NASA SBIR 00-I Solicitation

Project Summary

Contract Number: NAS2-01049
Proposal Number: 24.01-8755
Proposal Title: Autonomously Self-Repairing Circuits

Summary (Limit 400 words):
The purpose of our Phase-I research was to develop a methodology for implementing *autonomously self-repairing circuits*, based on a novel self-configurable architecture called the Cell Matrix. In order for the implemented circuit to be able to repair itself, it must contain not only information about itself, but also the necessary machinery for testing the underlying hardware, detecting and isolating faults, and working around faulty regions. The proposed approach involved designing a "supercell" to perform all these tasks. Each supercell contains a "genome" which specifies the target circuit. Supercells test the hardware for faults, isolate faulty regions from fault-free ones, configure new supercells, determine how to re-implement the target circuit given the current faults, and ultimately wire themselves together to implement the target circuit. This entire process is started by supplying a single, fixed configuration string to the system.

In Phase-I, we developed the supercell methodology, including techniques for analyzing individual cells within the Cell Matrix, handling faulty cells, and configuring new supercells in fault-free regions. We determined how supercells should interact, to cooperatively form a parallel, distributed system which implements the target circuit in the presence of faults. We then designed each of the systems inside a supercell. These designs were then translated to a cell-level specification, resulting in a Cell Matrix configuration string. That string was then used to configure a simulated Cell Matrix, resulting in implementation of the proposed approach. We allowed faulty regions to be introduced into the matrix, and successfully demonstrated the fault handling features of the system.

All Phase-I objectives were satisfied: all systems worked perfectly. For all test circuits, the simulations confirmed that the supercells correctly implemented them. By changing the locations of faults within the matrix, the system was seen to re-implement the target circuit by routing around the faulty regions. Moreover, the parallel operation of the system was verified, i.e., multiple supercells tested and configured multiple regions simultaneously. Thus, system throughput (number of cells being operated on at a given time) actually increased as the supercell count grew. This is an extremely important feature for future reconfigurable systems with extremely high gate counts, e.g., nano-scale devices.

We believe these results justify Phase-II continuation. Phase-I has demonstrated the feasibility and innovativeness of the Cell Matrix for implementing self-repairing circuits, as well as the versatility of the Cell Matrix itself.

Company Name:Cell Matrix Corporation
Principal Investigator: Lisa J. K. Durbeck
Authorized Contact Person: Lisa J. K. Durbeck
Authorized Contact Email: ld@cellmatrix.com
COTR:
COTR Email:

Form Printed on 08-15-01 05:10

Autonomously Self-Repairing Circuits

Cell Matrix Corporation

15 August 2001

Submitted as the Final Technical Report for
NASA SBIR Phase-I Contract #NAS2-01049

SBIR Rights Notice (Mar 1994)

These SBIR data are furnished with SBIR rights under Contract No. NAS2-01049. For a period of 4 years after acceptance of all items to be delivered under this contract, the Government agrees to use these data for Government purposes only, and they shall not be disclosed outside the Government (including disclosure for procurement purposes) during such period without permission of the Contractor, except that, subject to the foregoing use and disclosure prohibitions, such data may be disclosed for use by support Contractors. After the aforesaid 4-year period the Government has a royalty-free license to use, and to authorize others to use on its behalf, these data for Government purposes, but is relieved of all disclosure prohibitions and assumes no liability for unauthorized use of these data by third parties. This Notice shall be affixed to any reproductions of these data, in whole or in part.

Table Of Contents

Abstract

A self-repairing system based on a novel self-configurable architecture is described. The self-repairing system implements a desired target circuit on a virtual intermediate layer composed of “supercells.” These supercells are able to perform all the tasks necessary to implement a target circuit on imperfect hardware, including analysis of the hardware for faults, isolation of faults, synthesis of supercells while avoiding faults, and differentiation of the supercell layer into the target circuit. Moreover, all these operations occur in response to a fixed configuration string that specifies the target circuit. For a given target circuit, the same string is always used, regardless of the location or nature of faults in the hardware. Thus, self-repair is achieved without external fault analysis or configuration string recompilation. This allows the system to autonomously self-repair in response to a single “repair” command.

1. Introduction

In Phase I of this project, we have designed a system for implementing self-repairing circuits, using a novel reconfigurable platform called the Cell Matrix™. The system, which is implemented **entirely** on a Cell Matrix (i.e., there is no external controller), is able to autonomously reconfigure itself (*self-configure*) to implement a desired target circuit. In the event of a hardware failure, the system can be given a single “repair” command, to which it responds by again reconfiguring itself, thereby re-implementing the target circuit . This self-configuration includes performing an analysis on the Cell Matrix hardware, to determine which regions of the hardware are functioning correctly and which have faults in them. The results of this analysis are used to control the placement and routing of the target circuit within the Cell Matrix. In general, as the location of faults changes, each re-implementation of the target circuit may differ from the previous, based on the new faults. However, the configuration string which is sent to the system is, for a given target circuit, always the same.

Note that this system is completely distributed. There is no centralized processing, nor are there critical elements or irreplaceable components. Most any fault pattern can be handled by the system, provided enough Cell Matrix hardware is available. Excepting the memory which holds and delivers the configuration string, the entire system—including the fault analysis and isolation circuits and the dynamic circuit synthesis—is implemented on the Cell Matrix. All components are themselves subject to the system’s fault management, thereby making the system extremely robust.

This robustness is achieved by organizing the Cell Matrix as a collection of *supercells*. Each supercell contains an identical coded description (called a *genome*) of the desired target circuit. The supercells perform 3 major tasks:

1. analysis of the Cell Matrix hardware for faults, followed by the building of *guard walls* around faulty regions of the Cell Matrix;
2. configuration of non-faulty regions of the Cell Matrix to implement new supercells; and, once a collection of supercells has been implemented,
3. implementation of the desired target circuit on top of the supercell collection.

In order for the system to operate autonomously, i.e., without intervention, the same configuration string must be usable in all situations, regardless of the location and number of faults in the hardware. In other words, a single configuration string must be developed for a particular target circuit, independent of any hardware faults which may exist in the Cell Matrix platform. This imposes the minimal requirements for fault-free components, i.e., there must be a memory for storing the configuration string, and a mechanism for feeding this string to certain cells along an edge of the Cell Matrix. As long as those components work, the rest of the system can handle faults inside the Cell Matrix.

Section 2 will present background information on the Cell Matrix architecture. Section 3 describes the details of the supercell approach to self-repairing circuit implementation. Section 4 provides the details of the Cell Matrix implementation of a supercell, including details of how the supercell is used. Section 5 describes the results of a number of experiments, and Section 6 summarizes the results of this project.

2. Background

This work is based on a custom reconfigurable system called a Cell Matrix. A Cell Matrix is a hardware system composed of a number of reconfigurable elements (cells) connected in a regular fashion. It is similar to a Field-Programmable Gate Array (FPGA), but differs significantly in the following ways:

- The Cell Matrix is extremely fine grained. Individual cells implement small scale functions, such as simple logic or one-bit addition. Connections between cells are created by configuring cells to act as wires. This provides complete control over the configuration of circuits implemented on the hardware.
- The Cell Matrix is an infinitely-scalable architecture. All connections take place only between immediate neighbors, according to a pre-defined, system-wide topology. This has tremendous implications for future manufacturing technologies such as nanotechnology (Dur 2001).
- The Cell Matrix is *self-configurable*. Any cell within the system has the ability to read and write the configuration information in any of its neighboring cells. This means that configuration operations can be based on analysis of local information, including both pure data as well as cell configuration information. It also means that multiple cells can be configured in parallel, since there is no single external controller, nor is there a single, fixed-size communication channel through which configuration information must be sent.

The Cell Matrix is an inherently fault-resistant architecture, owing to very limited signal exchange among the reconfigurable elements. Despite this limited connectivity, the architecture is rich in its flexibility, and can be used to implement any standard digital circuit. Moreover, because the reconfigurable elements of the Cell Matrix are able to read and write other elements' configuration information, the system supports **self-organizing, self-configuring** circuits. This is a feature lacking in almost all other reconfigurable devices currently available. By using self-configurability, it is possible to implement dynamic circuits which change **autonomously** over time. One application of such circuits is to design systems which can reconfigure themselves after new faults have occurred in the hardware. Such systems are then said to be *self-repairing*.

2.1 Cell Behavior

The atomic unit of a Cell Matrix is called a cell. All cells within a Cell Matrix are identical to each other. Figure 2.1 shows an example of a single Cell Matrix cell. Such cells are connected to other cells using a fixed connection scheme, so that outputs from a cell are continuously provided to the inputs of neighboring cells, based on a fixed, system-wide topology, as shown in Figure 2.2. By properly configuring cells in such an interconnected set, digital circuits can be implemented.

Throughout this work, we concentrate only on four-sided cells, arranged in a two-dimensional array so that each cell has four adjacent neighbors[1]. Each cell has two inputs, labeled C and D, and two outputs (also labeled C and D) on each of its sides.

Each cell operates at any given moment in one of two modes: D-mode and C-mode. One cell's mode is effectively independent of the mode of other cells around it. It's mode is determined entirely by its C inputs. If any C inputs to a cell are 1, the cell is in C-mode; otherwise it is in D-mode.

2.2 D-Mode Behavior

In their simplest mode (called "D-Mode"), cells act as pure combinatorial processing elements. Cells accept inputs from neighboring cells, process them based on a per-cell *configuration memory* or *truth table*, and produce outputs accordingly.

[1] Note that cells on the edge of the entire matrix have only three neighbors, while cells in corners have only two neighbors. Such edge cases are important, as the unconnected sides can be made available to external connections, thereby allowing external control and analysis of the Cell Matrix.

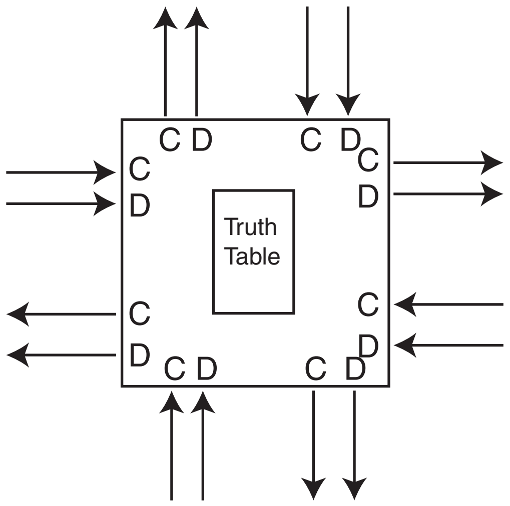


Figure 2.1
Basic Cell Matrx Cell Layout
Data enters and exits through the D lines
Cell's Mode is onctrolled by the C inputs
C outputs affect neighboring cells' modes.

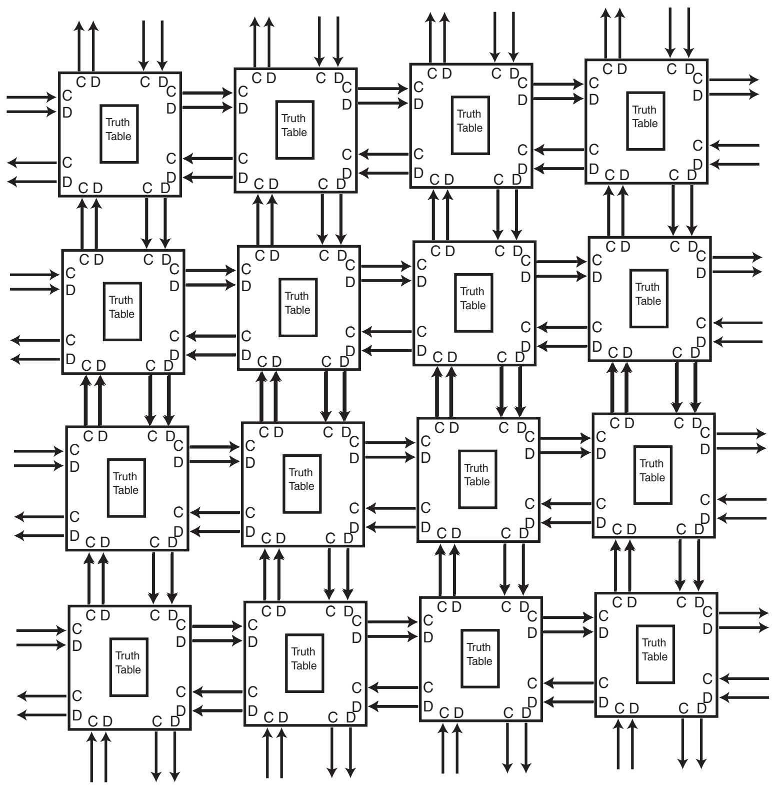


Figure 2.2
4x4 Collection of Cell Matrix Cells

Cells are typically configured to perform small-scale operations such as 2-, 3- or 4-input logical operations, multiplexing, one-bit addition, and so on. Cells are also used to implement wires, which pass data from a cell on one side to a cell on another, thereby allowing non-adjacent cells to exchange information. Collections of cells can be used to implement larger circuits, such as arithmetic units, complex logic, and so on. Collections of cells can also be used to implement sequential circuits, such as flip flops, memories, counters, and state machines. Therefore, a Cell Matrix can be used to implement traditional digital circuits, following standard digital circuit design methodologies.

2.3 C-Mode Behavior

The more unique aspects of a Cell Matrix arise when cells are operating in C-mode. A cell is placed in C-mode when any of its C inputs are 1. Assuming only one of a cell's C inputs is asserted, we may call the side where Cin=1 the "active side." A C-mode cell's behavior can then be described as follows:

1. the "last" bit (based on some pre-defined ordering) of the cell's configuration memory is presented to the D output of the active side;
2. on the rising edge of a system-wide clock, the D input on the active side is sampled; and
3. on the falling edge of the system-wide clock:

- the sampled D input is shifted into the beginning of the C-mode cell's configuration memory;
- the entire configuration memory is shifted by one bit; and
- the last bit of the configuration memory (which was formerly presented on the active side's D output) is lost..

This sequence of steps means that if cell X places cell Y into C-mode, cell X can read and write cell Y's configuration memory, via X's D inputs and outputs.

Since a cell's configuration memory completely defines the cell's D-mode behavior, the above mechanism allows a cell to analyze and modify the behavior of neighboring cells. Using the C-mode of cells, circuits can be built which process *circuit configuration information* in the same way that other circuits process numeric or logical data. As a result, circuits can be built which synthesize other circuits, which self-replicate, or which otherwise dynamically modify themselves.

Further details of the Cell Matrix can be found in the original Cell Matrix architecture patent (Mac1 1999), as well as (Mac3 1999) and the Company's Website (CMC).

3. Approach

Implementing a digital circuit on a reconfigurable platform P is normally a straightforward process. The circuit can be viewed as a graph, whose nodes are primitive elements (such as AND gates, flip flops, etc.), and whose edges represent connections among those elements. This graph is translated to a configuration string (a process normally called *place and route*), which, when sent to P, configures P's elements and interconnections so as to implement the desired circuit.

3.1 Circuit Configuration in the Presence of Faults

The process is more complicated if the hardware contains faults, such as elements which do not function properly, or interconnections which do not transmit data properly. If the locations of such faults can be determined, as well as the locations of any other elements or connections which do not behave properly because of those faults, then the place and route process can simply avoid those locations, treating them as unavailable resources.

Things are not as easy if the place and route process is not on-board the remote system. For example, if a system on a satellite has failed, it may not be practical to analyze the hardware from a ground station, re-run the place and route, and upload a new configuration string. Rather, it may be desirable to send a simple "repair" command and have the system reconfigure itself. This would permit the use of a single configuration string to configure the system at any time, regardless of where the faults may exist. The system would then only require a memory to store the configuration string, and a mechanism for delivering that string to the reconfigurable platform.

Note that adding an external fault locating system and a place and route processor is not an ideal solution, since these additional pieces are themselves subject to possible failures. Therefore, it is desirable that the reconfigurable system should not only implement the desired target circuit, but also whatever circuitry is necessary for determining *how* to implement the desired circuit. Figure 3.1a illustrates this situation. On reconfigurable platform P, there is not only the desired target circuit, but also the fault locator and the place and route algorithm. The location and exact

configuration of each of these elements may change, depending on the presence of faults within P, as shown in Figures 3.1b and 3.1c.

3.2 Circuit Configuration Using a Cell Matrix

The Cell Matrix possesses a natural self-duality which makes this type of system implementation possible. It is, in fact, exactly these types of self-referential systems for which the Cell Matrix was designed.

To implement a perfectly functioning circuit on a faulty hardware platform, four things must be done:

1. faulty regions of the hardware must be detected;
2. these faulty regions must somehow be isolated from other regions of the hardware, to prevent their malfunction from spreading;
3. the elements of the desired circuit must be implemented using only the good areas of the hardware; and
4. these elements must be interconnected, again using only the good areas of the hardware.

Moreover, each of these steps should be performed by the system itself, without any external control, other than the supplying of a fixed configuration string.

3.3 Supercell Approach to Circuit Configuration

In order to achieve these goals, we have implemented what is effectively an intermediate layer between the Cell Matrix hardware and the desired target circuit. This layer, called the *Supercell Layer*, is responsible for analyzing the underlying hardware, detecting and isolating faults, and then implementing the target circuit on top of itself. It is composed of higher-order cells called *supercells*, themselves composed of Cell Matrix cells. Figure 3.2 illustrates this conceptual organization. Note that the hierarchical view in Figure 3.2 is an organizational hierarchy, not a physical one. The supercell layer and target circuit layer do **not** consist of physically distinct hardware. The entire system is a single Cell Matrix, composed of perfectly identical cells.

3.4 Supercell Functional Behavior

The supercells perform all necessary functions for fault-free implementation of the target circuit. These functions will be described below in the following sections:

- 3.4.1 Fault Detection
- 3.4.2 Parallel Operation of Supercells
- 3.4.3 Loop Avoidance-The Link/Lock Network
- 3.4.4 Parallel Configuration of Supercells
- 3.4.5 The Supercell's Functional Block
- 3.4.6 Circuit Genome
- 3.4.7 Static ID Assignment
- 3.4.8 Dynamic ID Assignment
- 3.4.9 Traceback Generation and Parsing
- 3.4.10 Path Synthesis and the Steering Block

3.4.1 Fault Detection

The first task of a supercell is to locate faults within the hardware. This is done by testing each cell inside the Cell Matrix, via a series of tests which exercise the cell's basic functions (configurability, processing of inputs, delivery of outputs). Multiple regions, each the size of a supercell, are tested in parallel, so that they may be subsequently configured to act as new supercells.

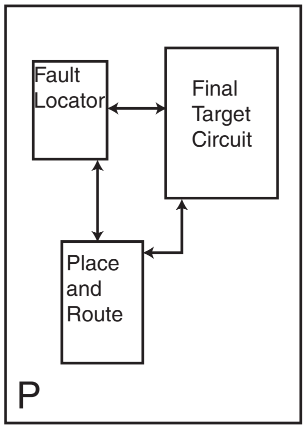


Figure 3.1(a)
Possible Implementation
of Self-Repairing
Circuit

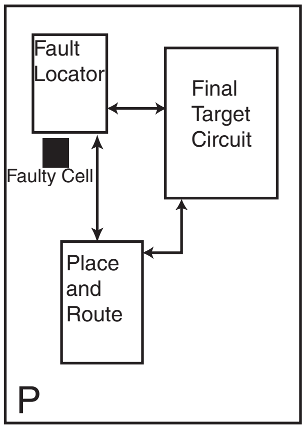


Figure 3.1(b)
Same System with
a Single Faulty Cell.
Fault Locator has
moved up.

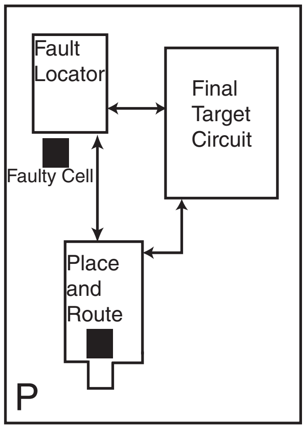


Figure 3.1(c)
Same System with
Multiple Faults.
Place and Route
block has been
reshaped.

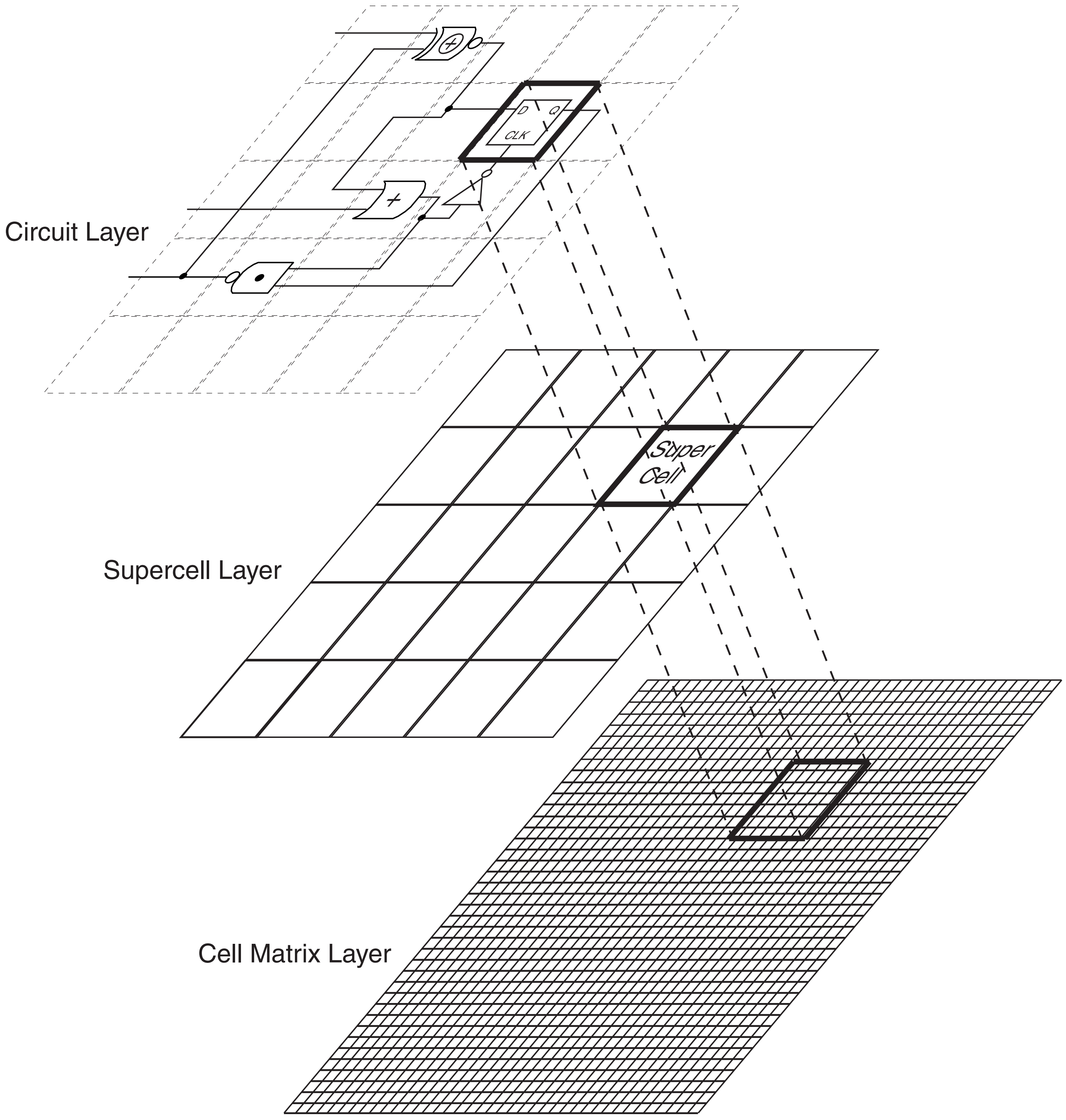


Figure 3.2
Hierarchical View of Supercell-Based Circuit Layout

Since a supercell is implemented on a Cell Matrix, the cells of the supercell have the ability to read and write neighboring cells' configuration information. Note however that *direct* control is only available over cells that it is physically connected to, i.e., the immediately adjacent ones. However, by carefully orchestrating the configuration of adjacent cells, they can be caused to convey configuration information to *their* immediate neighbors, and so on, as shown in the progression from Figure 3.3(a) to Figure 3.3(b). Such configurations of cells are called *wires*[2], and they are used extensively to allow a supercell to gain access to non-adjacent cells.

At any given time, fault detection is performed to only one side of a supercell, called the *current configuration side*. In Figure 3.4, the supercell is testing cells to its East. Within this region under test (RUT), wires are built to allow access to each cell. Cells within the RUT are accessed from all four sides (3 sides for cells along an edge, two sides for cells in a corner). Figures 3.5a-3.5f show how this multi-side access is achieved. Note that for each configuration of the test wire, two cells are being tested, each from two sides. Following this, the wire is extended one cell to the East, and the four tests are repeated (Figure 3.5f). At the end of the row, the wire is moved down on row, and the sequence repeats. This test ordering ensures that all cells within the RUT are tested from all internal side, while minimizing the total number of configuration operations needed to test all cells within the RUT.

Once all cells within a column of the RUT have been tested, the wire is rebuilt so that cells in the next column to the left will be tested, and so on. Special sequences are used for the edge cases.

From each side, the cell under test (CUT) is tested with the following two tests[3]:

1. the cell is configured as an inverter, and a test pattern of alternating 1s and 0s is sent to the cell. This tests the ability of the cell to implement a non-zero function, thus detecting faults in the configuration logic. It also forces the inputs and outputs to respond to and generate both 1s and 0s, thereby detecting stuck-at faults in the cell's D inputs and outputs.
2. the cell's configuration memory is loaded with an irregular test pattern, which is then read out of the cell's configuration memory. This tests the configuration memory's ability to store a given test pattern. Since the configuration memory is implemented as a shift register, this detects stuck-at faults in the memory itself, as well as any faults in the control logic responsible for shifting the memory, detecting the mode changes of the cell, and so on.

To perform each test, a GO signal is sent to the supercell, along with a test pattern and an expected return pattern. The GO signal indicates that the test pattern should be sent along the wire to the CUT. The output X of the CUT is transmitted back through the wire, and compared to the expected return pattern Y. Any difference between X and Y indicates a failure in the RUT, possibly in the CUT, or possibly in some other cell used to build the wire. Regardless of the cause, if $X \neq Y$, it indicates a problem in the behavior of some cell or cells within the RUT, and the RUT is deemed unusable. Note that because supercells are only placed in regions which have already been determined to be fault-free, test failures can be taken to indicate failures in the region under test, as opposed to failures in the circuit performing the test.

It is important to manage timing issues in comparing X and Y, since Y is a stored pattern, and is thus available before X, which is a generated pattern. The timing is synchronized to the system-wide clock, and comparisons are only made on the rising edge of that clock, while the patterns X and Y change on the falling edge.

Once a fault is detected, the supercell activates a *guard wall* along the side from which it is testing the RUT. This guard wall, which will be further explained in section 4, makes it impossible for any cell in the RUT to affect any cell inside the supercell. Note that the guard wall is actually part of the supercell itself, and therefore can be assumed to be composed of non-faulty cells. Figure 3.6 shows a supercell which has activated one of its guard walls following a test failure. Eventually, other supercells (on other sides of the RUT) will also detect the fault, and

[2] These configurations are disclosed in US Patent Application #09/702,574, filed 31 Oct 2000.

[3] Note that it is easy to add other tests and test patterns as well. For example, if hardware analysis revealed that a common failure mode was the shorting together of bits 8 locations apart in the configuration memory, a test pattern of 0000000011111111… could be used.

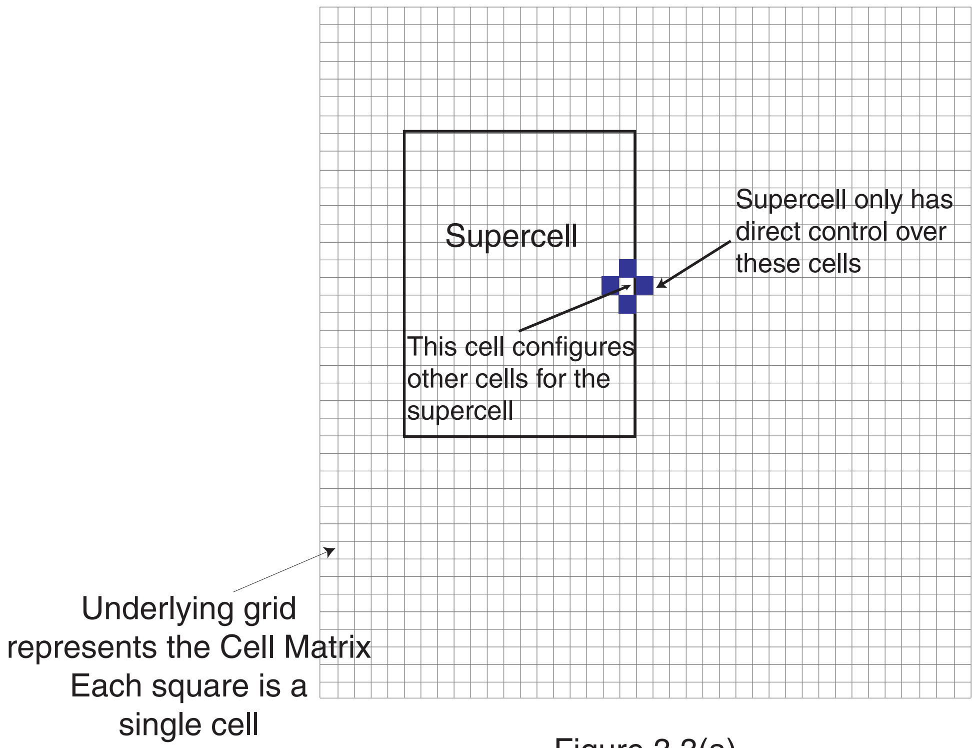

Supercell
Supercell only has direct control over these cells
This cell configures other cells for the supercell
Underlying grid represents the Cell Matrix Each square is a single cell

Figure 3.3(a)
Supercell's direct control is normally limited

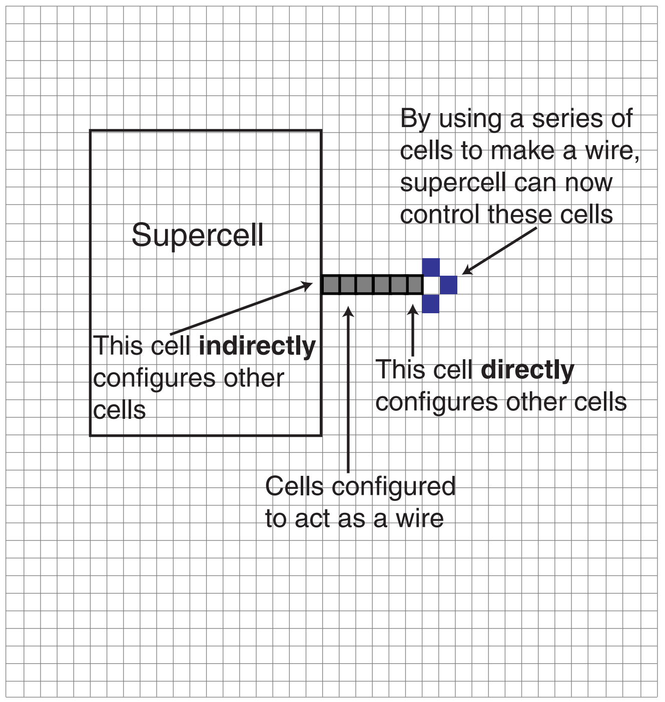

By using a series of cells to make a wire, supercell can now control these cells
Supercell
This cell **indirectly** configures other cells
This cell **directly** configures other cells
Cells configured to act as a wire

Figure 3.3(b)
6-Cell Wire used to control non-adjacent cells

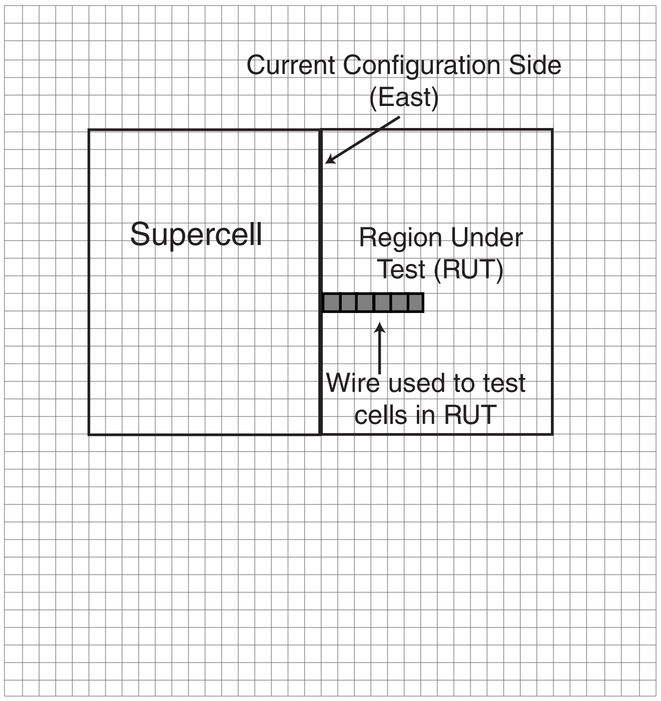


Figure 3.4
Supercell builds wires to test cells
in the Region Under Test

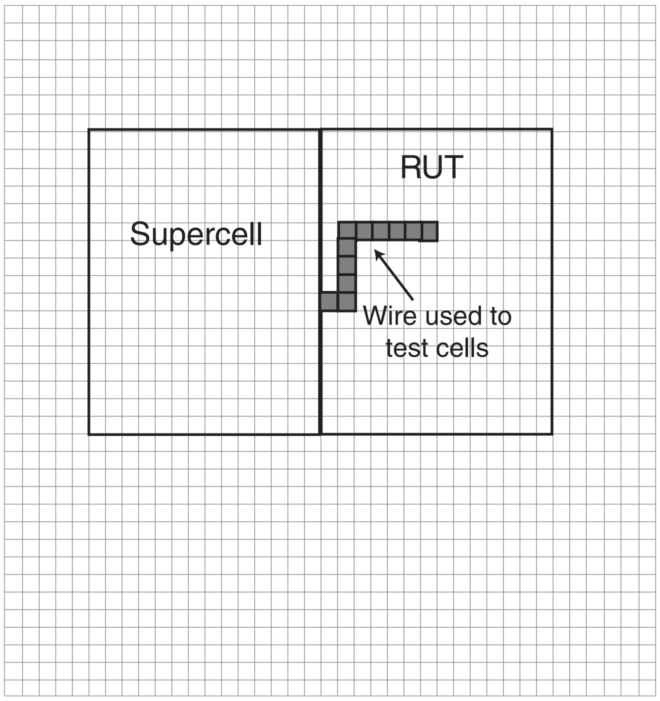


Figure 3.5(a)
Supercell has built a wire into the RUT

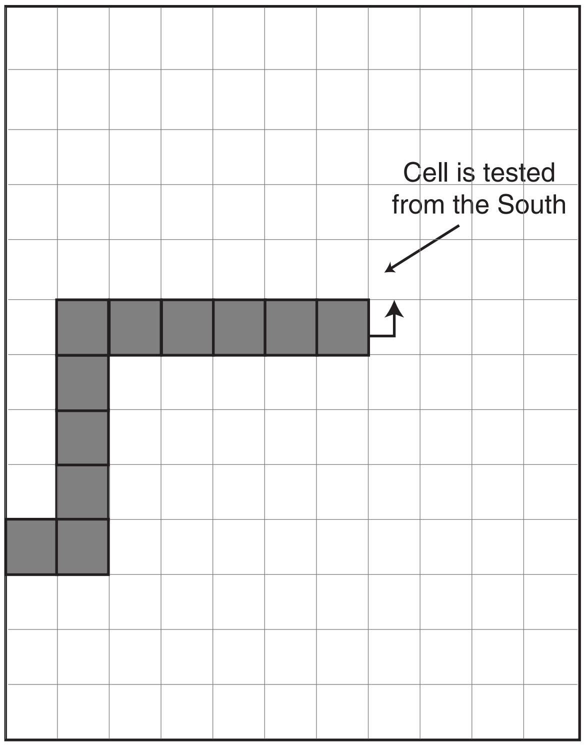


Figure 3.5(b)
One cell is being tested from the South
The supercell and part of the RUT are not displayed in this figure

