'TS_B1'
);

#build the OUT lines adj list
my %OUT_adj;
foreach my $line (@all_tile)
{
    if ($line =~ /(OUT\d) -> (\w+)/)
    {
        push @{$OUT_adj{$1}},$2;
    }
}

#build the ORM adj list
my %ORM_adj;
foreach my $line (@all_tile)
{
    if ($line =~ /([S]\w+) -> (OUT\d)/)
    {
        push @{$ORM_adj{$1}},$2;
    }
}

#########################################
#   Get the feedback paths 1 to 16
#########################################

#my %fb_1_16;
#my $cn = 0;
```

```
#foreach (@feedback_1_16_line)
#{
#     if ($_ =~ /(COLOR\d+): .* (\w+) (\w+)$/)
#     {
#    push @{$fb_1_16{$1}}, "$2->$3";
     #Test
     #$cn++;
     #print "$1 $2->$3\n";
#     }
#}
##########################################
#    Get the opposit short line of the double path for color 1 to 16
##########################################

#my %short_opposit;
#foreach (@feedback_color_l)
#{
#     if ($_ =~ /(COLOR\d+): \w+ OUT\d (\w+) (\w+) \w+ \w+$/)
#     {
#    $short_opposit{$1}{$3} = "$2";
#     }
#}

##########################################
#    build the graph G with groupment of the orthogonal lines
##########################################

#build the list of grouped vertices
my @group_vertex;
for($a = 0; $a <=23; $a++)
{
     push @group_vertex,"N_P$a:S_P$a";
     push @group_vertex,"E_P$a:W_P$a";
}

#build the grouped IRM for each color
my %group_edge;
foreach my $color (sort keys(%coloring))
{
     foreach (@{$coloring{$color}})
     {
          if (/[NS]_P(\d+) -> (\w+)/)
          {
               push @{$group_edge{$color}},"N_P$1:S_P$1";
               #push @{group_edge{$color}},"$2";
          }
          elsif(/[EW]_P(\d+) -> (\w+)/)
          {
               push @{$group_edge{$color}},"E_P$1:W_P$1";
               #push @group_edge,"$2";
          }
     }
     if(exists($feedback_GRM_res{$color}))
     {
          foreach (@{$coloring{$color}})
          {
               if(/[NS]_P(\d+)/)
               {
                    push @{$group_edge{$color}},"N_P$1:S_P$1";
               }
               elsif(/[EW]_P(\d+)/)
               {
                    push @{$group_edge{$color}},"E_P$1:W_P$1";
               }
          }
     }
}

#build the graph grouped
#my $G_group = Graph::Undirected->new;
#$G_group->add_edges(@group_edge);

################################################
################################################
#    Main program
################################################
################################################

##########################################
#    build the graph G
##########################################

my $G = Graph::Undirected->new;
$G->add_edges(@IRM_edges);
```

```
##################################################
#    Complete the color 1 to 16
##################################################

my %test_configuration = %coloring;
foreach my $color (sort keys(%test_configuration))
{
    ##################################
    #    only color 1 to 16
    ##################################
    if($color =~ /COLOR(\d+)/)
    {
        next if $1 > 16;
    }

    ##################################
    #    Find missing inputs
    ##################################
    #copy the @IRM_output
    my @output_rest = @IRM_output_list;
    for (1..scalar(@output_rest))
    {
        my $output = shift @output_rest;

        my $found = 0;
        foreach my $connection (@{$test_configuration{$color}})
        {
            if ($connection =~ / -> (\w+)/)
            {
                if ($1 eq $output)
                {
                    $found = 1;
                    last;
                }
            }
        }
        push @output_rest,$output if $found == 0;
    }
    #print "$color : @output_rest\n";

    ##################################
    #    Put CLK signals
    ##################################

    # the easy way by the 4 high fannout lines
    # GCLK0 for S0 and GCLK0 for S1 (Fannout but less signals to route)

    # check if the S0 CLOCK signal is missing
    foreach (@output_rest)
    {
        if($_ eq 'S0_CLK_B')
        {
            push @{$test_configuration{$color}}, 'GCLK0_->_S0_CLK_B';
            #print"S0_CLK_B is missing in $color\n";
            last;
        }
    }
    # check if the S1 CLOCK signal is missing
    foreach (@output_rest)
    {
        if($_ eq 'S1_CLK_B')
        {
            push @{$test_configuration{$color}}, 'GCLK0_->_S1_CLK_B';
            #print"S1_CLK_B is missing in $color\n";
            last;
        }
    }

    ##################################
    #    Put CE signals
    ##################################
    foreach (@output_rest)
    {
        if($_ eq 'S0_CE_B')
        {
            push @{$test_configuration{$color}}, 'V6N3_->_S0_CE_B';
            last;
        }
    }

    foreach (@output_rest)
    {
        if($_ eq 'S1_CE_B')
        {
            push @{$test_configuration{$color}}, 'V6N3_->_S1_CE_B';
            last;
        }
    }
```

```
####################################
#   Put SR signals
####################################
foreach (@output_rest)
{
    if($_ eq 'S0_SR_B')
    {
        push @{$test_configuration{$color}}, 'V6N1_->_S0_SR_B';
        last;
    }
}

foreach (@output_rest)
{
    if($_ eq 'S1_SR_B')
    {
        push @{$test_configuration{$color}}, 'V6N1_->_S1_SR_B';
        last;
    }
}

####################################
#   Put tri-state control signals
####################################
foreach (@output_rest)
{
    if($_ eq 'TS_B0')
    {
        push @{$test_configuration{$color}}, 'V6N0_->_TS_B0';
        last;
    }
}

foreach (@output_rest)
{
    if($_ eq 'TS_B1')
    {
        push @{$test_configuration{$color}}, 'V6N0_->_TS_B1';
        last;
    }
}

####################################
#   Put the BX and BY in the config
####################################
foreach (@output_rest)
{
    if($_ eq 'S0_BX_B')
    {
        my @neigh = $G->neighbors('S0_BX_B');

        #find if there is space to route the a signal to S0_BX_B
        my $good_node = 0;
        foreach my $adj (@neigh)
        {
            my $find = 1;
            foreach my $group_e (@{$group_edge{$color}})
            {
                if($group_e =~ /(\w+):(\w+)/)
                {
                    $find = 0 if $1 eq $adj;
                    $find = 0 if $2 eq $adj;
                    last if $find == 0;
                }
            }
            if ($find == 1)
            {
                $good_node = $adj;
                last;
            }
        }

        if ($good_node eq 0)
        {
            print "Error_:_No_connection_found_for_S0_BX_B\n";
        }
        else
        {
            #print "GOOD : found a S0_BX_B with $good_node\n";
            push @{$test_configuration{$color}}, "$good_node_->_S0_BX_B";
        }

        last;
    }
}
foreach (@output_rest)
{
    if($_ eq 'S0_BY_B')
    {
            my @neigh = $G->neighbors('S0_BY_B');

            #find if there is space to route the a signal to S0_BY_B
            my $good_node = 0;
            foreach my $adj (@neigh)
            {
                my $find = 1;
                foreach my $group_e (@{$group_edge{$color}})
                {
                    if($group_e =~ /(\w+):(\w+)/)
                    {
                        $find = 0 if $1 eq $adj;
                        $find = 0 if $2 eq $adj;
                        last if $find == 0;
                    }
                }
                if ($find == 1)
                {
                    $good_node = $adj;
                    last;
                }
            }

            if ($good_node eq 0)
            {
                print "Error_:_No_connection_found_for_S0_BY_B\n";
            }
            else
            {
                #print "GOOD : found a S0_BY_B with $good_node\n";
                push @{$test_configuration{$color}}, "$good_node_->_S0_BY_B";
            }

            last;
        }
    }
    ##################################
    #   Check which missing connection are left
    ##################################
    @output_rest = @IRM_output_list;
    for (1..scalar(@output_rest))
    {
        my $output = shift @output_rest;

        my $found = 0;
        foreach my $connection (@{$test_configuration{$color}})
        {
            if ($connection =~ / -> (\w+)/)
            {
                if ($1 eq $output)
                {
                    $found = 1;
                    last;
                }
            }
        }
        push @output_rest,$output if $found == 0;
    }
    #print "$color : @output_rest\n";
}

##############################################
#   Complete the color 17 to 22
##############################################

#routing added
my %routing_added;
my %routing_imp;

foreach my $color (sort keys(%test_configuration))
{
    ##################################
    #   only color 17 to 22
    ##################################
    if($color =~ /COLOR(\d+)/)
    {
        next if ($1 < 17)||($1 > 22);
    }

    ##################################
    #   Find missing inputs
    ##################################
    #copy the @IRM_output
    my @output_rest = @IRM_output_list;
    for (1..scalar(@output_rest))
    {
        my $output = shift @output_rest;
        my $found = 0;
        foreach my $connection (@{$test_configuration{$color}})
        {
            if ($connection =~ / -> (\w+)/)
            {
                if ($1 eq $output)
                {
                    $found = 1;
                    last;
                }
            }
        }
        push @output_rest,$output if $found == 0;
}
#print "$color : @output_rest\n";


##################################
#   Route the tri-state buffer signals
##################################

#if one of the T_IN[01] is missing
foreach (@output_rest)
{
        if (/(T_IN0)/)
        {
            push @{$routing_added{$color}},"S0_XQ_->_OUT2_->_$1";
        }
        elsif(/(T_IN1)/)
        {
            push @{$routing_added{$color}},"S1_XQ_->_OUT6_->_$1";
        }
}

#get the tri-state connection
foreach my $connection (@{$test_configuration{$color}})
{
        if($connection =~ /(\w+) -> (T_IN([01]))/)
        {
            my $node = $1;
            my $CLB_out = "S$3"."_XQ";
            my $T = "$2";

            #route the OUT direct feedback
            if($node =~ /OUT/)
            {
                my $OUT = $node;
                push @{$routing_added{$color}},"$CLB_out_->_$OUT";
            }

            #route the OUT -> GRM feedback
            if($node !~ /OUT/)
            {
                my $short;
                my $direction;
                if ($node =~ /([NSEW])_P(\d+)/)
                {
                    $short = "$1$2";
                    $direction = $1;
                }

                #check if it can be route directly from the OUT lines
                my $get = 0;
                foreach my $OUT_line (sort keys(%OUT_adj))
                {
                    foreach my $s (@{$OUT_adj{$OUT_line}})
                    {
                        if($s eq $short)
                        {
                            #print "single : $OUT_line -> $s\n";
                            push @{$routing_added{$color}},"$CLB_out_->_$OUT_line_->_$s_->_$node";
                            $get = 1;
                            last;
                        }
                    }
                }
                next if $get == 1;

                #check if it can be route with a double feedback
                #from the OUT lines

                my @line = grep /== $short\)/, @all_tile;
                my @adj_short;
                foreach my $l (@line)
                {

                    if ($l =~ /(([NSEW])\d+) == /)
        {
            if ($direction eq 'N')
            {
                push @adj_short,$1 if ($2 ne 'S');
            }
            elsif ($direction eq 'S')
            {
                push @adj_short,$1 if ($2 ne 'N');
            }
            elsif ($direction eq 'E')
            {
                push @adj_short,$1 if ($2 ne 'W');
            }
            elsif ($direction eq 'W')
            {
                push @adj_short,$1 if ($2 ne 'E');
            }
        }
    }

    my $find = 0;
    my $direct_conn;
    foreach my $last_S (@adj_short)
    {
        foreach my $OUT_line (sort keys(%OUT_adj))
        {
            foreach my $s (@{$OUT_adj{$OUT_line}})
            {
                #have to verify if there is no conflics
                my $ok = 1;
                foreach (@{$group_edge{$color}})
                {
                    if(/([NSEW])_P(\w+):([NSEW])_P(\w+)/)
                    {
                        $ok = 0 if ($last_S eq "$1$2");
                        $ok = 0 if ($last_S eq "$3$4");
                    }
                }
                next if $ok == 0;

                if ($last_S eq $s)
                {
                    #print "$OUT_line -> $s\n";
                    $direct_conn = "$CLB_out_->_$OUT_line_->_$s";
                    $find = 1;
                    last;
                }
            }
            last if $find == 1;
        }
        last if $find == 1;
    }
    if ($find == 1)
    {
        push @{$routing_added{$color}},"$direct_conn_==_$short_->_$node";
        #print"double : $direct_conn == $short\n";
    }
    else
    {
        #print "ERROR\n";
    }

    next if $find == 1;


    #check for the triple path

    my %adj_second_short;
    foreach my $sec (@adj_short)
    {
        my $local_sec = $1 if ($sec =~ /([NSEW])\d+/);

        my @line2 = grep /== $sec\)/, @all_tile;
        #print "@line2\n";
        foreach my $l (@line2)
        {
            if ($l =~ /(([NSEW])\d+) == /)
            {
                if ($local_sec eq 'N')
                {
                    push @{$adj_second_short{$sec}},$1 if ($2 ne 'S');
                }
                elsif ($local_sec eq 'S')
                {
                    push @{$adj_second_short{$sec}},$1 if ($2 ne 'N');
                }
                elsif ($local_sec eq 'E')
                {
                    push @{$adj_second_short{$sec}},$1 if ($2 ne 'W');
                }
                elsif ($local_sec eq 'W')
                {
                    push @{$adj_second_short{$sec}},$1 if ($2 ne 'E');
                }
            }
        }
    }
    #foreach my $sec (keys(%adj_second_short))
    #{
    #print "sec $sec : @{$adj_second_short{$sec}}\n";
    #}

    my $find2 = 0;
    my $direct_conn2;
    my $second_short;
    foreach my $sec (keys (%adj_second_short))
    {
        #have to verify if there is no conflics
        my $ok = 1;
        foreach (@{$group_edge{$color}})
        {
            if(/([NSEW])_P(\w+):([NSEW])_P(\w+)/)
            {
                $ok = 0 if ($sec eq "$1$2");
                $ok = 0 if ($sec eq "$3$4");
            }
        }
        next if $ok == 0;

        foreach my $last_S (@{$adj_second_short{$sec}})
        {
            foreach my $OUT_line (sort keys(%OUT_adj))
            {
                foreach my $s (@{$OUT_adj{$OUT_line}})
                {
                    if ($last_S eq $s)
                    {
                        #have to verify if there is no conflics
                        my $ok2 = 1;
                        foreach (@{$group_edge{$color}})
                        {
                            if(/([NSEW])_P(\w+):([NSEW])_P(\w+)/)
                            {
                                $ok = 0 if ($last_S eq "$1$2");
                                $ok = 0 if ($last_S eq "$3$4");
                            }
                        }
                        next if $ok2 == 0;

                        #print "$OUT_line -> $s\n";
                        $direct_conn2 = "$CLB_out_->_$OUT_line_->_$s";
                        $second_short = $sec;
                        $find2 = 1;
                        last;
                    }
                }
                last if $find2 == 1;
            }
            last if $find2 == 1;
        }
        last if $find2 == 1;
    }
    if ($find2 == 1)
    {
        push @{$routing_added{$color}},"$direct_conn2_==_$second_short_==_$short_->_$node";
        #print "triple : $direct_conn2 == $second_short == $short\n"
    }
    else
    {
        push @{$routing_imp{$color}},"$connection";
        #print "impossible: $connection\n";
    }
    #print "$connection : @adj_short\n";
        }
    }
}

#check if the T_IN are missing
foreach (@output_rest)
{
    if($_ eq 'T_IN0')
```

```
        {
            #push @{$test_configuration{$color}}, 'V6N0 -> TS_B1';
            last;
        }
    }
    foreach (@output_rest)
    {
        if($_ eq 'T_IN1')
        {
            #push @{$test_configuration{$color}}, 'V6N0 -> TS_B1';
            last;
        }
    }

    ##################################
    #   Route Feedback for BX and BY signals to G[1..4] or F[1..4]
    ##################################
    my @feedback;
    foreach my $connection (@{$test_configuration{$color}})
    {
        if($connection =~ /(\w+) -> ((S[01])_B([XY])_B)/)
        {
            my $node = $1;
            my $B_in = $2;
            my $B_out = "$3"."_$4"."B";
            my $CLB = $3;
            # See if a feeback from XB or YB can be route as a feeback to one
            # of the ressources already there for F[1..4] and G[1..4]

            foreach (@{$test_configuration{$color}})
            {
                if(/(\w+) -> (S[01]_[FG]_B\d)/)
                {
                    my $node_FG = $1;
                    my $FG = $2;

                    #find the short line
                    my $last_short;
                    if ($node_FG =~ /([NSEW])_P(\d+)/)
                    {
                        $last_short = "$1$2";
                    }
                    else
                    {
                        next;
                    }

                    #print "last short: $last_short\n";

                    #check if it can be route directly from the OUT lines
                    my $get = 0;
                    foreach my $OUT_line (sort keys(%OUT_adj))
                    {
                        #check if the OUT line have been use before
                        my $already = 0;
                        foreach (@{$routing_added{$color}})
                        {
                            if (/(OUT\d)/)
                            {
                                #print "$1\n";
                                $already = 1 if $1 eq $OUT_line;
                            }
                        }
                        next if $already == 1;

                        #check if the B_out can be connect to that OUT line
                        #my $cannot_be = 1;
                        #foreach my $CLB_out (sort keys(%ORM_adj))
                        #{
                        #    if ($CLB_out eq $B_out)
                        #    {
                        #    foreach (@{$ORM_adj{$CLB_out}})
                        #    {
                        #        $cannot_be = 0 if $_ eq $OUT_line;
                        #        last;
                        #    }
                        #    }
                        #}
                        #next if $cannot_be == 1;

                        foreach my $s (@{$OUT_adj{$OUT_line}})
                        {
                            if($s eq $last_short)
                            {
                                #print "single : $OUT_line -> $s -> $node_FG -> $FG\n";
                                push @feedback,"$OUT_line -> $s -> $node_FG -> $FG";
                                $get = 1;
                                last;
                            }
                    }
                }
            }
        }

        #check if the feedback can be route in ORM and pick one
        #my $cannot_be = 1;
        foreach my $CLB_out (sort keys(%ORM_adj))
        {
            if ($CLB_out eq $B_out)
            {
                for (1..scalar(@feedback))
                {
                    my $fb = shift @feedback;
                    #check if it's the same CLB
                    if ($fb =~ /(S[10])_[FG]_B/)
                    {
                        if ($1 ne $CLB)
                        {
                            push @feedback,$fb;
                            next;
                        }
                    }

                    my $fb_OUT = $1 if ($fb =~ /(OUT\d)/);