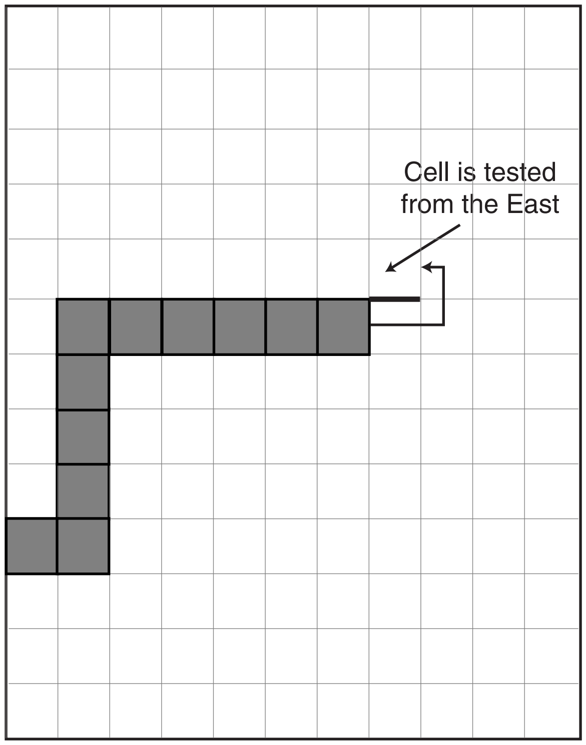


Figure 3.5(c)
Same cell is now being tested from the East
The dark line indicates tests which have already been performed

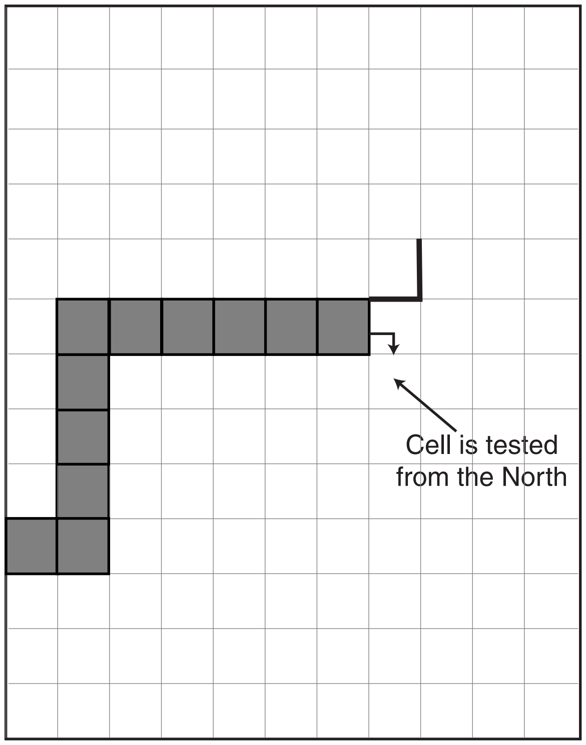


Figure 3.5(d)
A different cell is now tested from the North

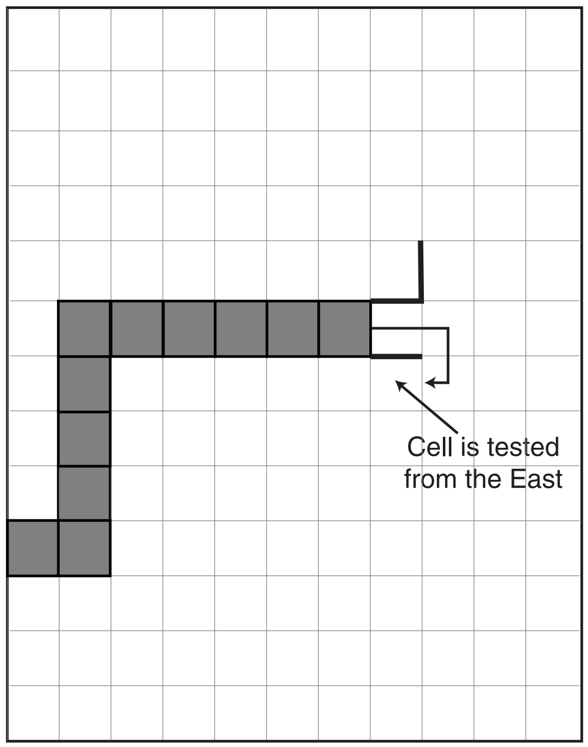


Figure 3.5(e)
Same cell is now tested from the East

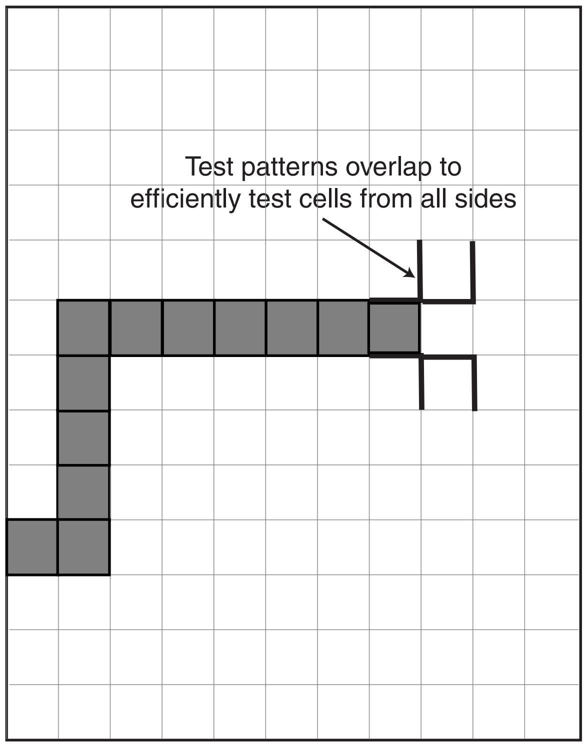


Figure 3.5(f)
Wire has been extended one cell to the East
and all four tests have been repeated

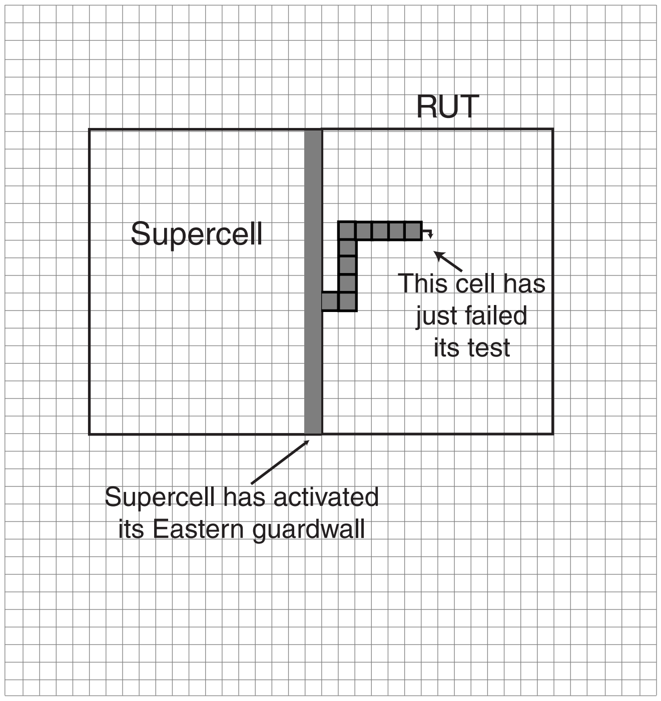

RUT
Supercell
This cell has just failed its test
Supercell has activated its Eastern guardwall

Figure 3.6
Supercell has detected a faulty cell in the RUT
and has responded by activating one of its guardwalls

activate their own guard walls. In this way, a collection of supercells will form a continuous ring of guard walls around the entire faulty area.

3.4.2 Parallel Operation of Supercells

The above description describes the behavior of a single supercells testing a single RUT. However, in a typical setup, there will be many supercells, all operating in parallel, each testing its own RUT. Figure 3.7 shows a typical situation. Again assuming regions to the East are being tested, there are five RUTs in this figure. Note that the test pattern, expected return pattern, GO command and wire-building commands are being sent to only one supercell (the upper left in this example). Each supercell which receives these inputs transmits them to *each* of its four sides in one of three ways:

- If the supercell's side is adjacent to another supercell, it transmits them as simple data. The receiving supercell accepts them as data, and transmits them according to these same rules.
- If the supercell's side is **not** adjacent to another supercell, and that side has been designated as the current configuration side (East in this example), then they are treated as live commands: the wire-building commands are used to configure cells to create wires, and if the GO command is asserted, the test pattern is sent through the wire, with the return pattern compared to the expected return as described above.
- If the supercell's side is not adjacent to another supercell, and that side has **not** been designated as the current configuration side, then nothing is transmitted through that side.

In this way, test patterns and commands can be sent to a single supercell, transmitted to the perimeter of the entire supercell collection, and new regions can be added in an orderly fashion. Thus, as the collection of supercells grows, so does the number of RUTs being tested simultaneously. **This is one key advantage of the Cell Matrix approach-the ability to perform fault detection on a number of regions in parallel.**

By performing the above tests while changing the current configuration side to each of North, South, West and East, faults end up completely isolated, as shown in Figures 3.8a-3.8d. Since cells are tested from all sides, each supercell which tests the RUT containing the fault will detect the failure, and activate its guard wall on the corresponding side. The result is a closed ring of guard walls, completely isolating the faulty cell from the properly-functioning supercells. Note that this also works for larger faults which span more than one supercell region.

3.4.3 Loop Avoidance-The Link/Lock Network

There is a technical detail to the above description. Since supercells transmit data to all their sides, there is a strong potential for transmission loops to form. If supercell A receives a GO command, and passes it to adjacent supercell B to its East, B would immediately turn around and transmit this back to supercell A. A and B have now formed a loop, and the GO command is trapped inside that loop. If the original transmission to A is terminated, A will still be receiving GO from B, and will continue to retransmit it back to B. Such a state is unrecoverable.

To avoid this loop problem, everything is built on top of a dynamically-built, loop-free network called a *Link/Lock (L2) Network*. The L2 network is assembled using a handshake between supercells consisting of two signals:

- a LINK output from supercell X to supercell Y means X is on the network, and Y should add itself to the network
- a LOCK output from supercell X means X is already on the network, and therefore no new LINK signals should be sent to X

The L2 network can be initiated from any supercell, which transmits "link" commands to all adjacent supercells which are not already on the network. Any supercell receiving a link command joins the supercell network, and in turn transmits link commands to each of *its* neighboring supercells which is not already on the network. When a supercell joins the network, it begins transmitting "lock" signals to all its sides, indicating that it is already on the network. In this way, a supercell X can tell if an adjacent supercell Y is already on the network by checking Y's lock output. If lock is not asserted, X can assert its LINK output, causing Y to join the network.

Once assembled, signals such as wire building commands can be piggybacked on top of the L2 network's link signals to achieve loop-free broadcasting of signals throughout the connected set of supercells. Note that signals can also be broadcast *from all* supercells back to the originator of the L2 network by transmitting in the reverse direction of the link signals.

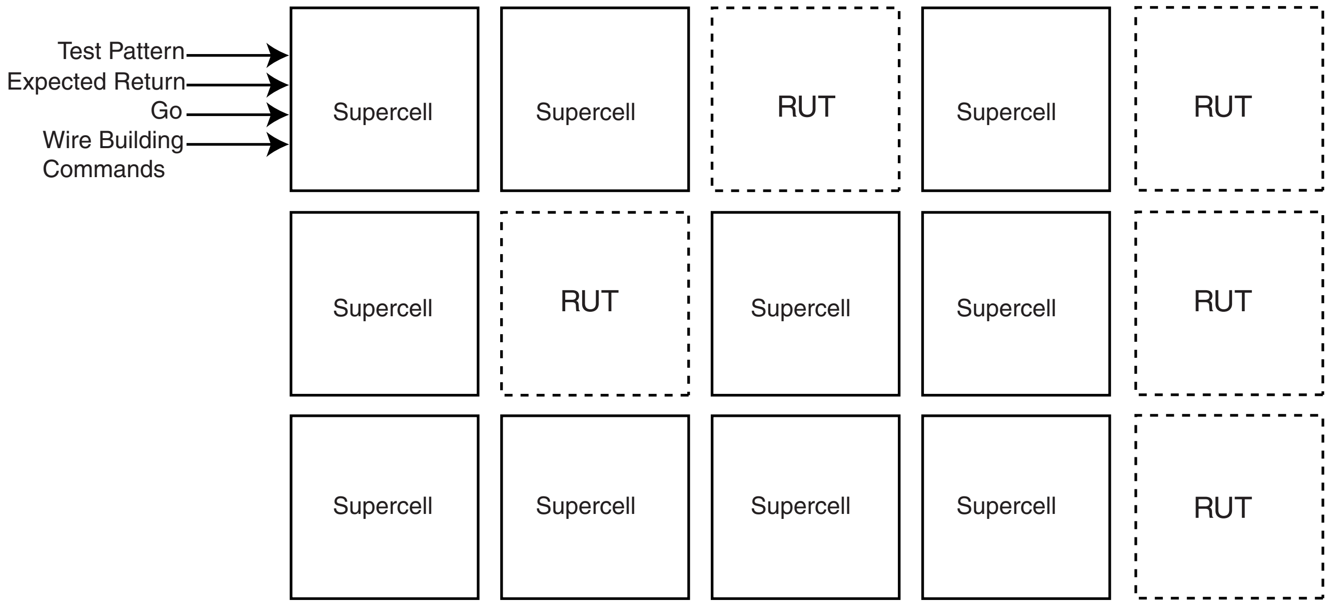


Figure 3.7
Set of supercells performing a parallel test
of five Regions (RUTs)
Only one supercell is sent the test-related commands.
That supercell transmits those commands to all other supercells.

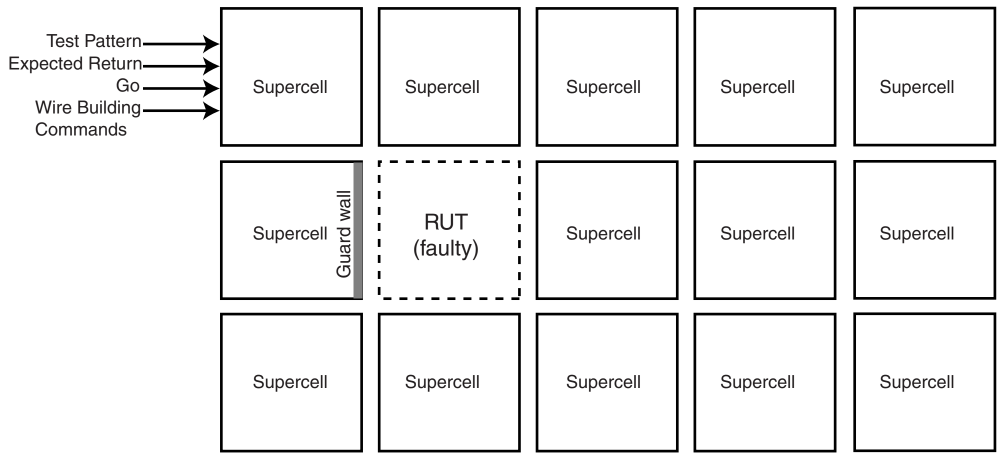


Figure 3.8(a)
RUTs which passed the Eastern test are now configured as supercells. One RUT contains a fault, causing the testing supercell to activate its Eastern guard wall (shown in gray).

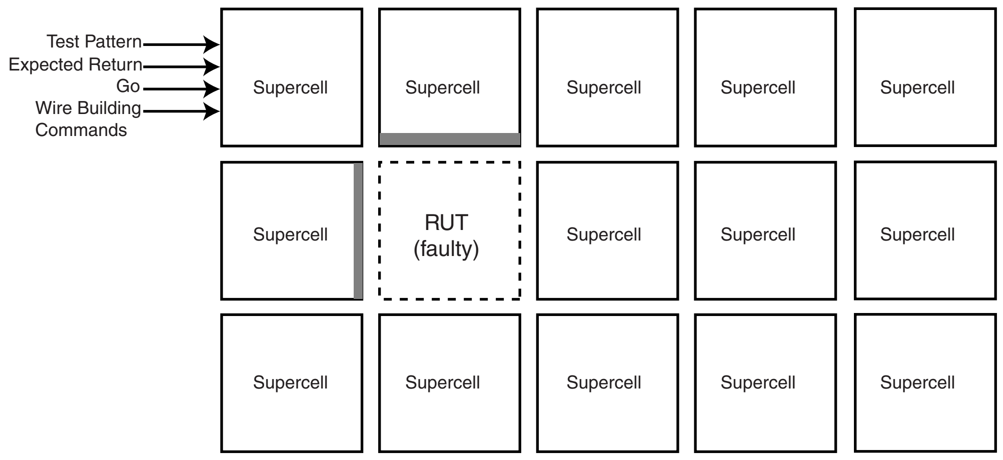


Figure 3.8(b)
Following a Southern test sequence, one supercell has now activated its Southern guard wall.
New RUTs are not shown.

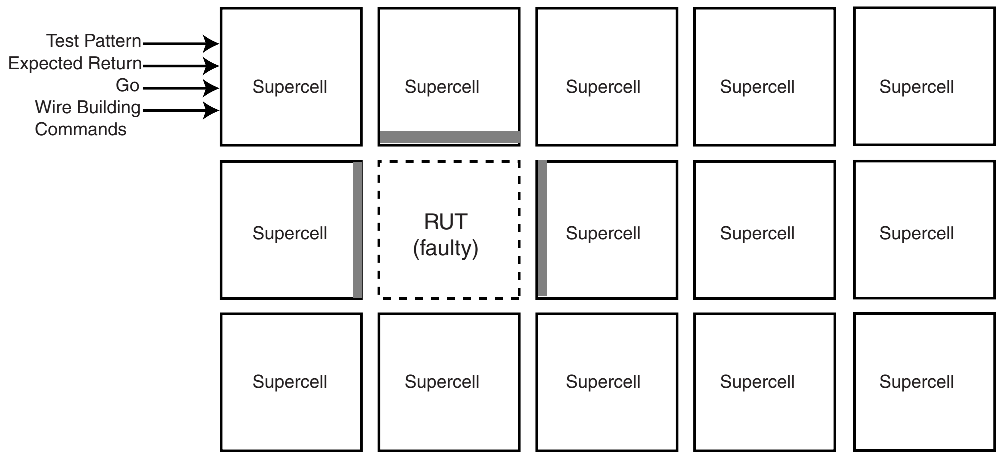


Figure 3.8(c)
Result of Western test sequence

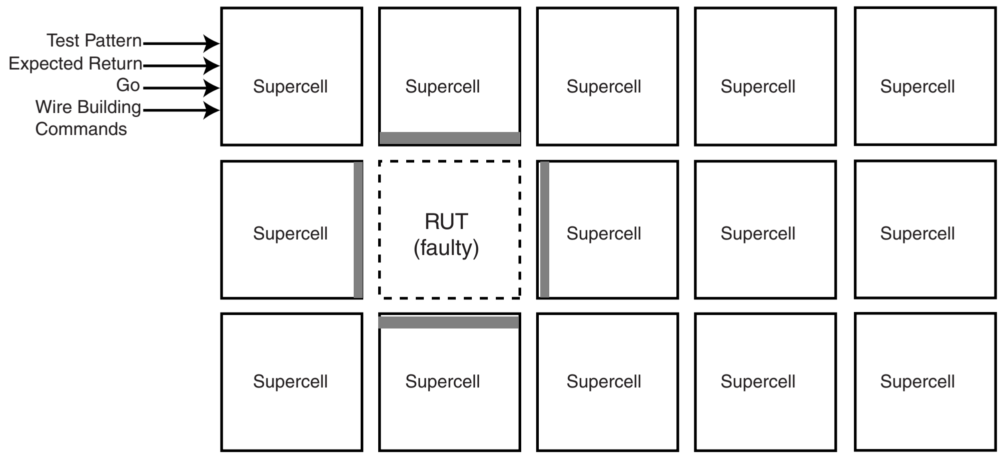


Figure 3.8(d)
Result of Northern test sequence.
Faulty RUT has now been completely
isolated by surrounding guard walls.

Figure 3.9 shows an example of the building of an L2 network. In this example, the supercell marked "*" is the initiator of the L2 network. Each arrow represents a link signal; the number next to an arrow shows the step at which the link signal was asserted. By building the L2 network in steps like this, the effect is that shortest paths are constructed, while longer paths are blocked by the LOCK signals. For example, the solid line shows a pathway (via the L2 network) from the initiator "*" to the supercell marked "+." This pathway is 5 supercells in length. There is another possible pathway, as shown by the dashed line. However, that pathway (which would have a length of 7) terminates in the supercell marked "-." When the L2 construction reaches node "-" (at step 6), node "+" has already joined the L2 network, and is thus asserting a LOCK signal to node "-." This is a direct consequence of the shorter path length of the solid path vs. the dashed path.

3.4.4 Parallel Configuration of Supercells
In addition to performing testing of the Cell Matrix in parallel, there is another more basic task which the collection of supercells must perform in parallel, and that is the configuration of new supercells. Using the techniques described above (3.4.1-3.4.3), an existing set of supercells can transmit configuration information to all perimeter supercells in some fixed direction. As is the case with fault testing, at any given time, configuration of new supercells occurs on only one side of the current supercell perimeter. In Figure 3.10, the indicated supercells are transmitting configuration information to the East. Non-perimeter supercells pass this information to adjacent supercells, while perimeter supercells use this information to build new supercells (by configuring Cell Matrix cells near their edge). The cells in regions marked with "*" are being configured to act as new supercells. As in the case of fault testing, the number of supercells being built in parallel increases as the size of the supercell network increases.

Note that as a new supercell is being constructed, the cells which will make up its edges may already be receiving signals from other adjacent supercells. For example, in Figure 3.11, three supercells have been constructed, and a fourth region ("*") is being configured as a new supercell. While "*" is being configured, SC2 and SC3 are sending Link/Lock signals and configuration information to the "*" region. The supercells have been carefully designed so that, as they are being configured, this always-present incoming information does not cause them to malfunction. Figure 3.12 shows how this is handled (again concentrating on building to the East). The incoming LOCK signal is inverted in the leftmost column of each supercell, and then inverted again inside the supercell. Since the leftmost column of cells is configured last (that's how the supercell configuration mechanism was written), the internal LOCK signal will be a one (inverted 0) until the leftmost column is configured. Because of this, the newly-forming supercell will not join the L2 network until the leftmost column has been built, at which point the internal circuitry for proper Link/Lock arbitration will have already been built. Similar control is applied for building to the North, West and South.

As soon as the configuration of cells within the "*" region is completed, the new supercell in that region will join the L2 network, and commence relaying and/or executing any commands which are sent to it.

Further details of how the configuration information is generated will be described in Section 4.

3.4.5 The Supercell's Functional Block
In implementing the final desired circuit on top of the supercell network (as in Figure 3.2), each node of the circuit will be implemented in a single supercell. For example, the circuit in Figure 3.13 contains 5 nodes. Each of these nodes will be implemented in a single supercell.

Accordingly, there must be some way to indicate what function each supercell is to perform in the final circuit. This per-supercell function is called the supercell's *functional block*. The functional block is the piece of the supercell which is incorporated in the final target circuit's implementation.

In this proof-of-concept version of the supercell, we have used an extremely simple functional block, consisting of only three Cell Matrix cells. In a more practical version, the functional block would be much larger, to allow functional units on the scale of state machines, larger memories, digital filters, and so on, instead of only simple gates and small-scale devices. However, regardless of the size of the functional blocks, their implementation and use is consistent with what is described here.

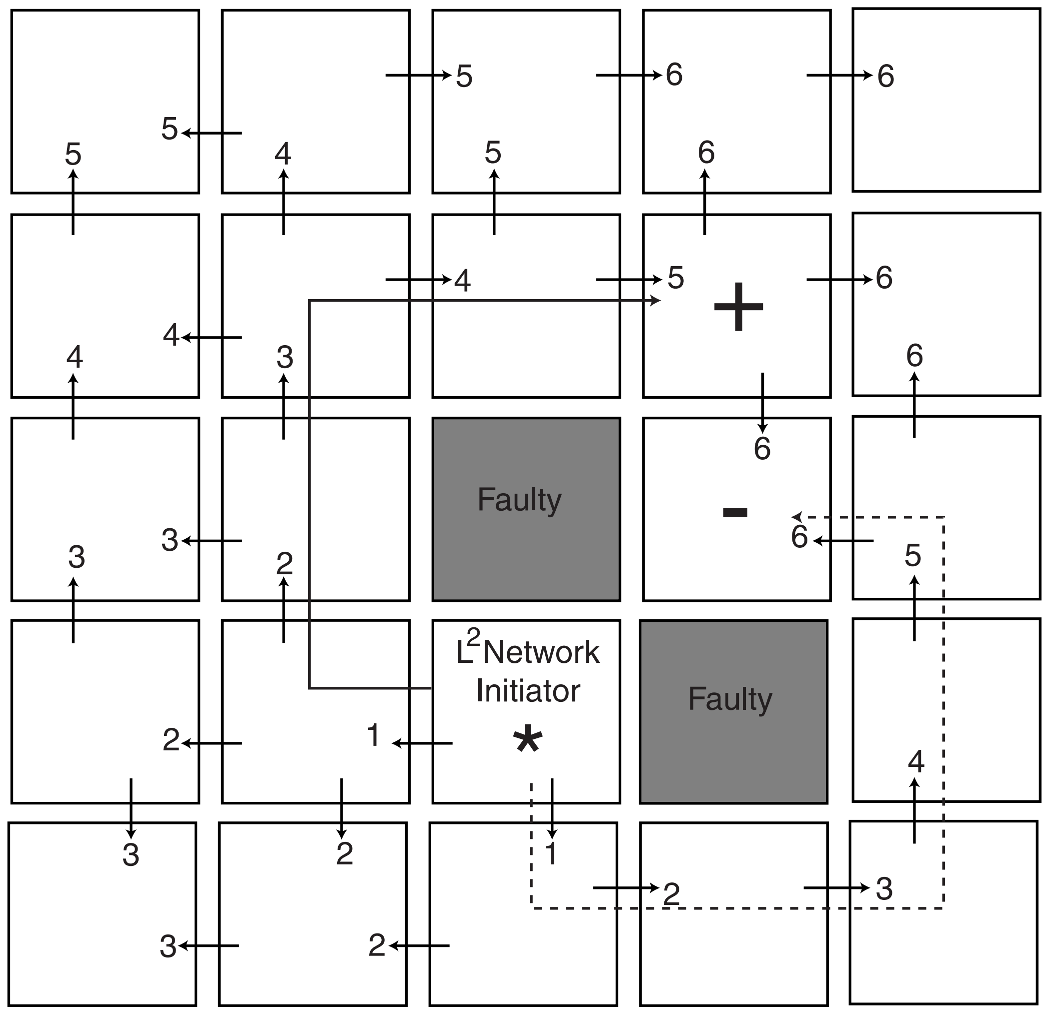


Figure 3.9
Sample Link/Lock Network
Supercell marked "*" is network initiator. Arrows indicate LINK outputs.
Numbers at the head of an arrow indicates the step at which that LINK output is activated.
Solid line shows resulting path from "*" to "+". Dashed line shows alternate path,
which is longer, and hence ends in "-"

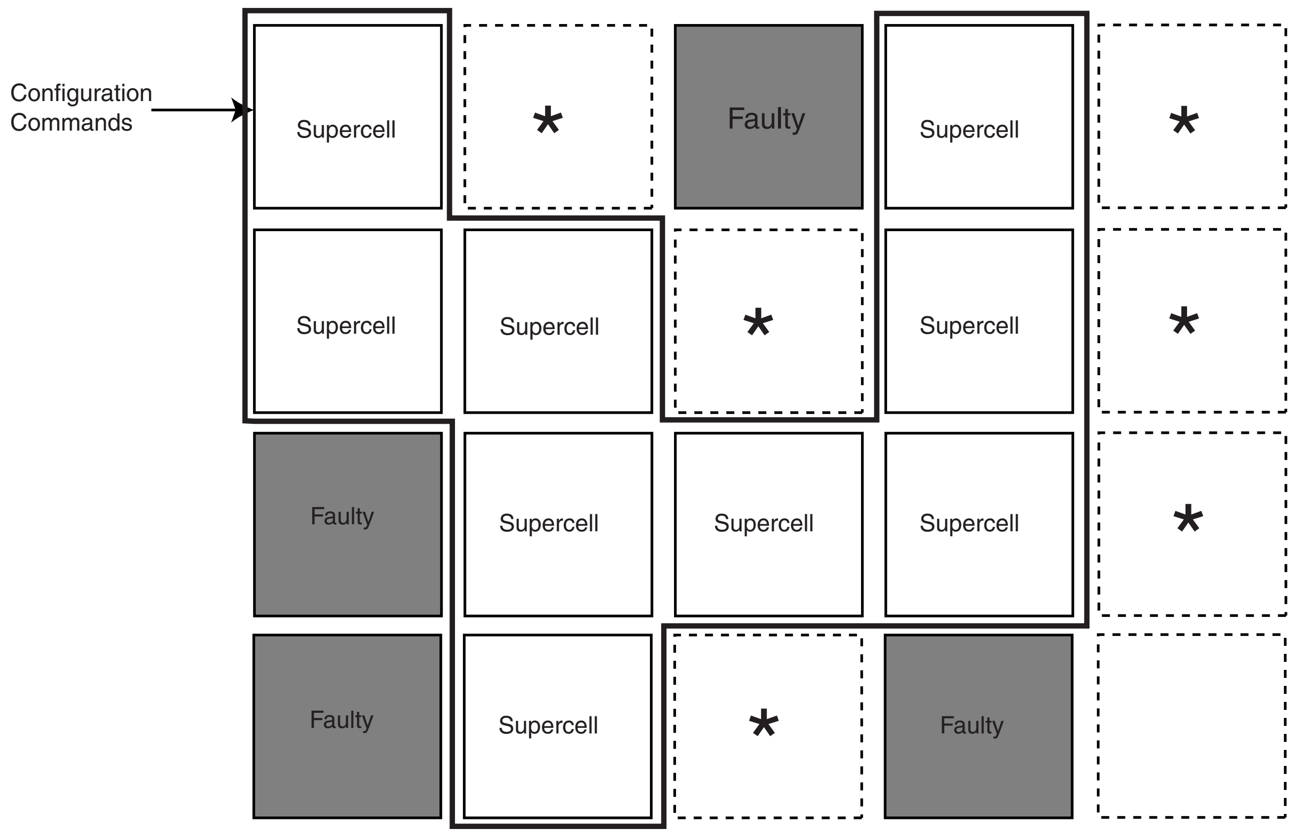


Figure 3.10
Supercells are configuring new supercells to the East of current perimeter (perimeter is shown by dark line).
Supercell in upper left corner receives external configuration commands.
Non-perimeter supercells pass the information to other supercells via the Link/Lock network.
Perimeter supercells configure regions to the East to act as new supercells.

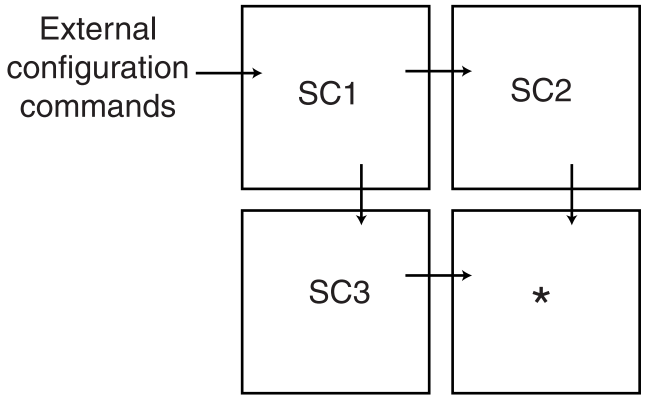


Figure 3.11
"*" is being configured by SC3 as a new supercell
While being configured, region "*" is also being sent LINK and other signals from the North by SC2. Supercells must handle receipt of these signals while they are being constructed.

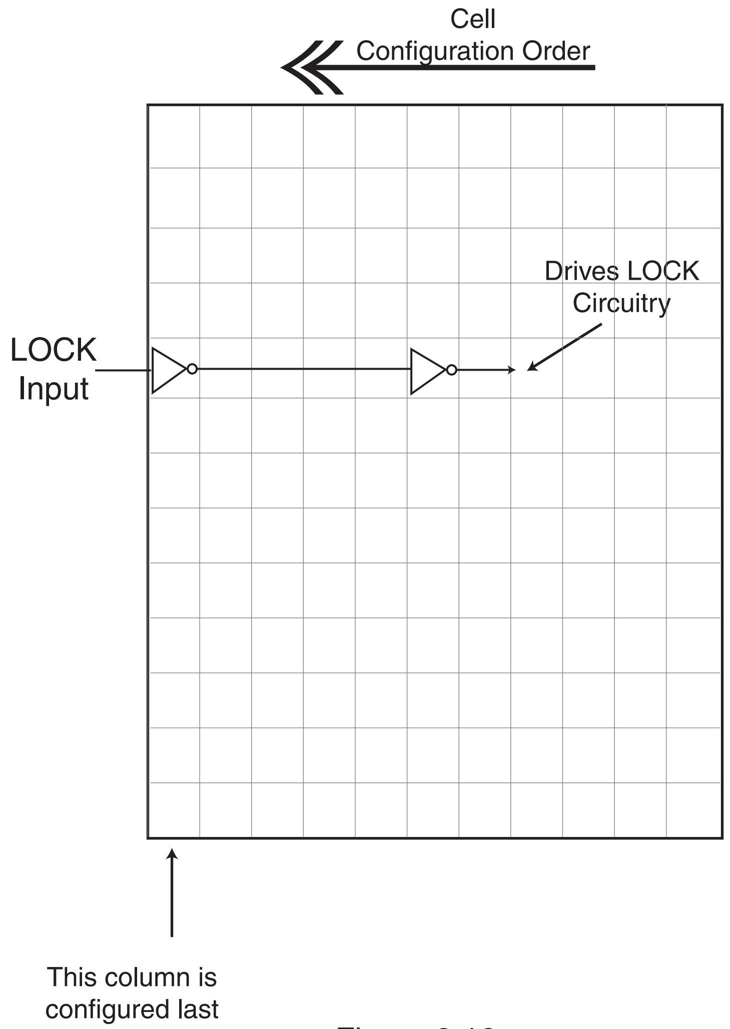


Figure 3.12
Internal LOCK signal is a twice-inverted copy of the external LOCK input. Prior to configuring the leftmost column, the internal LOCK signal will be 1, and the supercell will not initiate any outgoing LINK requests. When the leftmost column is built, the incoming LOCK signal is correctly passed to the internal LOCK circuitry. This mechanism is repeated on all four sides of the supercell.

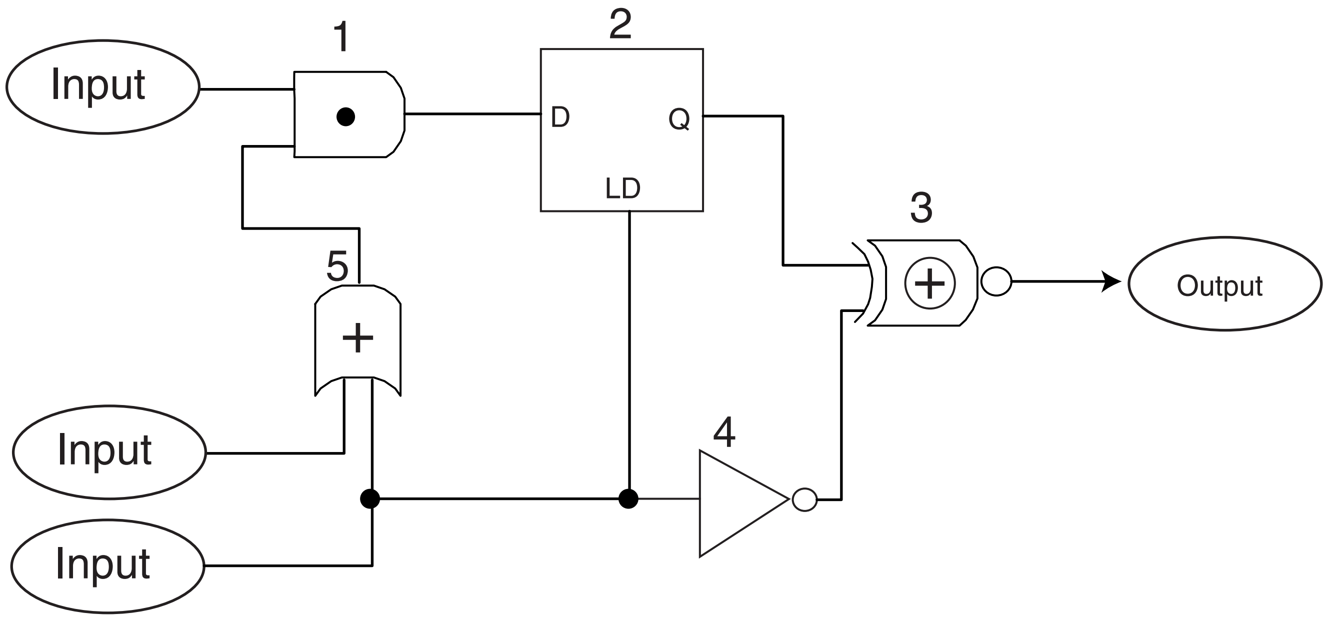


Figure 3.13
Sample Final Target Circuit
Node numbers are shown above components.
Each node will be implemented in a supercell.
Node assignments for Inputs an Outputs are not shown.