                    #check if the B_out can be connect
                    my $ok = 0;
                    foreach my $adj_OUT (@{$ORM_adj{$CLB_out}})
                    {
                        $ok = 1 if $fb_OUT eq $adj_OUT;
                    }
                    if($ok == 1)
                    {
                        push @{$routing_added{$color}},"$B_out_->_"."$fb";
                        last;
                    }
                    else
                    {
                        push @feedback,$fb;
                    }
                }
            }
        }
    }
    #print "$color\n";
    #if (exists($routing_added{$color}))
    #{
    #    print "@{$routing_added{$color}}\n";
    #    }
}
##################################################
#   Complete the color 23 to 26
##################################################

foreach my $color (sort keys(%test_configuration))
{
    ##################################
    #   only color 23 to 26
    ##################################
    if($color =~ /COLOR(\d+)/)
    {
        next if ($1 < 23);
    }

    ##################################
    #   Find missing inputs
    ##################################
    #copy the @IRM_output
    my @output_rest = @IRM_output_list;
    for (1..scalar(@output_rest))
    {
        my $output = shift @output_rest;

        my $found = 0;
        foreach my $connection (@{$test_configuration{$color}})
        {
            if ($connection =~ / -> (\w+)/)
            {
                if ($1 eq $output)
                {
                    $found = 1;
                    last;
                }
            }
        }
        push @output_rest ,$output if $found == 0;
    }
    #print "$color : @output_rest\n";


    ##################################
    #   supply the control signals (CE, SR, CLK, TS)
    ##################################

    # check if the S0 CLOCK signal is missing
    foreach (@output_rest)
    {
        if($_ eq 'S0_CLK_B')
        {
        push @{$test_configuration{$color}}, 'GCLK0_->_S0_CLK_B';
        #print"S0_CLK_B is missing in $color\n";
        last;
        }
    }
    # check if the S1 CLOCK signal is missing
    foreach (@output_rest)
    {
        if($_ eq 'S1_CLK_B')
        {
        push @{$test_configuration{$color}}, 'GCLK0_->_S1_CLK_B';
        #print"S1_CLK_B is missing in $color\n";
        last;
        }
    }

    ##################################
    #   Put CE signals
    ##################################
    foreach (@output_rest)
    {
        if($_ eq 'S0_CE_B')
            {
        push @{$test_configuration{$color}}, 'V6N3_->_S0_CE_B';
        last;
            }
    }

    foreach (@output_rest)
    {
            if($_ eq 'S1_CE_B')
            {
            push @{$test_configuration{$color}}, 'V6N3_->_S1_CE_B';
            last;
            }
    }

    ##################################
    #   Put SR signals
    ##################################
    foreach (@output_rest)
    {
            if($_ eq 'S0_SR_B')
            {
            push @{$test_configuration{$color}}, 'V6N1_->_S0_SR_B';
            last;
            }
    }

    foreach (@output_rest)
    {
            if($_ eq 'S1_SR_B')
            {
            push @{$test_configuration{$color}}, 'V6N1_->_S1_SR_B';
            last;
            }
    }

    ##################################
    #   Put tri-state control signals
    ##################################
    foreach (@output_rest)
    {
            if($_ eq 'TS_B0')
            {
            push @{$test_configuration{$color}}, 'V6N0_->_TS_B0';
            last;
            }
    }

    foreach (@output_rest)
    {
            if($_ eq 'TS_B1')
```

```
        {
            push @{$test_configuration{$color}}, 'V6N0_->_TS_B1';
            last;
        }
    }

    ###################################
    #   route a feedback for the lach clb output that going in the tri-state
    #   buffers
    ###################################

    my %OUT_1 = (
    COLOR23 => 'OUT5',
    COLOR24 => 'OUT3',
    COLOR25 => 'OUT5',
    COLOR26 => 'OUT4',
    );

    #my @OUT_av = ('OUT2', 'OUT3', 'OUT4', 'OUT5');

    my $T_IN0_fb;
    my $T_IN1_fb;

    #Route T_IN0 with OUT2 for all
    $T_IN0_fb = 'OUT2_->_T_IN0';


    #Route T_IN1 with OUT_1 for all
    #foreach my $local_OUT (@OUT_av)
    #{
    my $local_OUT = $OUT_1{$color};
    #print "$local_OUT : @{$OUT_adj{$local_OUT}}\n";
    foreach my $local_short (@{$OUT_adj{$local_OUT}})
    {
        if($local_short =~ /^([NSEW])(\d+)/)
        {
            my $node = "$1"."_P$2\n";
            chomp $node;

            foreach (@all_tile)
            {
                #if (/(([NSEW]_P\d+) -> (T_IN[01]))/)
                if (/(([NSEW]_P\d+) -> (T_IN1))/)
                {
                    #print "$2 -- $node\n";
                    my $IRM_c = $1;
                    my $local_node = $2;
                    my $T = $3;

                    if ($node eq $local_node)
                    {
                        #print "*\n";
                        #print "$local_OUT -> $local_short -> $IRM_c\n";

                        #check if there is no routing problems
                        foreach (@{$group_edge{$color}})
                        {
                            if(/(\w+):(\w+)/)
                            {
                                if (($1 ne $node)&&($2 ne $node))
                                {
                                    my $temp = "$local_OUT_->_$local_short_->_$IRM_c";
                                    #print "$color : $temp\n";
                                    #$T_IN0_fb = $temp if $T eq 'T_IN0';
                                    $T_IN1_fb = $temp if $T eq 'T_IN1';
                                }
                            }
                        }
                    }
                }
            }
        }
    }
    #}
#     push @{$routing_added{$color}},"S0_XQ -> "."$T_IN0_fb";
#     push @{$routing_added{$color}},"S1_XQ -> "."$T_IN1_fb";

    #supress the OUT lines used
#     for(1..scalar(@OUT_av))
#     {
#   my $temp = shift @OUT_av;
#
#   if ($T_IN0_fb =~ /^(OUT\d) ->/)
#   {
#       next if $1 eq $temp;
#   }
#
```

```
#     if ($T_IN1_fb =~ /^(OUT\d) ->/)
#     {
#         next if $1 eq $temp;
#     }
#     push @OUT_av, $temp;
#      }
      #print "@OUT_av\n";

      ###################################
      #    Need to route 1 feedback by CLB for the logic
      #    The feedback should be connect to an existing connnection in the IRM
      #    It should be connect to S[10]_F because S[01]_XQ is the output latched
      ###################################

      my $fb_CLB0;
      my $fb_CLB1;

      my %OUT_2 = (
      COLOR23 => ["OUT4","OUT3"],
      COLOR24 => ["OUT4","OUT5"],
      COLOR25 => ["OUT4","OUT3"],
      COLOR26 => ["OUT3","OUT5"],
      );

      #foreach my $local_conn (@{$test_configuration{$color}})
      #{

      #route OUT2 to S[01]_F CLB input
      my $T = '0';
      foreach my $OUT_line (@{$OUT_2{$color}})
      {
          my $find = 0;

          if ($T eq 'T_IN0')
          {
              $T = 'T_IN1';
          }
          elsif ($T eq 'T_IN1')
          {
              $T = 'T_IN0';
          }

          foreach my $OUT_adj (@{$OUT_adj{$OUT_line}})
          {
              next if $OUT_adj !~ /^[NSEW]\d+$/;
              #print "$OUT4_adj\n";
              foreach my $local_conn (@{$test_configuration{$color}})
              {
                  if ($local_conn =~ /(([NSEW])_P(\d+)) -> (S([01])_[FG]_B\d)/)
                  {
                      next if (($5 eq '0')&&($T eq 'T_IN1'));
                      next if (($5 eq '1')&&($T eq 'T_IN0'));

                      #        print "$5 -- $T\n";

                      #print "$local_conn\n";
                      my $T_num = "$5";
                      my $CLB_in = "$4";
                      my $short = "$2$3";
                      my $node = $1;
                      my $oppo;
                      $oppo = "S$1" if $node =~ /N_P(\d+)/;
                      $oppo = "N$1" if $node =~ /S_P(\d+)/;
                      $oppo = "W$1" if $node =~ /E_P(\d+)/;
                      $oppo = "E$1" if $node =~ /W_P(\d+)/;

                      #simple connection
                      if($short eq $OUT_adj)
                      {
                          #print "$color : $OUT_line -> $short -> $node -> $CLB_in\n";
                          my $temp = "$OUT_line_->_$short_->_$node_->_$CLB_in";
                          $find = 1;
                          $T = "T_IN$T_num" if ($T eq '0');
                          $fb_CLB0 = $temp if ($T eq 'T_IN0');
                          $fb_CLB1 = $temp if ($T eq 'T_IN1');
                          last;
                      }
                      #double connection
                      foreach (@all_tile)
                      {
                          if(/(\w+) == (\w+)/)
                          {
                              if(($1 eq $short)&&($2 ne $oppo))
                              {
                                  #print "$node : $1 == $2\n";
                                  if($2 eq $OUT_adj)
                                  {
                                        #print "$color : $OUT_line -> $2 == $1 -> $node -> $CLB_in\n";
                                        my $temp = "$OUT_line␣->␣$2␣==␣$1␣->␣$node␣->␣$CLB_in";
                                        $find = 1;
                                        $T = "T_IN$T_num" if ($T eq '0');
                                        $fb_CLB0 = $temp if ($T eq 'T_IN0');
                                        $fb_CLB1 = $temp if ($T eq 'T_IN1');
                                        last;
                                    }
                                }
                            }
                        }
                        last if $find == 1;
                    }
                }
                last if $find == 1;
            }
            #print "$T\n";
        }

        #print "$color : T0 : $T_IN0_fb\n";
        #print "$color : T1 : $T_IN1_fb\n";
        #print "$color : logic0 : $fb_CLB0\n";
        #print "$color : logic1 : $fb_CLB1\n";

        if (($fb_CLB0 =~ /-> S0_F_/)&&($fb_CLB1 =~ /-> S1_F_/))
        {
            push @{$routing_added{$color}},"S0_XQ␣->␣"."$T_IN0_fb";
            push @{$routing_added{$color}},"S1_XQ␣->␣"."$T_IN1_fb";
            push @{$routing_added{$color}},"S0_Y␣->␣"."$fb_CLB0";
            push @{$routing_added{$color}},"S1_Y␣->␣"."$fb_CLB1";
        }
        elsif (($fb_CLB0 =~ /-> S0_F_/)&&($fb_CLB1 =~ /-> S1_G_/))
        {
            push @{$routing_added{$color}},"S0_XQ␣->␣"."$T_IN0_fb";
            push @{$routing_added{$color}},"S1_YQ␣->␣"."$T_IN1_fb";
            push @{$routing_added{$color}},"S0_Y␣->␣"."$fb_CLB0";
            push @{$routing_added{$color}},"S1_X␣->␣"."$fb_CLB1";
        }
        elsif (($fb_CLB0 =~ /-> S0_G_/)&&($fb_CLB1 =~ /-> S1_F_/))
        {
            push @{$routing_added{$color}},"S0_YQ␣->␣"."$T_IN0_fb";
            push @{$routing_added{$color}},"S1_XQ␣->␣"."$T_IN1_fb";
            push @{$routing_added{$color}},"S0_X␣->␣"."$fb_CLB0";
            push @{$routing_added{$color}},"S1_Y␣->␣"."$fb_CLB1";
        }
        elsif (($fb_CLB0 =~ /-> S0_G_/)&&($fb_CLB1 =~ /-> S1_G_/))
        {
            push @{$routing_added{$color}},"S0_YQ␣->␣"."$T_IN0_fb";
            push @{$routing_added{$color}},"S1_YQ␣->␣"."$T_IN1_fb";
            push @{$routing_added{$color}},"S0_X␣->␣"."$fb_CLB0";
            push @{$routing_added{$color}},"S1_X␣->␣"."$fb_CLB1";
        }

        ##################################
        #   Route signal to OUT0, OUT1, OUT6, OUT7 to test the West and East tile
        #   connections in the IRM
        ##################################

        push @{$routing_added{$color}},"S0_XB␣->␣OUT0";
        push @{$routing_added{$color}},"S0_YB␣->␣OUT1";
        push @{$routing_added{$color}},"S1_XB␣->␣OUT6";
        push @{$routing_added{$color}},"S1_YB␣->␣OUT7";

        ##################################
        #   supply S[01]_B[XY]_B
        ##################################

        my @route_signal = ('S0_BX_B', 'S0_BY_B', 'S1_BX_B', 'S1_BY_B');

        foreach my $B (@route_signal)
        {
            my @B_adj;

            #find the adj_list
            foreach (@all_tile)
            {
                if (/(\w+) -> (S[01]_B[XY]_B)/)
                {
                    if ($2 eq $B)
                    {
                        push @B_adj, $1;
                    }
                }
            }
            #print "$B : @B_adj\n";
            #look for conflics in group_edge
        for (1..scalar(@B_adj))
        {
            my $adj = shift @B_adj;
            my $find = 1;
            foreach (@{$group_edge{$color}})
            {
                if (/(\w+):(\w+)/)
                {
                    if (($1 eq $adj)||($2 eq $adj))
                    {
                        $find = 0;
                        last;
                    }
                }
            }
            push @B_adj,$adj if $find == 1;
        }

        #print "$B : @B_adj\n";
        #look for conflics in the routing_added

        for (1..scalar(@B_adj))
        {
            my $adj = shift @B_adj;
            my $find = 1;
            foreach (@{$routing_added{$color}})
            {
                my $opo;
                if (/(([NS])(_P\d+))/)
                {
                    $opo = "S$3" if ($2 eq 'N');
                    $opo = "N$3" if ($2 eq 'S');

                    if (($1 eq $adj)||($opo eq $adj))
                    {
                        $find = 0;
                        last;
                    }
                }
                elsif(/(([EW])(_P\d+))/)
                {
                    $opo = "W$3" if ($2 eq 'E');
                    $opo = "E$3" if ($2 eq 'W');

                    if (($1 eq $adj)||($opo eq $adj))
                    {
                        $find = 0;
                        last;
                    }
                }
            }
            push @B_adj,$adj if $find == 1;
        }

        #print "$B : @B_adj\n";

        #put the routing in
        my $good_node = shift @B_adj;
        my $good_short = "$1$2" if ($good_node =~ /([NSEW])_P(\d+)/);
        push @{$routing_added{$color}}, "$good_short_->_$good_node_->_$B";
        #print "$good_short -> $good_node -> $B\n";

    }
    ##################################
    #   Have to be sure that all the LUTs input (F[1..4] and G[1..4]) are
    #   supply
    ##################################
    my @FG_rest;
    foreach (@output_rest)
    {
        if(/S[01]_[FG]_B\d/)
        {
            push @FG_rest,$_;
        }
    }

    foreach my $FG (@FG_rest)
    {
        my @FG_adj;

        #find the adj_list
        foreach (@all_tile)
        {
            if (/(\w+) -> (S[01]_[FG]_B\d)/)
            {
                if ($2 eq $FG)
                {
                    push @FG_adj, $1;
            }
        }
    }
    #print "$FG : @FG_adj \n";

    #look for conflics in group_edge
    for (1..scalar(@FG_adj))
    {
        my $adj = shift @FG_adj;
        my $find = 1;
        foreach (@{$group_edge{$color}})
        {
            if (/(\w+):(\w+)/)
            {
                if (($1 eq $adj)||($2 eq $adj))
                {
                    $find = 0;
                    last;
                }
            }
        }
        push @FG_adj,$adj if $find == 1;
    }
    #print "$FG : @FG_adj \n";

    #look for conflics in the routing_added

    for (1..scalar(@FG_adj))
    {
        my $adj = shift @FG_adj;
        my $find = 1;

        foreach (@{$routing_added{$color}})
        {
            #conflics in the first
            my $opo;
            if (/(([NS])(_P\d+))/)
            {
                $opo = "S$3" if ($2 eq 'N');
                $opo = "N$3" if ($2 eq 'S');

                if (($1 eq $adj)||($opo eq $adj))
                {
                    $find = 0;
                    last;
                }
            }
            elsif(/(([EW])(_P\d+))/)
            {
                $opo = "W$3" if ($2 eq 'E');
                $opo = "E$3" if ($2 eq 'W');

                if (($1 eq $adj)||($opo eq $adj))
                {
                    $find = 0;
                    last;
                }
            }

            #conflics for the double connection
            my $node_special;
            if (/([NSEW])(\d+) ==/)
            {
                $node_special = "$1"."_P$2";
                if($node_special eq $adj)
                {
                    $find = 0;
                    last;
                }
            }
        }
        push @FG_adj,$adj if $find == 1;
    }

    #print "$FG : @FG_adj \n";

    #put the routing in
    my $good_node = shift @FG_adj;
    my $good_short = "$1$2" if ($good_node =~ /([NSEW])_P(\d+)/);
    push @{$routing_added{$color}}, "$good_short_->_$good_node_->_$FG";
    #print "$good_short -> $good_node -> $FG\n";

}
#foreach (@{$routing_added{$color}})
#{
    #print "$color : $_\n";
    #}
}

############################
# ADD the impossible to route connections to another Testing cofiguration
############################

#push @{$routing_added{'COLOR17'}}, 'S0_XQ --> OUT1 --> N2 == E22 --> E_P22 ';
#push @{$routing_added{'COLOR17'}}, 'W20 == E20 == S2 --> S_P2 ';


foreach my $connection (@{$test_configuration{'COLOR17'}})
{
    #print "COLOR17 : $connection\n";
}

foreach my $connection (@{$routing_added{'COLOR17'}})
{
    #print "COLOR17 : $connection\n";
}



############################
# build paths of test configuration
############################

##################
# for 1 to 16
##################
foreach my $color (sort keys(%test_configuration))
{
    if ($color =~ /COLOR(\d+)/)
    {
        next if ($1 > 16);
    }

    ################
    #route orthogonal connections for the supply
    foreach my $IRM (@{$test_configuration{$color}})
    {
        #print "$IRM\n";
        if($IRM =~ /^(([NSEW])_P(\d+)) ->/)
        {
            my $node = $1;
            my $short1 = "$2$3";
            my $short2;
            $short2 = "S$1" if ($short1 =~ /N(\d+)/);
            $short2 = "N$1" if ($short1 =~ /S(\d+)/);
            $short2 = "W$1" if ($short1 =~ /E(\d+)/);
            $short2 = "E$1" if ($short1 =~ /W(\d+)/);

            my $fb = 0;
            foreach (@{$fb_rest{$color}})
            {
                if(/-> (\w+)$/)
                {
                    if($1 eq $node)
                    {
                        push @{$routing_added{$color}}, "$_";
                        #print "$color : $_\n";
                        $fb = 1;
                        last;
                    }
                }
            }

            if($fb == 0)
            {
                push @{$routing_added{$color}}, "$short2_==_$short1_->_$node";
                #print "$color : $short2 == $short1 -> $node\n";
            }
            #print "$color : $IRM\n";
        }
    }
}

##################
# for 17 to 22
##################
foreach my $color (sort keys(%test_configuration))
{
    if ($color =~ /COLOR(\d+)/)
    {
        next if (($1 < 17)||($1 > 22));
    }
    ##################
    #route orthogonal connections for the supply
    foreach my $IRM (@{$test_configuration{$color}})
    {
        #print "$color : $IRM\n";
        if($IRM =~ /^(([NSEW])_P(\d+)) ->/)
        {
            my $node = $1;
            my $short1 = "$2$3";
            my $short2;
            $short2 = "S$1" if ($short1 =~ /N(\d+)/);
            $short2 = "N$1" if ($short1 =~ /S(\d+)/);
            $short2 = "W$1" if ($short1 =~ /E(\d+)/);
            $short2 = "E$1" if ($short1 =~ /W(\d+)/);

            my $fb = 0;
            foreach (@{$routing_added{$color}})
            {
                #print "$_\n";
                if (/-> ([NSEW]_P\d+)/)
                {
                    if($1 eq $node)
                    {
                        #push @{$routing_added{$color}}, "$_";
                        #print "$color : $_\n";
                        $fb = 1;
                        last;
                    }
                }
            }

            if($fb == 0)
            {
                push @{$routing_added{$color}}, "$short2_==_$short1_->_$node";
                #print "$color : $short2 == $short1 -> $node\n";
            }
        }
    }
}
##################
# for 23 to 26
##################
foreach my $color (sort keys(%test_configuration))
{
    if ($color =~ /COLOR(\d+)/)
    {
        next if ($1 < 23);
    }

    ##################
    #route orthogonal connections for the supply
    foreach my $IRM (@{$test_configuration{$color}})
    {
        #print "$color : $IRM\n";
        if($IRM =~ /^(([NSEW])_P(\d+)) ->/)
        {
            #print "$color : $IRM\n";

            my $node = $1;
            my $short1 = "$2$3";
            my $short2;
            $short2 = "S$1" if ($short1 =~ /N(\d+)/);
            $short2 = "N$1" if ($short1 =~ /S(\d+)/);
            $short2 = "W$1" if ($short1 =~ /E(\d+)/);
            $short2 = "E$1" if ($short1 =~ /W(\d+)/);

            my $fb = 0;
            foreach (@{$routing_added{$color}})
            {
                #print "$color : $_\n";
                if (/-> ([NSEW]_P\d+)/)
                {
                    if($1 eq $node)
                    {
                        #push @{$routing_added{$color}}, "$_";
                        #print "$color : $_\n";
                        $fb = 1;
                        last;
                    }
                }
            }

            if($fb == 0)
            {
                push @{$routing_added{$color}}, "$short2_==_$short1_->_$node";
                #print "$color : $short2 == $short1 -> $node\n";
            }
        }
    }
    for (1..scalar(@{$routing_added{$color}}))
    {
        my $temp = shift @{$routing_added{$color}};
        if ($temp =~ /^([NSEW]\d+)->/)
        {
            my $short = $1;
            my $short2;
            $short2 = "S$1" if ($short =~ /N(\d+)/);
            $short2 = "N$1" if ($short =~ /S(\d+)/);
            $short2 = "W$1" if ($short =~ /E(\d+)/);
            $short2 = "E$1" if ($short =~ /W(\d+)/);

            push @{$routing_added{$color}}, "$short2 == $temp";
            next;
        }
        push @{$routing_added{$color}},$temp;
        #print "$color : $_\n";
    }
}

#print the result
foreach my $color (sort keys(%test_configuration))
{
    foreach my $conn (@{$test_configuration{$color}})
    {
        print "$color : $conn\n";
    }

    if (exists ($routing_added{$color}))
    {
        foreach my $add (@{$routing_added{$color}})
        {
            print "$color : $add\n";
        }
    }
}

exit;

##########################################################################
#            Functions
##########################################################################


##########################################################################
#Function Name: C1
#Inputs:
#(\@{list_of_edges_G})
#Output:      the color number 1 of the coloring
#        (the fonction find an entire coloring but return only)
##########################################################################

sub C1
{

    ####################################################################
    # put all the input of the function in arrays
    ####################################################################

    my @graph_edge = @{(shift(@_))};
    my @graph_edge_copy = @graph_edge;

    ####################################################################
    # Build the graph
    ####################################################################


    my $G = Graph::Undirected->new;
    $G->add_edges(@graph_edge);

    ####################################################################
    # Set all the edges color to 0
    ####################################################################

    for (1 .. (scalar(@graph_edge)/2))
    {
        my $color = 0;
        my $vs = shift(@graph_edge);
        my $vd = shift(@graph_edge);
        push @graph_edge,$vs;
        push @graph_edge,$vd;

        $G->set_attribute('color',$vs,$vd,$color);
        $G->set_attribute('color',$vd,$vs,$color);
    }
```

```
        ###############################################################
        # Star of the main loop
        ###############################################################

        my $alpha;
        my $beta;

        #test variable
        my $index = 1;
        for(1 .. (scalar(@graph_edge)/2))
        {

            #test
            #print "$index ";
            #$index++;

            my $vs = shift(@graph_edge);
            my $vd = shift(@graph_edge);
            push @graph_edge,$vs;
            push @graph_edge,$vd;


            ###############################################################
            # find the lower color missing at $vs
            ###############################################################

            $alpha = &lower_color(\$G,$vs);
            #test
            #print "$alpha ";


            ###############################################################
            #find the lower color missing at $vd
            ###############################################################

            $beta = &lower_color(\$G,$vd);
            #test
            #print "$beta ";

            ###############################################################
            # Check if alpha not equal beta
            ###############################################################
            if ($alpha != $beta)
            {
                ###############################################################
                #find an alternative path and swap color
                ###############################################################
                &alt_path(\$G,$vs,$vd,$alpha,$beta);
            }
            ###############################################################
            #color the edge (vs,vd) with beta
            ###############################################################
            $G->set_attribute('color',$vs,$vd,$beta);
            $G->set_attribute('color',$vd,$vs,$beta);

        }
        ###############################################################
        # return color 1
        ###############################################################

        my @color1_edges;

        #my $count = 1;
        #foreach my $color (1..25)
        #{
        my $color = 1;
        for(1 .. (scalar(@graph_edge)/2))
        {
            my $vs = shift(@graph_edge);
            my $vd = shift(@graph_edge);
            push @graph_edge,$vs;
            push @graph_edge,$vd;

            #if($G->get_attribute('color',$vs,$vd) == 1)
            if($G->get_attribute('color',$vs,$vd) == $color)
            {
                push @color1_edges,$vs,$vd;
                #print "COLOR$color:$vs - $vd\n";
            }

            #test
            #my $test = $G->get_attribute('color',$vs,$vd);
            #print "color $vs-$vd = $test\n";
        }
        #}
        return @color1_edges;

    }
```

```
#Return the lower missing color at the vertex
sub lower_color
{
    my $G = shift;
    my $vertex = shift;
    my $color = 1;
    my @color_list;

    my @adj_list = $$G->neighbors($vertex);


    #build a list of all the color
    foreach my $adj_vertex (@adj_list)
    {
        my $temp = $$G->get_attribute('color',$vertex,$adj_vertex);
        if ($temp != 0)
        {
            push @color_list, $temp;
        }
    }


    #find the smallest missing
    #@color_list = sort(@color_list);

    #test
    #print "@color_list: lowest missing ";
    my $got_it;
    for my $local_color (1..50)
    {
        $got_it = 1;
        foreach my $c (@color_list)
        {
            if ($c == $local_color)
            {
                $got_it = 0;
                last;
            }
        }
        if ($got_it == 1)
        {
            $color = $local_color;
            last;
        }
    }
    #my $prev_c = shift @color_list;
    #foreach my $c (sort(@color_list))
    #{
    #    if($prev_c == $c)
    #    {next;}
    #    elsif($c != $color)
    #    {last;}
    #    else
    #    {$color++;}
    #    $prev_c = $c;
    #}

    #test
    #print "$color\n";

    return $color;
}

sub alt_path
{
    my $G = shift;
    my $vs = shift;
    my $vd = shift;
    my $alpha = shift;
    my $beta = shift;

    #test
    #print "$G $vs $alpha $beta\n";


    my @alternate_path = &path(\$G,$vs,$alpha,$beta);


    if(@alternate_path)
    {
        &inv_path_color(\$G,\@alternate_path,$alpha,$beta);
    }
    else
    {
        @alternate_path = &path(\$G,$vd,$alpha,$beta);
        if(@alternate_path)
        {
            &inv_path_color(\$G,\@alternate_path,$alpha,$beta);
        }
        else
        {
            return;
        }
    }
}

sub path
{
    my $G = shift;
    my $vertex = shift;
    my $alpha = shift;
    my $beta = shift;

    my $temp_color = $beta;
    my @path_list;
    my $next_vertex;

    $next_vertex = &check_color(\$G, $vertex, $temp_color);
    if($next_vertex eq 0)
    {return @path_list;}

    while($next_vertex ne 0)
    {
        push (@path_list, $next_vertex);
        push (@path_list, $next_vertex);

        if($temp_color == $alpha)
        {$temp_color = $beta;}
        else
        {$temp_color = $alpha;}

        $next_vertex = &check_color(\$G, $next_vertex, $temp_color);


        # have to break the path when it is a loop because the program enter in
        # an infinite loop.

        # if the next_vertex is equal to vertex
        if ($next_vertex eq $vertex)
        {last;}

        my $flag = 0;
        foreach (@path_list)
        {
            if ($_ eq $next_vertex)
            {
                $flag = 1;
            }
        }
        last if $flag == 1;
        #test
        #print "$vertex:$next_vertex ";
        #print ".";
    }
    my $extra = pop @path_list;
    unshift (@path_list, $vertex);
    return @path_list;
}

sub inv_path_color
{
    my $G = shift;
    my @path = @{(shift(@_))};
    my $alpha = shift;
    my $beta = shift;

    #test
    #print "@path\n";

    for(1..(scalar(@path)/2))
    {
        my $start = shift @path;
        my $vertex = shift @path;

        if($$$G->get_attribute('color',$start,$vertex) == $beta)
        {
            $$$G->set_attribute('color',$start,$vertex,$alpha);
            $$$G->set_attribute('color',$vertex,$start,$alpha);
        }
        elsif($$$G->get_attribute('color',$start,$vertex) == $alpha)
        {
            $$$G->set_attribute('color',$start,$vertex,$beta);
            $$$G->set_attribute('color',$vertex,$start,$beta);
        }
    }
}
```

```
sub check_color
{
    my $G = shift;
    my $vertex = shift;
    my $color = shift;

    my $edge_color;

    my @vertex_adj = $$$$G->neighbors($vertex);
    foreach my $v_end (@vertex_adj)
    {
        $edge_color = $$$$G->get_attribute('color',$vertex,$v_end);
        if($edge_color == $color)
        {
            return $v_end;
        }
    }
    return 0;
}
###########################################################################
#Function Name: read_file
#Inputs:
#   name of the file you want to open
#Output:
#   the array that contain all the line of the file
###########################################################################
sub read_file
{
    my ($filename) = @_;
    open(FILE,"$filename")||die("could_not_open_the_file");
    my @file = <FILE>;
    close(FILE);
    chomp(@file);
    return @file;
}
```

Appendix C

Execution Script

```
#!/usr/bin/env bash
##########################################################################
# Name: running_script
# Author: Christian Giasson
# Date: July 21st 2004
# Note: This bash script executes all programs in sequence to get all the
# test configurations.
##########################################################################

#ORM coloring
ORM='../good_coloring_clean'
$ORM/good_coloring.pl    -a $ORM/ORM_CONF_2
                         -b $ORM/CLB_CONF_2
                         -c $ORM/OUT_adj_list > EDGE_COLORING

#complete ORM coloring
COMP='../complete_ORM_coloring'
$COMP/complete_ORM_coloring.pl
                         -a ./EDGE_COLORING
                         -b $COMP/CLB_CONF_2
                         -c $COMP/OUT_adj_list > EDGE_COLORING_C

#find feedback paths to the CLB inputs for all edges in each color
PATH_1_16='../color_path_1_16_clean'
$PATH_1_16/color_path_1_16.pl
                         -a ./EDGE_COLORING_C
                         -b $PATH_1_16/CLB_CONFIG_2_4_v3
                         -c $PATH_1_16/LIST_CONF
                         -d $PATH_1_16/PATH_SINGLE
                         -e $PATH_1_16/PATH_DOUBLE >FEEDBACK_PATH_1_16

#complete the coloring of the IRM for color 1 to 16
IRM='../color_IMUX_clean';
$IRM/color_IRM.pl        -a $IRM/small_IRM
                         -b ./FEEDBACK_PATH_1_16 > IRM_1_16

#get the rest of the IRM that is not color yet
$IRM/color_IRM_rest.pl   -a $IRM/small_IRM
                         -b ./FEEDBACK_PATH_1_16 > IRM_REST

#get the IRM connection that are not connected on node lines
$IRM/gen_IRMG_sup.pl < $IRM/all_tile.txt > IRM_SUP

#merge the rest of the node line connections and the one that
#are not connected to the node lines.
cat IRM_REST IRM_SUP > IRM_TOTAL

#color the rest of IRM and get test configuration 17 to 26
$IRM/color_IRM_part2.pl -a ./IRM_TOTAL
                        -b $IRM/feedback_17_22 > IRM_17_26