To allow maximum flexibility in the use of the cells making up the functional block, each cell's behavior can be specified independently of the other cells within the block. To avoid having to specify the entire truth table of a cell, one layer of indirection is added. Figure 3.14 shows the mapping table from cell codes to actual cell configurations. For example, to configure a cell within the block to act as an inverter, the code **6** is used, while an AND gate is coded as a **7**.

The functional block is shown in Figure 3.15. The three cells making up the block are labeled 1, 2 and 3. The actual output of the functional block is labeled f_{out}, and is generated by the Eastern output of cell 3. This output can be connected (via the supercell network) to the inputs of other functional blocks. The inputs to the functional block are labeled i_1 and i_2, and are delivered to the southern inputs of cells 1 and 2 respectively. These inputs can be connected (via the supercell network) to the f_{out} outputs of other functional blocks. By connecting these inputs and outputs appropriately, the desired target circuit is constructed.

Additionally, there are two special I/O lines connected to the Western edge of cell 1. These are used for transmitting information between the functional block and the outside world. They are directly connected to cells on the western edge of the supercell. Thus, functional blocks on supercells in the western-most column of the supercell network may be accessed from outside the Cell Matrix. This is the mechanism used for writing and reading data to and from the final target circuit.

Each of the three blocks can be configured independently. For example, the functional block in Figure 3.16 reads an external input, inverts it, and ANDs it with input i_2. The result is inverted in cell 3 and sent as the functional block's output f_{out}. Thus this functional block implements a NAND with one inverted input. The block is specified by the ordered triple 6, 7, 6.

Figure 3.17 shows the functional block for a D-flip flop. The D input is sent to cell 1, the LD signal is sent to cell 2, and the Q output is sent from cell 3. When LD=0, Q is latched, while when LD=1, the D input is loaded into the feedback path. The ordered triple for this block is (2,19,9).

In this fashion, arbitrary functional blocks may be specified. Again, we have chosen a simple 3-cell block for this project, but it is relatively easy to implement larger-scale functional blocks, thereby decreasing the number of supercells required for a given target circuit.

In the current supercell design, we allow up to 64 different single cells which can be used to build a functional block. However, we have only defined twenty cells for the particular tests we ran. The choice of cells is unconstrained though, and whatever cells are needed for the anticipated desired circuits can be added to the supercell design.

The next section describes how the functional block's specification is encoded into the circuit's genome, and how the connections among those blocks are specified. Section 4 will discuss the cell-level encoding of the genome inside the supercell.

3.4.6 Circuit Genome

Each supercell contains a *genome* for the final desired target circuit. The genome is a coded description of the circuit, and contains two sets of information:

1. the functional blocks which will compose the nodes in the final circuit; and
2. the connections among the inputs and output of those functional blocks.

For the sake of genome specification, each node of the circuit is assigned a numeric node ID. The mapping from nodes to IDs is somewhat arbitrary. It may be helpful to number nodes which are closely connected with IDs which are numerically close, but this is not necessary, and in some cases doesn't even make sense.

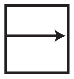
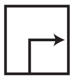
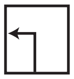
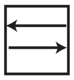
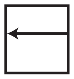
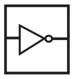
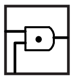
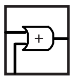
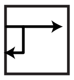
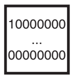
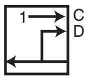
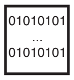
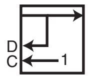
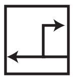
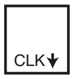
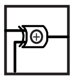
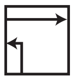
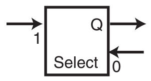
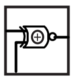
Cell Code	Configuration	Description
1		DW->DE
2		Input->DE
3		Input->DW
4		DW<->DE
5		DE->DW
6		NOT
7		AND
8		OR
9		Flip Flop Feedback
10		1/128 Crystal
11		Crystal Controller
12		High Frequency Crystal
13		Crystal Controller
14		Flip Flop Feedback
15		Toggle Flip Flop
17		XOR
18		Input->DW, DW->DE
19		MUX
20		XNOR

Figure 3.14
Functional Block Cell Codes

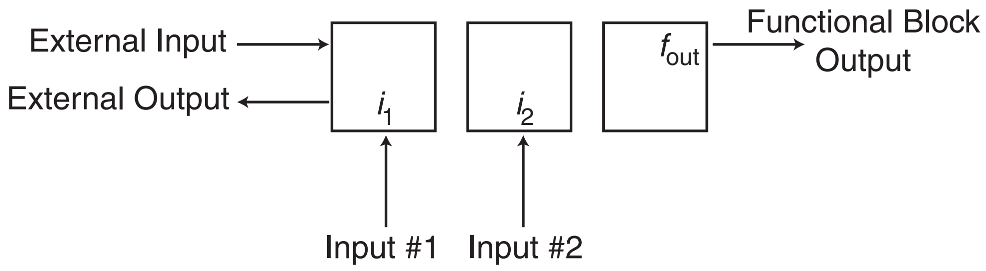


Figure 3.15
Supercell's Functional Block

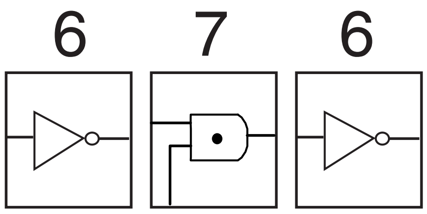


Figure 3.16
Functional Block for NAND Gate with One Inverted Input
Cell codes are shown above each cell

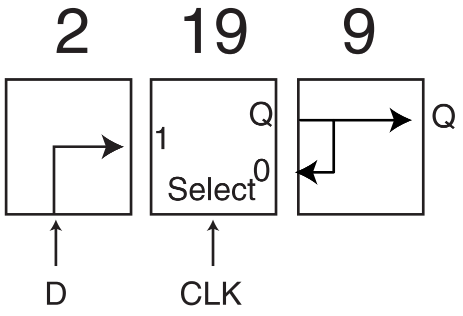


Figure 3.17
Functional Block for D-Flip Flop
Clock is sent to middle cell's Southern input (block input #2)
D is sent to Western cell's Southern input (block input #1)
Q is sent to block's output

The one case in which the choice of node ID assignment **is** critical is for nodes which will interact with the outside world, i.e., nodes which should be accessed form outside the system. Since these nodes will need to be implemented by supercells along an edge of the Cell Matrix, they should be given the lowest node IDs (beginning with ID 1). This will be explained further in Section 4.

Each piece of the genome consists of a set of 6 numbers, describing:

1. the node number of this functional block;
2. the node number of the functional block supplying **this** node's first input;
3. the node number of the functional block supplying this node's second input;

4-6. the block numbers for this functional block's cells, as described above in Section 3.4.5

Figure 3.18 shows a sample circuit (the same one in Figure 3.13). Each node has been numbered with sequential numeric IDs, beginning with the input and output nodes. Note that the external output node has been assigned the same node number (1) as one of the input nodes. This is not a problem, since the external I/O lines are bidirectional.

Figure 3.19 shows the same circuit implemented as a set of interconnected functional blocks. For example, node 1's first input is sent to its external output (on the Western edge of cell 1); its external input (from the Western edge of cell 1) is sent to its f_{out} output. Node 4 receives its first input from node 1's output, and its second input from node 5's output. In this way, the circuit of Figure 3.18 is translated into a set of functional blocks with appropriate interconnects among them.

Figure 3.19 also shows the makeup of each functional block in terms of the cell code for each of the three cells in the block. For example, node 4 is composed of cells 2, 7 and 1, as specified in Figure 3.14

The circuit of Figure 3.19 can be translated directly into a genome, which simply noting the node numbers and cell codes of each block, and the sources for each block's inputs. Thus, the genome for this circuit is:

1,8,0,4,1,1,
2,0,0,1,1,1,
3,0,0,1,1,1,
4,1,5,2,7,1,
5,2,3,2,8,1,
6,4,3,2,19,9,
7,3,0,2,6,1,
8,6,7,2,17,1

Each row in the above list is called a *block*, and describes a single node in the final circuit. The first number of each block is the node's number, the second and third numbers are the node numbers of the nodes supplying inputs to this node, and the fourth through sixth numbers are the cell codes for this node's functional blocks, as shown in Figure 3.20. Note that unconnected inputs are specified with the special node number 0, which means "no connection."

The mapping from a circuit as in Figure 3.18 to a functional description as in Figure 3.19 is completely trivial to automate. For example, OrCAD Capture will translate schematic diagrams into Verilog, which is just a list of nodes and connections among nodes. Translating from a node such as "AND" to a standard triple of functional block cells such as "(2,7,1)" can also be easily automated, by using a simple lookup table. Therefore, it is relatively easy to automate the process of genome compilation from a schematic diagram, or from a text based language such as Verilog. However, creating such a compiler requires making firm decisions about what cells will be used to create functional blocks, how large functional blocks should be, and so on. For a proof-of-concept, it is thus easier to compile circuits into genomes by hand, as was done in this project.

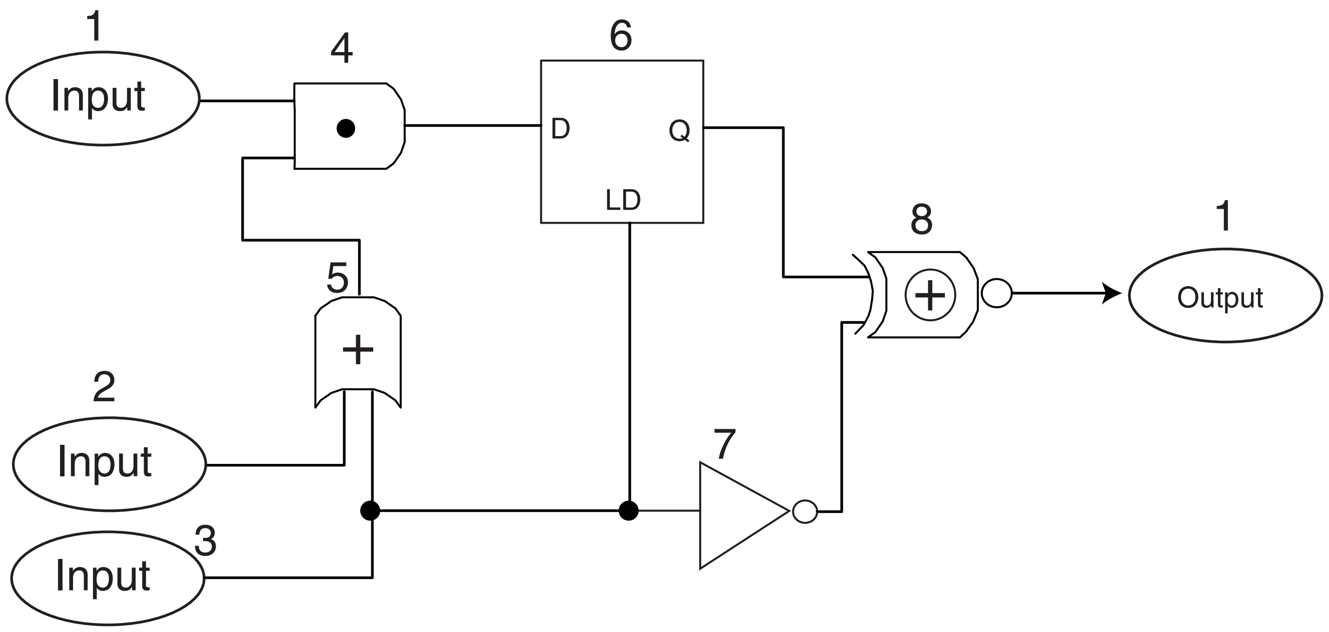


Figure 3.18
Sample Final Target Circuit
Node numbers are shown above components.

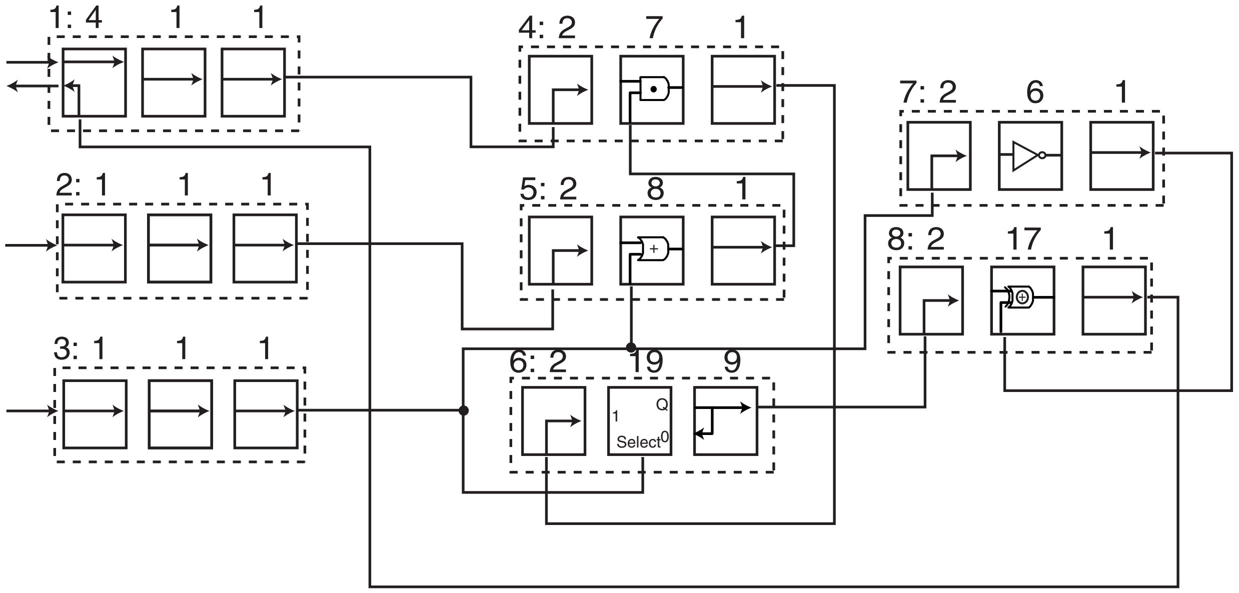


Figure 3.19
Functional Block Numbering, Cell Codes and Interconnect
for Circuit of Figure 3.18
Each block is indicated by a dashed rectangle.
Node number appears on the left (N:)
Cell codes appear on the right above each cell

THIS NODE	FIRST INPUT	SECOND INPUT	FUNCTIONAL BLOCK		
Node Number	Node Number	Node Number	Cell Code, Cell 0	Cell Code, Cell 1	Cell Code, Cell 2

Figure 3.20
Genome Block Layout
Corresponds to a single node, whose number is given in the first segment.
Describes inputs to this node, as well as this node's functional block composition.
Functional Block Cell Codes are shown in Figure 3.14.
The complete genome consists of a number of these blocks,
one for each node in the final target circuit.

3.4.7 Static ID Assignment

Since all nodes within the circuit are identified with numeric node IDs, there must be some way to assign a numeric node ID to each supercell within the supercell system. The easiest way to accomplish this is to use relative positional information. The IDs which are assigned in this way are called *Static IDs*, or *SID*s, for reasons which will become apparent soon.

For simplicity, each SID can be viewed as an ordered pair (r,c), where r represents the row in which the supercell sits and c represents the column, both relative to the first (upper-leftmost) supercell. The first supercell is hardwired to have an SID of (1,1). All other supercells dynamically assign themselves SIDs based on the SIDs of any adjacent supercells. In Figure 3.21, the supercell labeled "*" has been assigned SID (r,c), based on the SID of its neighboring supercells. For example, the supercell to the left has SID(r,c-1). Since supercell "*" is in the next column to the right, it's SID is (r,(c-1)+1).

Of course, the supercell "*" will also note that the supercell to its right has SID (r,c+1), and, based on that, will want to assign itself SID (r,(c+1)-1). Similar comments apply to the SID received from above and below. However, all four dynamic SID calculations result in the same final SID, (r,c).

Figure 3.22 shows a piece of the circuit for performing this SID calculation. Note that each side receives not only r and c, but also a VALID signal from each adjacent supercell[4]. The circuit makes the necessary adjustment (increment or decrement) on either r or c, and then zeros out all the bits if VALID is false. The results from all four sides can thus be bitwise-ORd to produce the final (r,c) for this supercell. This final (r,c) SID is then transmitted to adjacent supercells, so that they may further arbitrate their own SIDs.

It is important to note that, if each supercell were to transmit and receive to all of its adjacent supercells, one would create self-feeding loops wherever a 1 was transmitted between two supercell. This situation is avoided by using the Link Lock network (Section 3.4.3 above). SIDs are input from all sides, but are only transmitted on sides where a LINK output is asserted. If there is no LINK output, a serial stream of all 0s will be effectively transmitted, which will be interpreted as INVALID by the receiving supercell, and thus ignored in SID calculations.

It is also important to note that there will always be at least one pathway to each supercell within the system. If there was no pathway available on the L2 network, then there would be no way for the supercell to have been created in the first place.

The numeric data for r and c are transmitted as serial bitstreams. These are sent LSB first, which is important for incrementing/decrementing them. Note that all supercells perform these same operations on the same bits of the bitstream at the same time. In other words, all supercells will process bit 0 of the SID bitstream at the same time, adding or subtracting one, comparing to neighboring supercell's SIDs, and so on. Then all supercells will proceed to bit 1, conditionally changing that bit, and so on. Thus, other than propagation delay, the time required to assign SIDs is independent of the number of supercells, and depends only on the number of bits in the SID serial bitstream.

3.4.8 Dynamic ID Assignment

The above scheme for assigning SIDs works quite well, but there is one problem. Because SIDs are based on positional information, there is no guarantee that a particular SID will be present in the final set of supercells. For example, in Figure 3.23, the middle supercell in a 3x3 set has not been configured, because of the presence of a hardware fault within that region. Therefore, there is **no** supercell with SID (2,2).

Therefore, if we used SIDs inside our genome, and we assign a node to SID (2,2), that node would be missing in the final circuit, and our circuit would not work. The problem is that the genome encoding must be location-independent, while SIDs are extremely location dependent.

[4] In the actual circuit implementation, r and c are sent in as a single serial bitstream, proceeded by a VALID bit.

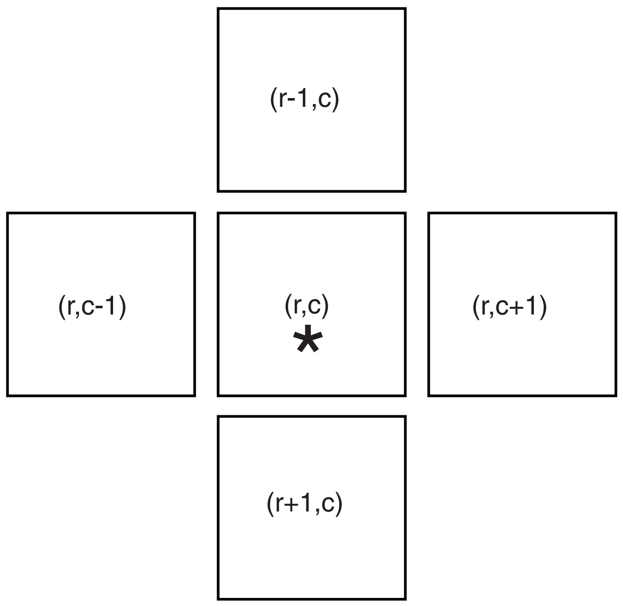
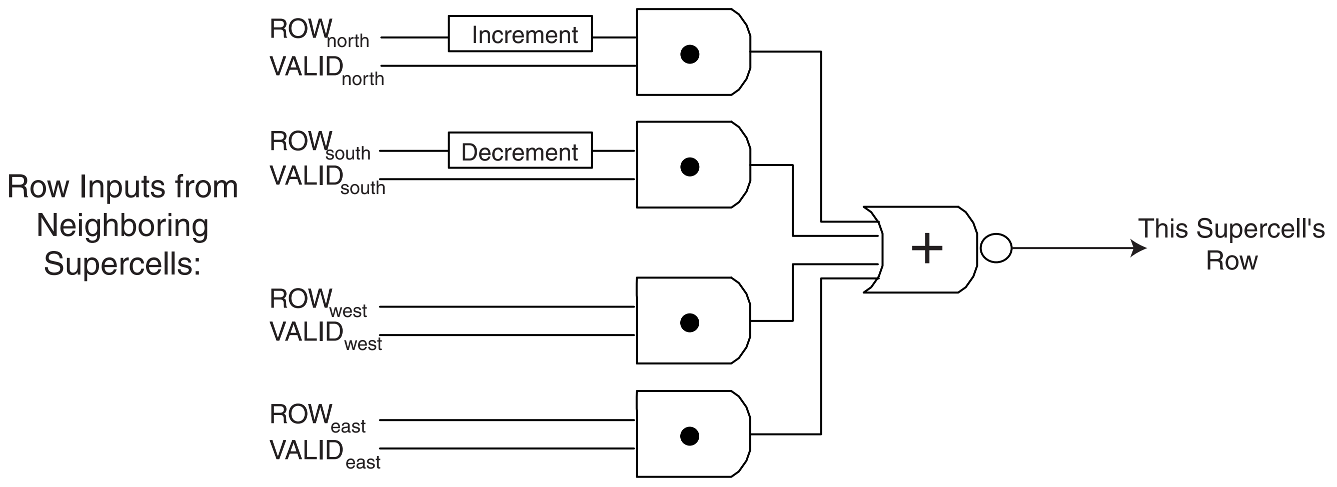


Figure 3.21
Supercell "*" can compute its (r,c) address
from the address of any of its neighboring supercells.
If multiple neighboring supercells are available,
all computed addresses should end up identical.



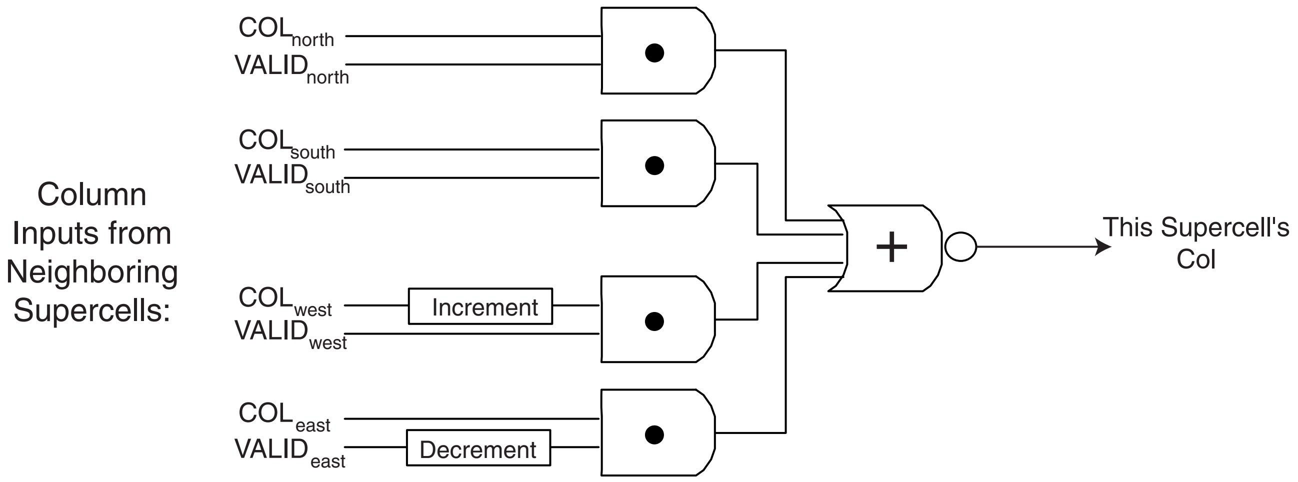


Figure 3.22
Row and Column calculation inside a Supercell
Su[ercell can compute its own row and column by adjusting the row and column information provided by neighboring supercells.
The VALID signals are 0 if there is no neighboring supercell.

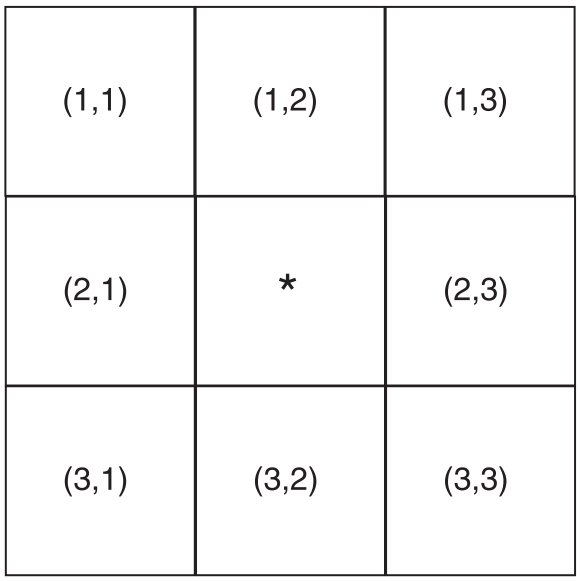


Figure 3.23
3x3 Set of Supercells with Missing Supercell in Middle
Because static IDs (shown inside each supercell) are position-based,
NO supercell has been assigned Static ID (2,2)

The solution to this problem is to assign *Dynamic IDs*, or *DID*s, to each supercell, such that the set of assigned DIDs begins with 1 and has no gaps. In this way, if we have say at least eight supercells in our final system, we will be guaranteed that we have DIDs of 1, 2, 3, 4, 5, 6, 7 and 8. We therefore only need to have at least as many functioning supercells as we have nodes in our circuit. The location of those nodes becomes irrelevant.

To achieve this DID assignment, we require a mapping from SID→DID such that the set of assigned DIDs is contiguous. One way to achieve such a mapping is to simply order the SIDs and the desired DIDs, and then map them in that order (first SID→first DID, second SID→second DID, and so on).

The supercells are able to dynamically perform this mapping by themselves. To achieve this, they assign themselves a single integer value based on their SID(r,c), using the formula

$$(r,c) \rightarrow 2^{32} * c + r.$$

The resulting integers thus supply an ordering to the SIDs of each supercell.

The set of supercells assign themselves DIDs using the following algorithms:

1. Determine the smallest SID among all supercells which have **not** been assigned a DID
2. Assign that supercell the next consecutive DID (beginning with 1)
3. Repeat until there are no supercells with unassigned DIDs

The only tricky part of this algorithm is step 1. To determine the smallest SID, a parallel comparison is performed among all unassigned supercells. Specifically, each supercell compares its own SID with any SIDs being sent by neighboring supercells. The minimum SID of this set of SIDs is then broadcast to all adjacent supercells. However, unlike the SID assignment, this broadcast is only done along the **reverse** direction of the LINK outputs in the L2 network. Figure 3.24 illustrates the situation. The arrows show the LINK outputs, as would be present in an L2 network initiated from the supercell (1,1) in the upper left corner. For example, supercell (3,3) generates a LINK output signal to both the supercells (4,3) and (2,3).

In this configuration, supercell (3,3) would compare SIDs being sent by supercells (4,3) and (3,4) to its own SID, and would transmit the minimum SID to supercell (2,3). Note again that these transmission are occurring in the backwards direction along each LINK output. Also, each supercell only compares its own SID if it itself has not yet been assigned a DID. Otherwise, the comparison is only made on the incoming SIDs.

With each supercell acting in this fashion, the result is that the initiator of the L2 network (supercell (1,1)) can determine the global minimum SID by comparing the SIDs it is receiving (and possibly its own SID). The minimum of that comparison will be **the** minimum of all unassigned supercells' SIDs in the system.

Next, supercell (1,1) broadcasts this minimum SID to all other supercells along the L2 network (in the forward direction along the LINK output lines). Each supercell receives this minimum SID, and compares it to its own SID. If all bits match, then that supercell knows it has the minimum SID, and it assigns itself the next consecutive DID. Proceeding in this way, all supercells are assigned DIDs.

There are some important details which should be noted here:

- To compare the magnitude of two serial bitstreams, it is best to transmit the streams MSB first. With that ordering, as soon as the two streams differ, the stream containing a 0 is know to be smaller than the stream containing the 1. However, SIDa were generated LSB first for efficient increment/decrement operations. Therefore, each supercell contains a **bit reversal** circuit, which transposes bits within a serial bitstream. This is a simple circuit which simply captures bits, and replays them in a reversed order.
- Since comparisons can be made on a bit-by-bit basis, there is no looping in this minimum-SID calculation. Rather, the first bit of **all** unmapped SIDs are compared simultaneously, and the first bit of the global minimum is known immediately. Thus, again, except for propagation delays, the time to compute this first bit is independent of the number of supercells under consideration. Similar

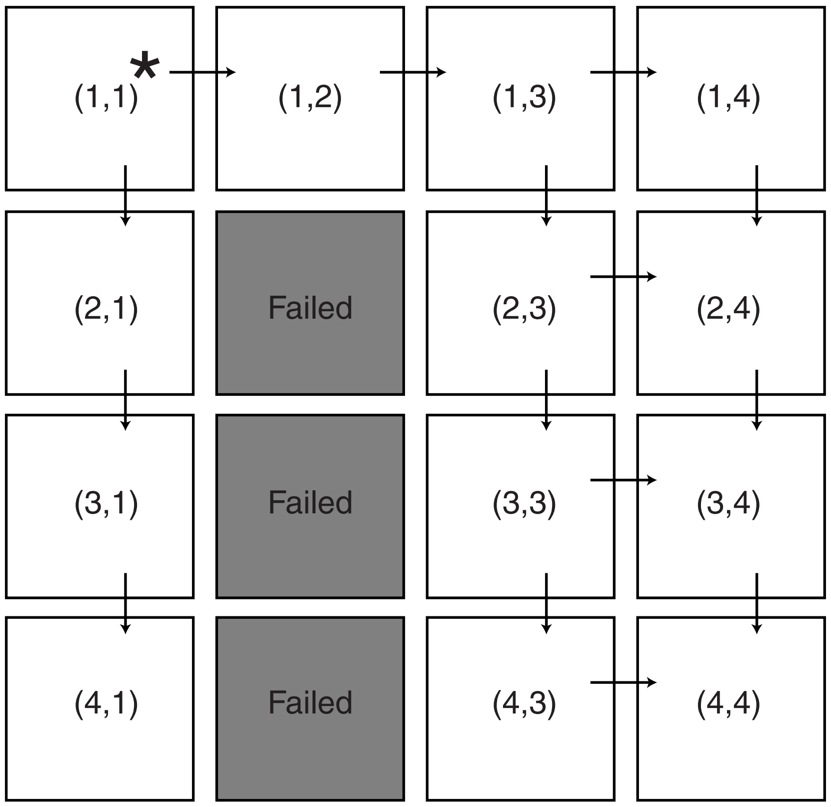


Figure 3.24
DID Arbitration in a 4x4 set of supercells with 3 failed regions
Arrows show LINK outputs on a Link/Lock network initiated from supercell (1,1) ("*")
DID comparison flows in reverse arrow direction

- comments apply for the second bit, the third bit, and so on. Thus the time required to determine the global minimum SID is a function of the SID length, not the number of SIDs being compared[5].
- As soon as supercell (1,1) has determined the first bit of the global minimum SID, and broadcast this bit to all supercells, each supercell can compare that bit to its own SID's first bit, and determine if it is still a candidate for being the global minimum. In this way, once the global minimum SID has been completely determined, the supercell with the matching SID has also been determined. There is no need for an additional comparison step for the matching supercell to identify itself. The comparison is performed at the same time that the global minimum SID is being computed.
- Each supercell retains a copy of the next DID to assign, and increments it after each step of this algorithm. Thus, there is no need to broadcast the next DID to all supercells. Each supercell retains its own copy of this variable. This is an example of distributing the processing throughout the hardware, rather than performing the processing in one location and distributing the data. It is a key advantage of fine-grained reconfigurable hardware over traditional memory/CPU systems.
- As with SID assignment, supercells will on appear in the system at all unless there is an L2 path to them. Therefore, all supercells will eventually be assigned DIDs with this algorithm.

There is one further enhancement to the above algorithm. In our research, we learned that assigning nodes to supercells which are adjacent to failed regions makes circuit construction more difficult. For example, in Figure 3.25, the supercell labeled "*" has failed cells on three sides. As a result, connections to and from "*" can only be made through one side of that supercell, namely via the supercell to its left. If "*"'s output was being sent to many other nodes, it would be difficult to implement this routing.

To account for this situation, we have implemented additional circuitry which detects the presence of failed regions in adjacent locations. During DID assignment, if any adjacent region is marked as FAILED (i.e., does not contain a properly-working supercell), then the supercell will remove itself from consideration as a global minimum SID. In Figure 3.26, the supercell marked "*" will detect missing supercells around it, and therefore "*" will not allow itself to be considered in the minimum SID algorithm. In fact, only the supercells marked with "+" are now candidates for being global minimum SIDs, and therefore are the only supercells which will eventually have DIDs assigned. The result is that an expanded region (indicated by the darkened line in Figure 3.26) is created around the actual failed regions, and no DIDs will be assigned to supercells within this expanded region. Note however that the good supercells which are not assigned DIDs **can still** be used for building connections among supercells. The algorithm does **not** discard supercells next to failed regions. It merely avoids assigning DIDs to those supercells, i.e., it does not assign those supercells to act as nodes in the final target circuit.

This feature, which is called *edge sensing*, can be enabled or disabled by changing the configuration of a single cell within the final supercell circuit.

3.4.9 Traceback Generation and Parsing

Once DIDs have been assigned to all supercells, the next step in circuit assembly is to process the circuit genome. Each piece of the genome contains 4 pieces of information:

1. the DID of the node (called the DST node) described by this piece of the genome;
2. the DID of the node (called the SRC1 node) supplying the DST node's first input;
3. the DID of the node (called the SRC2 node) supplying the DST node's second input; and
4. the coded description of the DST node's functional block.

Synthesis of the functional block has already been described (Section 3.4.5 above). What remains in order to implement the target circuit it to build the pathways from the SRC nodes to the DST nodes via the supercell network. The actual path synthesising mechanism will be described in Section 3.4.10 below. However, before pathways can be built, the system must determine exactly what path to use to transmit data from a SRC node to a DST node. The method of determining such a pathway is called *traceback generation*.

[5] This is a fundamental characteristic of circuits built on the Cell Matrix. The ability to create circuits which operate efficiently in parallel instead of looping sequentially is a recurring theme in Cell Matrix circuit design.

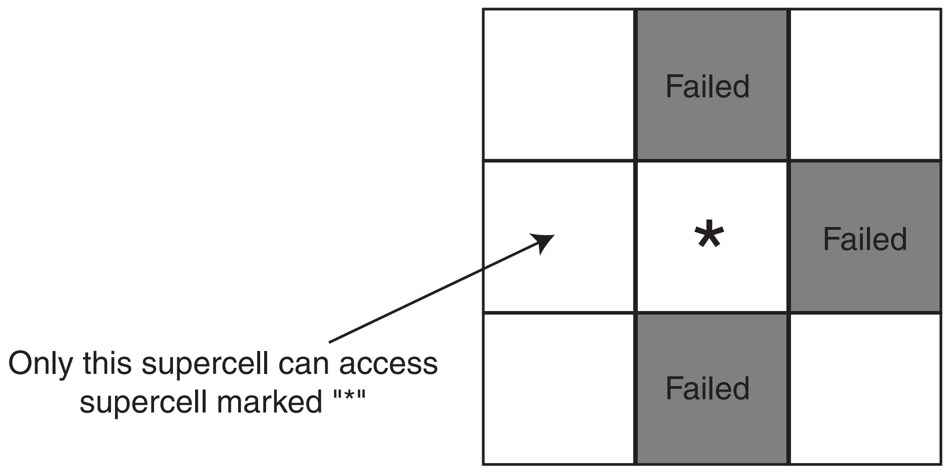


Figure 3.25
Effect of failed regions on supercell access
Supercell "*" can only be accessed from one side.
This is a potential problem if "*" implements a node
whose output is used by many other nodes
(and therefore needs lots of pathways connected to it).

+		Failed		+	+	+
+	+	*	Failed		+	+
+		Failed		+	+	+
+	+		+	+	+	+
+	+	+	+	+	+	+

Figure 3.26
Edge Sensing in a network of supercells with failed regions
"*" can only be accessed from one side, and therefore should not be used to implement a node in the genome.
Edge Sensing causes supercells adjacent to failed regions to NOT be assigned DIDs (indicated by lack of "+" inside).
Therefore, all interior supercells which ARE assigned DIDs ("+") are guaranteed to be surrounded by functioning supercells.

Each supercell contains its own (identical) copy of the final circuit's genome. This genome is basically a prescription for how to assemble the target circuit, with each block of the genome (Figure 3.20) describing a single DST node's inputs and its functional behavior. By successively processing each of these genome blocks, the final circuit is assembled, one node at a time.

Each supercell also maintains a *current block* pointer, indicating which block is being processed. This pointer is advanced simultaneously in all the supercells, so that all supercells are always processing the same genome block at any given time. By using the current block pointer to pick a genome block, **all** supercells know the DID of a current DST node, as well as the DIDs of the SRC1 and SCR2 nodes to be connected to the DST node's inputs.

Traceback generation begins with each supercell comparing its own DID to the current genome block's DST. Exactly one supercell should recognize the DST DID as matching its own DID. That supercell now becomes the DST node.

The DST node next assembles a Link/Lock (L2) network, beginning with itself, and broadcasting to all reachable supercells in the supercell collection. Depending on the presence of already-built pathways among the supercells, not all supercells will be reachable. There are only a finite number of connections possible between two supercells. In Figure 3.27, pathways have previously been built from (3,2)$\rightarrow$(3,3) and from (2,3)$\rightarrow$(3,3). These are shown by the dotted lines. The L2 network subsequently built from DST node (2,2) is shown in solid lines. (2,2) can access all supercells except (3,3), since the two sides from which (3,3) can be accessed are in use. Note that this is a deliberately simplified example. In the actual supercell implementation, it is possible to access each supercells via **three**[6] independent channels on each side. Thus, a typical (internal) supercell can be accessed via 12 different paths (4 sides * 3 channels per side). See Section 3.4.10 below for more details on channels.

After the DST node has assembled its L2 network, it sends a signal (via that L2 network) to all connected supercells, requesting initiation of a traceback. At this point, all supercells (which, recall, each have a copy of the current genome block) compare their DID to the SRC1 DID[7]. The supercell which matches declares itself the SRC1 node. Upon receiving the traceback request from DST, SRC1 broadcasts a *traceback string*, using the reverse LINK outputs of the L2 network (exactly as in the DID assignment, Section 3.4.8). Note that since SRC1 received the traceback request, there is guaranteed to exist at least one pathway from DST to SRC1 on the current L2 network. All that remains is for DST to discover exactly what that pathway is.

Figure 3.28 shows a particularly unpleasant 5x5 collection of supercells. Supercells which are grayed out are missing, presumably because of hardware failures. DIDs are indicated inside each supercell (we're assuming edge sensing, as described at the end of Section 3.4.8, has been disabled). Node 5 is DST, and node 15 is SRC1. The L2 network, constructed from node 5, is indicated by the arrows from supercell to supercell. When SRC1 receives the traceback request, it broadcasts a single *Valid* bit in the reverse direction of all arrows coming into it.

Each supercell which receives this valid bit performs the following steps:

- notes the direction D the valid bit arrived from[8];
- broadcasts the valid bit to its adjacent supercells (again on the reverse LINK outputs);
- broadcasts a coded representation of direction D to all adjacent supercells (again in the reverse LINK outputs);
- broadcast any remaining data received following receipt of the valid bit.

[6] The choice of using three channels is somewhat arbitrary. The design can easily be extended to allow more channels.

[7] In this section we will describe traceback generation from DST to SRC1. This is then followed by path synthesis between those nodes. Following that, the entire process (traceback generation and path synthesis) is repeated for the DST$\leftarrow\rightarrow$SRC2 pathway.

[8] If a valid bit is received from more than one side, a priority ordering is used to pick a single side. The current ordering is N (highest priority), S, W, E (lowest priority). This ordering can be randomized for increased flexibility-see Section 7.

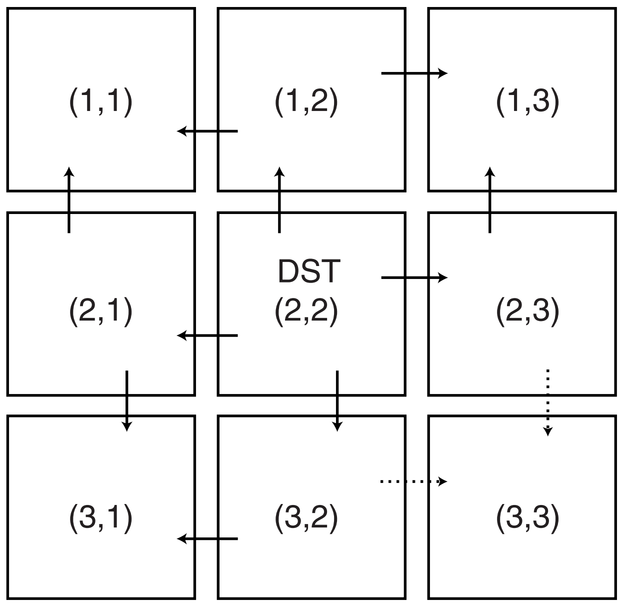


Figure 3.27
Example of Routing Contention
Supercell at (2,2) "DST" has initiated a Link/Lock network
Supercells (3,2) and (2,3) already have pre-existing paths built to (3,3),
shown by the dashed lines. Therefore, DST's L2 network can not reach (3,3).

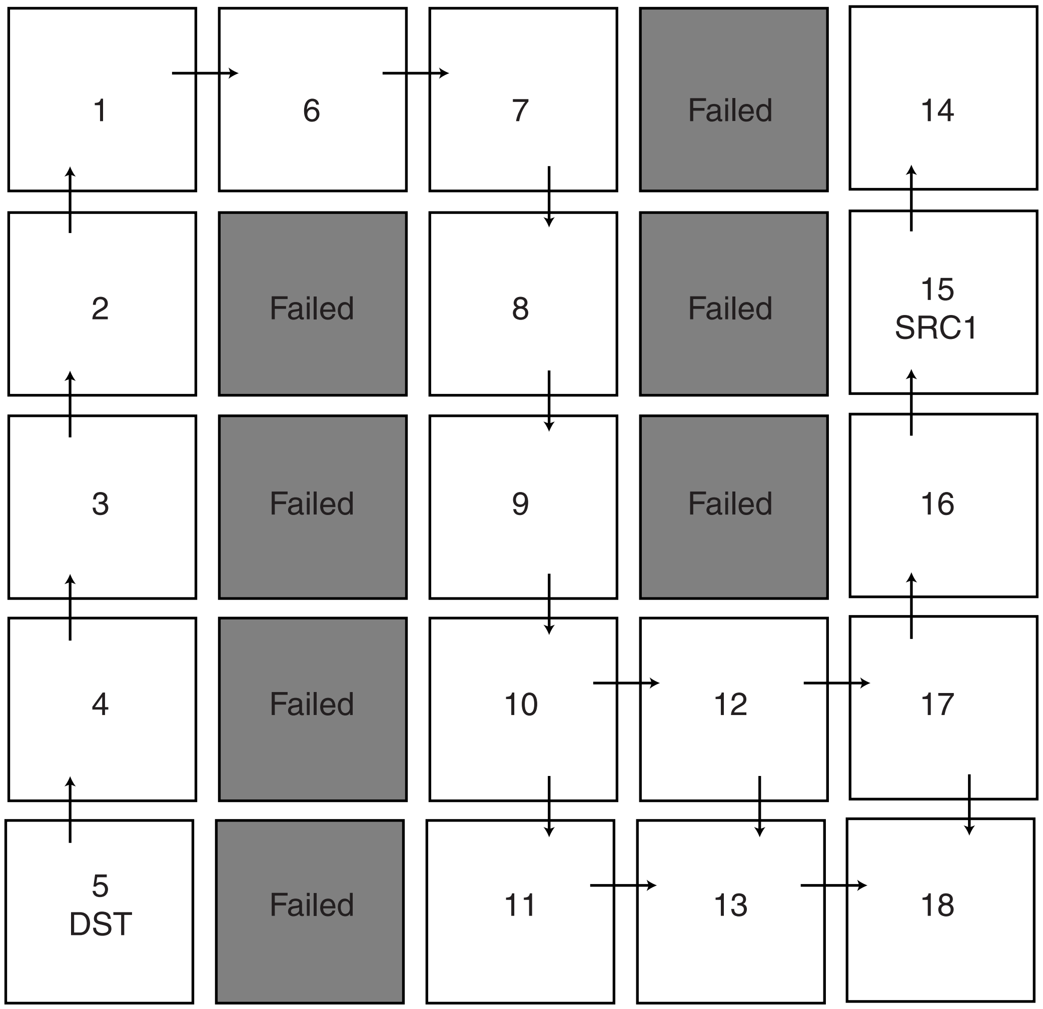


Figure 3.28
Link Lock Network in set of 5x5 Supercells with Failed Regions
Network has been generated by DST supercell
Dynamic IDs (DIDs) are shown inside each supercell
Arrows indicate LINK outputs

The effect of this algorithm is that each node appends the arrival direction D to the beginning of the traceback string, and re-broadcasts the traceback string, with a valid bit in the beginning.

An example will help to clarify this. In Figure 3.29, the string shown inside each node represents the string it broadcasts to adjacent nodes. Node 15 is SRC1, and thus broadcasts a valid bit "V" to node 16.

Node 16 receives a valid bit (from node 15), and then performs the above steps. It notes that V was received from the North ("N"), then broadcasts the valid bit, followed by "N." No other data is received by node 16, thus its full transmission is the string "VN." This will be broadcast to node 17.

Node 17 receives the string "VN," notes its arrival from the North, broadcasts a valid bit V, the direction "N," and finally the remaining received string "N." Thus node 17's full transmission is "VNN." This will be broadcast to node 12. Note that node 18 does not receive a traceback string. This indicates that, on the current L2 network, node 18 has no pathway to node 15. This is not a problem, as node 5 is the initiator of the L2 network, and thus only connectivity from node 5 to other nodes is important.

Node 12 receives the string "VNN" and notes that it arrived from the East. It then broadcasts a valid bit V, followed by "E," followed by the remaining data it receives, in this case the string "NN." Thus node 12 broadcasts the string "VENN."

Looking at this string closely, we see that it is simply a map for traversing the supercell network to arrive at SRC1 from node 12. Thus, from node 12, you can reach SRC1 by moving East (to node 17), then North (to node 16) and then North (to node 15).

In a similar manner, node 10 receives "VENN" and broadcasts "VEENN." Node 9 broadcasts "VSEENN" and so on. Finally, node 5 will receive the string "VNNNEESSSEENNN" from node 4. Node 5 notes the arrival direction (N), inserts that after the valid bit, and thus has the full traceback string "VNNNNEESSSEENNN." This is a complete map of how node 5 can reach node 15 in the current L2 network.

The DST supercell stores this traceback string in an internal memory (actually all supercells store their traceback string). DST can then parse this string, one character at a time, and build a pathway between itself and SRC1. The path synthesis mechanism is explained in the next section.

3.4.10 Path Synthesis and the Steering Block

Ultimately, supercells need to exchange information with one another. To do so, pathways must be built among the supercells. Since the supercell is the basic atomic unit for the finished circuit, and since only hardware covered by supercells can be guaranteed to be defect free, the pathways must actually be built **through** the supercells. The easiest way to accomplish this is to include pre-existing path segments inside each supercell. These segments handle the actual transmission of data through one supercell. Steering of the path is accomplished by dictating on which side of the supercell each segment exits.

The self-configurability of the Cell Matrix can be put to good use to accomplish this. The supercells include circuits which are called *multi-channel wires*, and are disclosed in a pending patent applications (allowed July 2001) (Mac 2001). Such wires are used to transfer binary data from one cell to another, such as is needed for implementation of the final target circuit. However, multi-channel wires can also carry cell configuration information to the cells at the end of the wire. And, because of the self-configurability of cells, multi-channel wires can carry configuration information **for building more multi-channel wires**. In this fashion wires can be efficiently extended.

The process of path synthesis is thus a sequence of wire configuration steps, where each configuration builds a path from the current supercell to the desired next supercell. This process, dictated by the traceback string, continues until the desired cell (SRC1) has been reached. Following this, SRC1 may send data to DST along the just-constructed path (the paths are bidirectional).

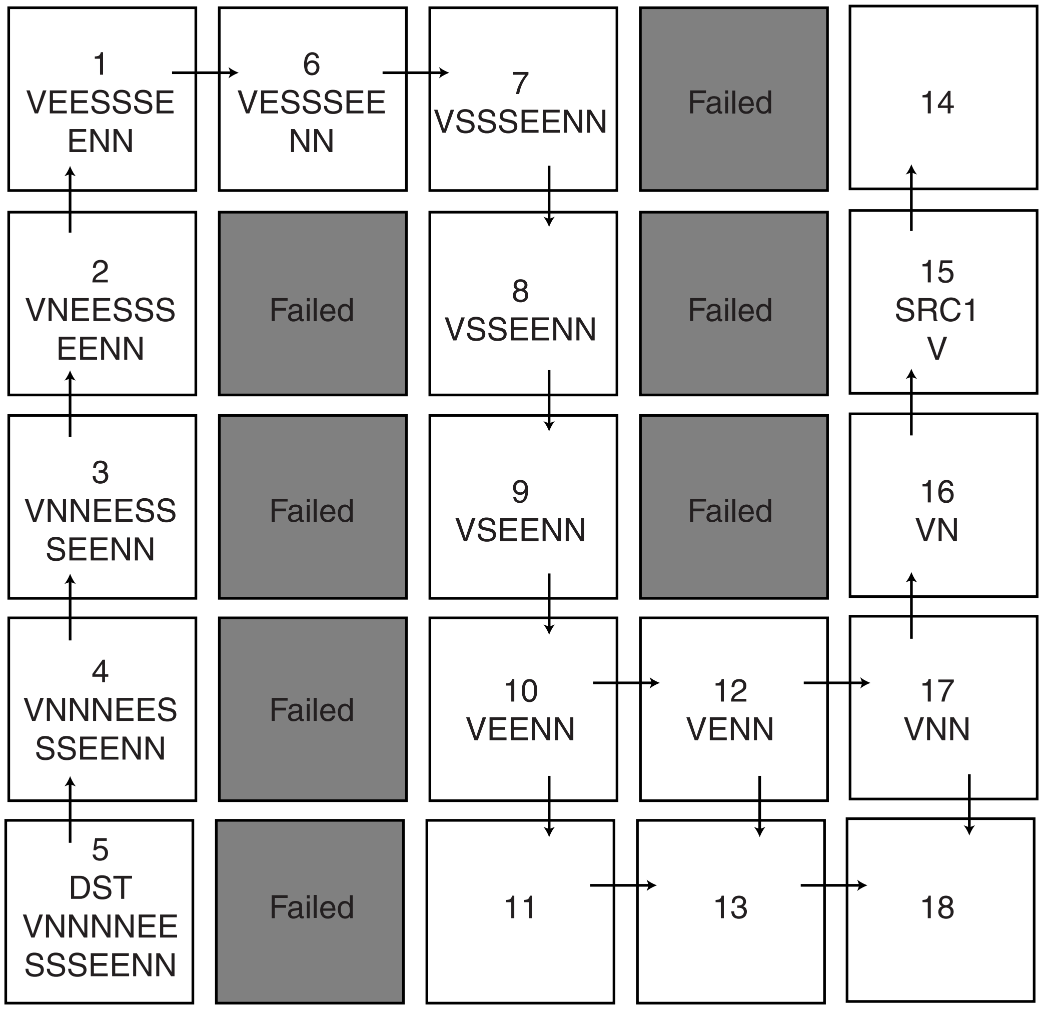


Figure 3.29
Traceback Generation using same Link Lock Network from Figure 3.28
Traceback packets are passed in the **reverse** direction of the LINK outputs (arrows).
Current traceback string **to be broadcast** is shown inside each supercell.
V is the valid signal. Other characters show directions.
Final traceback string is NNNNEESSSSEENN.

As mentioned above, it is desirable to have pre-existing wire segments inside each supercell, rather than trying to build a complete path from one end of a supercell to another. This is not only faster, but is also much easier, since building wires through already-existing circuitry can be difficult.

Figure 3.30 illustrates a supercell containing a circuit called a *steering block*, which is used for extending paths through the supercell. The inputs and outputs along the edges represent incoming and outgoing paths. Tiling these supercells connects inputs to outputs, and vice versa. The paths inside the steering block are actually multi-channel wires, capable of carrying data as well as being extended. The areas marked with "?" are breaks in the wires (unconfigured cells). Figure 3.30 also shows a 2x2 array of supercells, containing this supercell ("*"), another supercell to its West ("W") and another to its North ("N"). We're ignoring the supercell in the upper left.

Suppose now that W's traceback string is telling it to build the indicated path: East, then North. W's Eastern output is connected to point (1) in Figure 3.30. Therefore, by outputting configuration commands to its Eastern output, W has access to a wire inside *'s steering block. In particular, it has access to the unconfigured cells in region (2) of *.

If the cells in (2) are configured to also act as a wire, the two wire segments w1 and w2 will be joined, and subsequent commands sent to (1) will be transmitted to the unconfigured cells in region (3). Configuring these cells to act as a wire, w2 and w3 will be joined, and now supercell W has access to cells in region (4). At this point, W can send configuration commands to build a bent wire, one which turns 90 degrees clockwise, followed by a straight segment. This will complete a wire which links input (1) to output (5). Thus, supercell W has built a pathway from its Eastern output to the Southern input of supercell N.

The same process can now be repeated to configure N's steering block to route its Southern input to any of its output sides, and so on. In this way, one supercell can synthesize a path through the set of supercells, steering the path out each supercell's chosen side.

There are actually four independent sets of path segments inside the steering block, one for each input side. Thus, the above path from *'s Western input to its Northern output can exist simultaneously with, say, another path from South to West. Figure 3.30 is a simplified version of the actual steering block. For example, if the cells in region (7) were configured to turn and connect to output (8), the last segment of the path would cross wire segment w4, which may already have been configured as part of say a W→E path. Additional path segments exist inside the steering block to allow these crossings to occur without interfering with other already-built paths.

It is possible to generate wire-configuration sequences for building straight wires, as well as for turning clockwise or counter-clockwise. However, configuring wires which make turns is more complicated than configuring straight wires, and also requires more steps (11 instead of 3). Therefore, a special mechanism (disclosed to NASA ARC in a New Technology Report dated 19 July 2001) is employed for making turns. Rather than configuring bent wires to achieve a turn, bent wires are pre-configured inside the wire breaks (e.g. region (2) in Figure 3.30) in the steering block. A turn is accomplished by simply attaching to this bent piece of wire. If instead a straight segment is desired, the pre-existing bent segment is overwritten with a straight segment.

Figures 3.31(a)-(c) illustrate this mechanism. Figure 3.31(a) is an unconfigured path segment, with wire breaks in regions (1) and (3) and a pre-existing bent wire in region (2). Input W is connected to neither E nor S. In Figure 3.31(b), the break in region (1) has been filled with a short wire segment. Input W is now connected to output S. In Figure 3.31(c), regions (1) and (3) have both been filled with a straight wire segment, and the cells in region (2) have been overwritten with another straight segment. In this configuration, input W is connected to output E. Thus, W can be steered either South or East by configuring only straight line segments.

As described above, in building an L2 network prior to traceback generation, it is necessary to know which steering block segments are available for use, and which have already been allocated to other paths. When segments are configured as in Figure 3.31(b), it means that the output side (S in this case) is being used. When the straight segment which connects to the bent wire is configured, one of its cells is made to output a signal (*InUse*) on a particular side. Pre-existing circuitry picks up this signal, and transmits it to the L2 generating logic, thus preventing

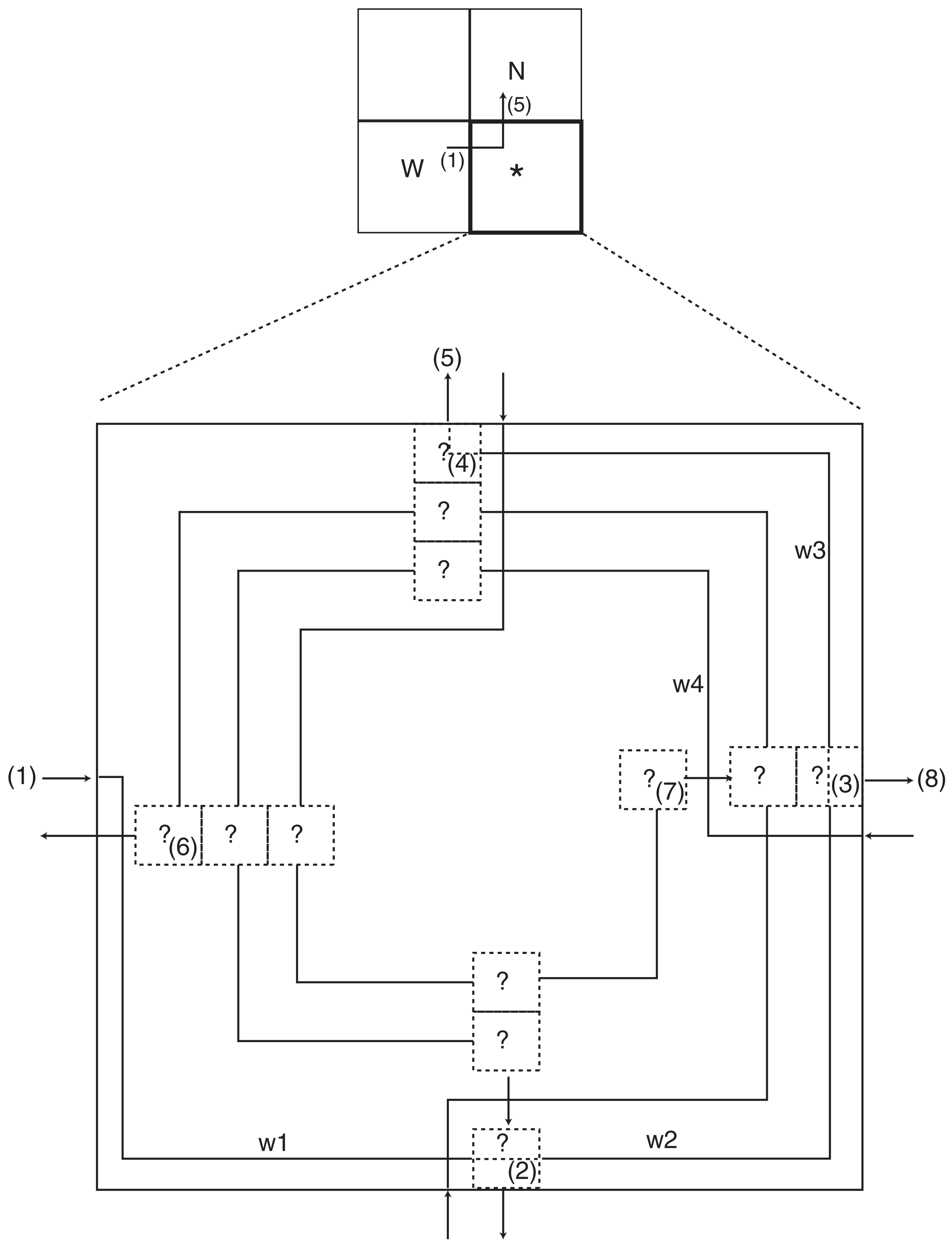


Figure 3.30
Supercell's Steering Block
Wire building commands are sent through inputs such as (1). Regions marked "?" contain unconfigured cells, in which wire segments (shown dashed) are built.
A straight segment in (2) joins w1 and w2. A bent segment in (4) joins w3 and (5).

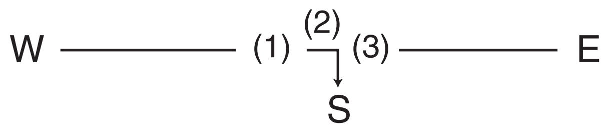


Figure 3.31(a)
Example of Wire Turning by Building Only Straight Segments
Regions (1) and (3) contain unconfigured cells.
W is connected to neither E nor S

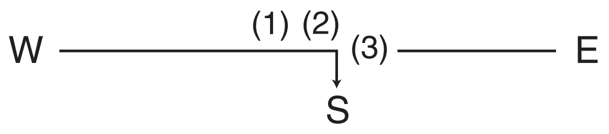


Figure 3.31(b)
Clockwise Turn
A straight segment has been built in region (1)
W is now connected to S

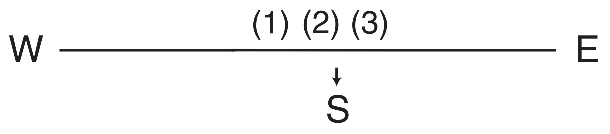


Figure 3.31(c)
Straight Extension
A straight segment has been built in regions (1), (2) and (3)
W is now connected to E

this side's output from being used for more than one path. Note that only output protection is required. When a sueprcell's input I is in use, an adjacent supercell's output O is also in use. Since no additional paths will built which output to O, only one path will be built to input I.

A final detail of the steering block, which is omitted in Figure 3.30, is the concept of *channels*. In Figure 3.30, each input side can be independently directed to any of the steering block's output sides. For greater flexibility in routing, in the actual steering block, the circuitry in Figure 3.30 is replicated three times. This allows up to three independent paths from each input side to any output sides.

Given the above mechanisms in each supercell's steering block, the DST cell can read its traceback string, and for each direction contained therein, issue wire building commands to synthesize a path through the set of supercells. Upon reaching the end of the traceback string, there will be a complete path from DST to SRC1, including the appropriate InUse flags.

Figure 3.32 is a screen capture from a simulation run on a simplified supercell (made when we were developing the steering block). Black regions are unconfigured cells, gray regions are configured cells which are outputting all zeros, and white regions are cells which have at least one non-zero output. The simulation was run to build a path from supercell 1 to supercell 4. To allow the built path to be detected, supercell 4's functional output was set to one, thus causing all cells along the built path to show up in white in the simulator.

As can be seen in this figure, the path enters supercell 1 from the West, exits to the East into supercell 3, then exits to the South into supercell 4. The final path in supercell 4 (which turns in 4 directions but appears to go nowhere) will be explained in Section 4. Note also the InUse lines annotated in Figure 3.32 These are generated at the exit point from each supercell, and transmit the InUse signal back to the L2 generating logic in the upper left corner of each supercell (see Figure 4.1 for a fully annotated block diagram of the final supercell).

4. Implementation

The final supercell is a 270x270 block of Cell Matrix cells. Figure 4.1 shows a block diagram of the supercell layout. This is a screen capture of a single supercell inside a simulator, enhanced via edge detection to highlight certain regions. The following major sections are identified:

- L2 Build Area: responsible for controlling creation of an L2 network.
- L2 Shutdown Area: controls the disassembly of an L2 network.
- SID Parser: reads and interprets Static ID (SID) data from neighboring supercells, in order to generate this supercell's own SID.
- SID Bit Reversal Logic: Changes bit ordering for subsequent serial comparison of SIDs.
- DID Sorter: Arbitrates selection of own and incoming Dynamic IDs (DIDs) to select local minimum DID.
- Traceback Logic Generation: Controls processing of incoming traceback string and concatenation of own directional information.
- Traceback Storage: Stores up to 1,024 bits of traceback information (enough for a path containing 126 supercells).
- Traceback Parser: Analyzes the stored traceback string to drive path synthesis.
- *Channel Analysis: Controls generation of InUse flags.
- Path Synthesis Library: Contains configuration information for building wires.
- Genome: Contains coded description of final target circuit to be implemented.
- *Path Synthesis Inputs and Outputs (Channels A-C): Entry and exit point for path synthesis commands from supercell to supercell.
- Functional Block Cell library: Contains configuration information for components of supercell's functional block.
- Path Synthesis State Logic: Controls Traceback Parser and Path Synthesis Library to send configuration commands through the Path Synthesis Outputs.
- Genome Serializer: Converts parallelized genome back to non-phase-shifted serial format.

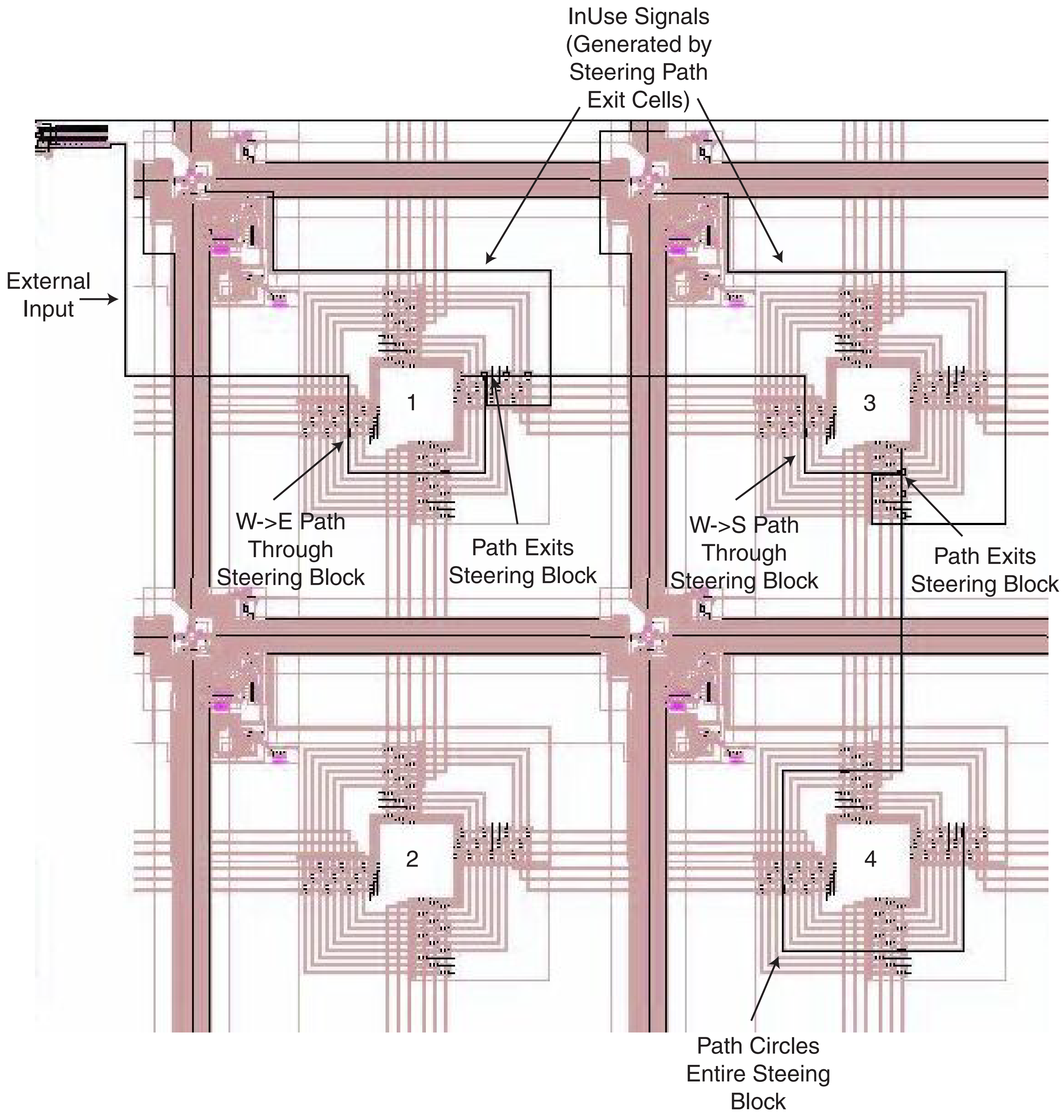

InUse Signals (Generated by Steering Path Exit Cells)
External Input
1
3
W->E Path Through Steering Block
Path Exits Steering Block
W->S Path Through Steering Block
Path Exits Steering Block
2
4
Path Circles Entire Steeing Block

Figure 3.32
Screen Capture of Synthesized Path Through 4 Supercells
Input to West of supercell 1 exits to the East into supercell 3,
then exits to the South into supercell 4.

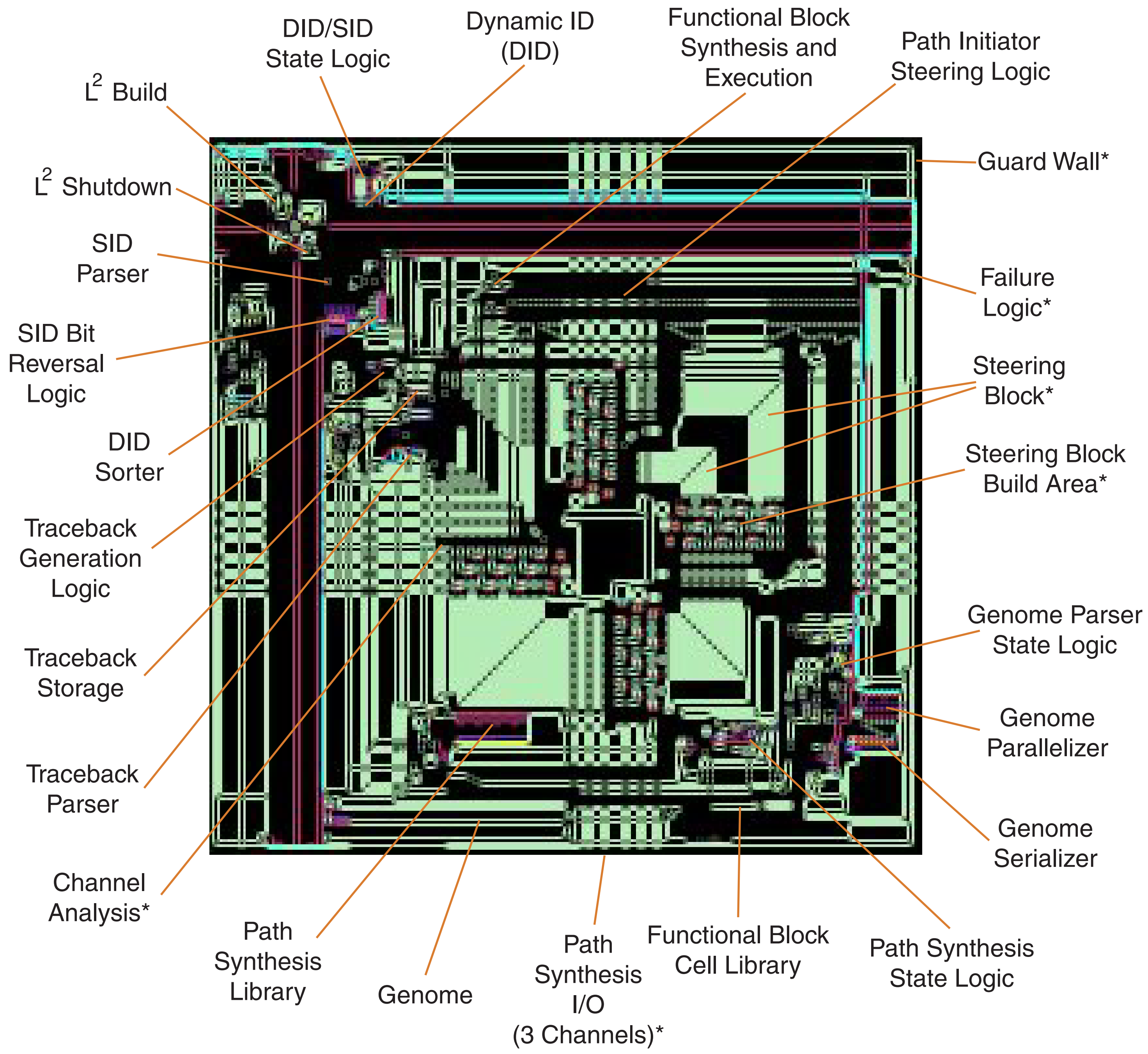


Figure 4.1
Supercell Block Diagram
Supercell is composed of a 270x270 array of Cell Matrix cells.
Components marked with "*" are present on all four sides of the supercell.

- Genome Parallelizer: Converts phase-shifted serial data inside genome to temporary parallel format for subsequent conversion to non-phase-shifted serial data.
- Genome Parser State Logic: Controls parsing of genome, identification of DST/SRC1/SRC2 nodes, and top-level control of other state machines.
- *Steering Block Build Area: Area of broken wires inside steering block, use to selectively connect pathways around or out-of the steering block.
- * Steering Block: Pre-built pathways for routing data through a Supercell, with externally-controlled exit points.
- *Failure Logic: Controls analysis of test data, records failures and activate Guard Wall.
- Path Initiator Steering Logic: Steers initial path synthesis commands to proper Path Synthesis Outputs. Also connects functional block's inputs to beginning of synthesized path.
- *Guardwall: Ring of guard cells for isolating faulty cells.
- Functional Block Synthesis and Execution: Configures the working functional block of a supercell.
- DID: Stores supercell's Dynamic ID.
- DID/SID State Logic: Controls all SID/DID circuitry.

Features marked with a "*" appear on all four sides of the supercell, though only one instance has been annotated in Figure 4.1.

4.1 Supercell Usage

This section describes how the supercell is used: its I/O ports, the commands it accepts, and how it is customized for a particular genome.

4.1.1 I/O Locations

The supercell contains a number of I/O lines along its edges. Table 1 shows the I/O assignment along the Western edge of a supercell. Most of these lines are not used from outside the supercell network, but are used to exchange information with other supercells. A few lines (described in the following sections) are used for sending the first Cell Matrix test commands into the Cell Matrix, configuring the first supercell, subsequently testing and configuring additional supercells, initiating genome parsing and final circuit synthesis, and finally interacting with the synthesized target circuit. Note that the set of I/O lines is replicated on the other three sides of the supercell (with 3 exceptions noted below). The I/O lines are arranged so that one supercells' output is adjacent to a neighboring supercell's corresponding input, and so on. However, if all configuration strings are sent into the Western edge of the Cell Matrix, then the location of the I/O lines along other edges of the supercells is irrelevant.