#Get feedback for the TC 17 to 26 and configuring all the GRM in such a way
#that there is no conflics in the routing of the signals.
$IRM/completion.pl       -a $IRM/small_IRM
                         -b IRM_1_16
                         -c IRM_17_26
                         -d FEEDBACK_PATH_1_16
                         -e $IRM/feedback_17_22
                         -f $IRM/all_tile.txt > FINAL_PATH
```

blank page - no text

Appendix D

Test Configurations

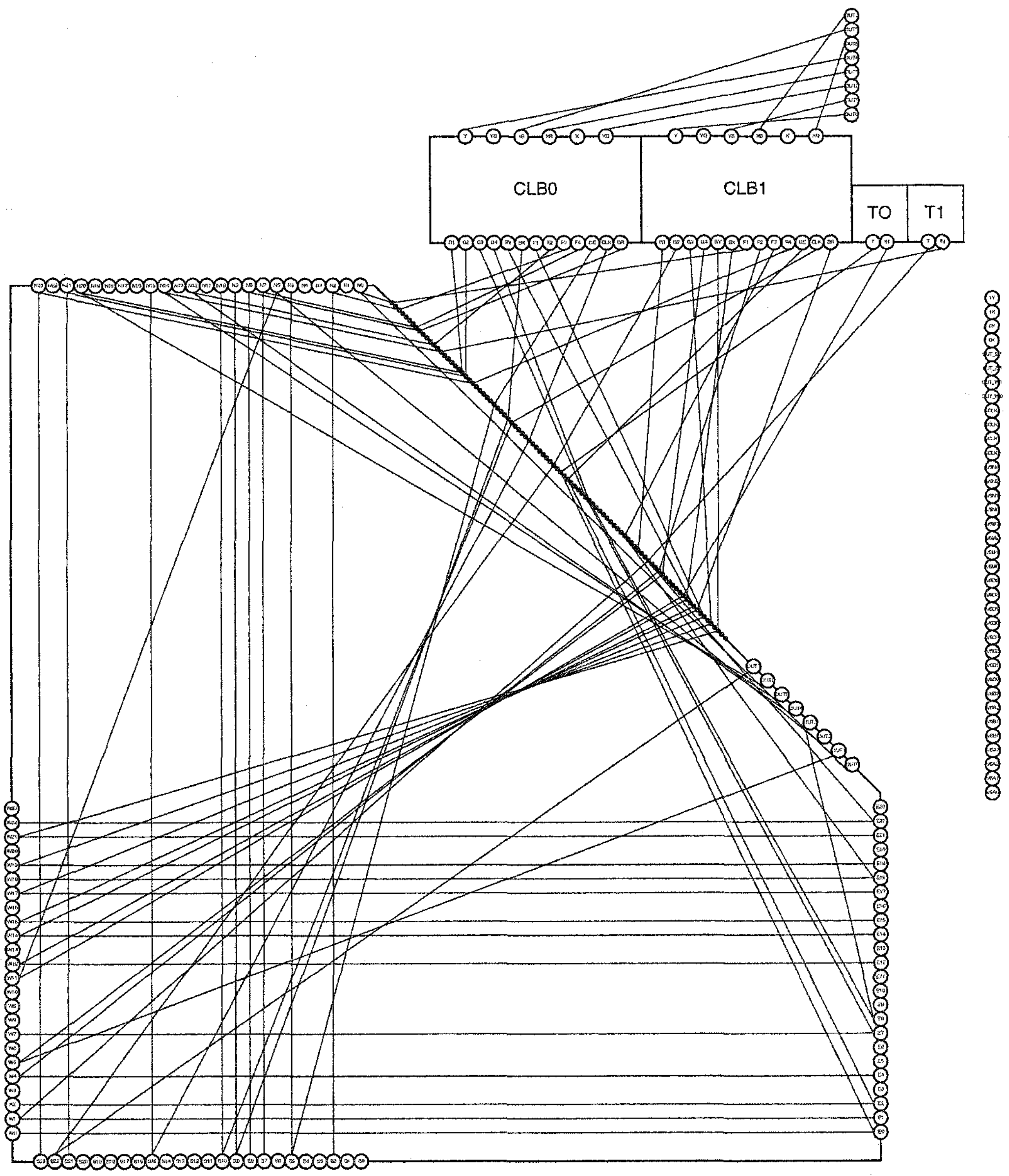