The entire *circuit synthesis process*, from empty Cell Matrix to final implementation of the desired target circuit, is driven by sending in a series of configuration strings through ten inputs to the Cell Matrix, and hardwiring an additional 11th input to a value of one. Of those ten inputs though, three are responsible for general configuration control, one is used for building the first supercell, two are used for fault testing, and the other four are used sporadically. In reality, a single input accounts for over 99% of the actual information sent to the Cell Matrix during the circuit synthesis process.

4.1.2 Supercell Circuit Synthesis Procedure

The circuit synthesis process involves the following steps:

1. Set LINK IN (33) and [0,0]FLAG (17) TRUE. This directs the first supercell to act as the initial master [0,0] supercell. The LINK IN line causes a L2 network to be built as supercells are successively configured.
2. Set the Side Select (M1/M0) inputs to 11 to indicate activity to the East. Send the fault detection sequence (for the East) into the Cell Matrix. This sequence will test areas about to be configured as supercells, activating guard walls between good supercells and faulty regions. This fault detection sequence is independent of the desired target circuit.
3. Feed the supercell configuration sequence (for the East) into the Cell Matrix. This causes a next set of supercells to be configured along one perimeter of the existing supercell collection. Note that for a specific genome (i.e., a particular target circuit), there is a single fixed configuration string for supercell configuration.

Column #	Direction (Relative to supercell)	Description
4*	Output	Serialized Dynamic ID
11	Output	Serial Data Exchange (All serial data)
17*	Input	[0,0] Flag (Supercell is at [0,0])
21	Output	State Signal (Start ID assignment)
22	Input	Edge Sense (Adjacent supercell exists)
23	Output	COMPARE (Expected return from fault test)
24	Output	GO (Do the fault test)
25	Output	CC (Configuration Channel of wire)
26	Output	PC (Program Channel of wire)/ GENOME GO+ (Start parse and build)
27	Output	BREAK (Misc wire control)/CHANNEL INC+
28	Output	M1-Side Select (MSB)
29	Output	M0-Side Select (LSB). [M1,M0]=00:N 01:S 10:W 11:E /CHANNEL CLEAR+
30	Output	Link (L2 network link)
31	Output	Lock (Already on L2 network-do not send link)
32	Input	Lock
33	Input	Link
34	Input	M0/CHANNEL CLEAR+
35	Input	M1
36	Input	BREAK/CHANNEL INC+
37	Input	PC/GENOME GO+
38	Input	CC
39	Input	GO
40	Input	COMPARE
41	Input	Edge Sense
42	Input	State Signal
52	Input	Serial Data Exchange
73*	Bidirectional	Functional Block I/O
90	Bidirectional	Shutdown

* These lines appear only on the Western edge of the supercell
+These lines are multi-use. The second function applies after the STATE signal has toggled.

Table 4.1
Major I/O Locations Along Western Edge of Supercell.
Most inputs correspond to complimentary outputs.

4. Repeat steps 2 and 3 three times; once using configuration sequences for the South, once for the West, and once for the North. For each sequence, [M1,M0] is set accordingly. Note that, if desired, the sequences can simply be concatenated into a single long configuration string, as opposed externally arbitrating which configuration strings are sent[9].

[9] In general, separate configuration strings sent in succession or sent repeatedly can always be combined into a single long configuration string. We will suppress further comments on this fact.

Steps 2-4 are repeated n times in order to synthesis up to $(2n-1)^2$ supercells.

5. Toggle the STATE (42) line. This causes the ID assignment engines to start. The system must be left to run for n clock cycles in order to assign n dynamic IDs. During this period, configuration strings of all 0s may be delivered to the system. Note that following this, the multi-use lines (M0, BREAK and PC) take on their secondary roles.
6. Set PC (37) to TRUE, and wait as many clock cycles as the longest path through the supercell network (normally 2N for an NxN network, but 4N is safer, as it allows for indirect routes around faults).
7. Set SHUTDOWN (90) to TRUE. This break the L2 network so that other supercells may rebuild it from themselves. This also drops PC to most supercells.
8. Set PC to FALSE. Now PC has dropped in all supercells, thus initiating executing of the genome parsing engines. Wait one clock cycle
9. Set SHUTDOWN to FALSE. The genome parsing engines are now all running, and circuit synthesis begins.

From this point forward, no outside intervention is necessary. The collection of supercells acts collectively as a parallel state machine, arbitrating node assignments among themselves, differentiating into the functional blocks necessary to implement the final circuit, and establishing connectivity among themselves.

Completion of the circuit synthesis process can be detected by adding a flag to the end of the circuit's genome, so that the last piece of the circuit which is assembled will assert a particular output. That output can then be sensed to indicate completion of the circuit synthesis.

4.2 Configuration String Generation

The fault detection strings are fixed, regardless of the desired target circuit. Using Java code, these strings are generated once, based on the size of the sueprcell (270x270) and the direction in which fault testing is to occur (N/S/W/E). The resulting configuration strings are approximately 9.9 million cycles in length.

The configuration strings used for synthesizing supercells depend on the genome of the desired target circuit. They are generated as follows:

1. As described in Section 3.4.6, a genome describing the desired target circuit is generated by hand (in the future, this step could be automated by implementing a translator from say Verilog. Such is translator is straightforward to create).
2. The genome is placed in a file and sent to a compiler we have developed (a C program called "genome_compile"). This program reads a fixed base supercell binary file (fulltrc.bin.saved), embeds a coded copy of the genome into it, and creates a new binary image (fulltrc.bin) containing the genome.
3. fulltrc.bin is sent to a Java program called "final_fault" along with a side specification (n, s, w or e). The program creates a file "final_fault_x.seq." where x is the indicated side. This .seq file is the final configuration string for synthesizing a set of supercells along an edge (indicated by x) of the supercell collection's perimeter.

The resulting configuration strings for supercell synthesis are approximately 1.2 million cycles in length.

Once the final target circuit has been synthesized, it will be necessary to interact with it from outside the Cell Matrix. This is done via the Functional Block I/O lines (row 73 in each supercell). As described above, inside the circuit's genome, nodes with low Dynamic IDs (DIDs) should be assigned to external I/O functions, so that they will be placed along the Westernmost edge of the Cell Matrix. There are two things to note here:

1. If the edge sensing feature (Section 3.4.8) has been enabled, then supercells along this Westernmost edge will not normally be assigned DIDs, since they are not themselves surrounded by supercells. Specifically, they are not adjacent to supercells to their West, since there are no Cell Matrix cells at all to their West. This situation can be avoided, however, by asserting the EDGE SENSE input to each supercell. This input is situated at row 22 along the Western edge of each supercell. Note that fir the upper left supercell (the supercell tagged as [0,0]), it is also necessary to assert the EDGE SENSE input along the Northernmost row. The EDGE SENSE input in that row is located at column 41.
2. Because of the possibility of faults near the Western edge of the Cell Matrix, it is not possible to predict the exact location of a supercell with a given DID. For this reason, there is an output available (only along a supercell's Western edge) which continually sends the supercell's DID. This output is located at row 4, and can

be used to identify which supercells are assigned which DIDs. In this way, external circuitry can locate and attach to the external I/O lines of the final synthesized circuit.

4.3 Supercell Internals

This section presents some of the details concerning the internal operation of supercells.

4.3.1 Guard Walls

Because of the behavior of a cell when it is in C-mode (configuration mode), it is possible to build circuits which will isolate a section of the Cell Matrix from another sections. Such isolation circuits are generally called *Guard Walls*. Many different types of guard walls can be implemented. Figure 4.2 shows a simple one, which is two cells wide. In Figure 4.2, the region labeled GOOD is assumed to have perfectly-functioning cells, while the region labeled BAD may have one or more faulty cells. Note that the good region includes the guard wall. This is critical, since the guard wall can not be guaranteed to work properly if it itself contains faulty cells.

Each cell in the column labeled the active column is asserting its Western C output, thereby placing the neighboring cell in the passive column into C mode. Recall, a cell in C mode is not allowed to assert any of its C outputs. In particular, it is impossible for a cell in the passive column to modify the configuration of a cell in the active column (remember, we have placed the guard wall such that the passive and active columns are guaranteed to be operating properly). If the cells in the active column are configured to output only 0s to their East, North and South, then no action by cells inside the BAD region can affect any cells inside the GOOD region. This is thus a perfect isolation circuit.

Of course, in Figure 4.2, there is nothing stopping a cell in the BAD region from affecting cells to the North of the GOOD region, and then affecting GOOD cells from the North. By extending guard walls around the perimeter of a region, a *Guard Ring* can be built to completely isolate a region.

4.3.1.1 Semi-Permeable Guard Cells

In the case where a region is completely surrounded by guard cells, the question arises, how do any external circuits interact with the guarded circuit? *Semi-Permeable Guard Cells* allow controlled interaction through a guard wall. Specifically, they allow data to pass bi-directionally through the wall. However, if a cell in the passive column is placed in C-mode, even for a brief instance, the active cell will **then** assert its C output, thus activating the guard wall (and blocking subsequent data exchanges through that section of the wall). To implement such a circuit, all that is necessary is to have the passive cell output a 1 to the active cell. If the passive cell is placed in C mode, its D output will drop. This is detected by the active cell, which responds by asserting C out to the passive cell, thus switching to a standard guard cell arrangement.

Both types of guard cells are important in circuits such as supercells. If a neighboring region is detected to be bad, it is necessary to prevent both C and D inputs to certain cells. For example, certain D inputs cause configuration of cells elsewhere in the network, or cause state machines to begin execution. These inputs must be guarded with standard guard cells. However, since there are no supercells to the left of the Westernmost supercells, the fault testing logic will conclude that there are failed cells to the left, and will activate the guard walls along the Western edge of those supercells. If these walls were composed only of standard guard cells, there would be no way to interact with those Westernmost supercells. Therefore, semi-permeable guard cells are used for I/O lines such as the serialized DID and the external functional block I/O.

4.3.2 Serial Data Formats

Most serial data used inside the supercell and exchanged among supercells is coded in a special *double bit* format[10]. In this format, a particular bit stream in represented as a longer bitstream, where each bit appears twice in succession. For example, the bitstream "1101001" would be represented as "11110011000011." The reason for this

[10] This has previously been disclosed in a New technology Report, "Technique for processing serial data without system-clock frequency doubling."

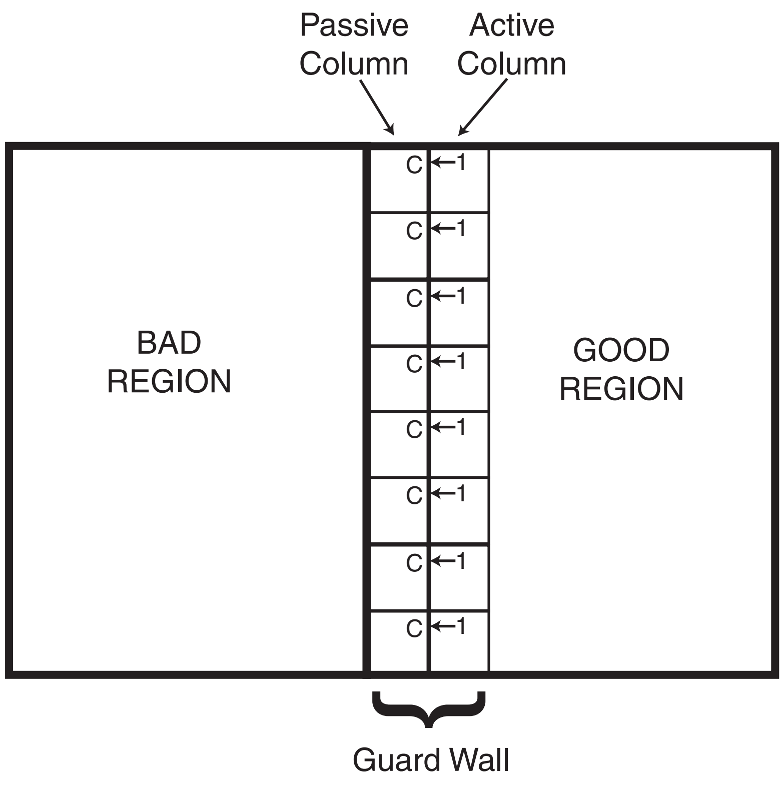


Figure 4.2
Simple Guard Wall
The cells making up the actual guard wall are all contained in the good region.
Cells in the Active Column prevent cells in the Passive Column from entering D mode.
Therefore, cells in the bad region have no access paths to cells in the good region.

is that it is easy to generate a signal (called a *high frequency clock* or HFC) inside the Cell Matrix which toggles every time the system-wide clock drops (1→0). This means the HFC is actually generated at **half** the system clock frequency, while bit streams being written to and read from a cell's configuration memory are clocked at the same frequency as the system clock (twice the HFC's frequency). Instead of doubling the frequency of the HFC (which requires introducing delay circuits and complicates timing), the double bit format effectively doubles the HFC frequency, by halving the datarate of the serial bit streams.

Throughout the remainder of this discussion, we will simply talk about the actual binary data streams, understanding that they are actually implemented with twice as many bits, unless indicated otherwise.

4.3.2.1 Special Bit Streams

A number of standard bit streams are used for parsing serial data. Throughout this section, the "first" bits are the ones which enter or exit a C-mode cell first. Most bit streams are viewed as 64 bits (half the size of a cell's configuration memory), or as multiples of 64 bits.

VALID BITS: Most bit streams begin with a single TRUE bit to indicate that valid data follows. For example, in Static ID (SID) arbitration, all SIDs begin with a 1. In this way, if there is no supercell on a particular side, the SID received from that side will be all 0s, including the valid bit. This SID will thus be considered invalid, and will not contribute to the SID calculation.

VSYNC/HSYNC: SIDs are stored in 64 bits, broken into two pieces. The first 32 bits correspond to the column, and the last 32 bits correspond to the row. Therefore, a valid column address is indicated by checking the first bit of the SID, while a valid row address is indicated by checking bit 32. Of course, these bits are valid or invalid together. The VSYNC bit stream has a single 1 in the beginning. When ANDed with an SID, it generates a pulse at the beginning of the column data. HSYNC has a single 1 in bit position 32, and is used to detect the beginning of the row data.

Note that this is an exception to the standard bit doubling. Figure 4.3 shows the actual behavior of VSYNC relative to a bit-doubled bit stream ("DATA"). Suppose the actual (undoubled) DATA stream is 110000000… The first actual bit of VSYNC is 0, so VSYNC actually looks like 0100000000…

At t_1, the first DATA bit (VALID) *should* be available, but may be delayed due to propagation delay. At t_2, the actual DATA valid bit is available, and remains stable through t_5 (because of the bit doubling). Therefore, when the VSYNC signal goes high at t_4, we AND it with the true DATA VALID bit. In other words, we are guaranteed that we can sample a stable VALID bit on the rising edge of VSYNC.

In general, special bit streams which are used to analyze other serial streams may use this technique to avoid race conditions.

DSYNC: This is an alternating pattern of 1s and 0s used to pick off the actual data bit from a VALID-prefixed bit stream. The undoubled DSYNC looks like 0001010101010101…0101

4.3.3 Genome Packing

The genome of a target circuit is stored in a series of Cell Matrix cells. Recall (Figure 3.20) that each block of the genome contains six pieces. Each piece is represented by an integer between 0 and 255. Using the standard ordering, each piece is stored as an 8 bit integer, immediately followed by the bits of the next piece. There are no valid bits contained in the genome block's serial data. However, it is understood that all node numbers are non-zero. Therefore, a DST node of 0 is used as a special *End Of Genome* flag.

Since each piece of the block is 8 bits and there are 6 pieces, a single block requires a total of 48 bits (which is doubled to 96 bits as usual). Since a Cell Matrix cell's configuration memory can hold 64 (doubled) bits, each block is stored in a single cell. The extra 16 bits are reserved.

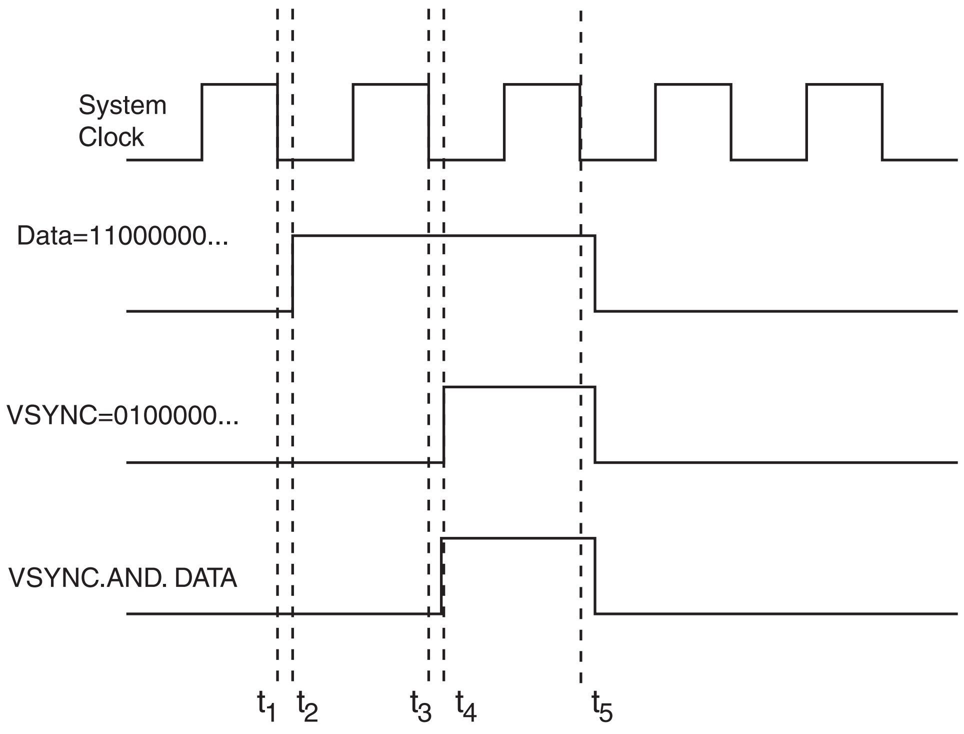


Figure 4.3
Use of VSYNC for glitch-free sampling of DATA.
DATA bits are doubled, while VSYNC only pulses on 2nd bit.
Therefore, when VSYNC rises, we know DATA is and has been stable.

With 8 bits per piece, the maximum node number is 255. Since node 0 is invalid, this means there can be 255 nodes in the final circuit. Of course, this number can be extended if desired by modifying this genome packing scheme. Also, note that this allows up to 256 different cell codes for the configuration of the supercell's functional block. However, the current cell library only supports 64 cell codes. The two most significant bits of each cell code in the genome are ignored.

4.3.4 Traceback Packing

The traceback string used to control path synthesis is stored as a series of 4-bit packets, formatted as:

$$V\ S_1\ S_0\ T$$

V is a valid bit, and indicates a valid segment in the traceback string (in particular, the beginning of a traceback string). S_1 and S_0 indicate the side through which the traceback path passes, and is encoded as follows:

S_1	S_0	Side
0	0	E
0	1	N
1	0	W
1	1	S

T is the *terminal* flag, indicating the end of the traceback string (and hence the end of the path).

A single Cell Matrix cell can thus store 16 of these traceback packets. In the current supercell implementation, 8 cells are used for storing the full traceback, thus allowing paths up to 128 supercells in length.

4.3.5 Path Synthesis Details

There are two details which have been skipped in the previous discussion about the steering block and path synthesis. These details concern the initiation of the path synthesis, and the final connection between a synthesized path and the inputs to a functional block.

Path synthesis initiation is achieved through a simple multiplexing circuit called the *Path Initiator Steering Block* (PISB), which is shown in Figure 4.4. When the DST supercell is ready to synthesize a path, it uses the first traceback packet to direct the PISB to steer its wire building commands into the appropriate exit path of the steering network. Thus, if the first path direction should be to the East, the PISB's SIDE inputs will be set so that the wire building commands are directed to the supercell to the East. The CHANNEL inputs indicate which channel (A, B or C) should be used. The InUse input indicates that the other inputs are valid. Note that there are two independent PISBs, since a DST supercell may build up to two pathways to connect to the SRC of its functional block's two inputs.

The same PISB is also used to direct data received from a SRC cell into an input of the functional block.

4.3.6 Delivery of Functional Block Output

At the other end of a synthesized path is a SRC cell, whose functional block's output is to be delivered through the synthesized path to the DST cell. Recall the steering block of Figure 3.30 Input (1) feeds wire building commands to the cells in region (2). Building a straight wire segment connects W1 to W2 and moves control to the cell sin region (3). Another straight segment moves control to region (4). Normally, a bent segment would be synthesized here, to connect W3 to output (5). However, if an additional straight segment is built, W3 will be connected to W5, which terminates at region (6). At first, such a sequence of builds makes no sense. A path coming from the West should be made to exit to either the South, East or North. There is no reason for a path arriving from the West to exit to the West.

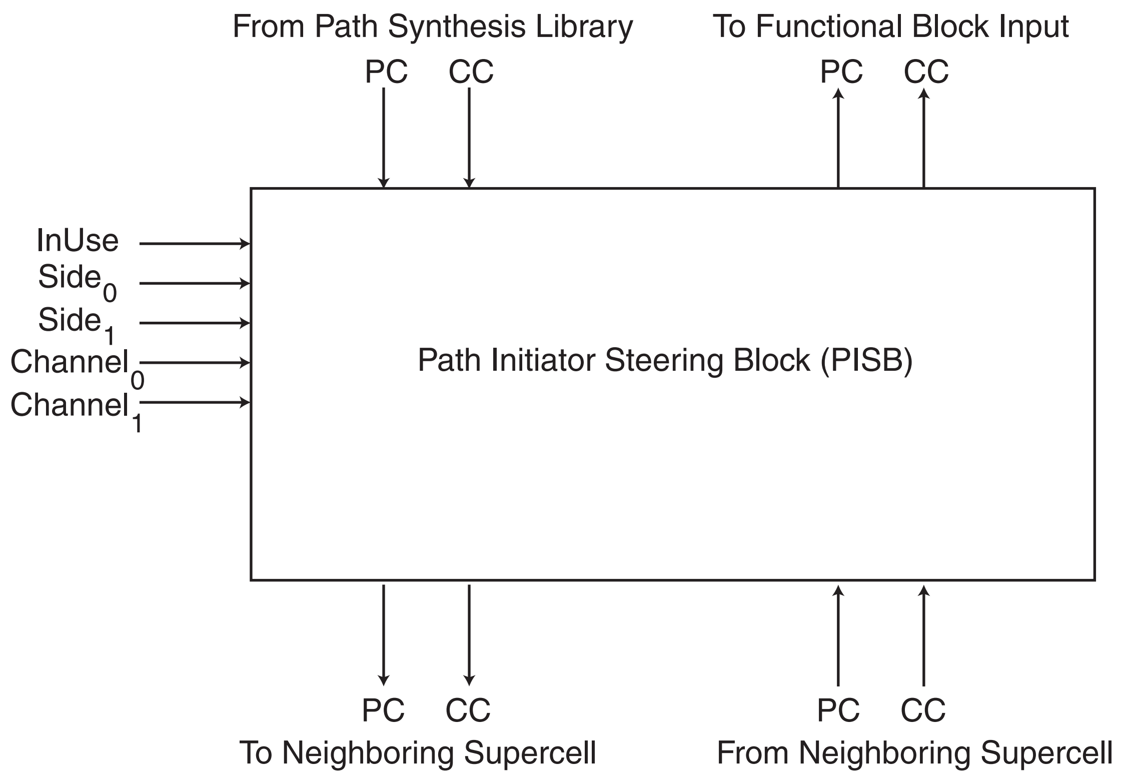

From Path Synthesis Library
PC
CC
To Functional Block Input
PC
CC
InUse
Side0
Side1
Channel0
Channel1
Path Initiator Steering Block (PISB)
PC
CC
To Neighboring Supercell
PC
CC
From Neighboring Supercell

Figure 4.4
Path Initiator Steering Block
Used to initiate path synthesis, and to connect path to a functional block input.
PC/CC from top left can be routed to a neigboring supercell's path synthesis I/O
PC/CC from a built path can be routed to a functional block input
Inputs on the left control the routing.
Supercell contains two PISBs, one for each functional block input.

This special sequence of builds is reserved for the terminal stage of path synthesis. Upon reaching the desired SRC supercell, four straight wire segment builds are issued, bringing control to region (6) in Figure 3.30 Region 6, as well as the termination points of all steering block paths, are connected to that supercell's functional block's output. Normally, this has no effect, since the path around the steering block exits before the termination point. However, when the path is built with four straight segments, the functional block's output will be sent around the steering block, into the next supercell along the path, continuing through all supercells in the path, finally reaching the DST supercell, where the network multiplexer will direct this path to one of the DST supercell's functional block's inputs. In this way, a SRC node's output is connected to a DST node's input.

4.3.7 Channel Switching

Recall that the steering block contains three independent channels. The choice of a channel to use is made globally. That is, during a specific path synthesis, **all** supercells will be working on the same channel. Channel 0 means all InUse flags will be ignored. This channel is used exclusively prior to starting of the genome parsing engine. Channel A is the default channel during path synthesis. If no path can be found from DST to SRC, all supercells will switch to channel B, and another path finding attempt is made. If no path can be found, all supercells will switch to channel C, and a final attempt is made. If no path is found, a *Hard Fail* is signaled, and the system fails. Section 7 will discuss options for dealing with this situation.

Channel switching can be initiated by asserting the CHANNEL INC signal. Note that prior to doing so, the current L2 network (which was built with channel restrictions) must be shutdown. CHANNEL INC is then asserted, after which the L2 network is rebuilt. L2 assembly with CHANNEL INC asserted ignores the InUse flags, thereby ensuring that **all** supercells receive the increment command. Following this, the L2 network can be shutdown, CHANNEL INC is de-asserted, and the L2 network is rebuilt. This L2 network will pay attention to InUse flags on the current channel.

Similarly, the CHANNEL CLEAR signal can be asserted to switch every supercell to channel 0.

5. Results

The behavior of supercells was simulated using a series of Cell Matrix cell-level simulation tools. This section explains how these simulations were performed, and discusses the results of the simulations.

5.1 Fault Testing and Parallel Configuration

As discussed in our first progress report, simulation of the fault testing and supercell configuration sequences is extremely slow on a sequential machine. Therefore, these tests were performed on a simplified version of the supercell, one which is 44x44 Cell Matrix cells. The process of constructing these sequences and the operation of these sequences is completely general, and extends quite naturally to supercells of any size and structure.

The figures used in this section are screenshots from the simulator's output. White regions are unconfigured cells, dark regions are configured cells outputting all 0s, blue regions are cells with at least one non-zero D output, and red regions are cells with at least one non-zero C output. Note that one of the failed cells will not have any impact on the tests which were run.

The normal configuration string consists of a repeating pattern of substrings for testing and configuring regions to each side. For example, the configuration might be:

> Eastern Test/Eastern Config/Northern Test/Northern Config/Western Test/Western Config/Southern Test/Southern Config.

However, in these tests, we varied the side on which testing and configurations were performed, mainly for more efficient simulation.

Throughout this section, we will sometimes refer to supercells by their [row,col] coordinates, where the upper left corner is [0,0].

Fault testing and supercell configuration sequences were first created (as described in Section 4 above), after which a series of tests were performed. Figure 5.1 shows the initial, empty Cell Matrix. The two crosses correspond to two failed cells, which we have simulated by configuring them to assert all their C outputs. In this configuration, these cells can not be accessed or reconfigured, and therefore are essentially faulty[11]. Starting with this initial matrix, a series of fault test and supercell configuration sequences were sent to the matrix.

Figure 5.2 shows the initial supercell synthesis, at location [1,0]. Normally, this would be initiated from the upper left corner of the matrix. In this case though, the build is being initiated from one block to the south. This allows for easier testing of the configuration steps which build to the North.

The next step is a fault testing sequence to the East. Figure 5.3 shows the system's status after this step. The fault, which was placed in a particularly critical location, prevented most of the test sequence from working. Many cells in the region under test were not reachable. However, as
soon as the first test failed, the controlling supercell detected the failure, and activated its guard wall on the East, seen as a red line.

The next step attempts to build supercells to the East of the current perimeter. However, since there is only one supercell configured, and its Eastern guard wall is activated, this step results in no change in the state of the Cell Matrix.

Figure 5.4 shows the result of the next step, which is a fault testing sequence to the North. Since all cells in that region are functioning properly, all tests pass, and the Northern guard wall is not activated. The pattern of cells in the Northern region are leftover from the test, but do no harm.

In Figure 5.5, a Northern supercell configuration sequence has been sent to the Cell Matrix, resulting in a new supercell to the north of the original. Note that the cells which were leftover from the previous test sequence have been successfully overwritten.

In Figure 5.6, another Eastern test sequence has been run. Again, all cells pass all tests, and a series of configured cells remain in the test region. Figure 5.7 shows the result of a subsequent Eastern supercell configuration sequence. The entire supercell collection now contains 3 good supercells, and one failed one.

Figure 5.8 shows the results of Southern test sequence. The region at the bottom left of the figure has passed all tests. However, the region containing the failed cell has failed at least one test. As a result, the supercell above that region has activated its Southern guard wall. Figure 5.9 shows the subsequent Southern supercell configuration sequence.

Figure 5.10 shows another Eastern test sequence. In this case, there are three supercells along the Eastern perimeter of the supercell collection. Therefore, three regions are tested in parallel. Of course, one region has already failed and is not tested because of the guard wall. The other two regions are tested, and pass all tests. Figure 5.11 shows the subsequent Eastern supercell configuration sequence results. The two new supercells are configured in parallel.

Next, a Northern test sequence is run (Figure 5.12). The supercell below the failed region detects the failure, and asserts its Northern guard wall. Also, all supercells in the top row will test the regions to their North. Since there are no Cell Matrix cells to the North of these supercells, these tests will quickly fail. As a result, each supercells in the top row has activated its Northern guard wall. The subsequent Northern supercell configuration sequence has no affect, since there are no non-failed regions along the Northern perimeter of the supercell collection.

[11] Additional tests were also performed using a fault-simulation feature of the simulator, allowing us to simulate stuck-at faults. The fault-detecting sequences worked perfectly in this scenario as well.

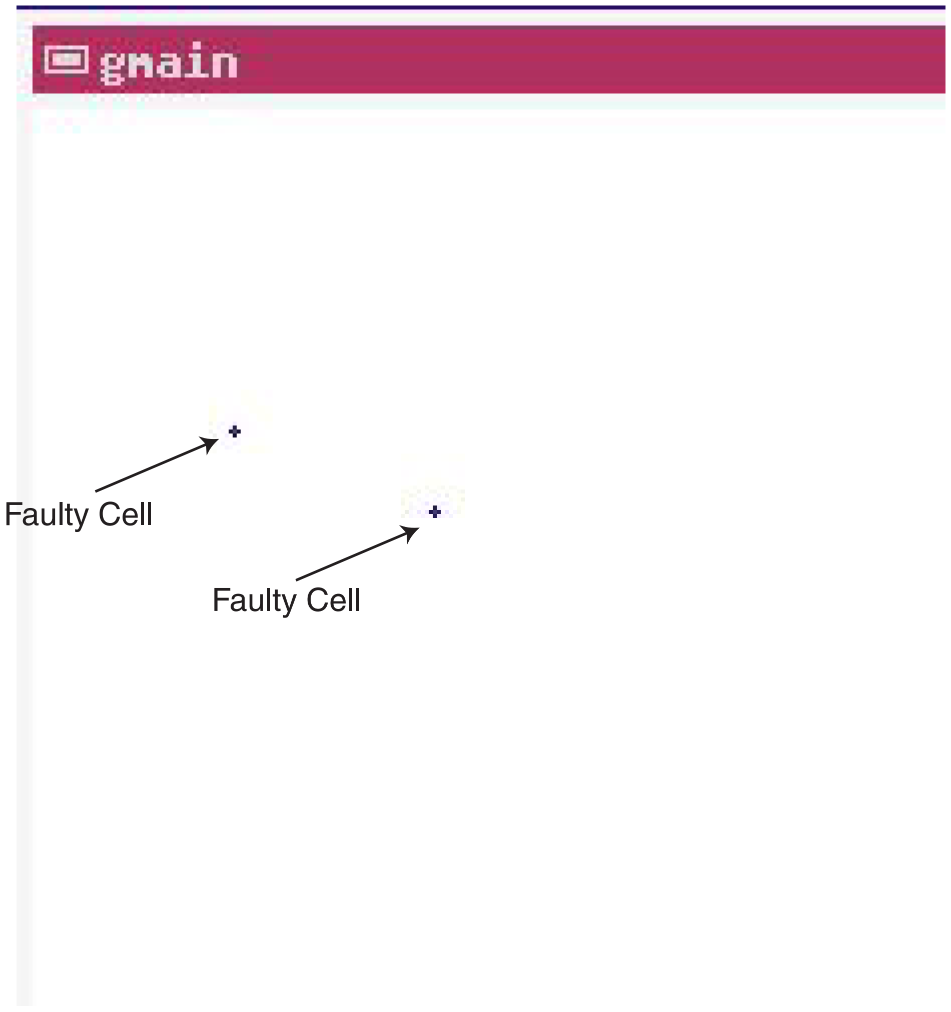


Figure 5.1
Empty Cell Matrix Containing Two Failed Cells

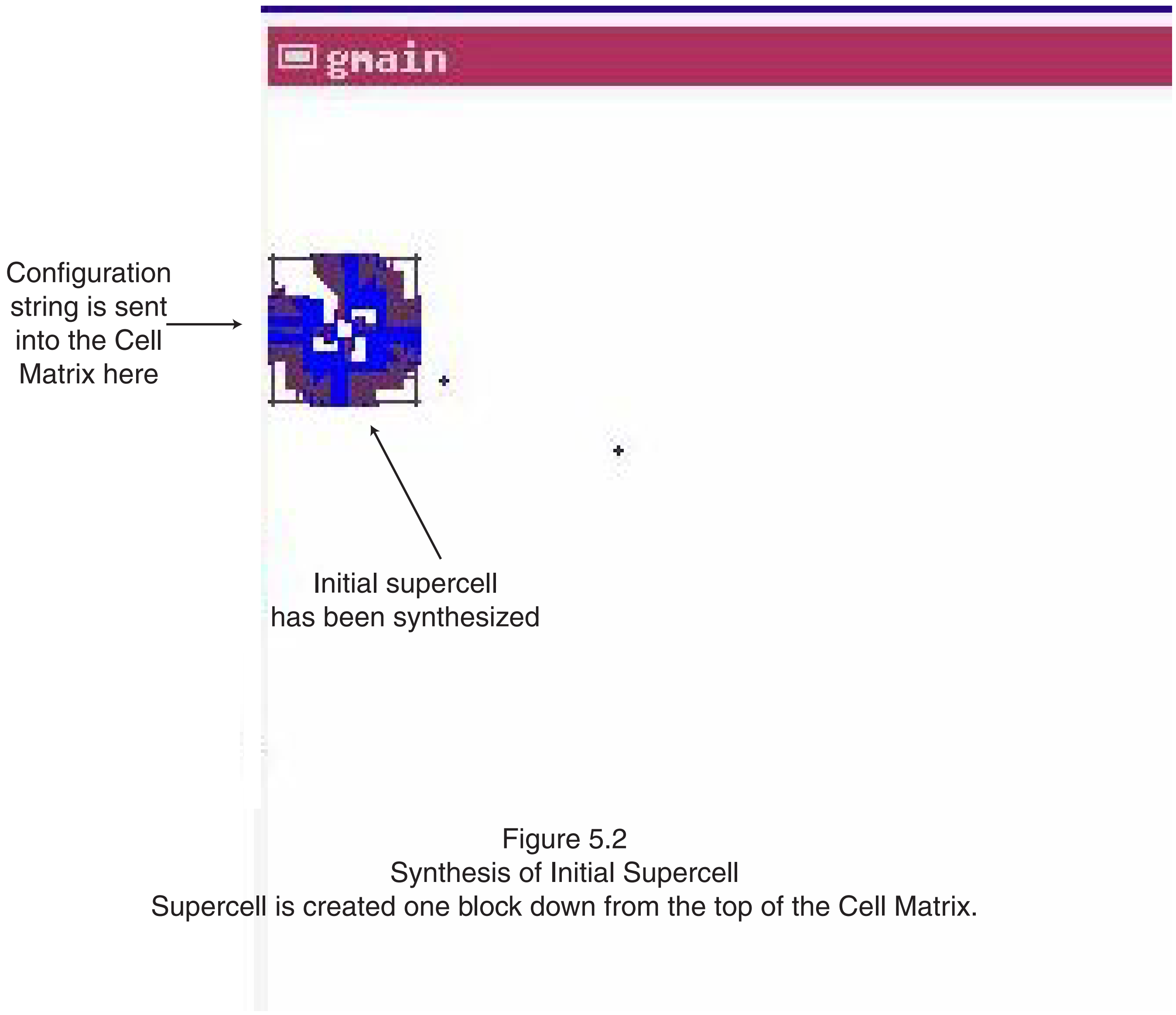

gmain
Configuration string is sent into the Cell Matrix here
Initial supercell has been synthesized

Figure 5.2
Synthesis of Initial Supercell
Supercell is created one block down from the top of the Cell Matrix.