Figure D.1: Test Configuration 1

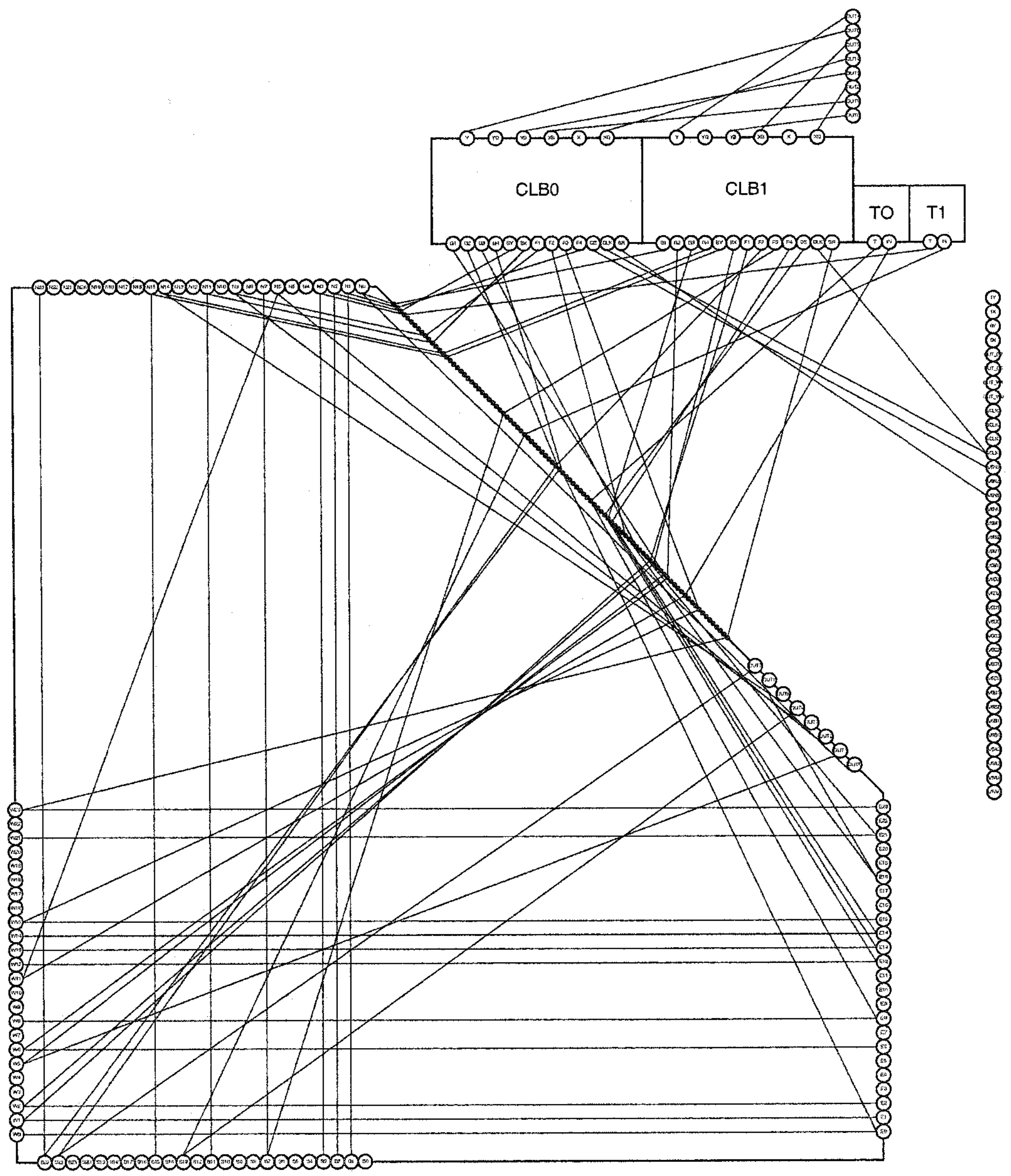


Figure D.2: Test Configuration 2

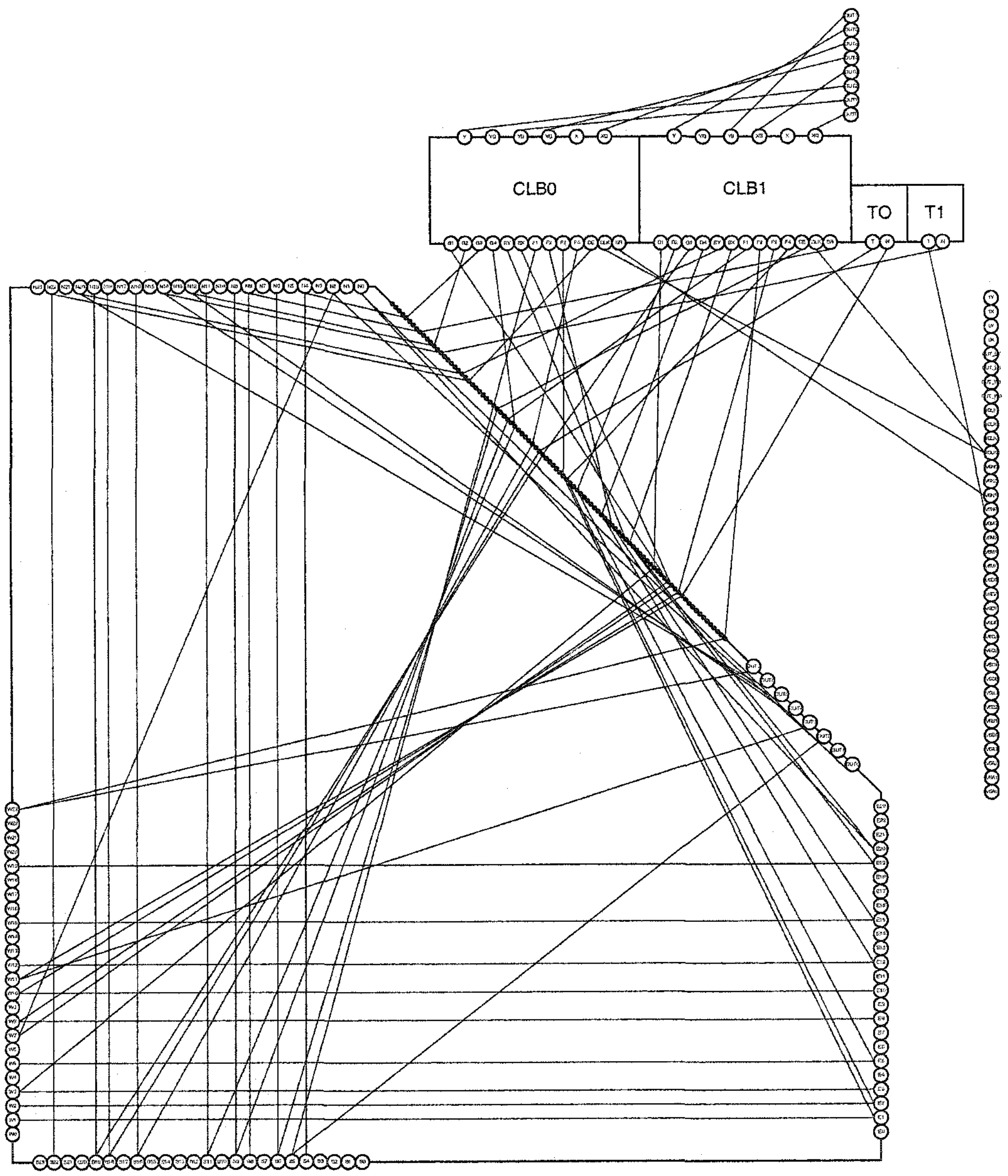


Figure D.3: Test Configuration 3

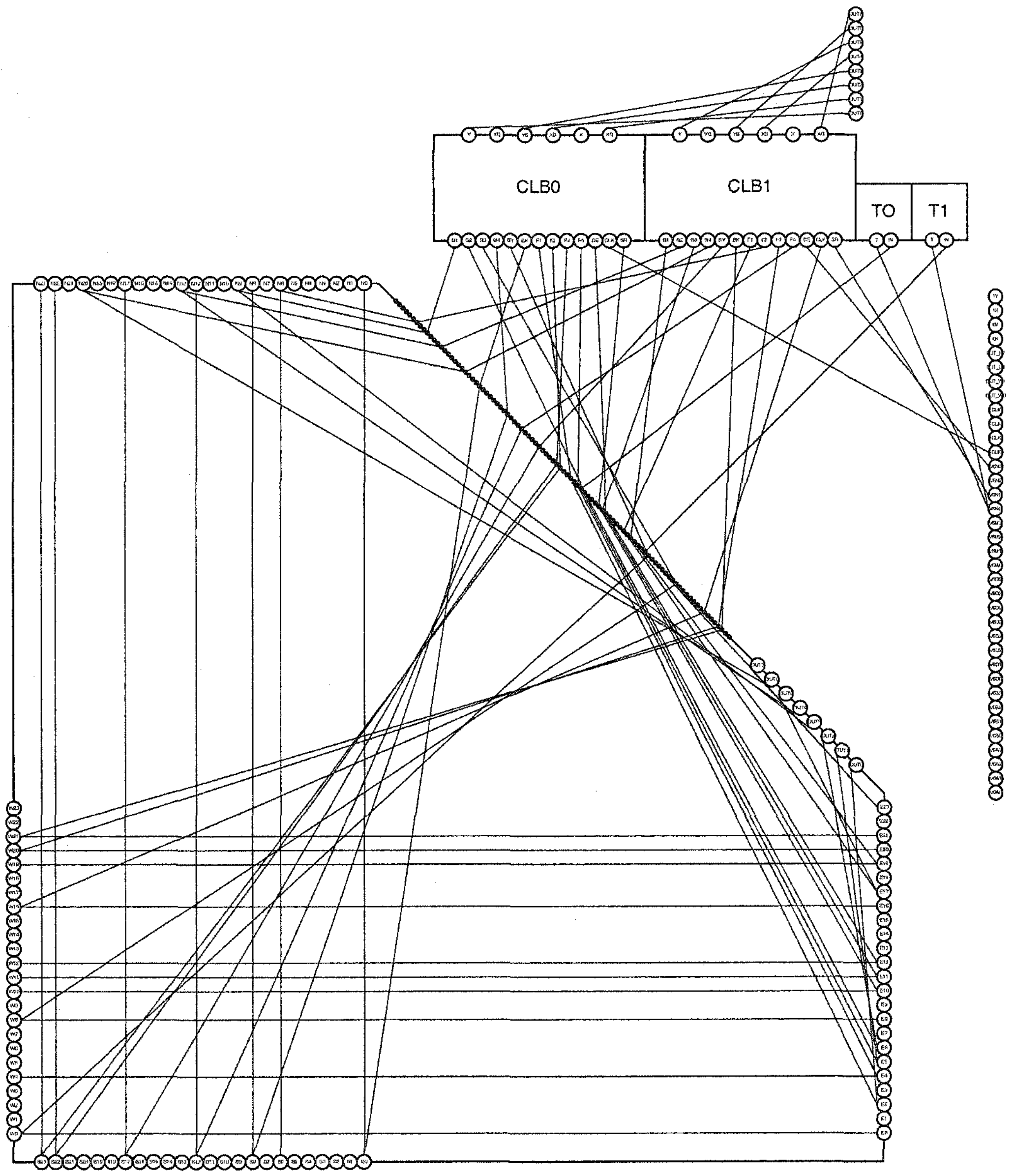


Figure D.4: Test Configuration 4

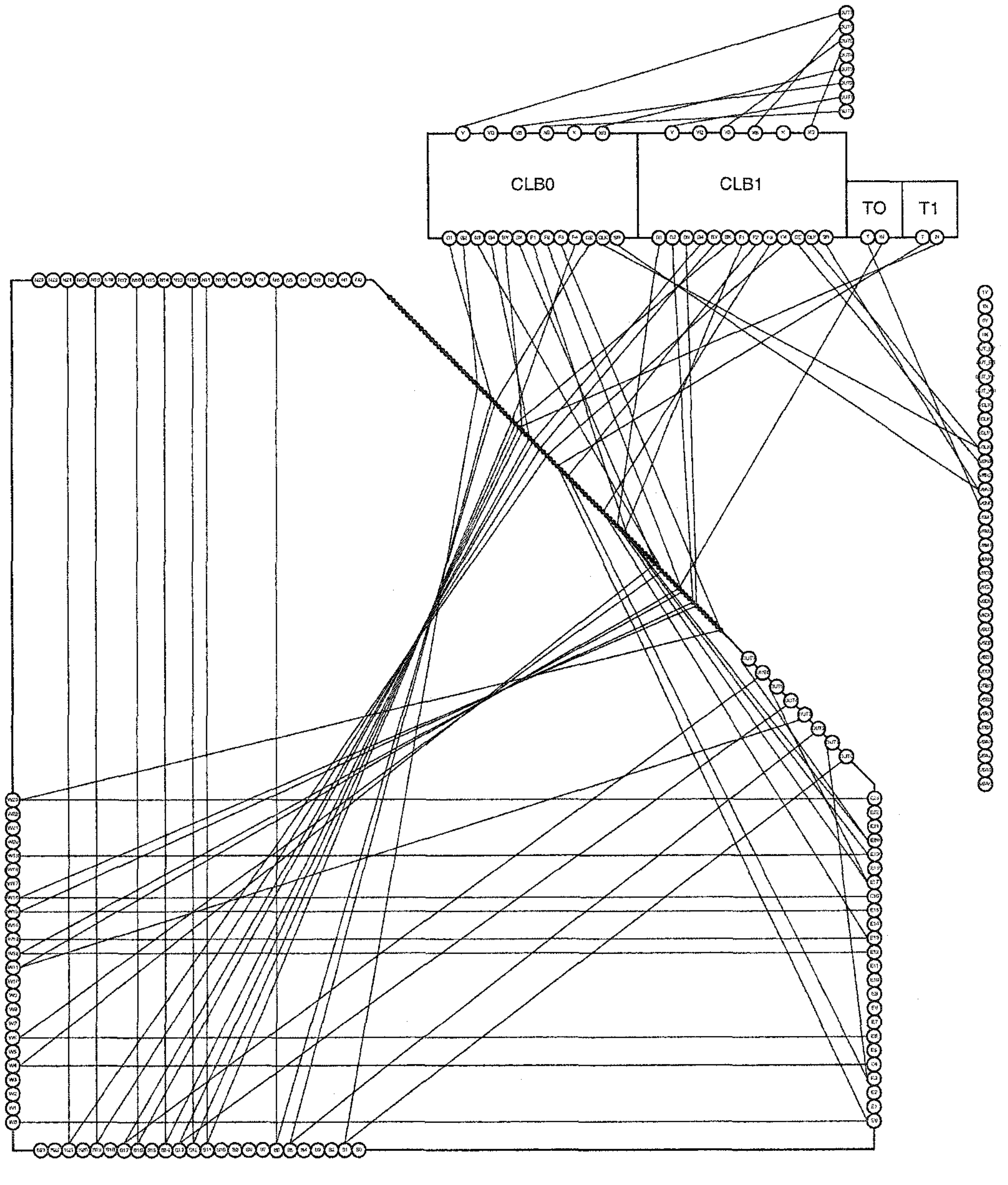


Figure D.5: Test Configuration 5

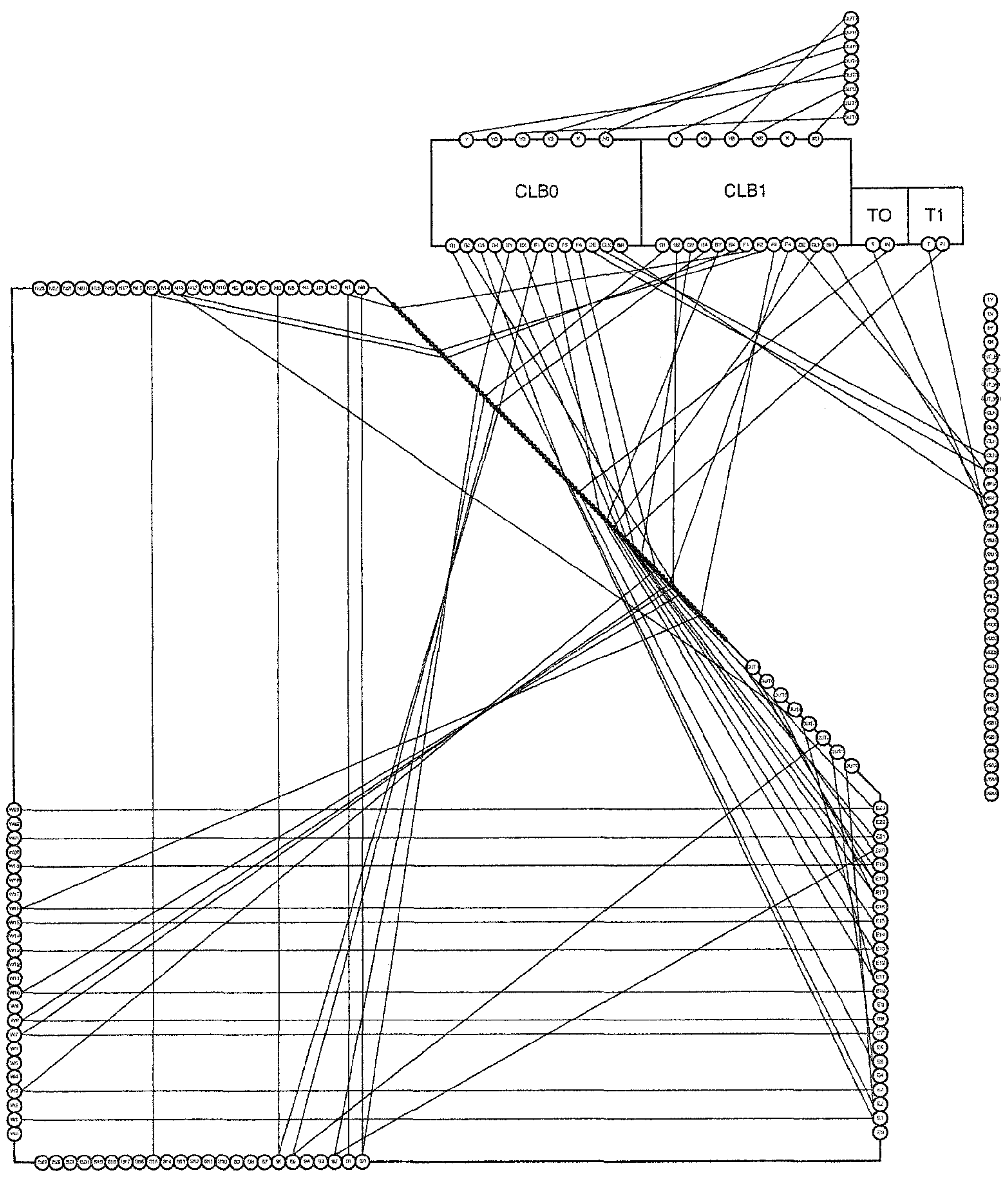


Figure D.6: Test Configuration 6

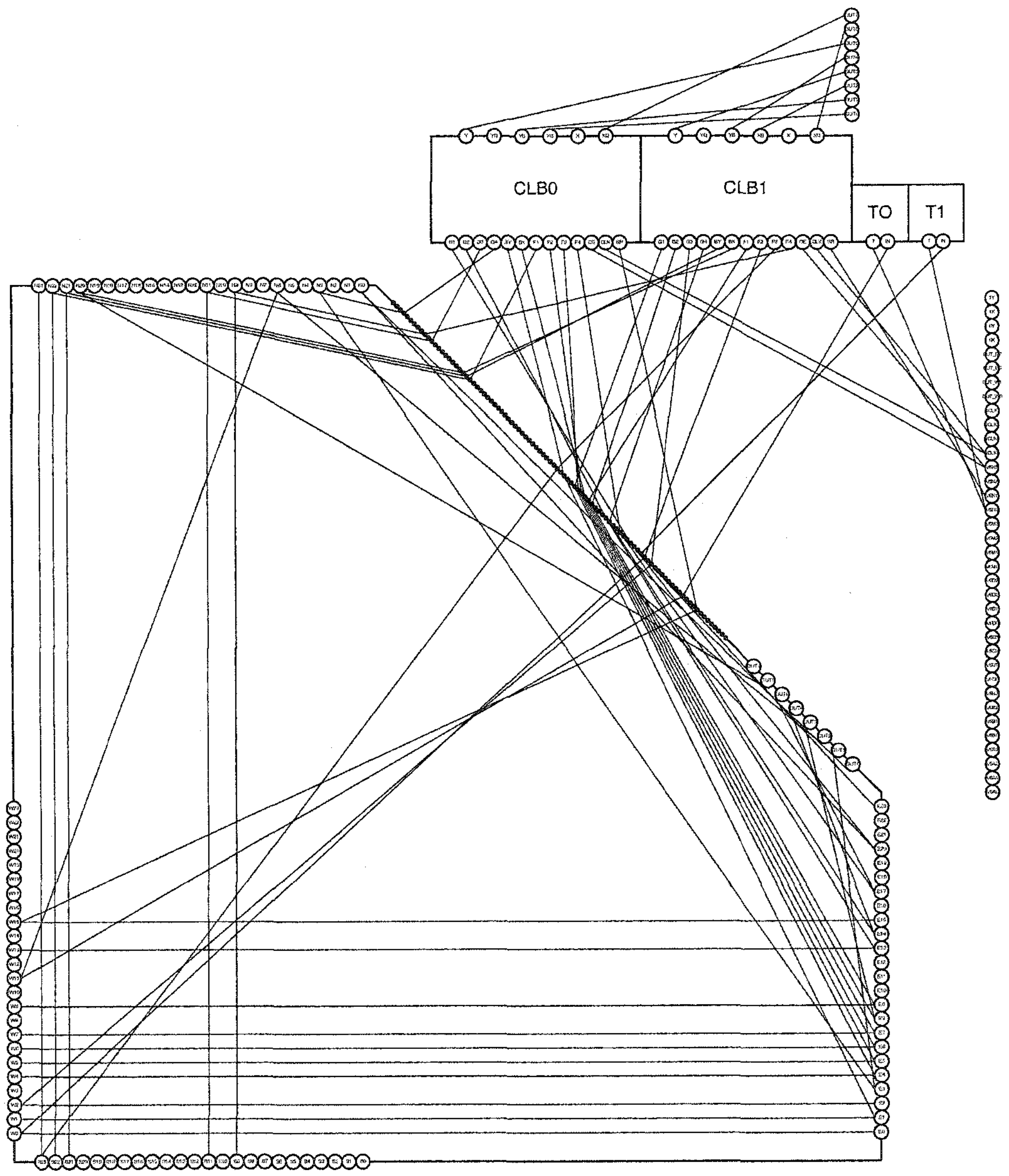


Figure D.7: Test Configuration 7