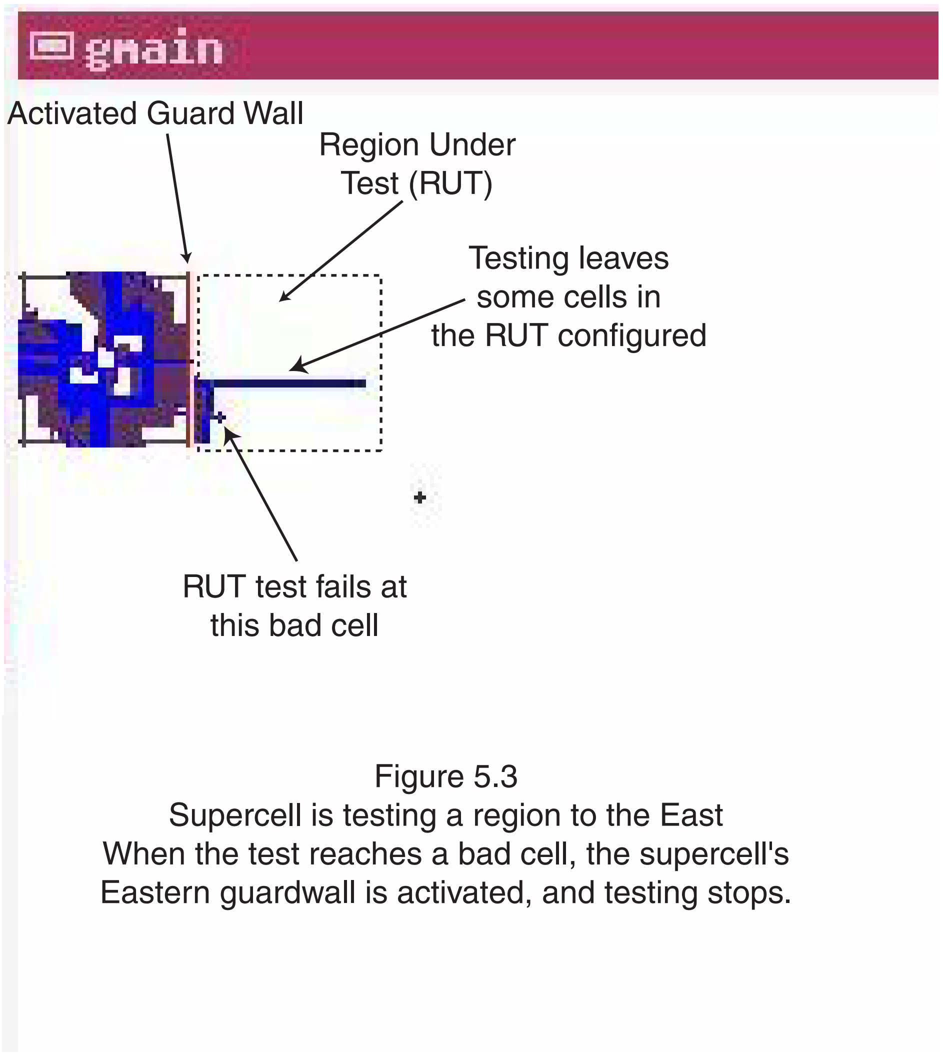

gmain
Activated Guard Wall
Region Under Test (RUT)
Testing leaves some cells in the RUT configured
RUT test fails at this bad cell

Figure 5.3
Supercell is testing a region to the East
When the test reaches a bad cell, the supercell's
Eastern guardwall is activated, and testing stops.

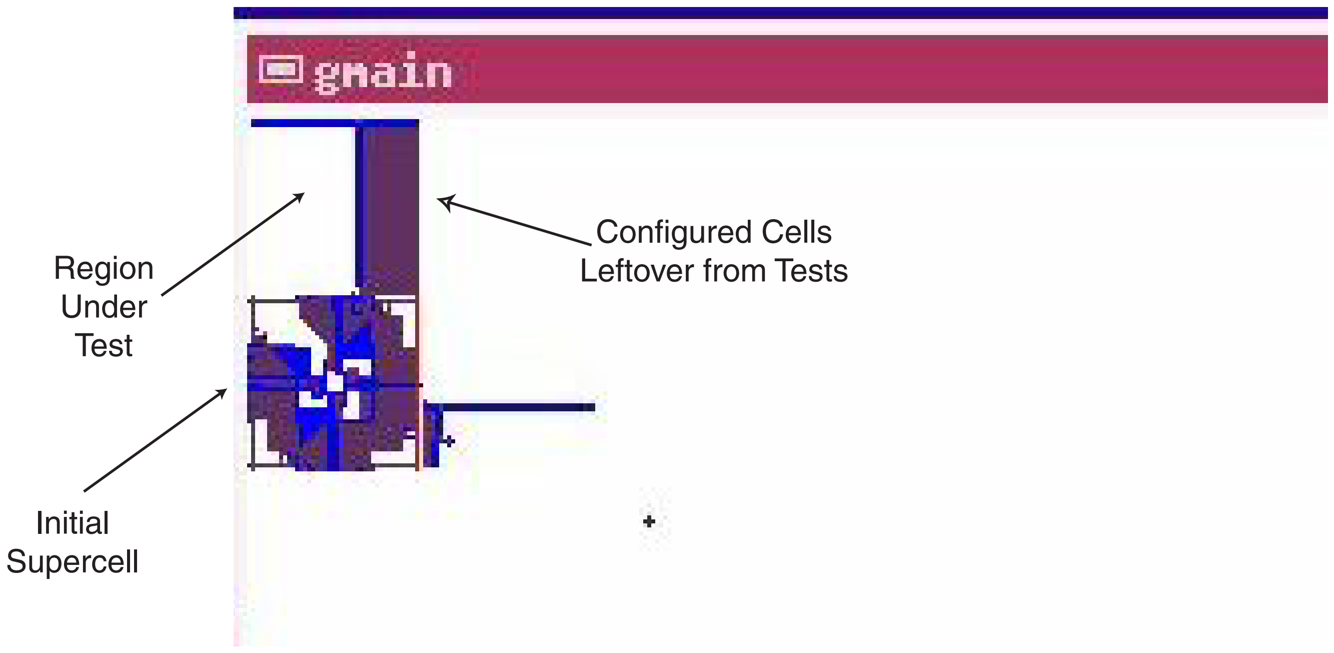


Figure 5.4
Initial supercell has finished testing the region to the North
The dark areas in the RUT indicate cells with non-empty configuration memories.
This pattern is indicative of a successful test.

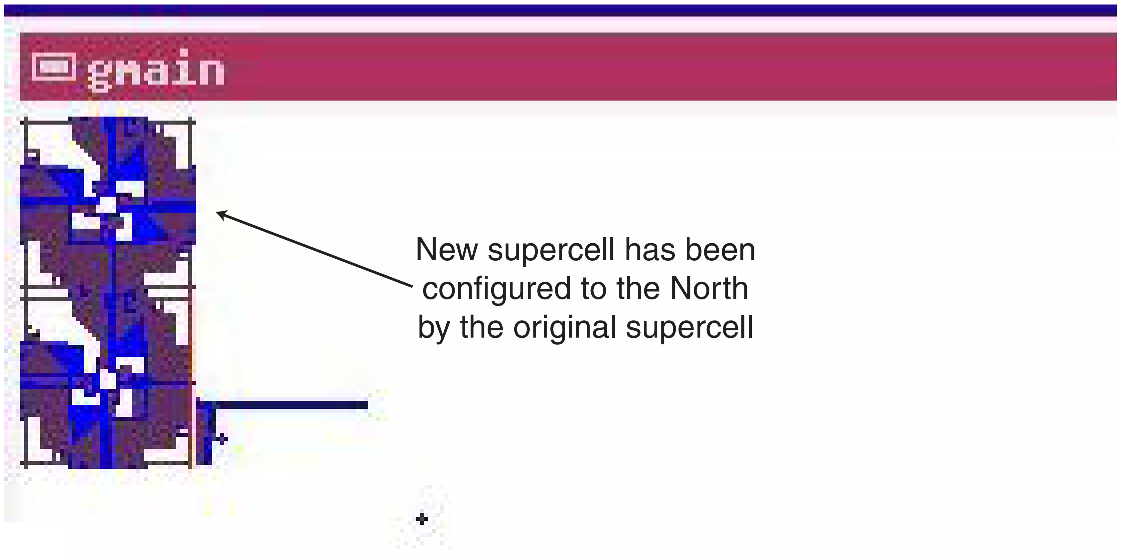

gmain
New supercell has been configured to the North by the original supercell

Figure 5.5
Initial supercell has configured a new supercell to the North.

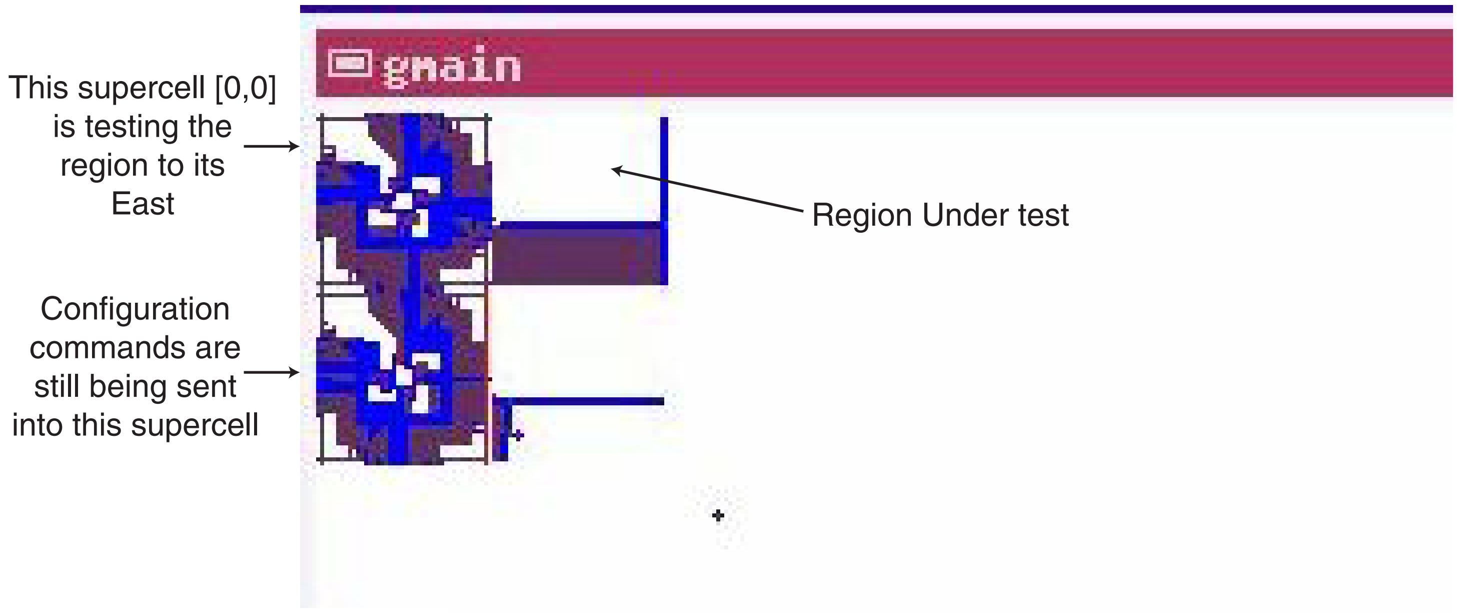

gmain
This supercell [0,0] is testing the region to its East
Configuration commands are still being sent into this supercell
Region Under test

Figure 5.6
Configuration commands sent into the initial supercell are directed to supercell [0,0], which then tests the region to its East. The RUT has passed all tests.

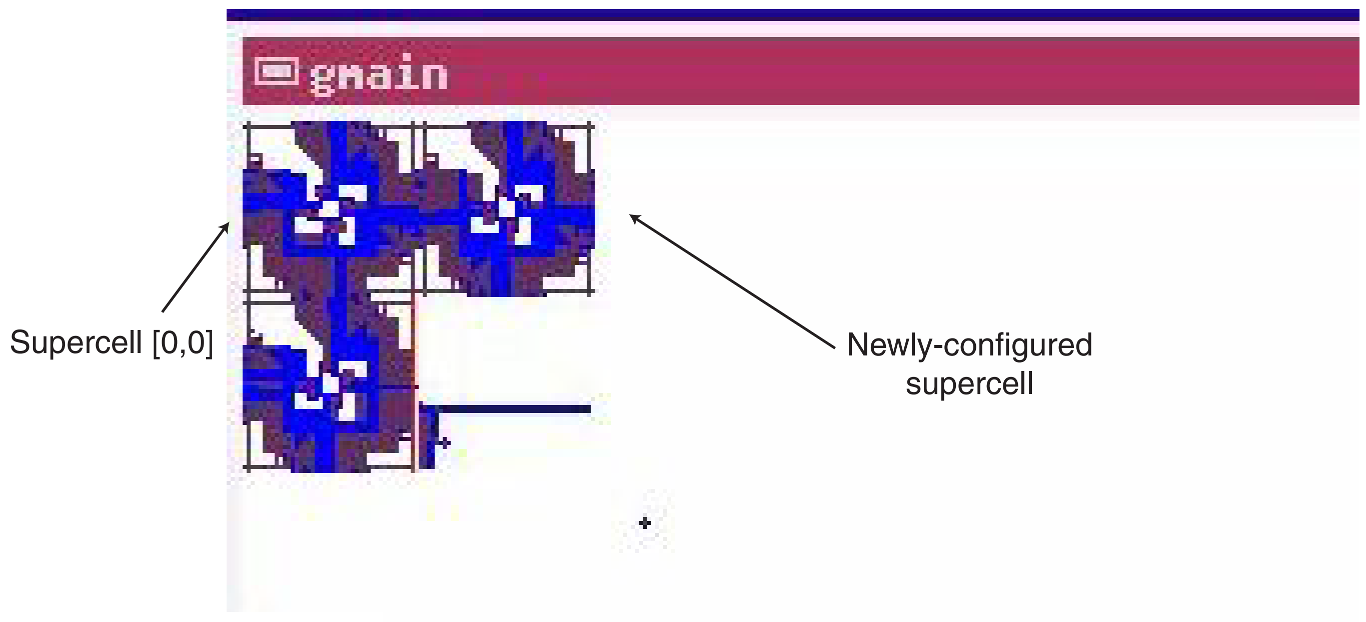


Figure 5.7
Supercell [0,0] has configured a new supercell ([0,1]) to its East

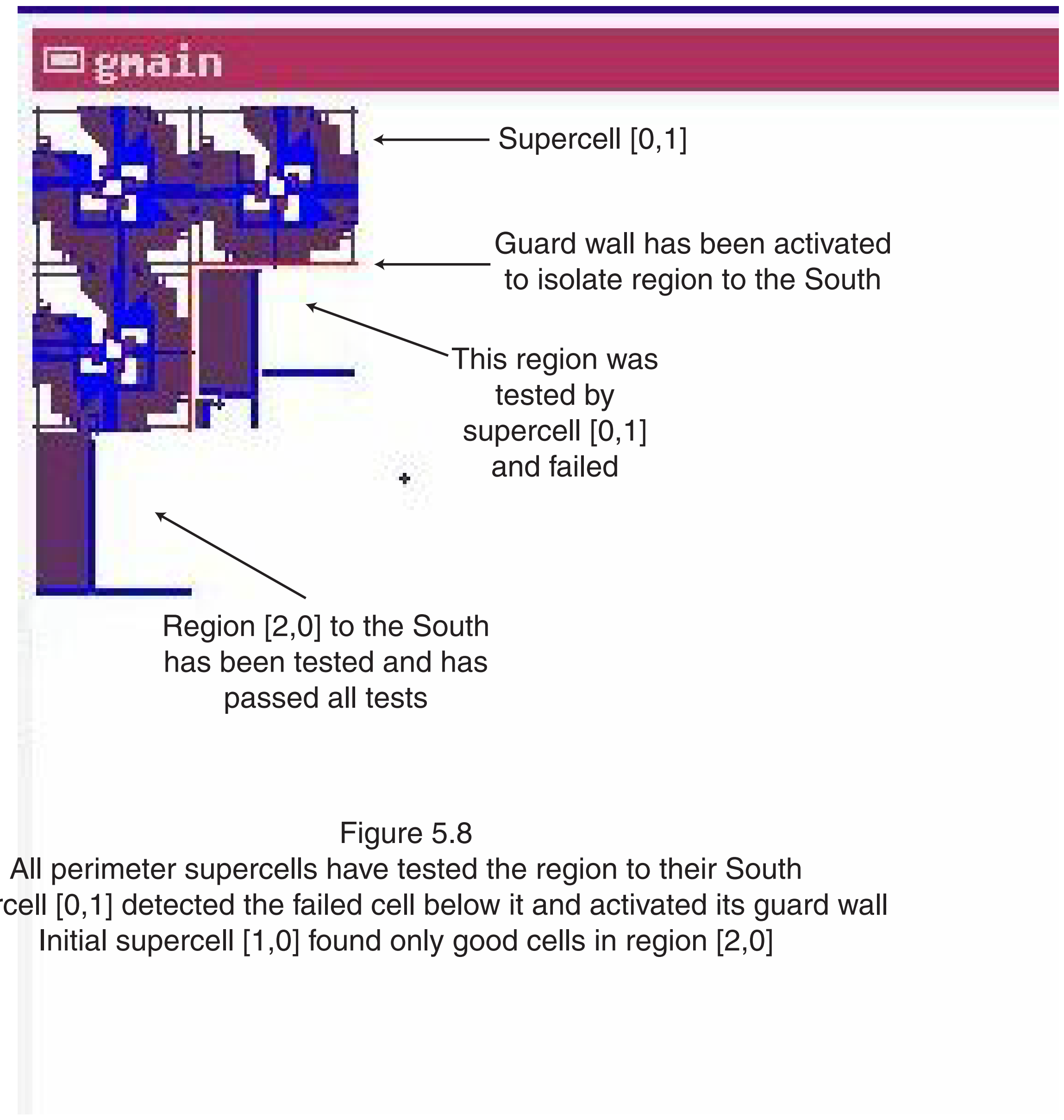

gmain
Supercell [0,1]
Guard wall has been activated to isolate region to the South
This region was tested by supercell [0,1] and failed
Region [2,0] to the South has been tested and has passed all tests

Figure 5.8
All perimeter supercells have tested the region to their South
Supercell [0,1] detected the failed cell below it and activated its guard wall
Initial supercell [1,0] found only good cells in region [2,0]

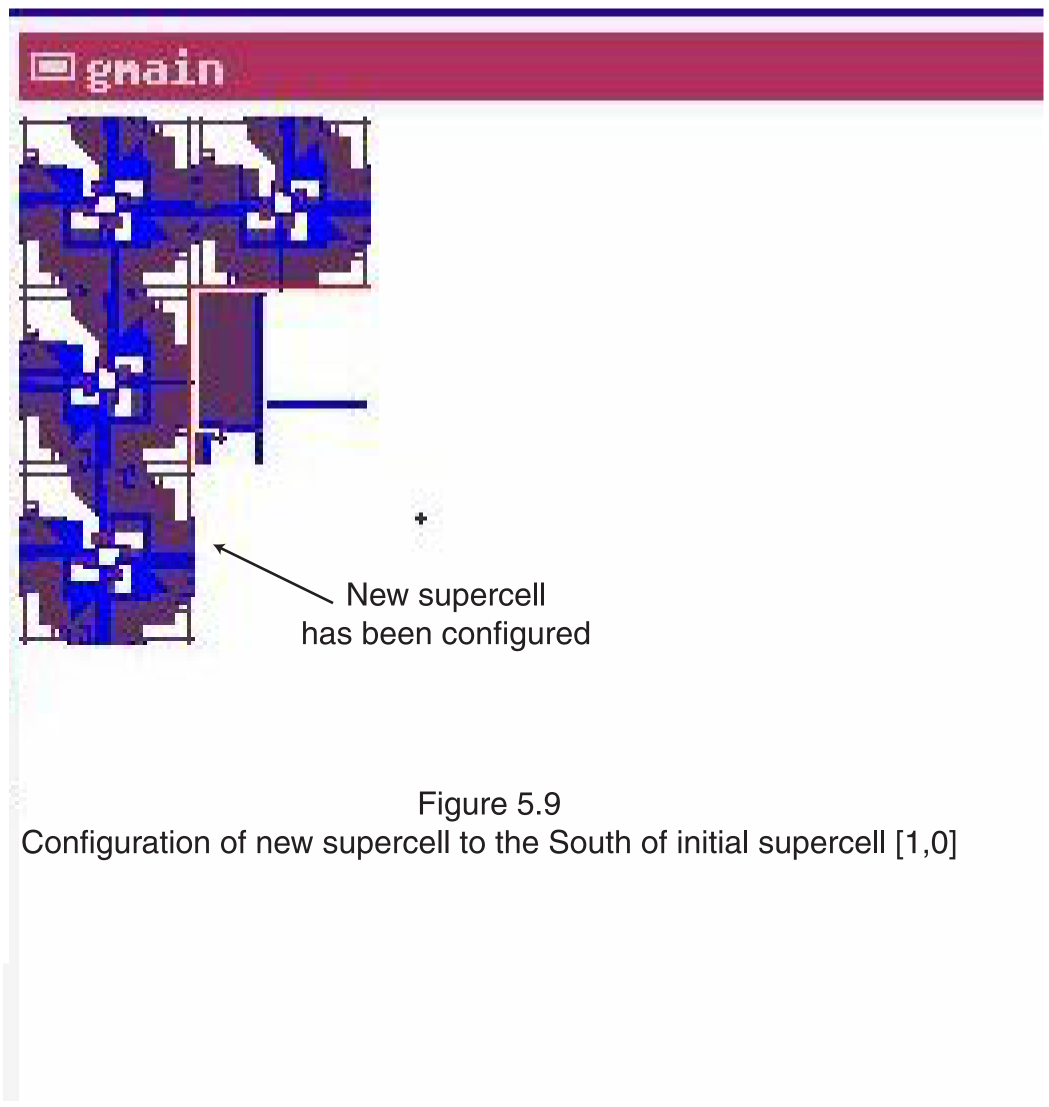


Figure 5.9
Configuration of new supercell to the South of initial supercell [1,0]

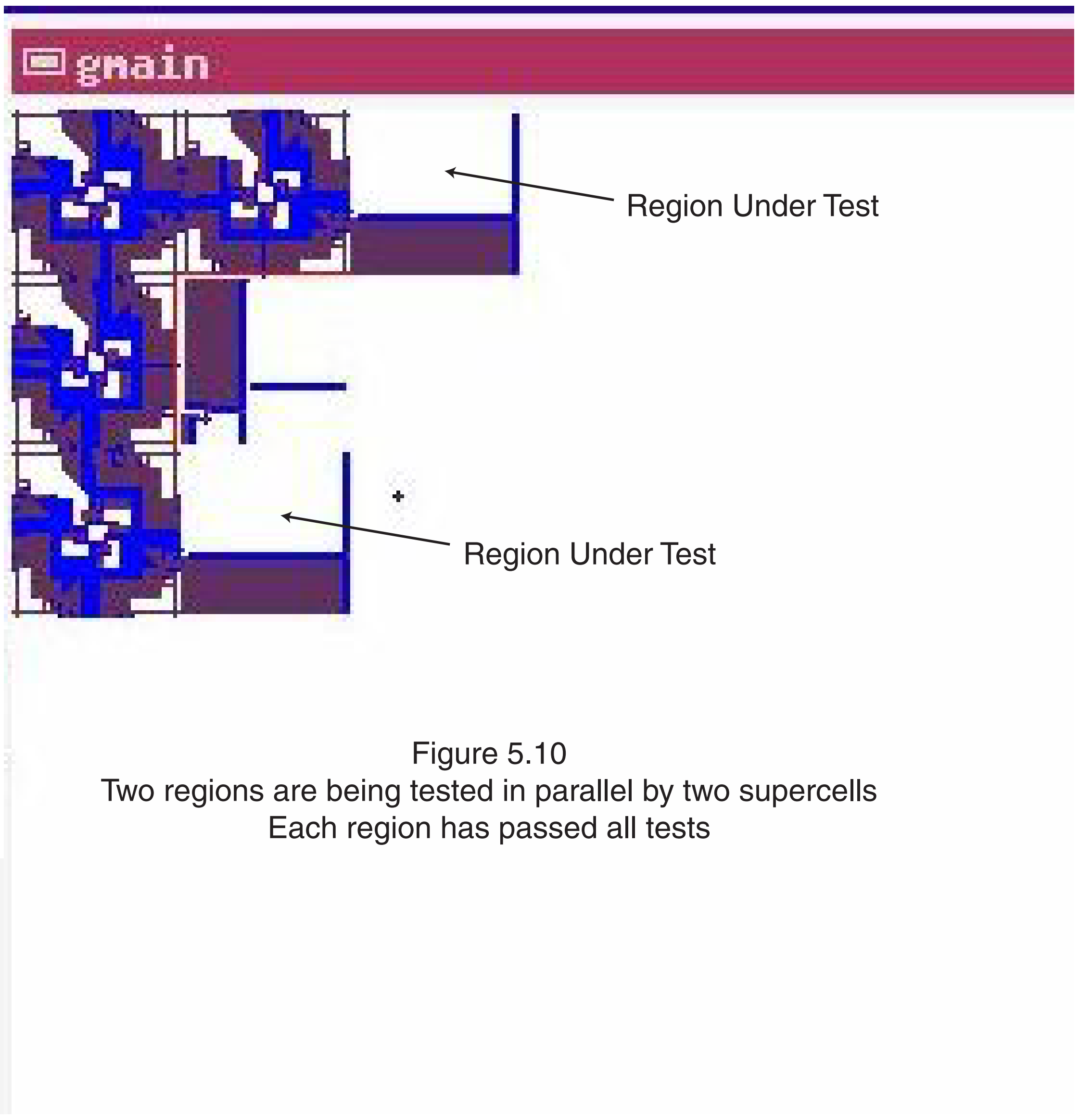


Figure 5.10
Two regions are being tested in parallel by two supercells
Each region has passed all tests

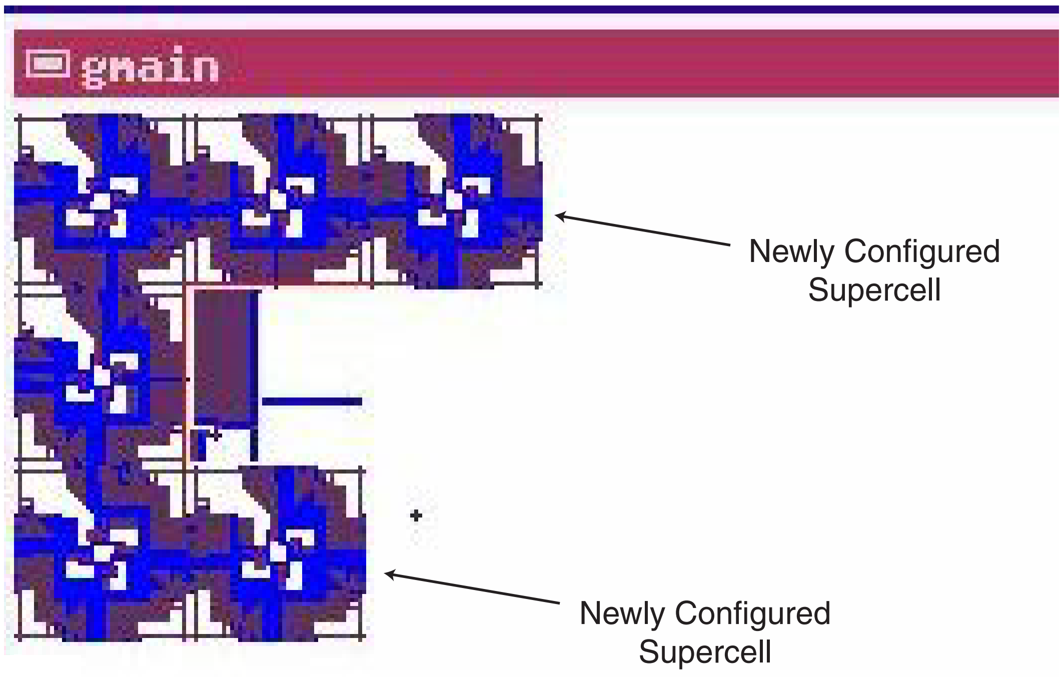


Figure 5.11
Two new supercells have been configured in parallel

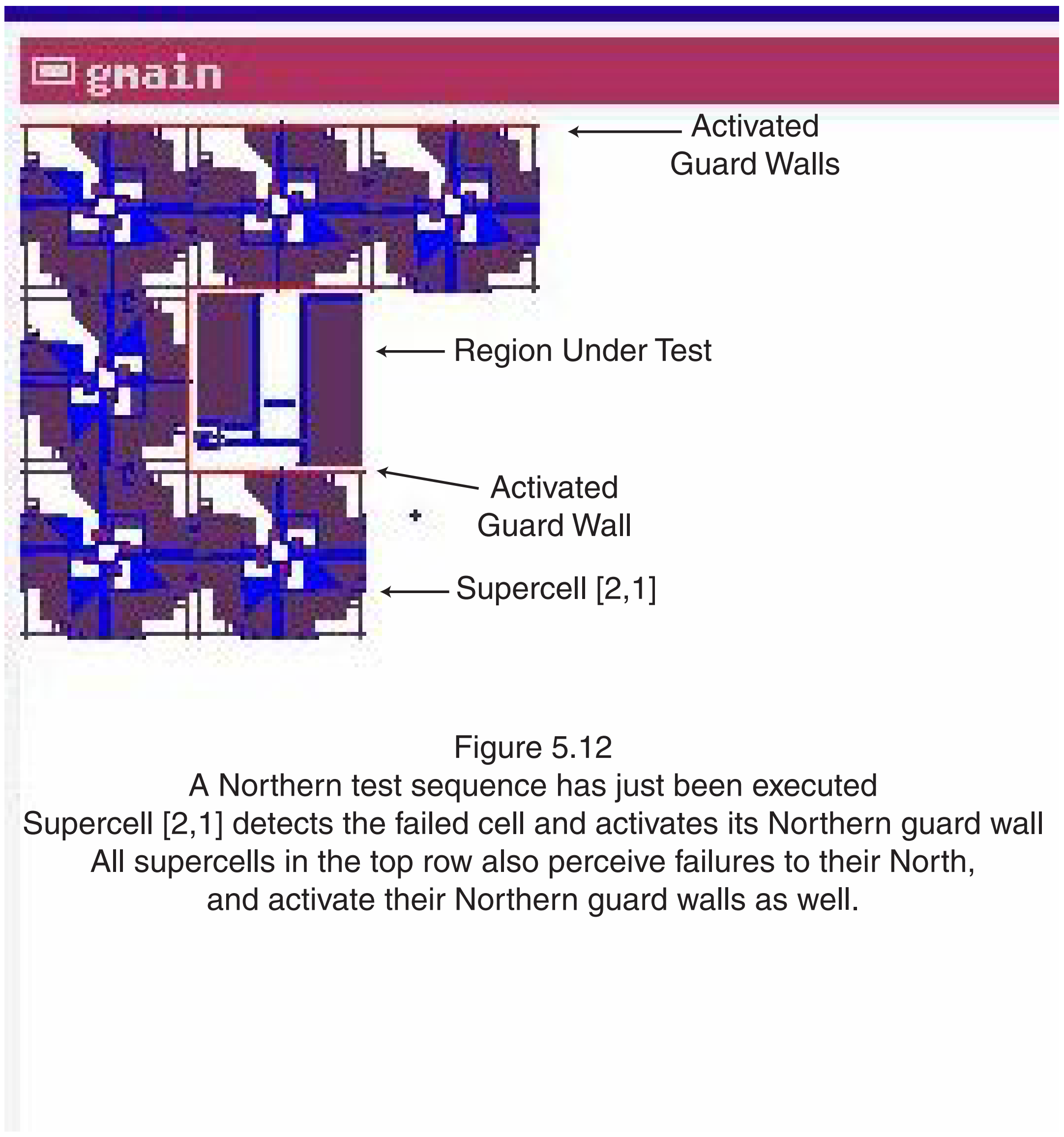


Figure 5.12
A Northern test sequence has just been executed
Supercell [2,1] detects the failed cell and activates its Northern guard wall
All supercells in the top row also perceive failures to their North,
and activate their Northern guard walls as well.

Figure 5.13 shows a Southern test sequence. Now, four regions are tested in parallel. Again, the region with the failed cell fails the test, and the testing supercell activates its Southern guard wall. This is followed by a Southern supercell configuration sequence. Figure 5.14 shows the results of that sequence, where three new supercells have been configured.

Finally, Figure 5.15 shows the result of a Western test sequence. The region containing the failed cell fails this test, and thus the testing supercell's Western guard wall is activated. Note that, at this point, the faulty cell has been completely ringed by guard walls. It is isolated from the properly-functioning supercells. Additionally, all supercells in the leftmost column have decided that there are faulty cells to their West (in fact, there are **no** cells to their West, and hence the test sequences fail). As a result, each supercells in the leftmost column has activated its Western guard wall, **except for the initial supercell**, whose Western guard wall logic is disabled by an external input. This is necessary because, if the Western guard wall were activated, it would be impossible to send further configuration commands into this supercell.

This sequence of tests proves a number of things:

- the ability of a collection of supercells to test regions of the Cell Matrix, to detect failed cells, and to isolate those failed cells;
- the ability to performed such tests in parallel;
- the ability to performed such tests non-invasively, that is, without interfering with Cell Matrix cells outside the region under test;
- the ability of the system to configure new supercells in parallel, and
- the ability of a set of supercells to immediately begin using newly configured supercells for subsequent tests and for configuration of new supercells.

5.2 Circuit Synthesis Tests

Having verified the operation of the fault testing sequences and the supercell configuration sequences, all future testing was performed by simply loading a tiled collection of supercells into the simulator. Tools were developed to control this placement, allowing gaps in the tiling to simulate failed regions.

After the tiled supercells were loaded into the simulator, the supercell circuit synthesis algorithm (Section 4.12 above) was run, causing the set of supercells to perform the ID arbitration and genome parsing steps. A post-simulation analysis tool was also developed, to allow generation of a human-readable summary of the system's current state.

The first test was the construction of a 1-bit adder. Figure 5.16 shows the circuit being assembled, along with the node assignments. The circuit contains 8 nodes, but they have been numbered with gaps between them, to reduce the number of nodes which are placed directly adjacent to other nodes. However, the external I/O points (nodes 1-3) are sequential, since we need all I/O points to be placed on the leftmost edge of the Cell Matrix (so that external circuitry may access them). The circuit adds A, B and Cin, and generates Cout and Sum.

The genome for this circuit is:

```
1 11 0 18  1 1
2 13 0 18  1 1
3  0 0  1  1 1
5  1 2  2 17 1
7  1 2  2  7 1
9  3 5  2  7 1
11 3 5  2 17 1
13 7 9  2 17 1
```

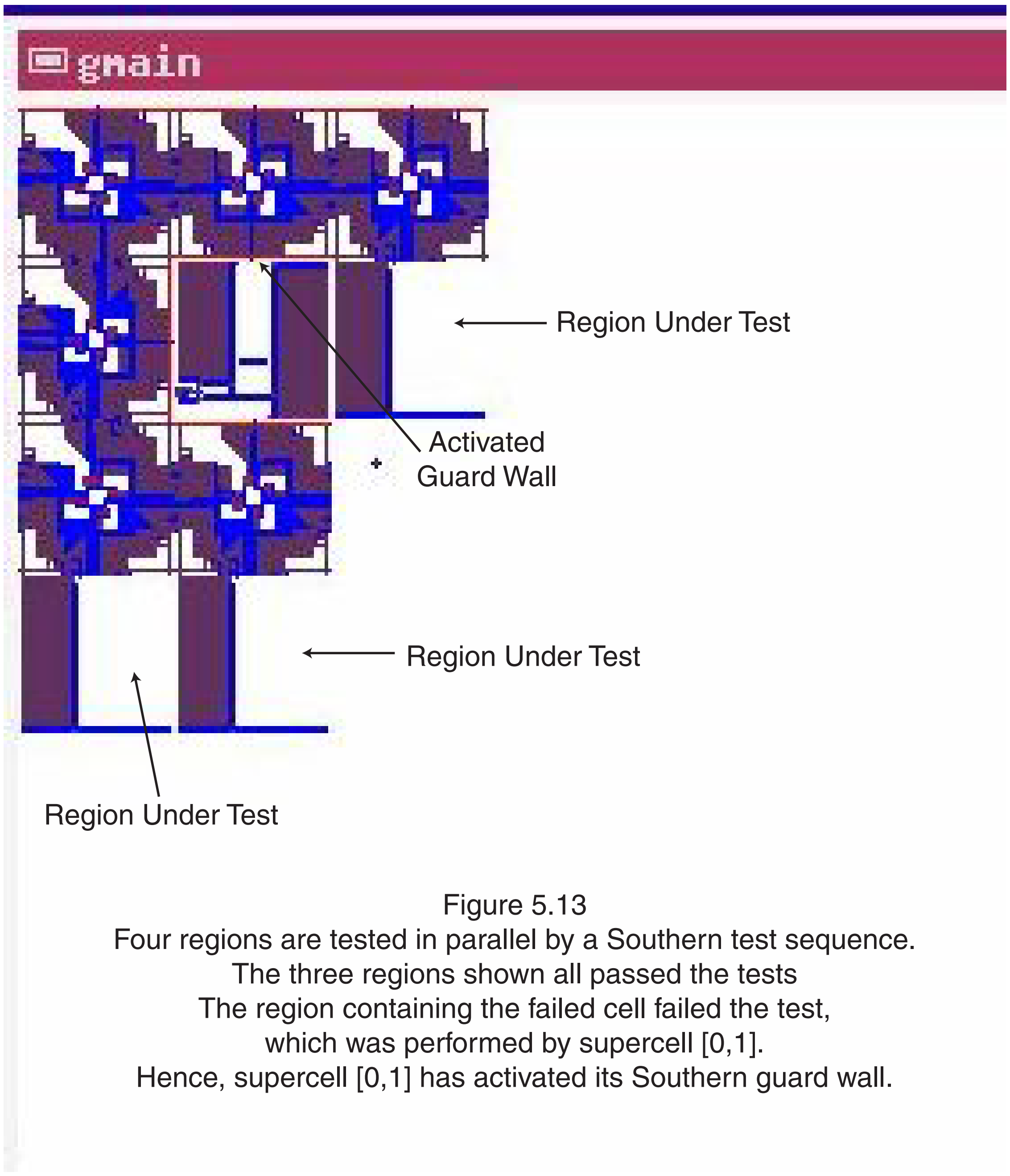


Figure 5.13
Four regions are tested in parallel by a Southern test sequence.
The three regions shown all passed the tests
The region containing the failed cell failed the test,
which was performed by supercell [0,1].
Hence, supercell [0,1] has activated its Southern guard wall.