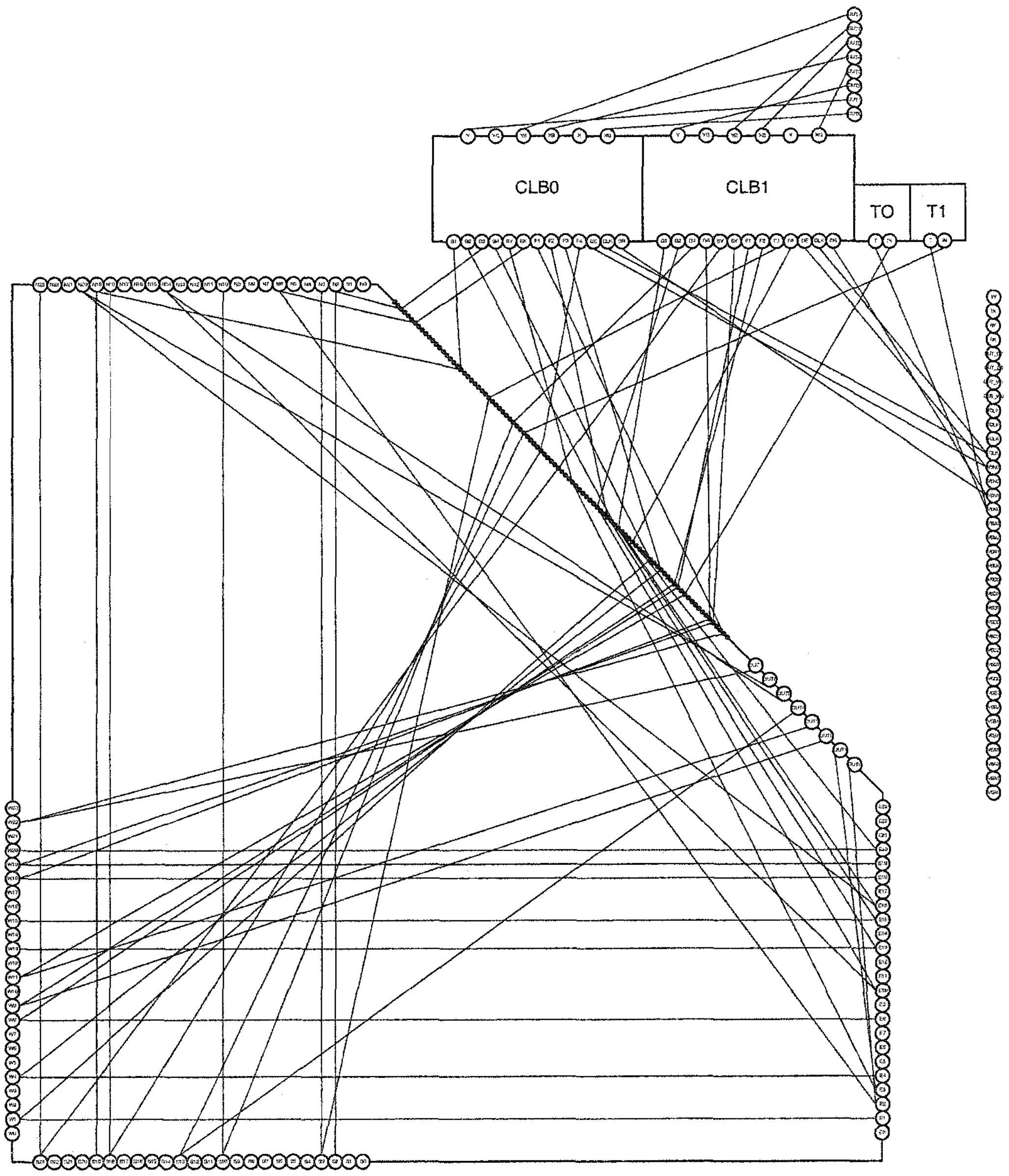


Figure D.8: Test Configuration 8

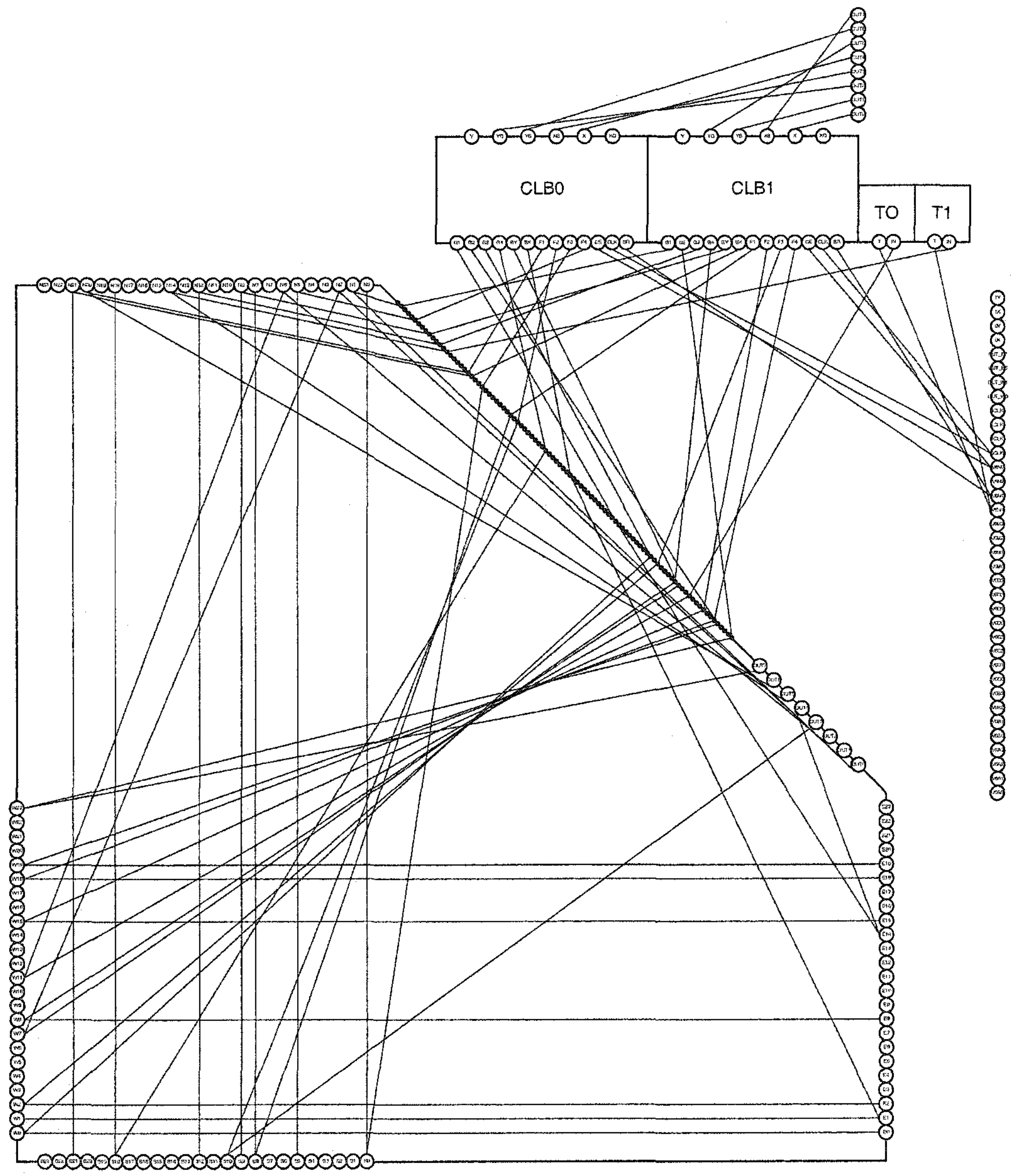


Figure D.9: Test Configuration 9

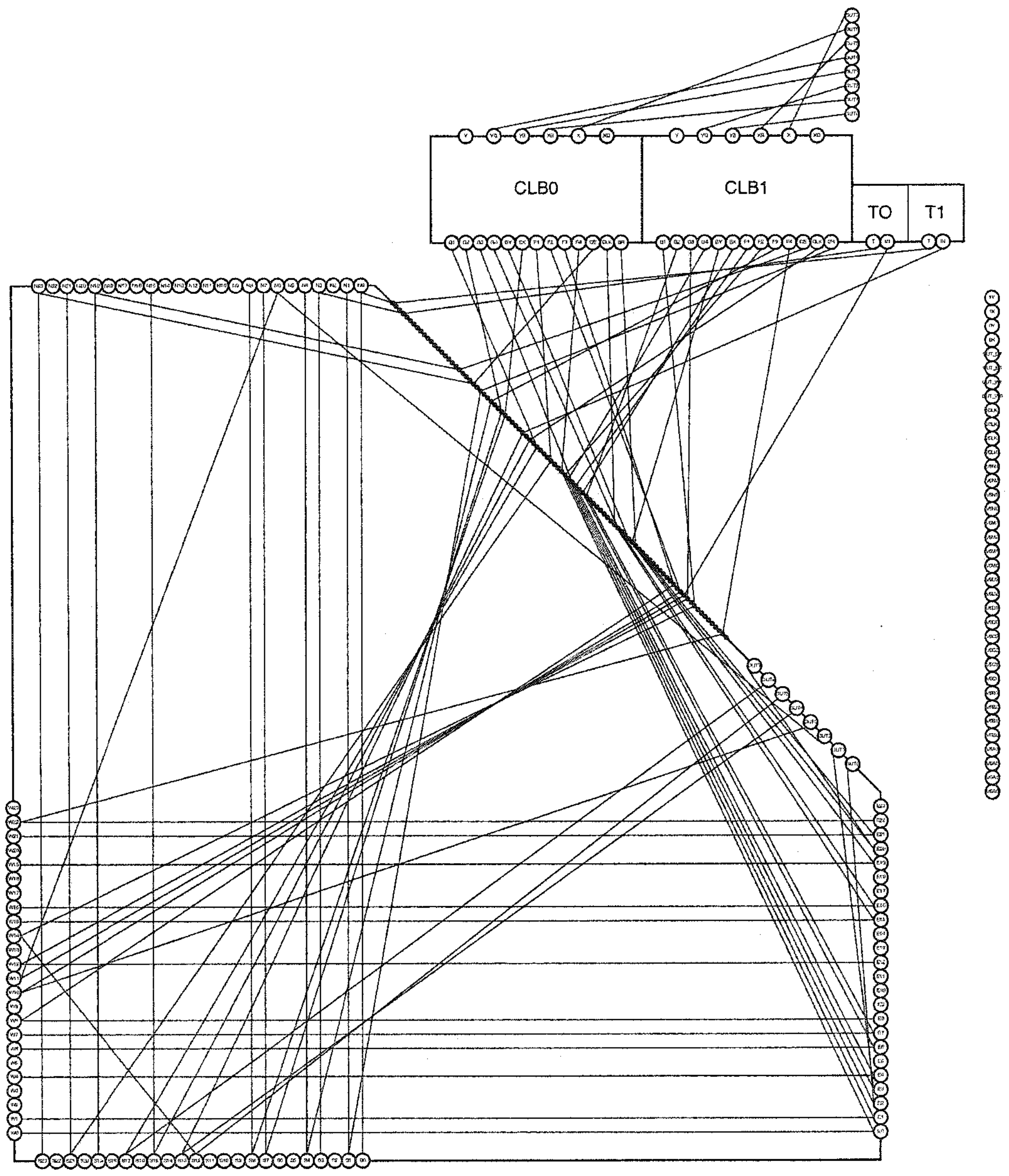


Figure D.10: Test Configuration 10

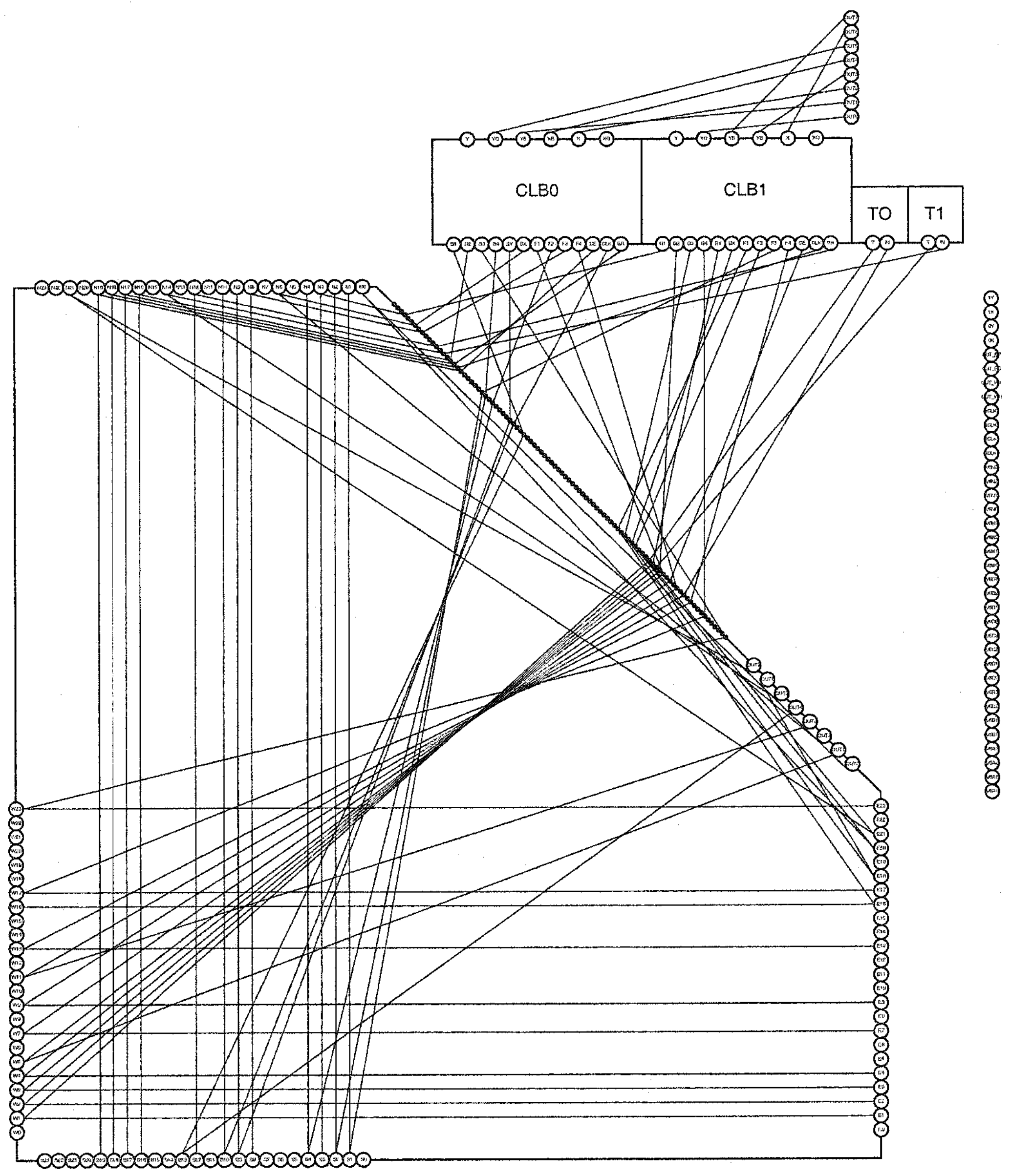


Figure D.11: Test Configuration 11

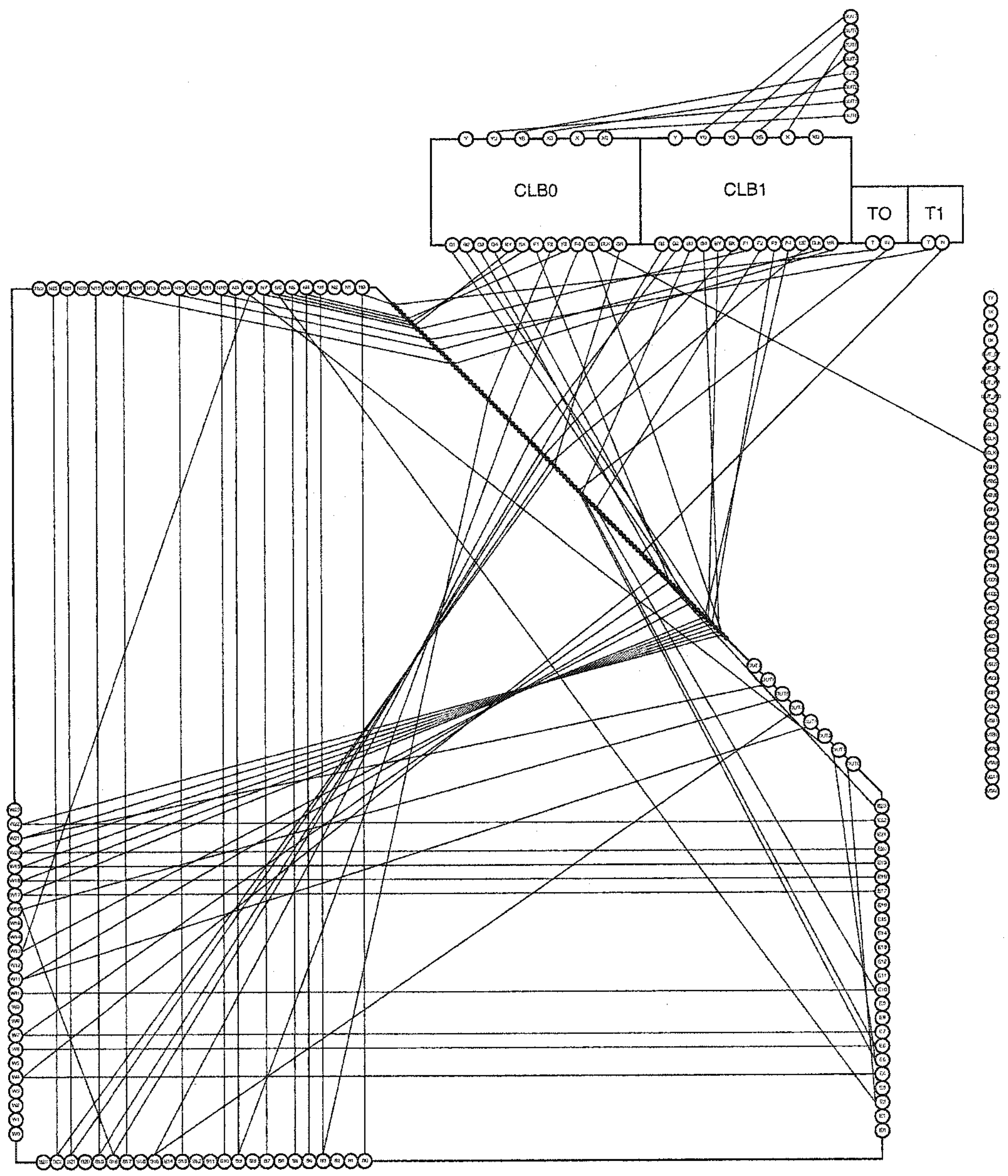


Figure D.12: Test Configuration 12

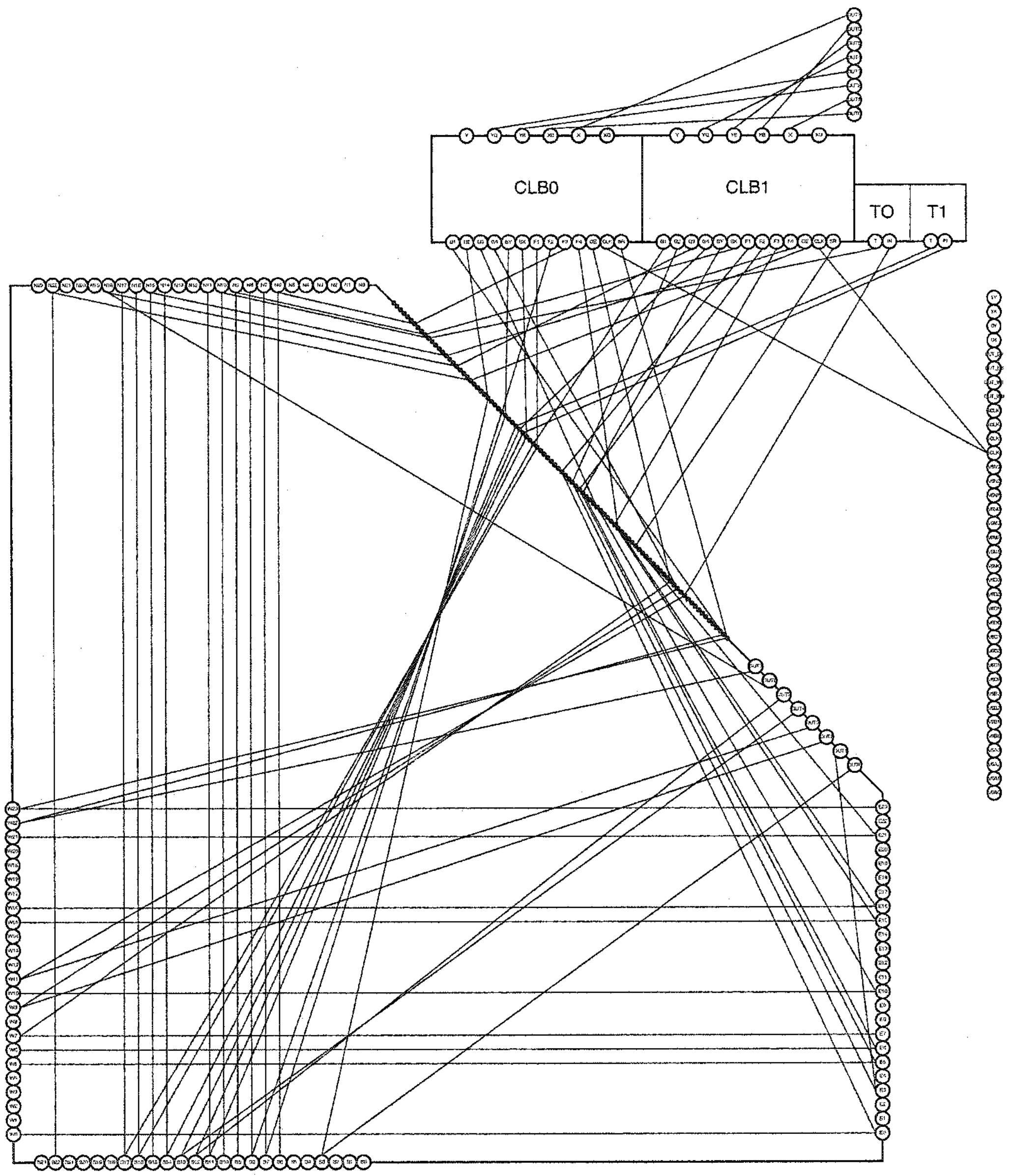


Figure D.13: Test Configuration 13

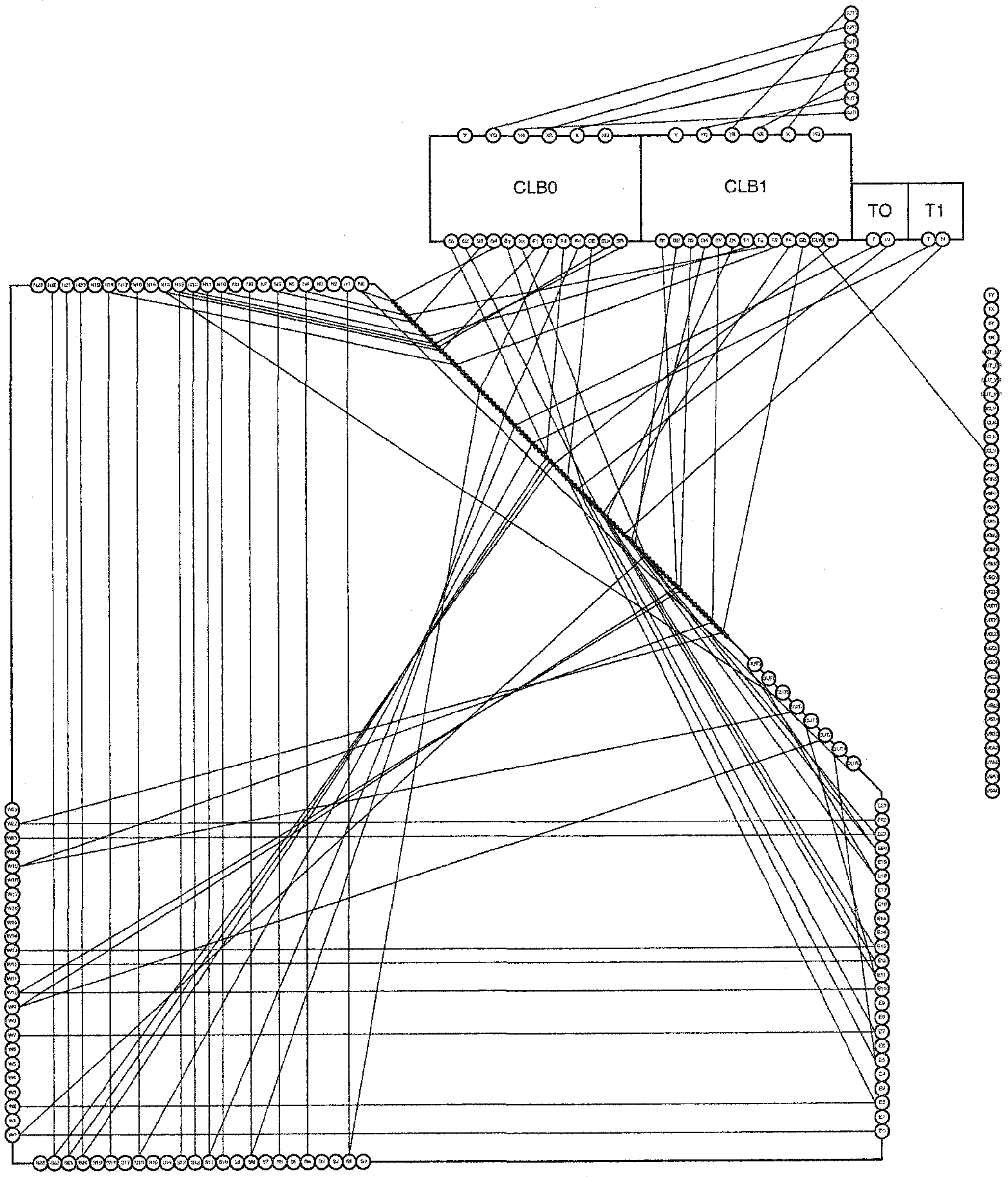


Figure D.14: Test Configuration 14

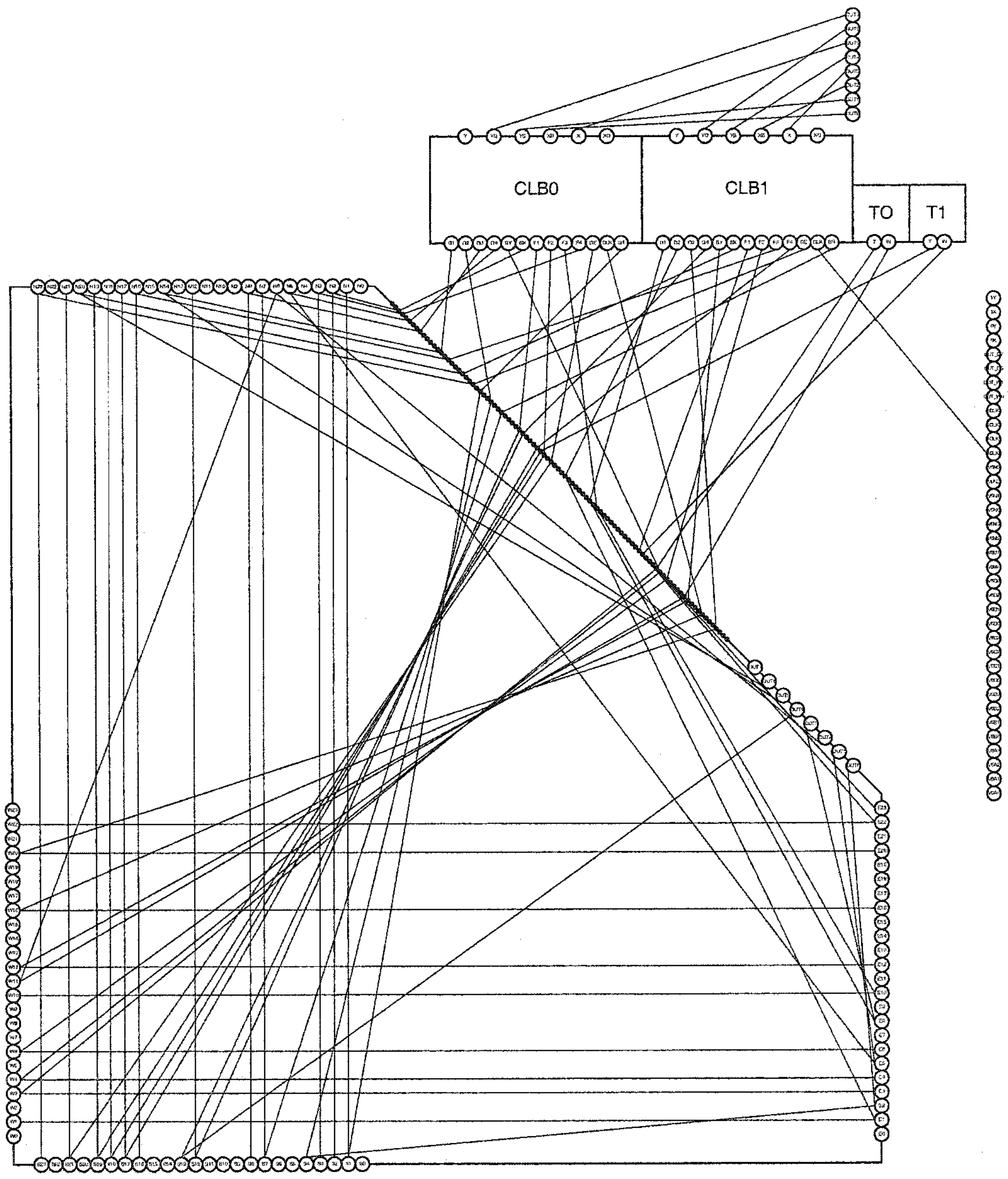


Figure D.15: Test Configuration 15

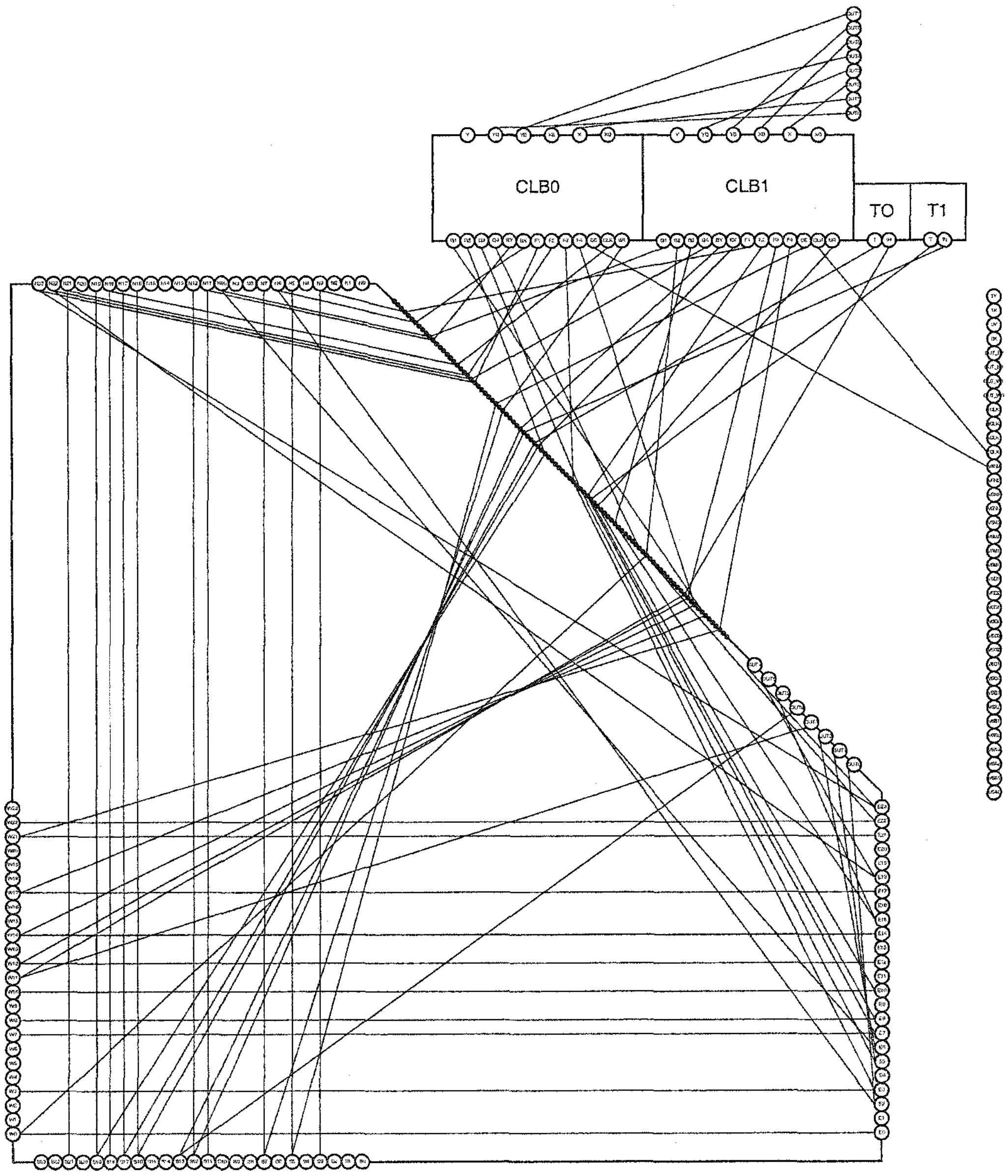


Figure D.16: Test Configuration 16

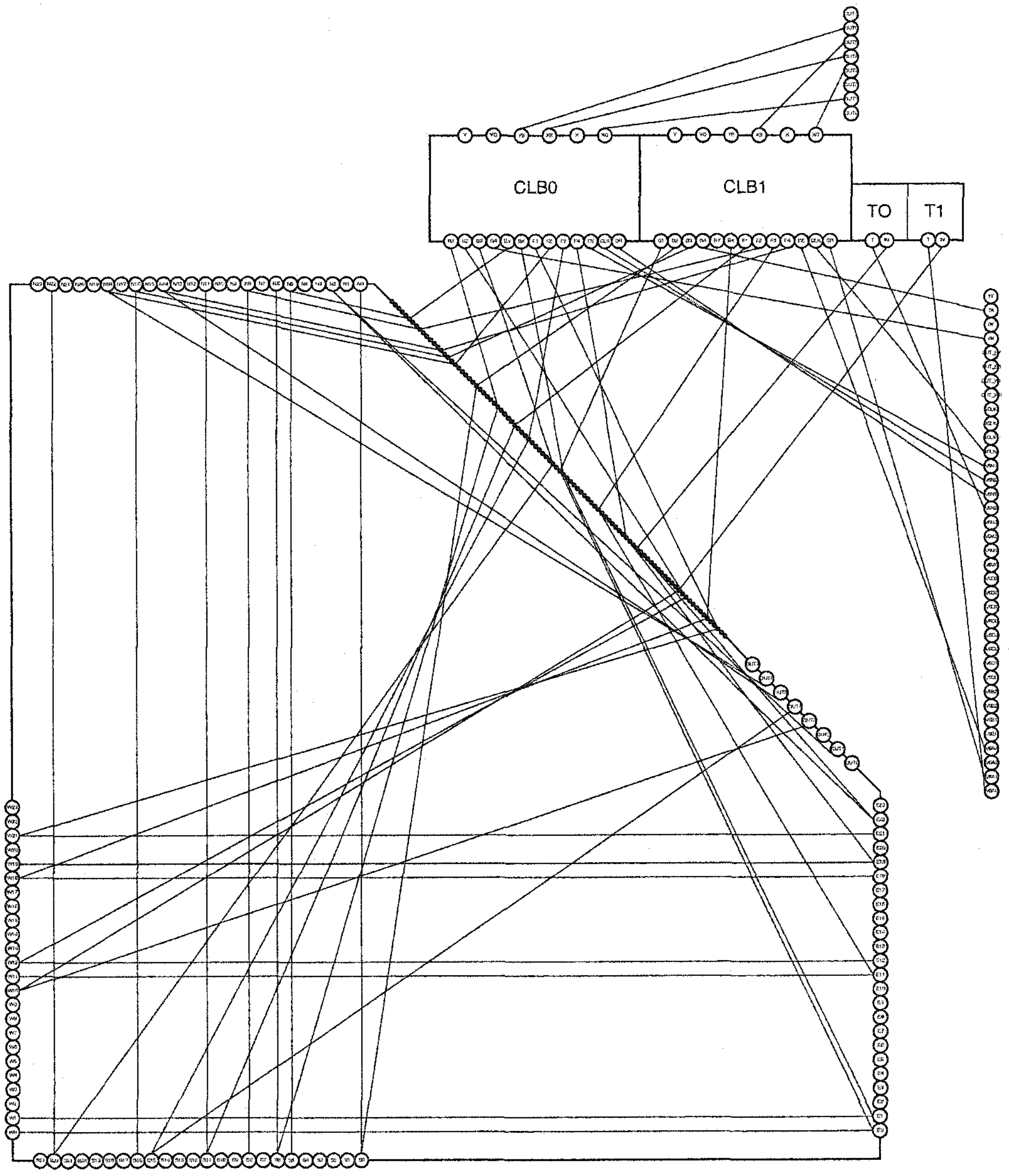


Figure D.17: Test Configuration 17

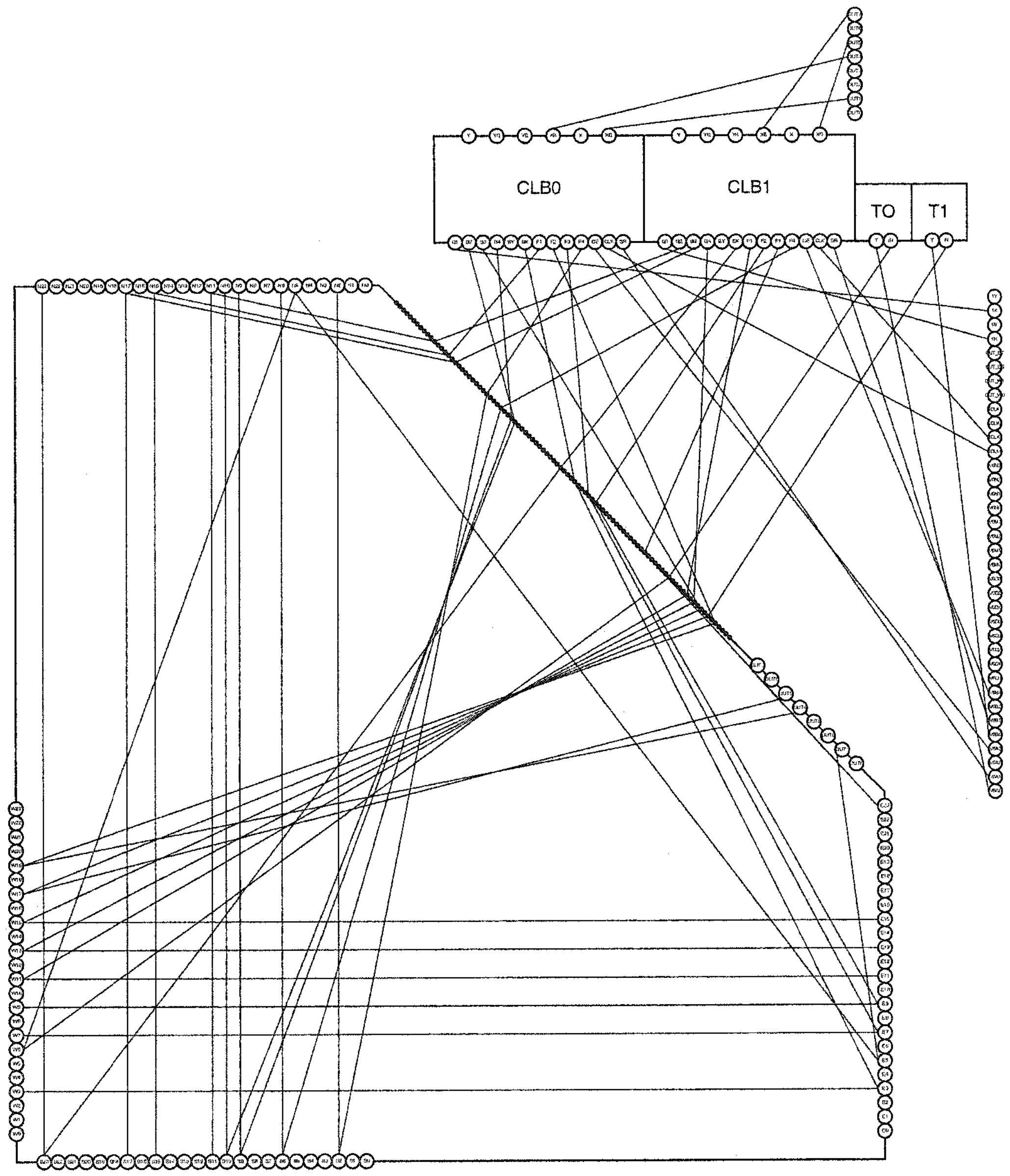


Figure D.18: Test Configuration 18

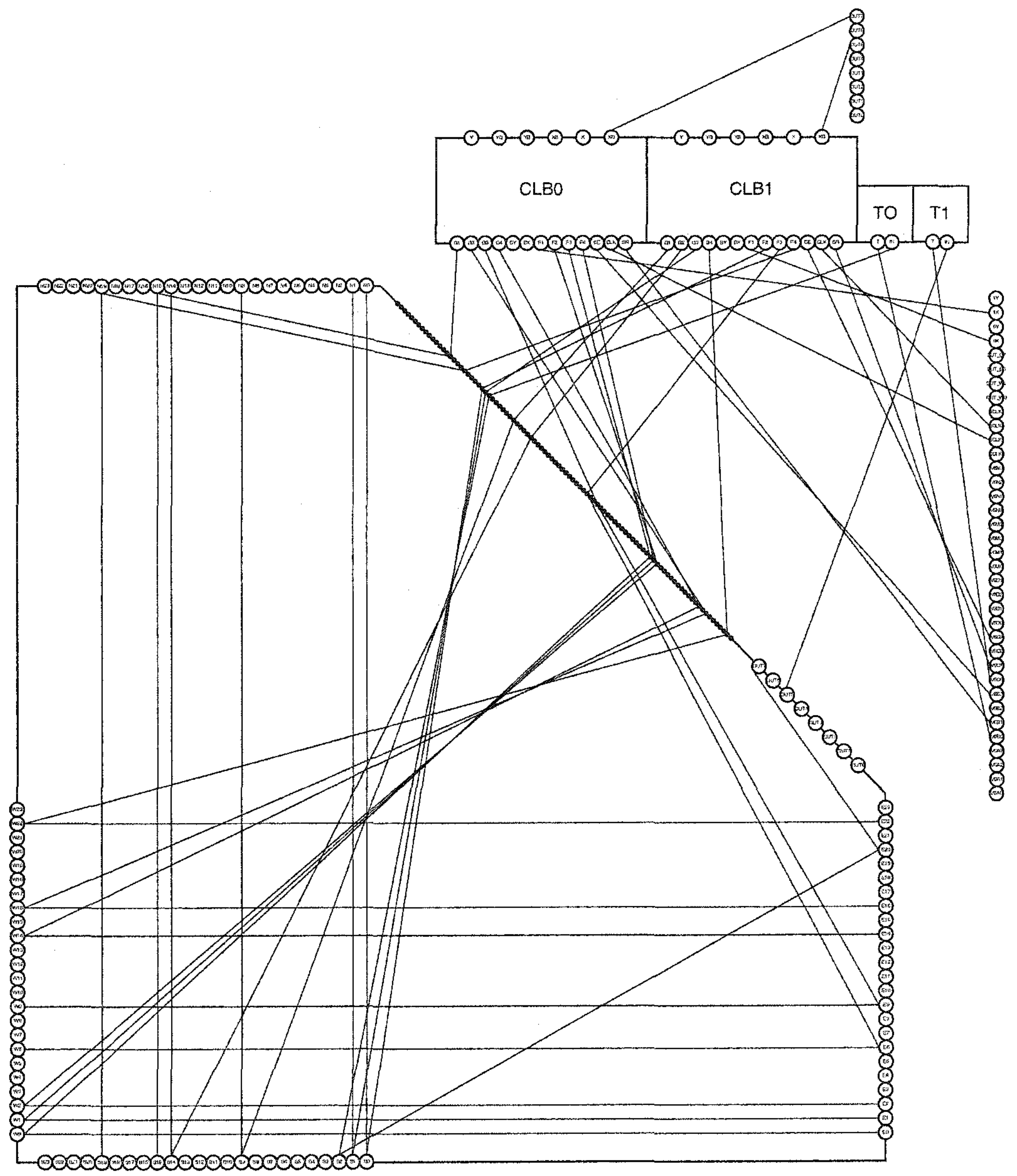


Figure D.19: Test Configuration 19

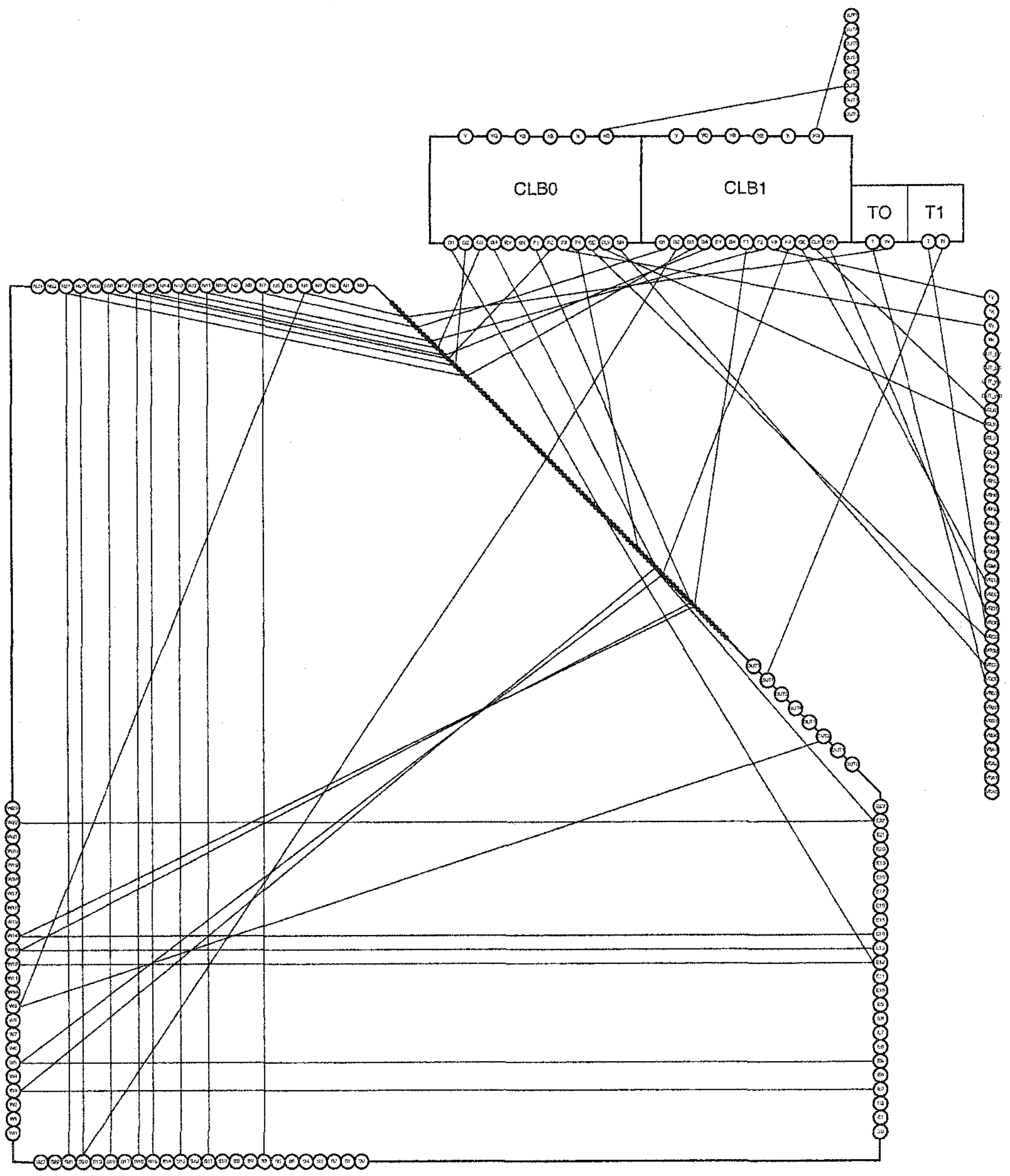