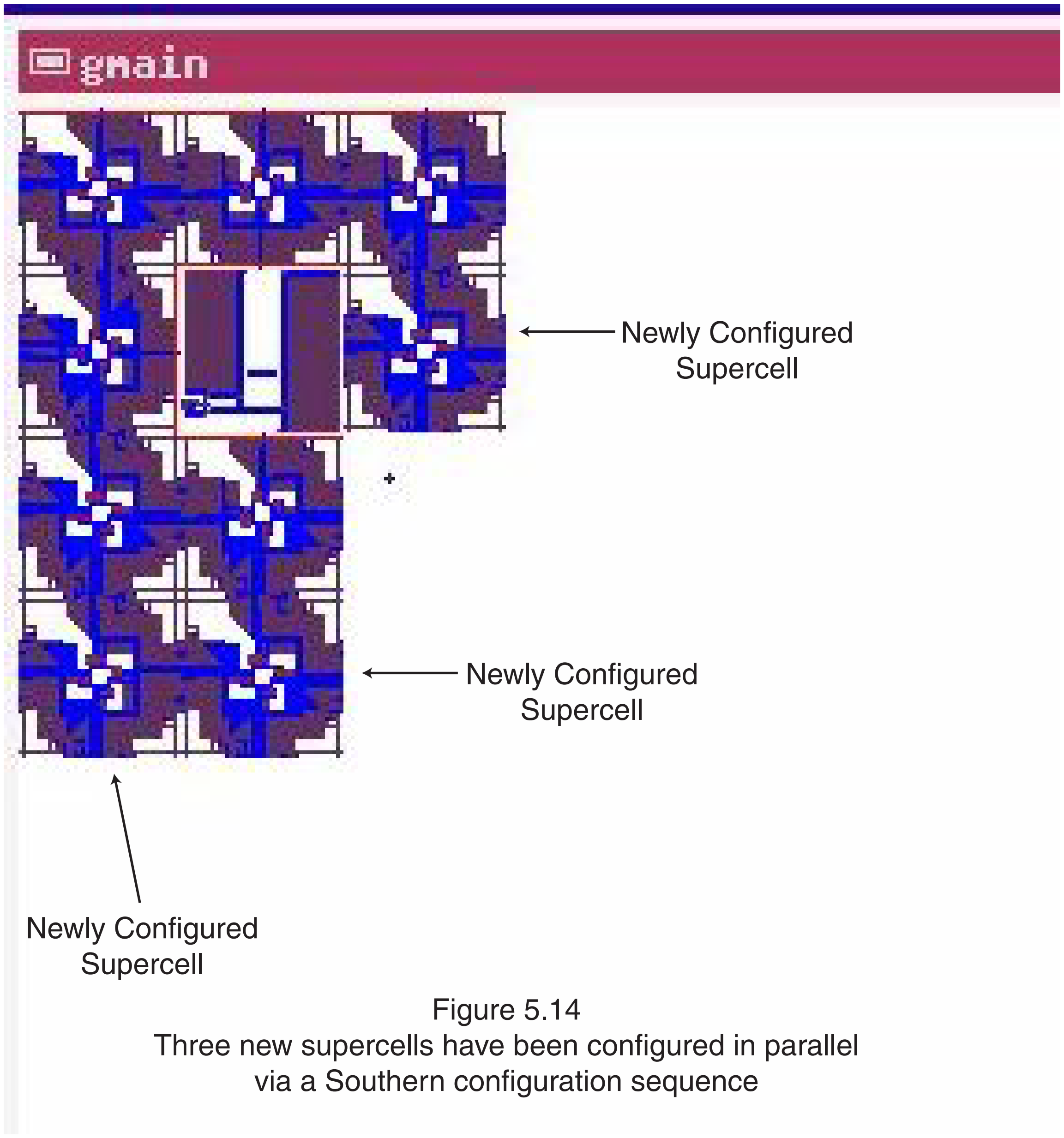

gmain
Newly Configured Supercell
Newly Configured Supercell
Newly Configured Supercell

Figure 5.14
Three new supercells have been configured in parallel via a Southern configuration sequence

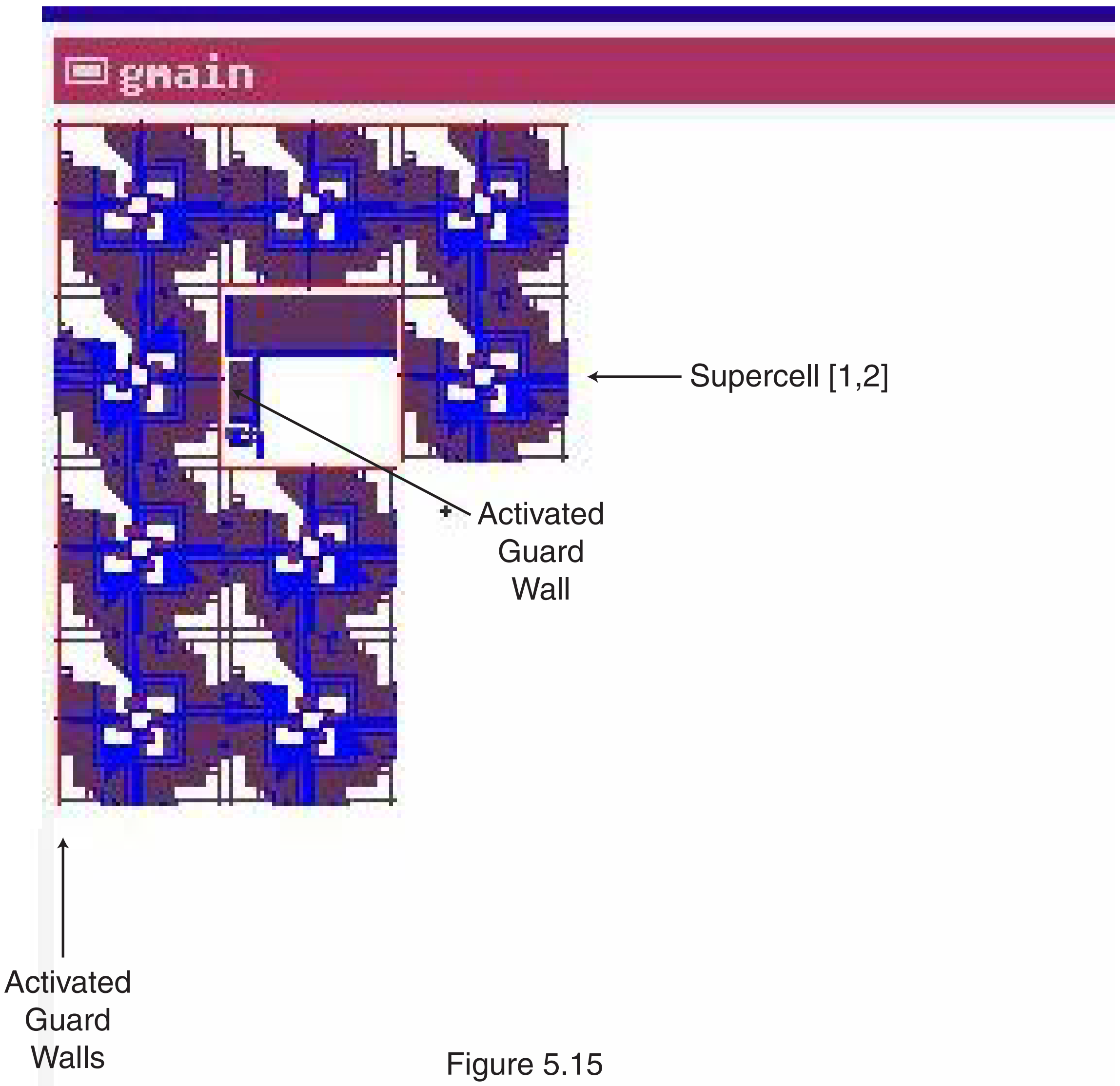


Figure 5.15
Results of Western test sequence
Supercell [1,2] detects the failed cell to its West, and thus activates its Western guard wall. Additionally, all supercells in the leftmost column have no cells to their West, and therefore their test sequences fail. Thus all Western guard walls in that column are activated, except for the initial supercell which was configured without a Western guard wall.
The faulty cell is now completely surrounded by a guard ring.

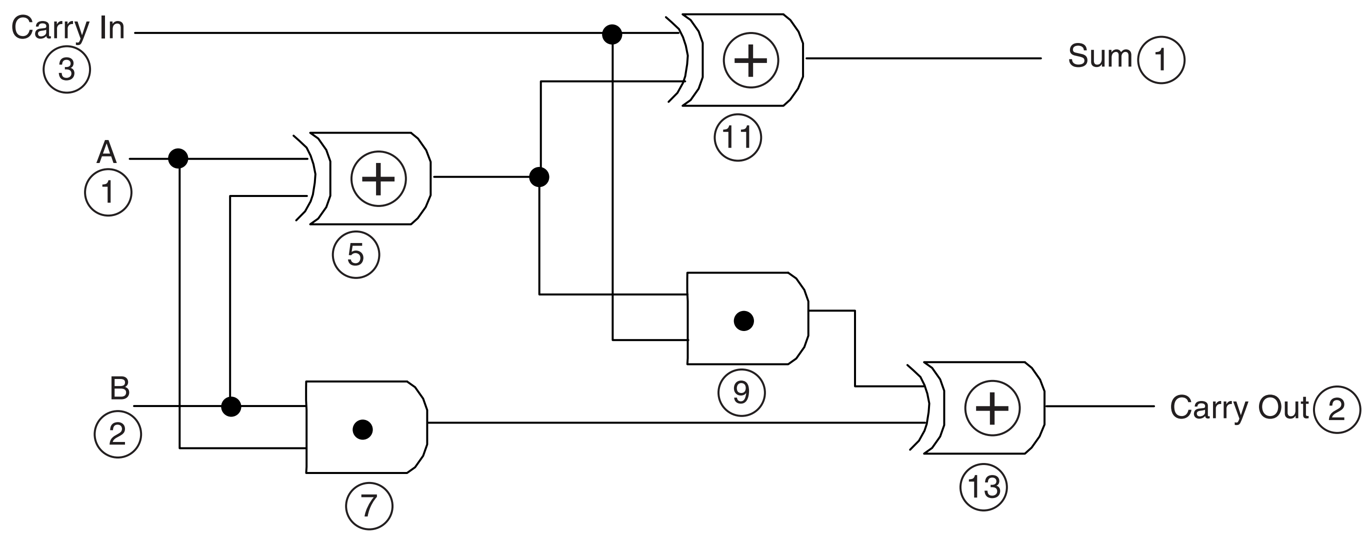


Figure 5.16
Simple 1-Bit Adder
Node numbers are shown inside circles.
Inputs and outputs are given low node numbers, so they will be placed along the left of the Cell Matrix.

The simulator was run through the parsing and processing of the first genome block, i.e., until the first DST supercell's inputs were connected to their SRC supercells, and DST's functional block was built. Each square in Figure 5.17 represents one supercell. The DID of each supercell is shown in the upper left corner. Failed supercells are grayed out. This figure corresponds to the state after parsing the first block of the genome (1 11 0 18 1 1).

Figure 5.17 shows the following:

- The supercell in the upper left corner has been assigned DID 1.
- That supercell's functional block has been configured with the following equations:
  - f_0: DE=W DW=S This transfers input 1 to the external I/O output, and transfers the external I/O input to the West.
  - f_1: DE=W This transfers the input from the East (=the external input) to the West
  - f_2: DE=W Thus transfers the input from the East (=the external input) to the West.

  Hence, this functional block performs two actions: it copies its first input to the external output, and it copies the external input to its own functional block output. This functional block corresponds to the "18 1 1" piece of the genome.
- A path has been built from node 11 to node 1. The path beings in node 11, whose functional block's output will drive the path. The node terminates after node 2, where it enters node 1 from the South. This corresponds to the "1 11" piece of the genome block.

Figure 5.18 shows an effectively equivalent circuit.

Figure 5.19 shows the state after the next block of the genome is parsed. Node 2 has been configured, and a path has been built from node 13 to node 2. Note that this path crosses the previous path in node 6. The concentric circles in node 6 correspond roughly to the concentric paths inside a supercell's steering block.

The third block of the genome simply specifies the behavior of node 3, but no path synthesis is performed. This is because node 3 is purely an external input node. Figure 5.20 shows the state after block 3 is processed.

Figure 5.21 shows the result of parsing the fourth genome block. Here, node 5 is being fed from nodes 1 and 2. From node 1, the path moves East, then South. From node 2, the path moves South then East.

Figure 5.22 shows the result of parsing the fifth genome block. Here, node 7 is also to be driven from nodes 1 and 2. Since node 1 is a corner node, is has only two sides with which to work. Since node 5 already built paths which use both these sides, node 1 can not connect to node 7 on channel A. It thus switches to channel B. The pathways here are 1→2→3→6→7 and 2→5→6→7. These can be discerned by following the paths around the circles.

Figures 5.23-5.25 show the remained of the circuit synthesis. Figure 5.25 illustrates the final implementation of the target circuit.

A test vector command was also added to the simulator. This command allows the external I/O inputs to be set in a succession of supercells along the left edge of the matrix. The command then displays the external I/O outputs from the leftmost supercells. The following segment from the simulator output shows the test results for this circuit:

```
>> v 000<00000000>
>> v 001<10000000>
>> v 010<10000000>
>> v 011<01000000>
>> v 100<10000000>
>> v 101<01000000>
>> v 110<01000000>
>> v 111<11000000>
>> <<<EOF>>>
```

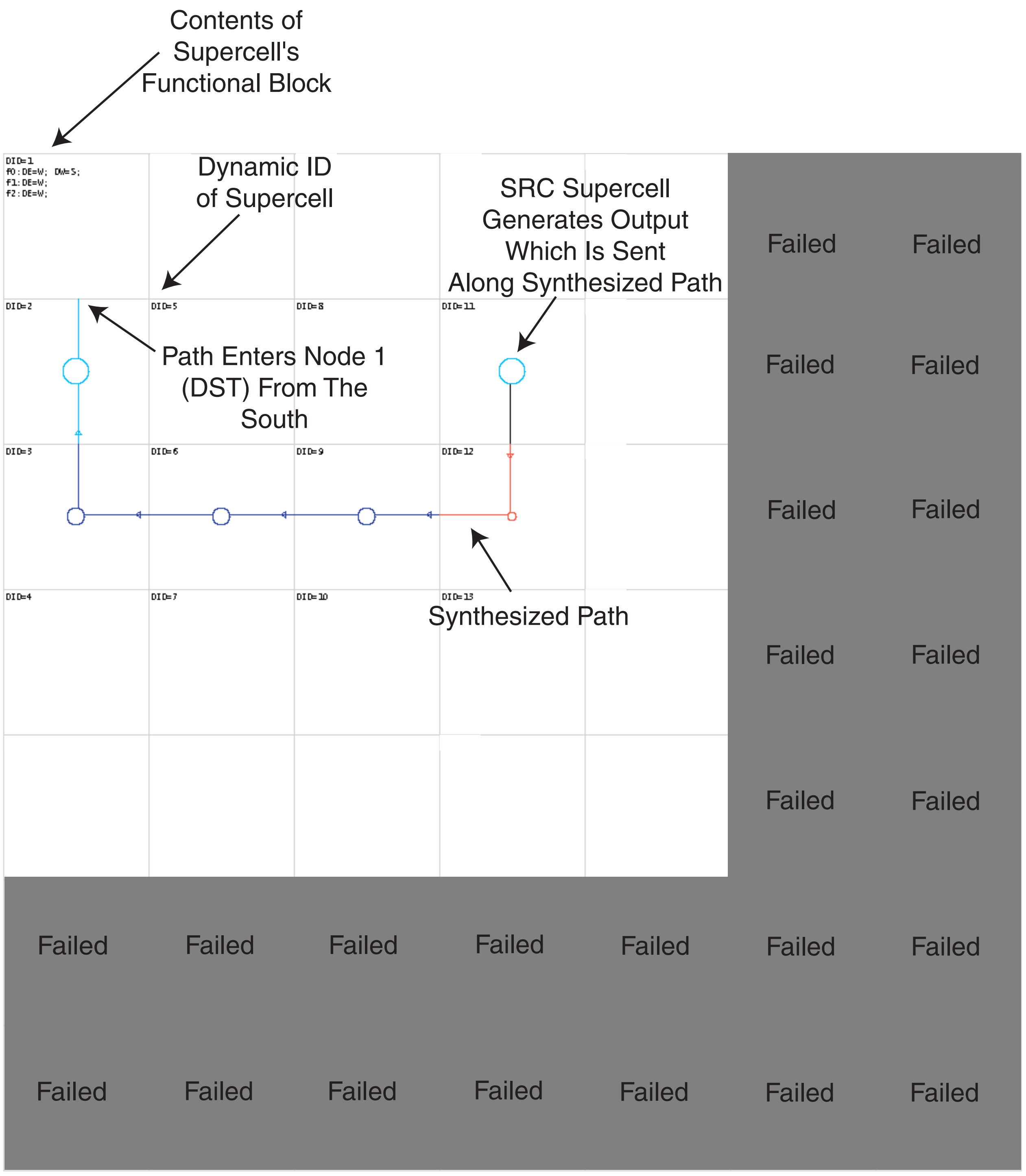

Contents of Supercell's Functional Block
Dynamic ID of Supercell
SRC Supercell Generates Output Which Is Sent Along Synthesized Path
Path Enters Node 1 (DST) From The South
Synthesized Path
DID=1
f0:DE=W; DW=S;
f1:DE=W;
f2:DE=W;
DID=2
DID=5
DID=8
DID=11
DID=3
DID=6
DID=9
DID=12
DID=4
DID=7
DID=10
DID=13
Failed

Figure 5.17
Status After Parsing First Block of Genome
Each block represents a single supercell. SRC supercell is node 11, DST supercell is node 1. Path from SRC travels through nodes 12, 9, 6, 3 and 2. Circles in the middle of these supercells roughly correspond to the steering block. Edge Sensing is ENABLED, so most edge cells are not assigned a DID. Nodes 1, 2 and 3 have their EDGE SENSE inputs wired to 1, allowing them to be assigned DIDs.

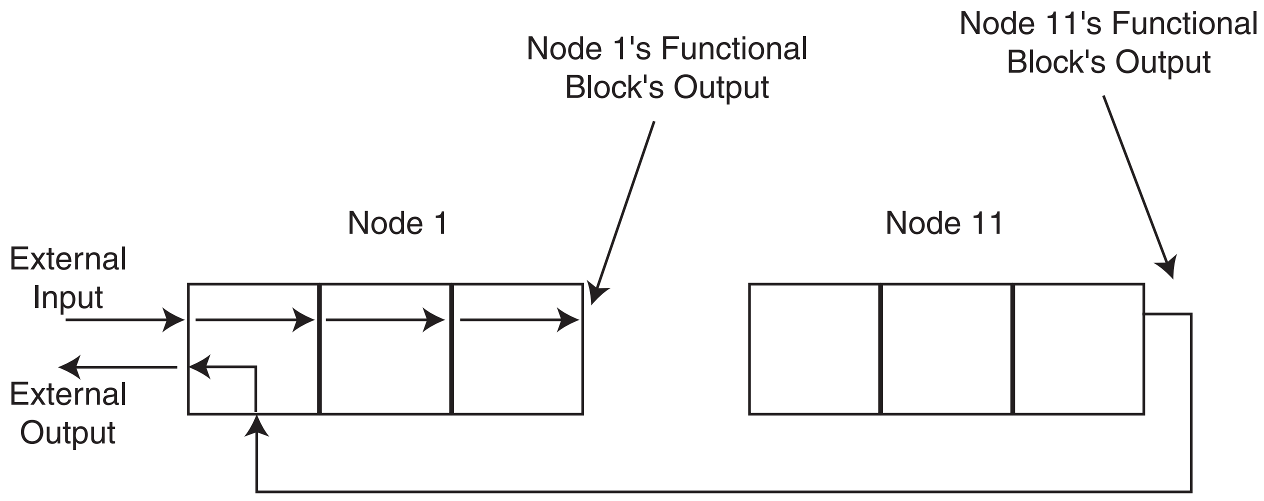

Node 1's Functional Block's Output
Node 11's Functional Block's Output
Node 1
Node 11
External Input
External Output

Figure 5.18
Equivalent Block Diagram for Figure 5.17
Node 1's functional block has been specified to pass its external input to its functional output (which is not yet connected to any inputs)
Node 1 also passes its first input to its external output.
Node 1's first input is connected to node 11's functional block's output.
Therefore, node 11's output can be read from supercell 1's external output port.
Node 11's functional block has not yet been specified.

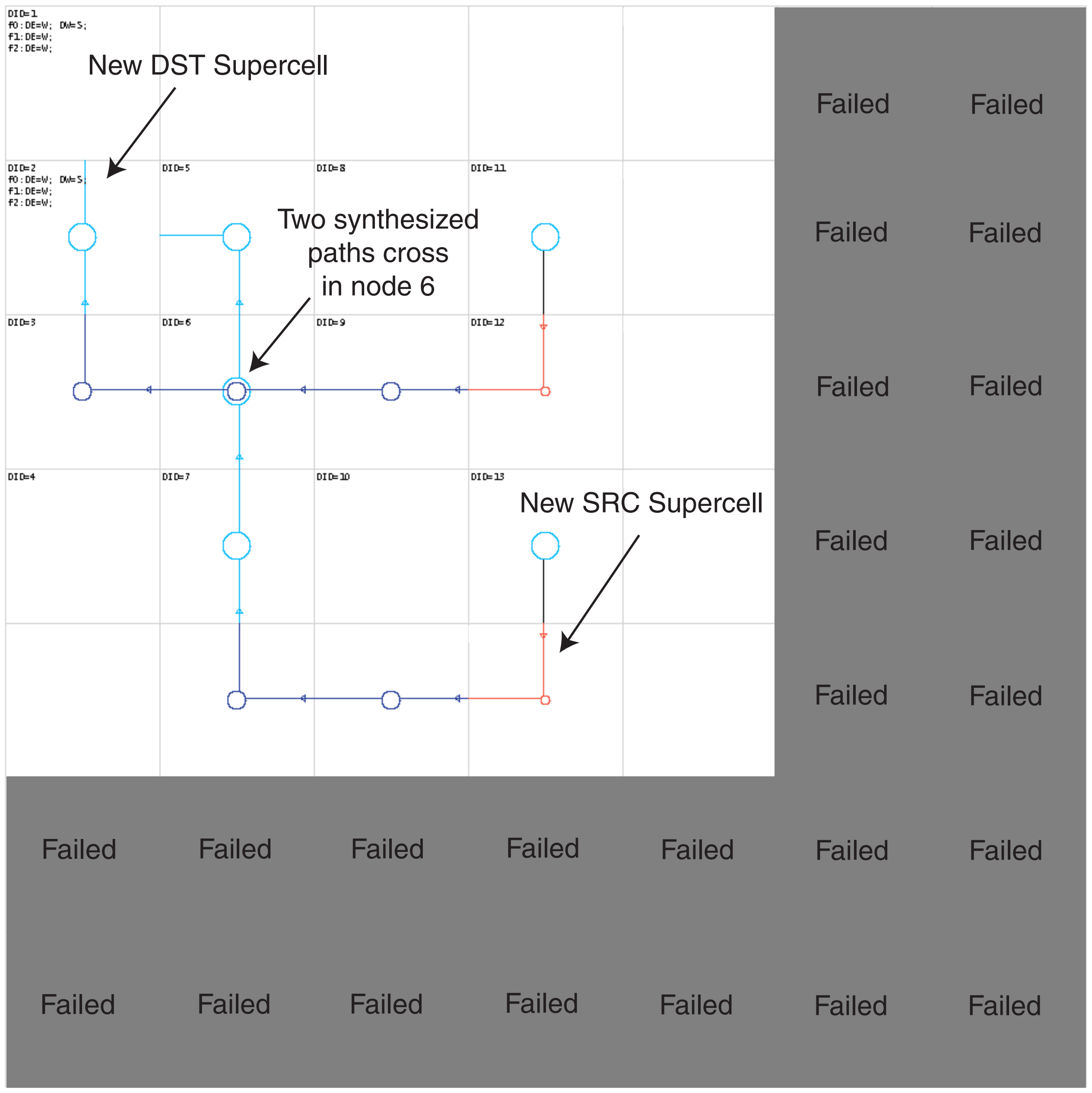


Figure 5.19
Second Block of Genome Has Been processed
New path has been synthesized from node 13 to node 2
Two paths cross in node 6, but travel around separate paths
in the steering block, and therefore don't interfere with each other.

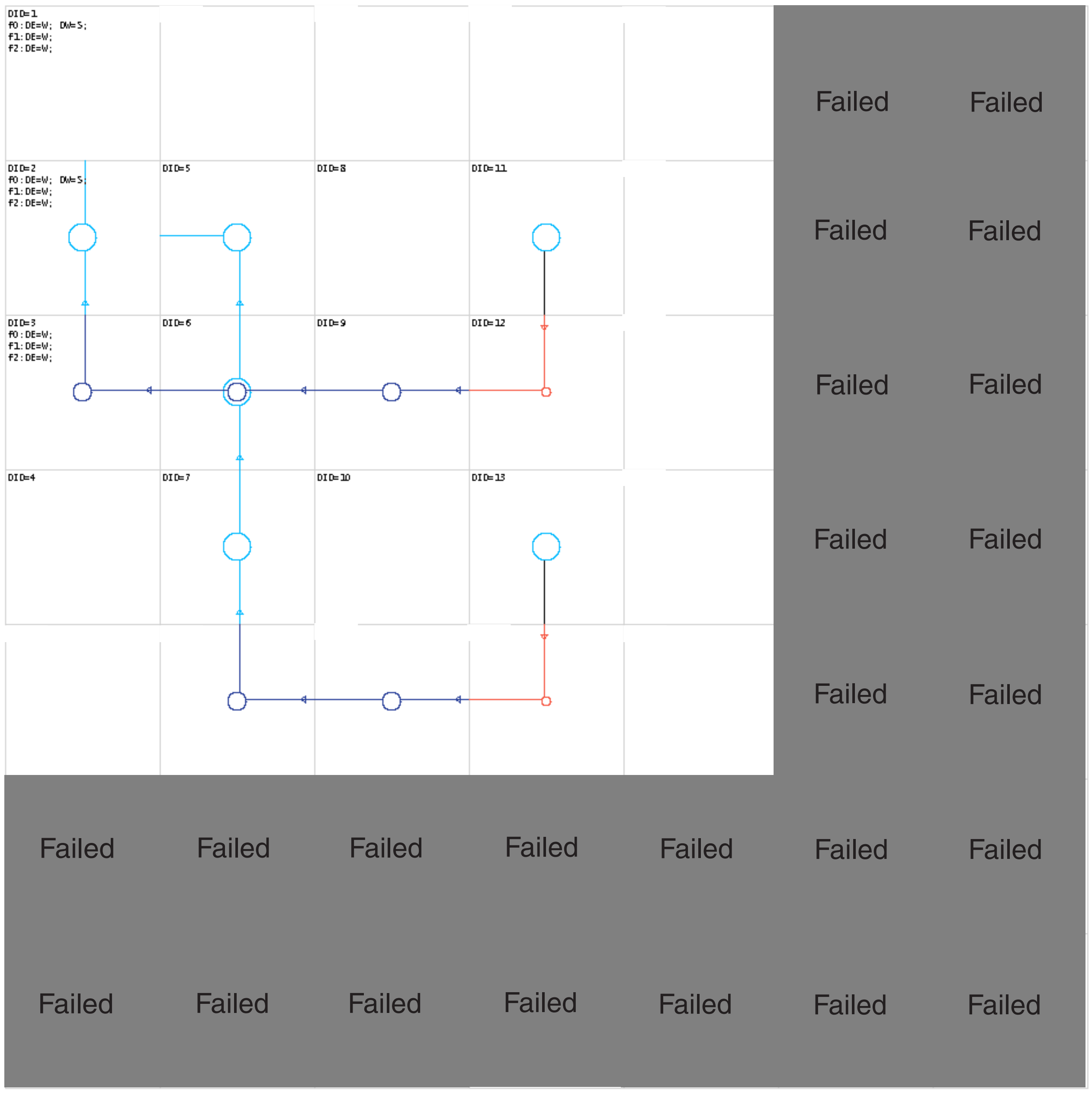


Figure 5.20
Configuration After Third Block of Genome Has Been Parsed

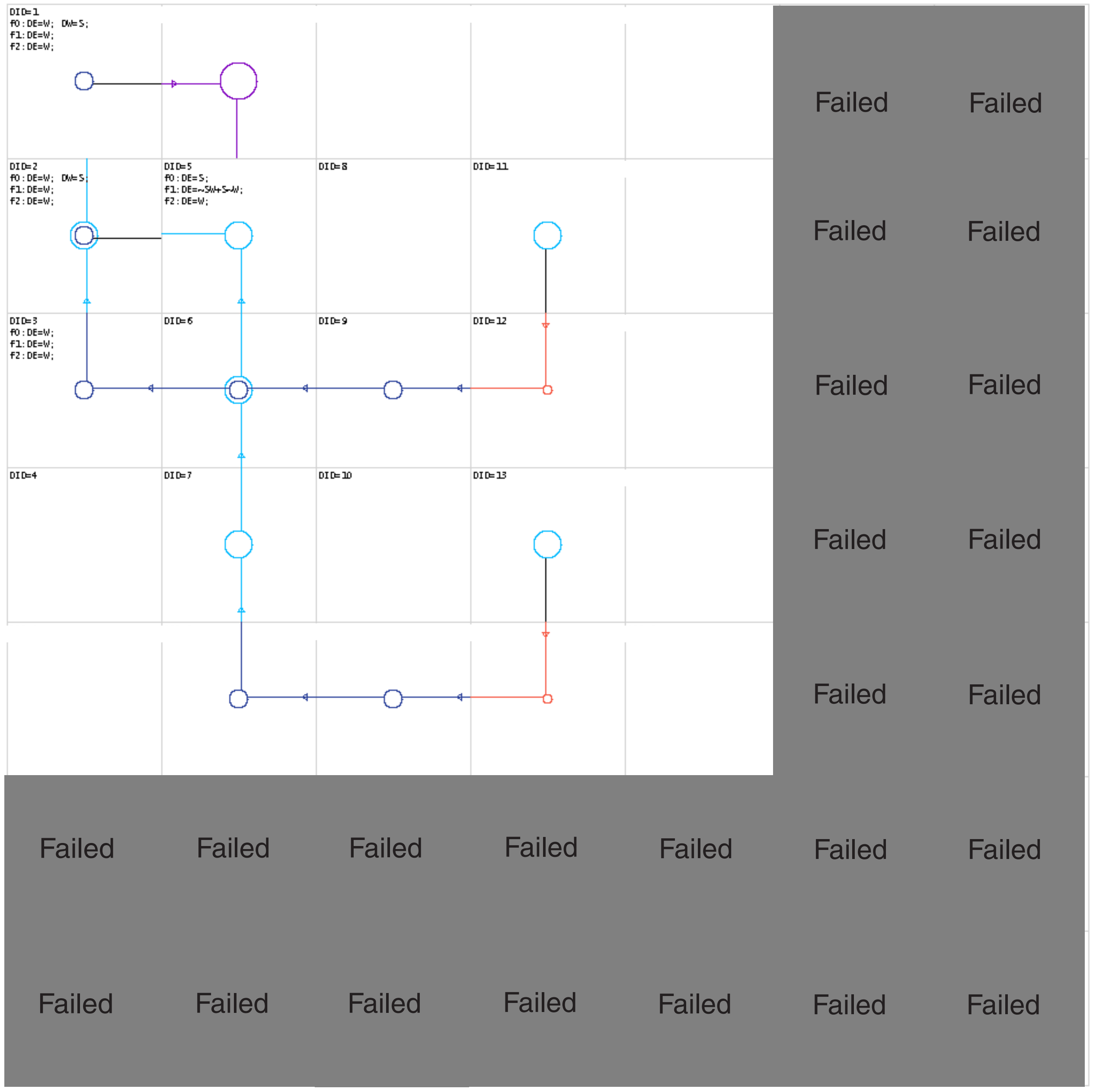


Figure 5.21
Configuration After Fourth Block of Genome Has Been Parsed

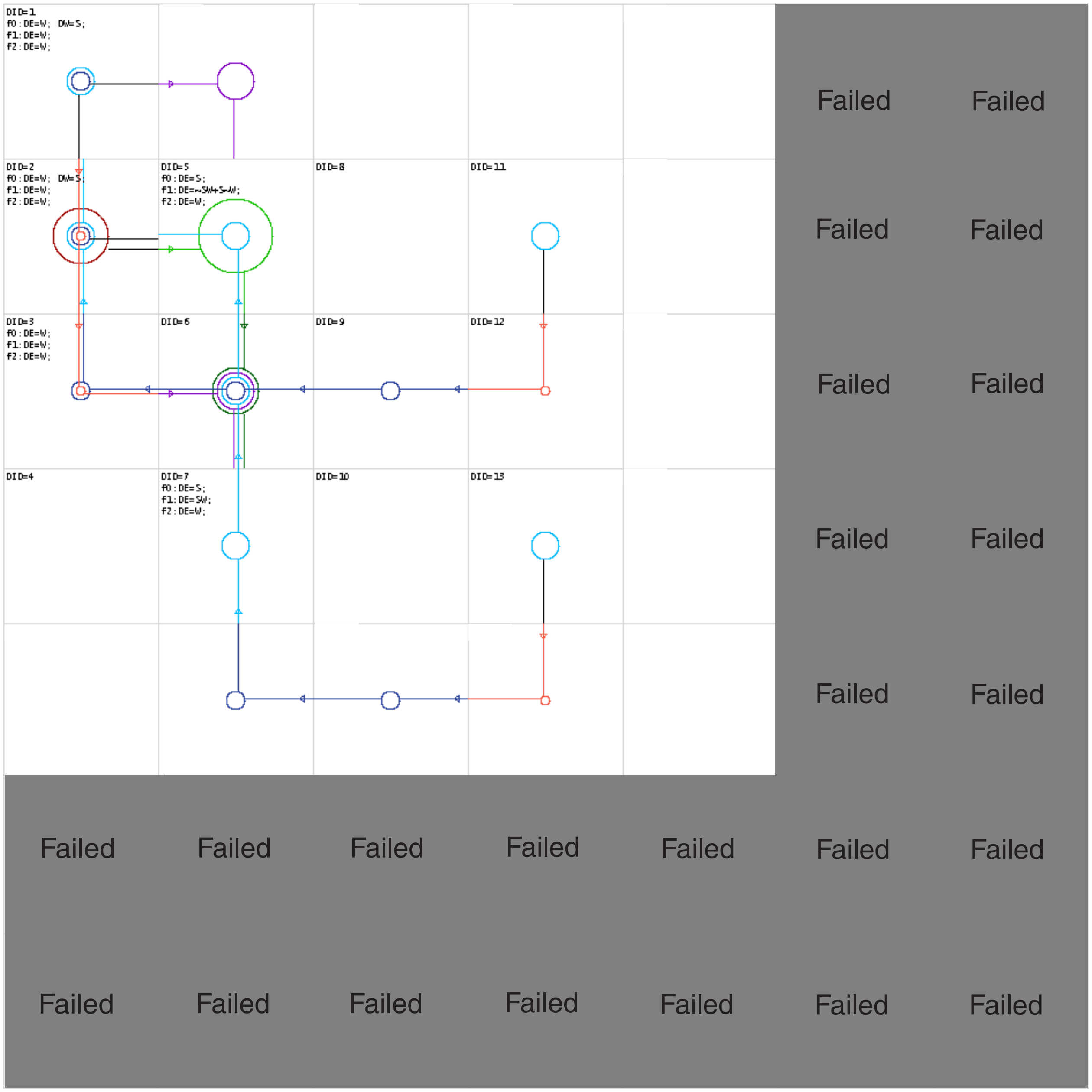


Figure 5.22
Configuration After Fifth Block of Genome Has Been Parsed
Node 7 (most recently configured node) receives inputs from nodes
1 and 2.