Figure D.20: Test Configuration 20

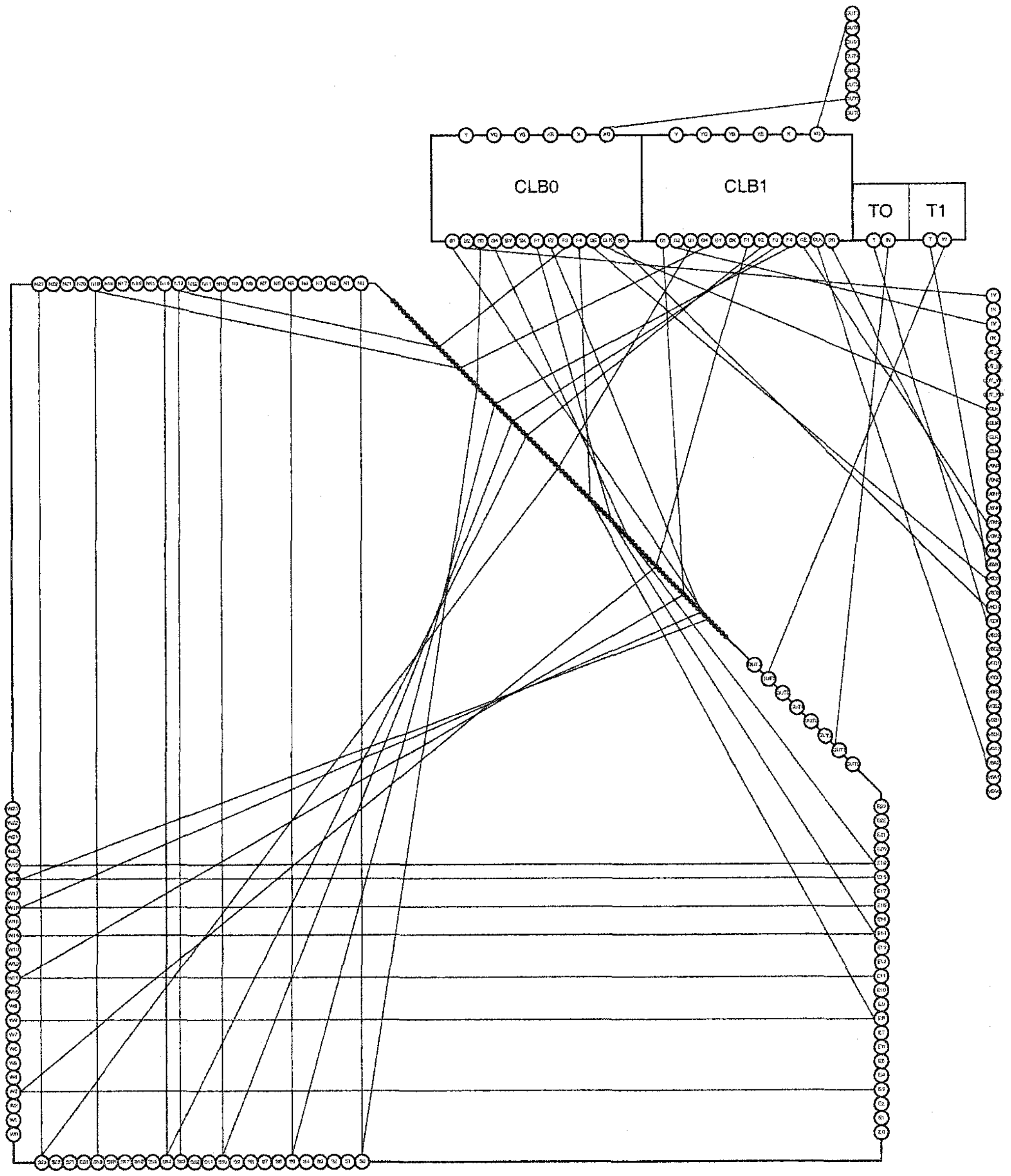


Figure D.21: Test Configuration 21

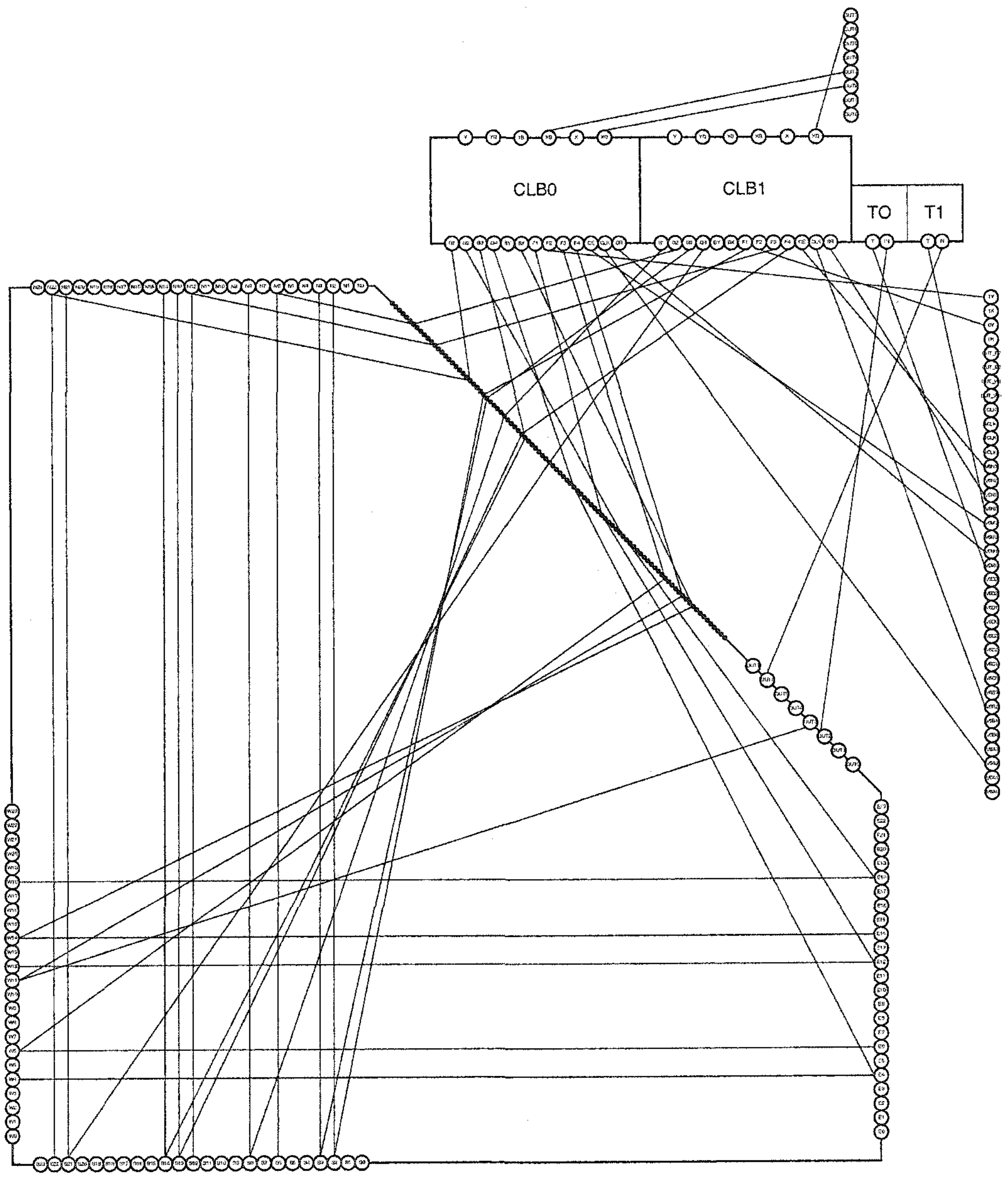


Figure D.22: Test Configuration 22

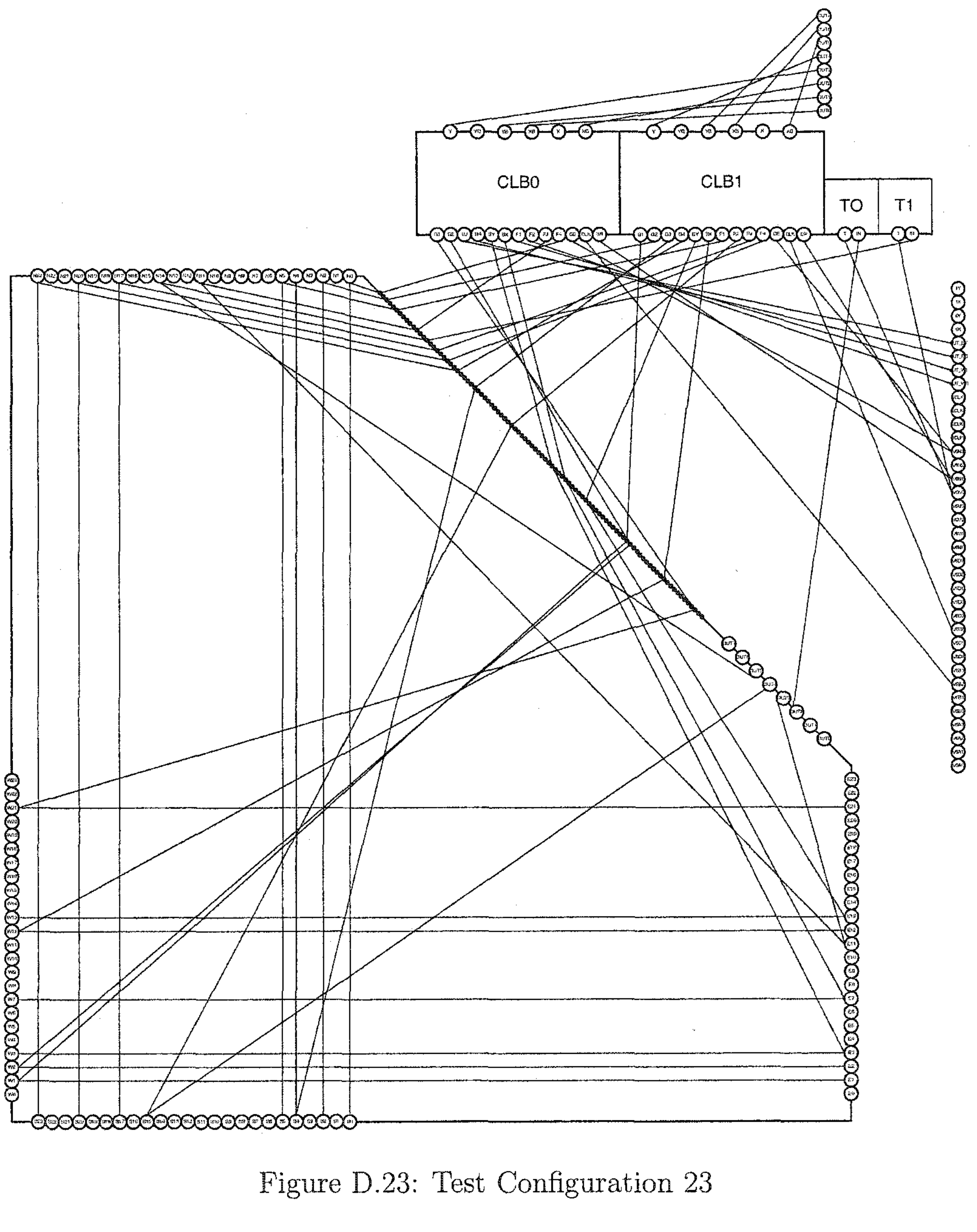


Figure D.23: Test Configuration 23

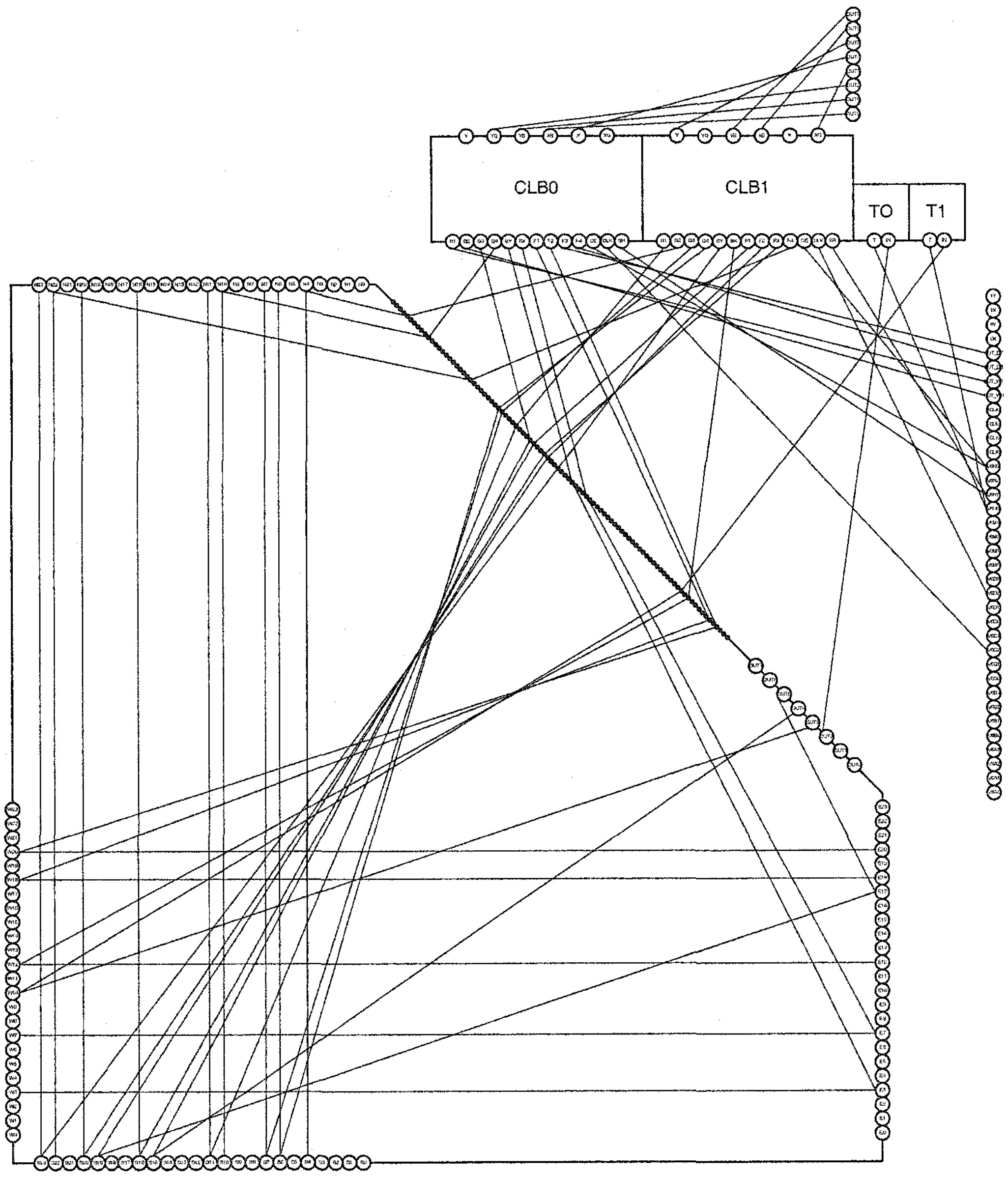


Figure D.24: Test Configuration 24

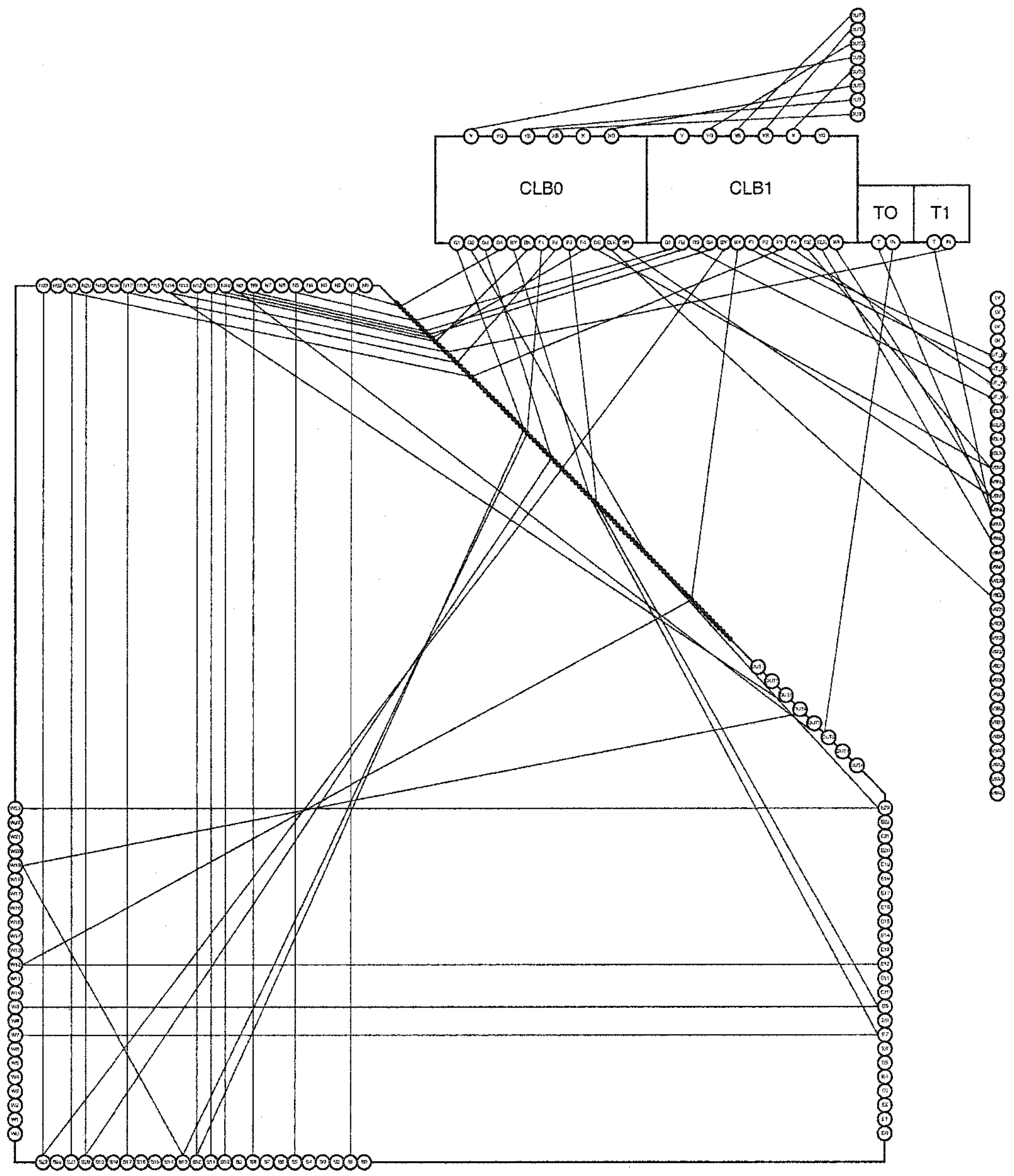


Figure D.25: Test Configuration 25

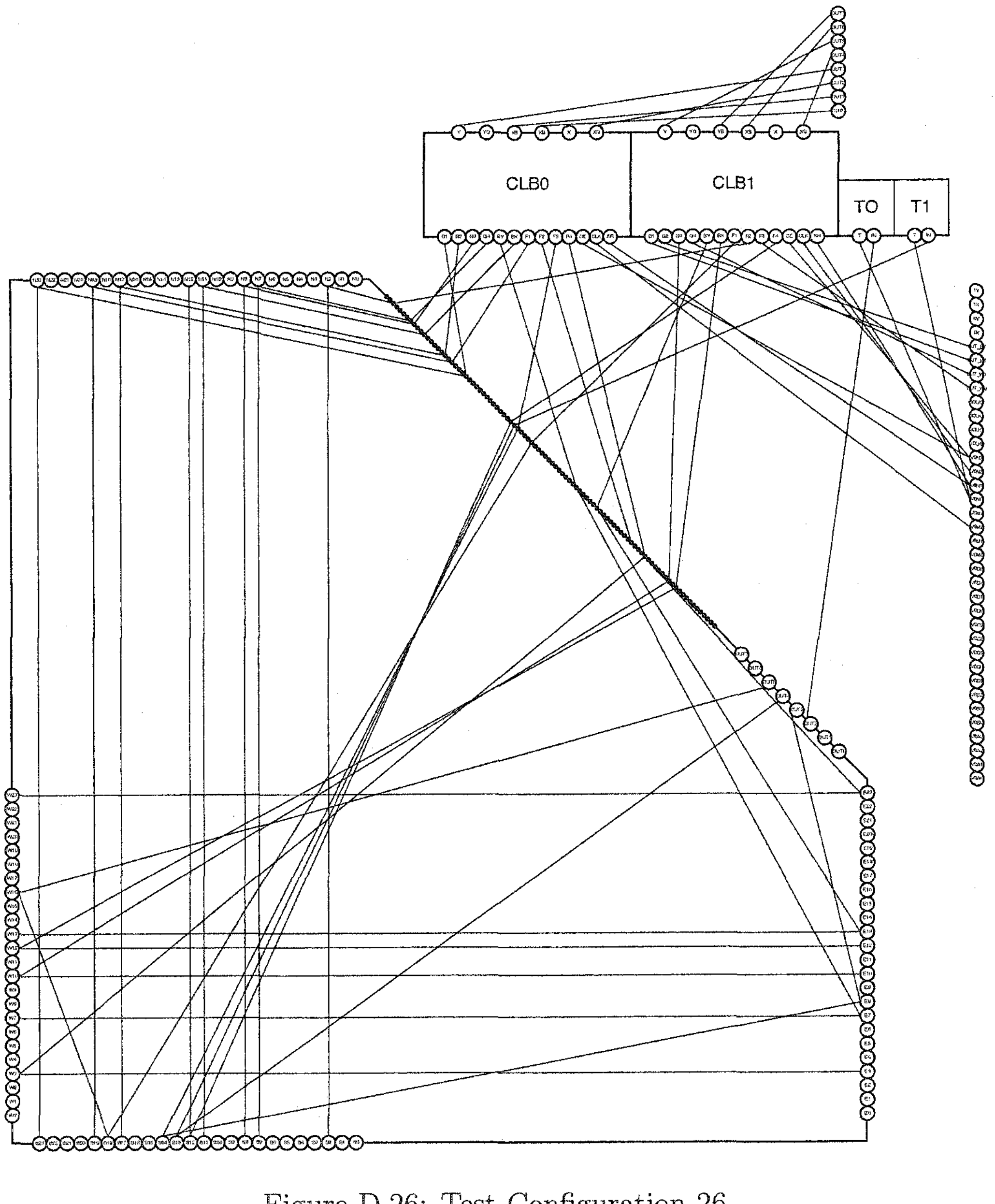


Figure D.26: Test Configuration 26