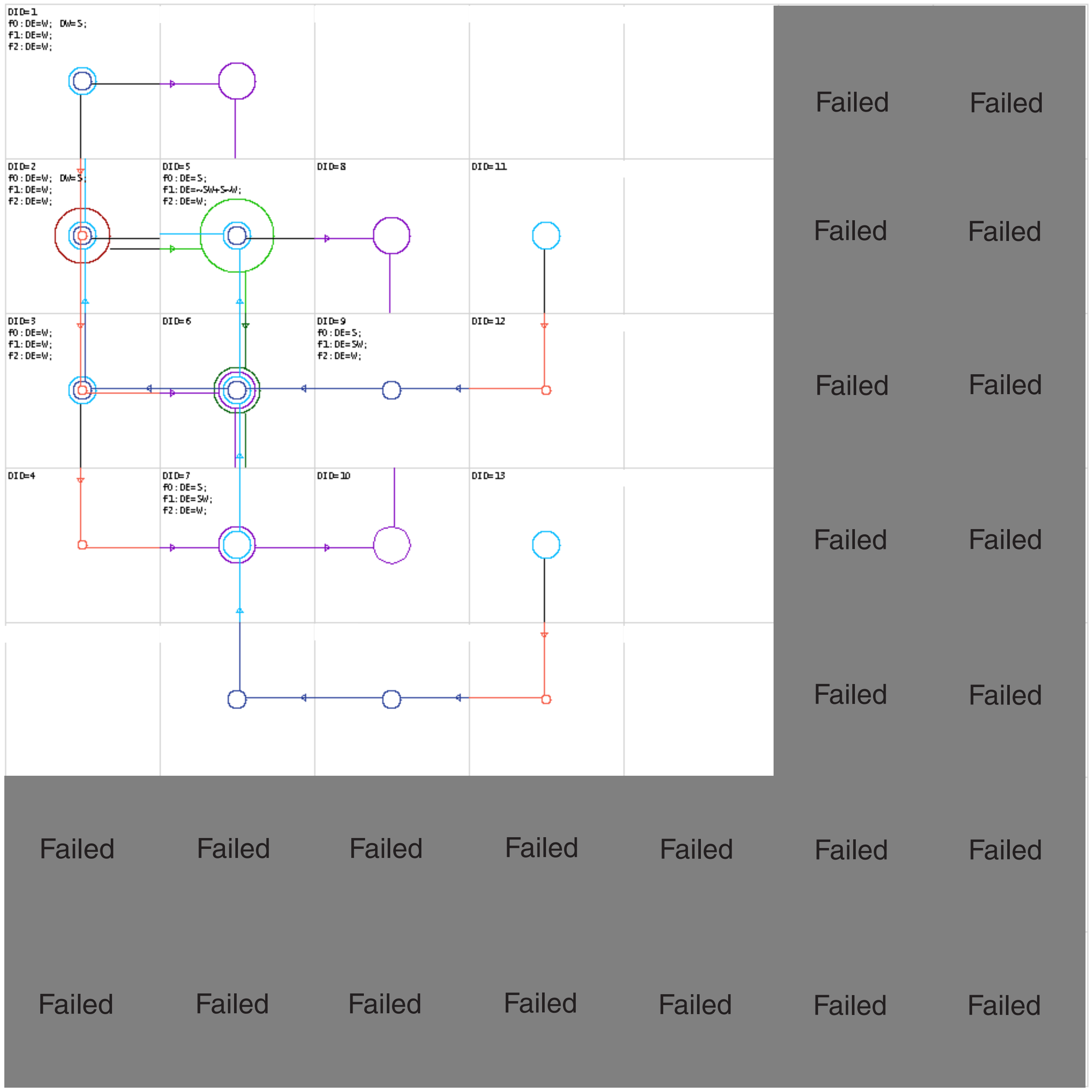


Figure 5.23
Configuration After Sixth Block of Genome Has Been Parsed

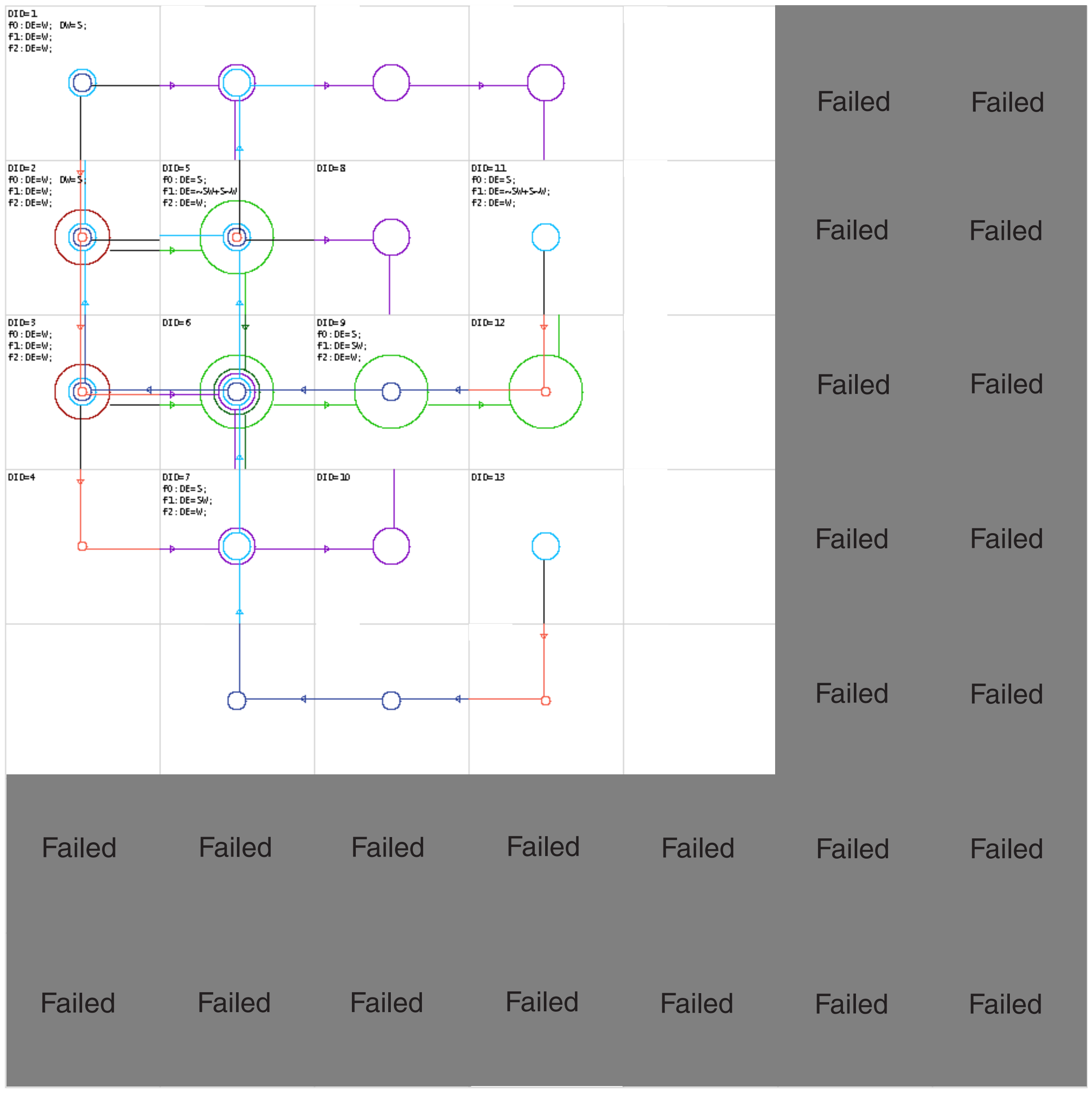


Figure 5.24
Configuration After Seventh Block of Genome Has Been Parsed

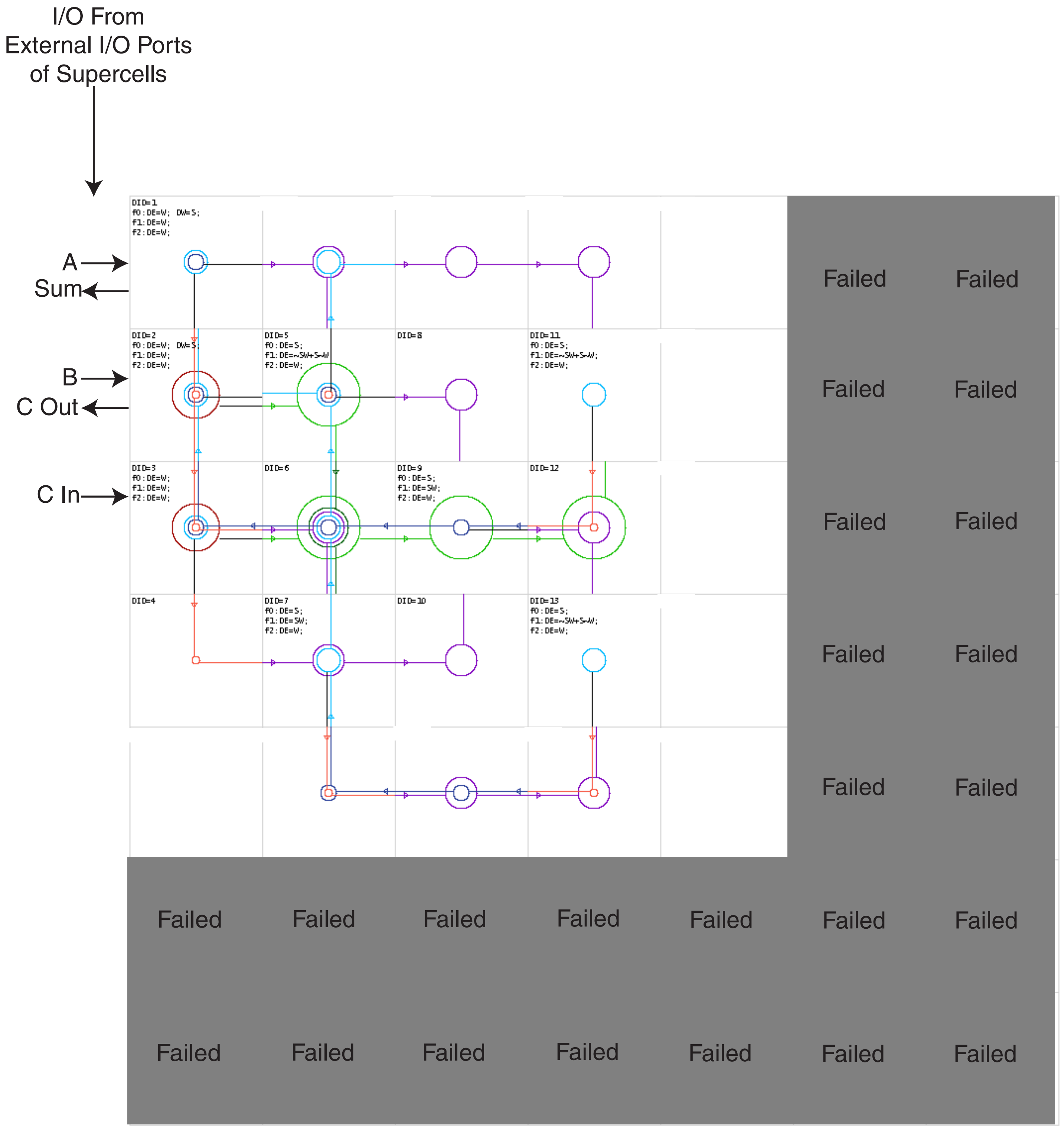


Figure 5.25
Configuration After Eigth and Final Block of Genome Has Been Parsed
A, B and Carry In are sent into the External Input on supercells
[0,0], [1,0] and [2,0] respectively.
Sum can be read from supercell [0,0]'s External Output.
Carry Out can be read from supercell [1,0]'s External Output.

The three binary digits following the “v” command are the inputs sent to the supercells along the left of the matrix, with the first digit corresponding to the upper left supercell. The output values shown in the <angle brackets> are ordered similarly. Thus, for example, the “v 110” command sends a 1 to supercells 0 and 1 (counting from the top), and a 0 to supercell 0. This corresponds to setting A=1, B=1 and Cin=0. The output vector <01000000> corresponds to Sum=0 and Cout=1. As can be seen, the adder functions correctly, producing the correct Sum and Cout for all combinations of A, B and Cin.

Figure 5.26 shows another implementation of the same circuit, **from the same genome**. In this case, five regions were assumed to contain faulty Cell Matrix cells. Note that every row and every column contains a faulty supercell. Despite this scattering of faults, the supercell collection is still able to implement the desired target circuit.

Remember, for these tests, the supercells were directly loaded into the simulator. However, had they been configured using fault testing sequences and supercell configuration sequences, **the exact same sequences** would have resulted in the layouts of either Figure 5.25 or Figure 5.26, depending on where the faults were located.

Figure 5.27 shows another test circuit, a simple three bit ripple counter, implemented with three toggle flip flops. The genome for this circuit is:

```
    1 0 0  1  1  1
    2 5 0  3  0  0
    3 6 0  3  0  0
    4 7 0  3  0  0
    5 0 1 14 15 16
    6 0 5 14 15 16
    7 0 6 14 15 16
```

A single clock input is sent to node 1. The 3-bit output is read from nodes 2, 3 and 4. The simulator output for the final implemented circuit is:

```
>> v 0<00000000>
>> v 1<00000000>
>> v 0<01000000>
>> v 1<01000000>
>> v 0<00100000>
>> v 1<00100000>
>> v 0<01100000>
>> v 1<01100000>
>> v 0<00010000>
>> v 1<00010000>
>> v 0<01010000>
>> v 1<01010000>
>> v 0<00110000>
>> v 1<00110000>
>> v 0<01110000>
>> v 1<01110000>
>> <<<EOF>>>
```

As can be seen, when the clock input drops from 1→0, the 3-bit output counts up one, finally rolling over from 7 (111) to 0 (000). Note that the MSB is on the right.

Figure 5.28 shows the final implementation of this circuit. Note that while the circuit contains 7 nodes, it was successfully implemented in a 4x3 set of supercells, containing only 8 functioning supercells. In fact, the additional supercell (DID=8) is not used in the final configuration.

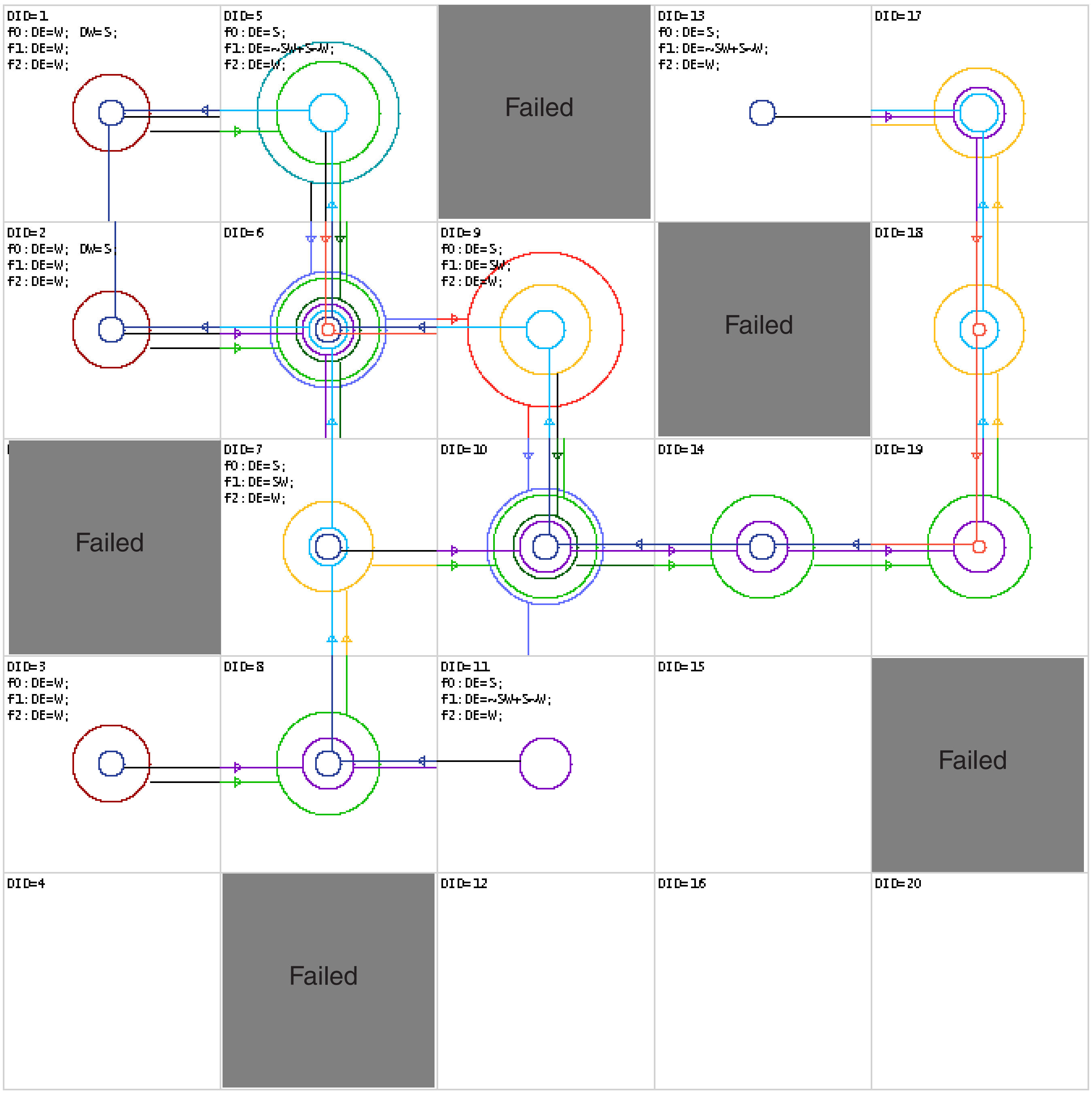


Figure 5.26
Same Circuit as Figure 5.16 (Same Genome)
Edge Sense has been disabled, but failed cells have been introduced.
The supercells still successfully organize into the target circuit.

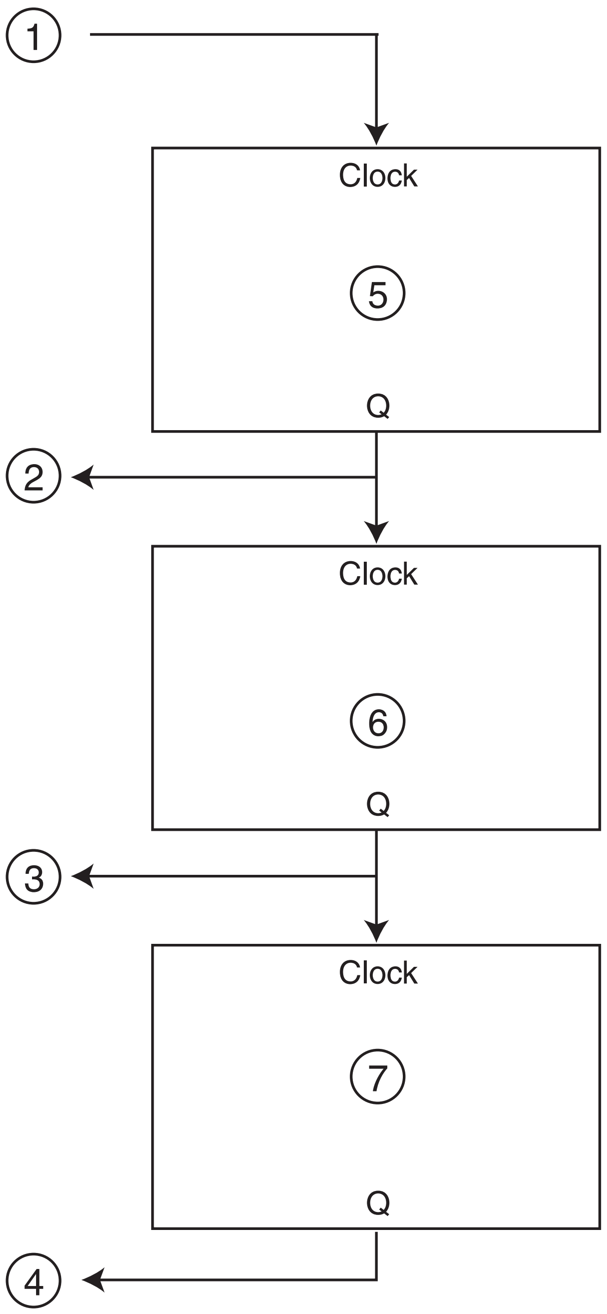


Figure 5.27
3-Bit Counter
Blocks are toggle flip flops
Node numbers are shown inside circles

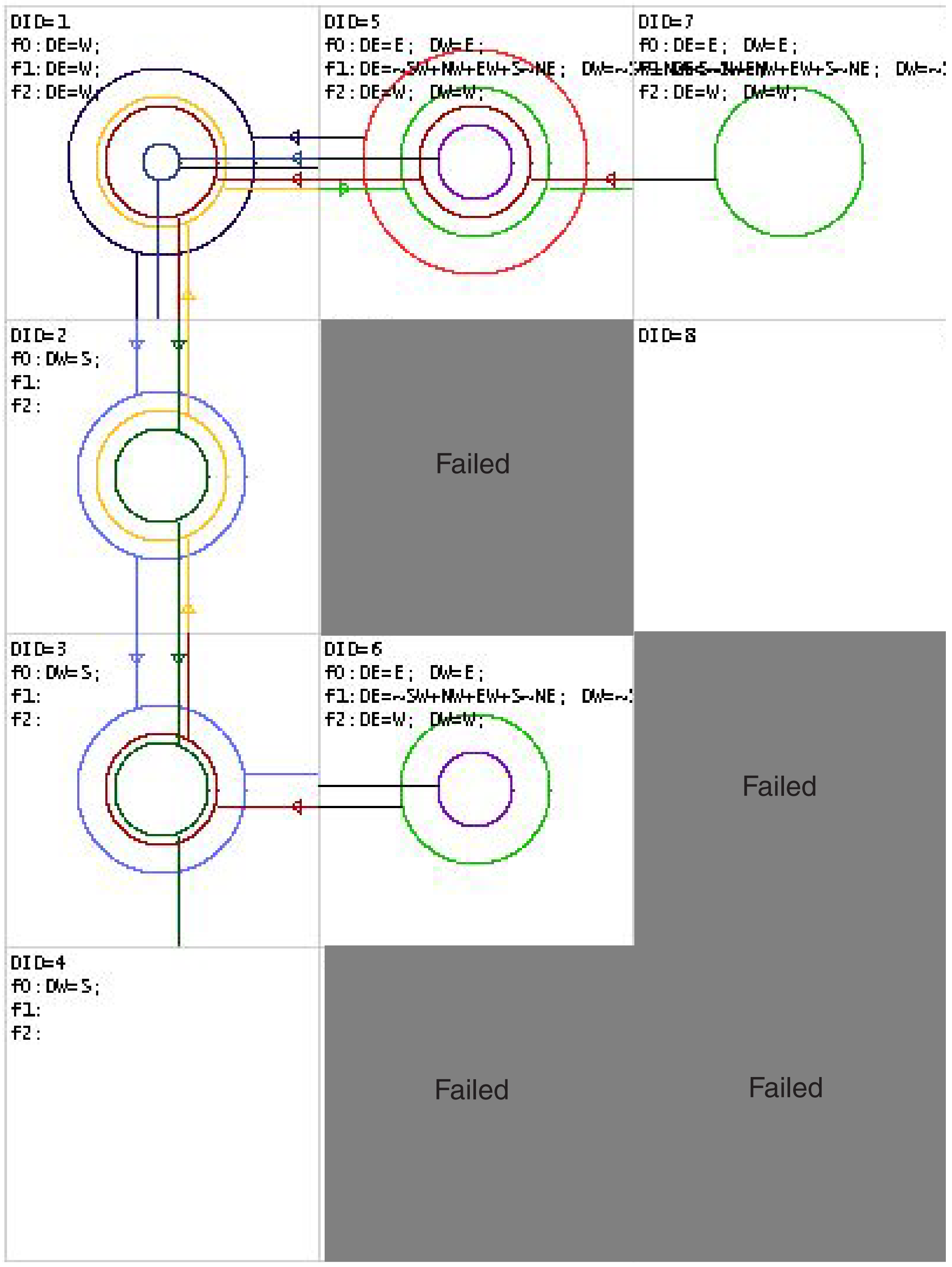


Figure 5.28
Supercell Implementation of Three Bit Adder
Clock is input to external I/O port on node 1. Outputs come from nodes 2 (LSB), 3, and 4 (MSB). The f1 piece of the functional block for the flip flops (nodes 5, 6 and 7) has the Boolean expressions:

$$DE=((N+\overline{S})W) + (SE\overline{N})$$
$$DW=((N+\overline{S})W) + (S\overline{E}\overline{N})$$

The remaining tests were performed on the circuit of Figure 5.29, a simple 3-bit linear feedback shift register (LFSR) with taps at bits 0 and 2. The shift register is constructed explicitly from pairs of D flip flops, clocked alternately on a pair of non-overlapping clocks. The genome for this circuit is:

```
1   7  0 18  1 1
2  11  0 18  1 1
3  15  0  3  0 0
5  17  1  2 19 9
7   5  2  2 19 9
9   7  1  2 19 9
11  9  2  2 19 9
13 11  1  2 19 9
15 13  2  2 19 9
17  7 15  2 20 1
```

The LFSR's output pattern is: 000→100→010→101→110→011→001→000. The LFSR is clocked by toggling CLK 1 (0→1→0) and then CLK 2 (0→1→0).

Figure 5.30 shows a supercell implementation of this circuit on a perfect supercell tiling (no faults). The routing congestion through node 8 is indicative of the clustering of assigned DID nodes near node 8.

The output test vectors for the simulation of this circuit were as follows (comments have been added for clarity):

```
>>> v 00<00000000>      ; Initial output is 000
>>> v 10<00000000>
>>> v 00<00000000>
>>> v 01<10000000>      ; CLK 2 raises; LFSR output 100
>>> v 00<10000000>
>>> v 00<10000000>
>>> v 10<10000000>
>>> v 00<10000000>
>>> v 01<01000000>      ; Output=010
>>> v 00<01000000>
>>> v 00<01000000>
>>> v 10<01000000>
>>> v 00<01000000>
>>> v 01<10100000>      ; Output=101
>>> v 00<10100000>
>>> v 00<10100000>
>>> v 10<10100000>
>>> v 00<10100000>
>>> v 01<11000000>      ; Output=110
>>> v 00<11000000>
>>> v 00<11000000>
>>> v 10<11000000>
>>> v 00<11000000>
>>> v 01<01100000>      ; Output=011
>>> v 00<01100000>
>>> v 00<01100000>
>>> v 10<01100000>
>>> v 00<01100000>
>>> v 01<00100000>      ; Output=001
>>> v 00<00100000>
>>> v 00<00100000>
>>> v 10<00100000>
>>> v 00<00100000>
>>> v 01<00000000>      ; Output=000
```

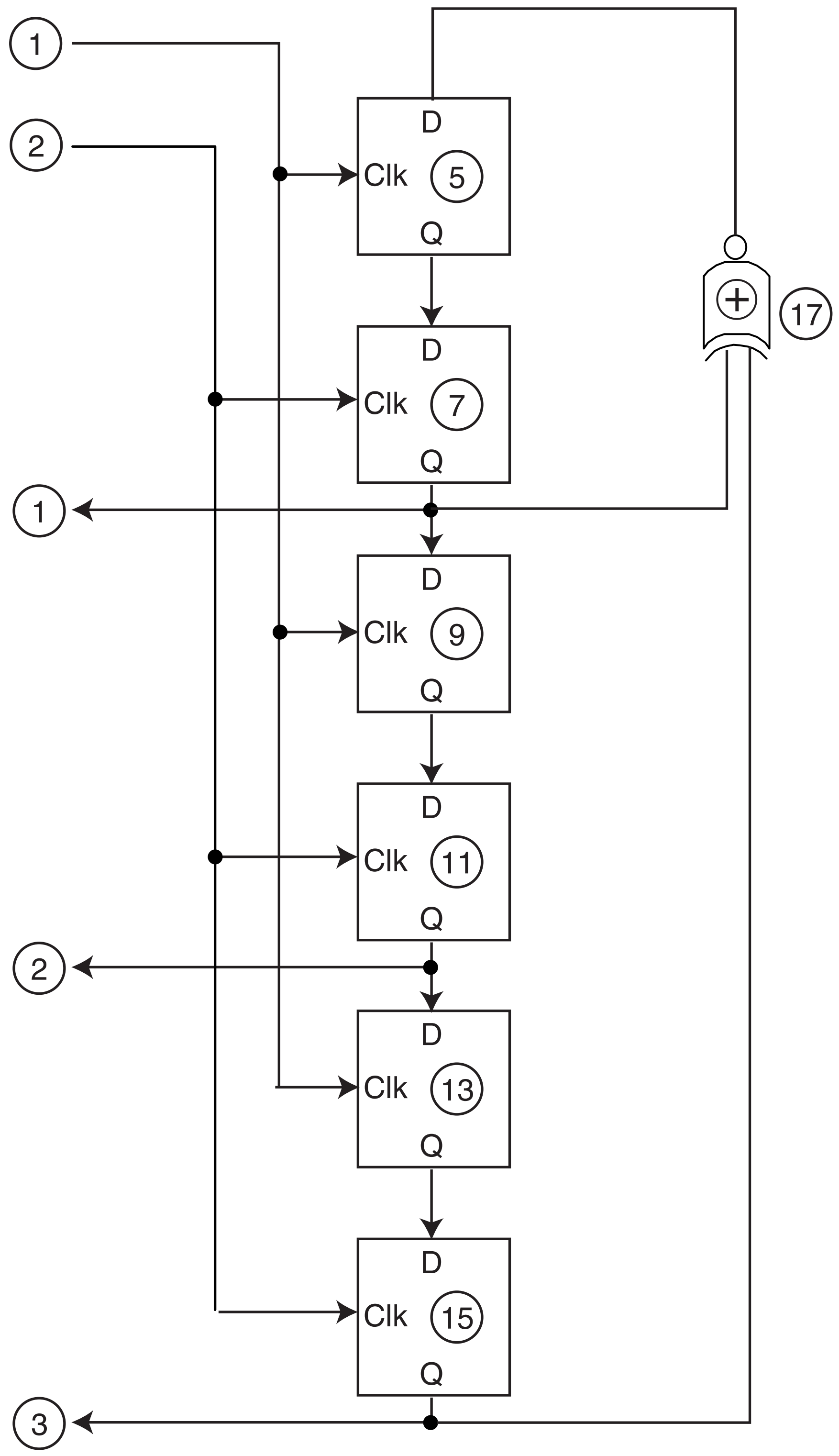


Figure 5.29
Three Bit LFSR
Non-overlapping clocks are sent into nodes 1 and 2
Three bit output is available from nodes 1, 2 and 3
Register shifts on rising edge of input 2
Nodes 5, 7, 9, 11, 13 and 15 are D flip flops

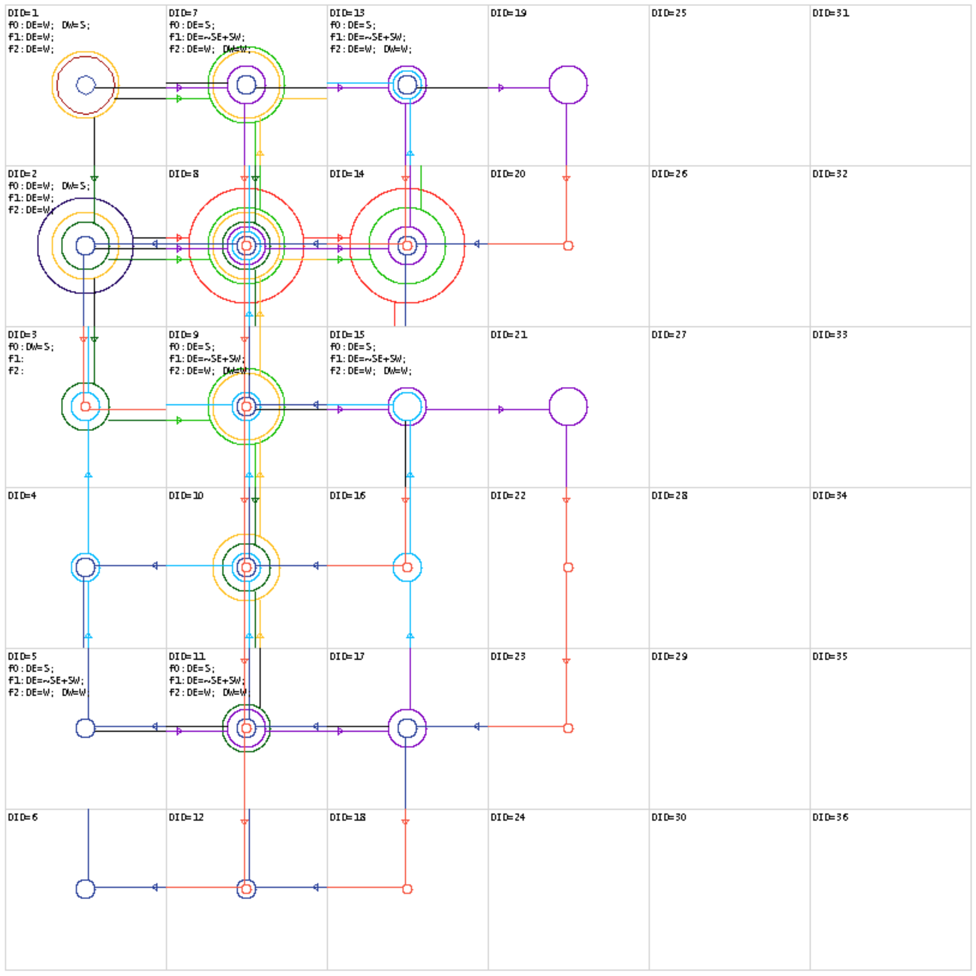


Figure 5.30
Supercell Implementation of Three Bit LFSR
This layout was performed with supercells with the EDGE SENSE feature disabled
No faults were present in the underlying hardware.

```
>>> v 00<00000000>
>>> <<<EOF>>>
```

Figures 5.31-5.38 show additional circuit synthesis results. In these runs, the edge sense feature was enabled, preventing DIDs from being assigned to supercells adjacent to failed regions. The EDGE SENSE input was driven for the supercells along the left (except for the second supercell from the top). Figure 5.31 corresponds to a run on a perfect tiling, with no faulty regions. Note that in general, supercells along the edge of the matrix are not assigned DIDs. For example, the supercells along the top row are not assigned DIDs, other than the supercell in the upper left corner (recall, the upper left corner supercell had its external EDGE SENSE inputs driven high).

Each subsequent figure corresponds to the addition of another faulty region. For example, in Figure 5.32, the supercell which was node 14 in Figure 5.31 has been marked as failed. Because of the edge sense feature, the supercells which were formerly nodes 9, 15, 19 and 13 are no longer assigned DIDs. This shifts the location of assigned DIDs in the tiling, and therefore changes the layout of the final circuit. However, in each case, the final circuit is successfully wired together, and works perfectly.

Figure 5.38 shows the most difficult case, where seven faults have been introduced. Again, every row and every column contains a faulty region, yet the system is still able to wire together a perfectly functioning final circuit.

This sequence of tests resembles what might occur during operational use of the LFSR. When a fault is detected, the system is reset (all cells' configuration memories are cleared), and the configuration string is again sent into the matrix. Each time this is done, different regions are marked as failed, due to the presence of different faults in the hardware. Each time however, the collection of supercells manages to self-assemble into the original circuit, simply using a different layout of the nodes and a different corresponding interconnection scheme among those nodes. Thus, even after a number of failures scattered throughout the matrix (as in Figure 5.38), the system successfully self repairs.

Figure 5.39 shows the final test, again the same LFSR, but this time with a single large region of failed cells introduced into the middle. Again, the system successfully wires together the final circuit, despite the presence of many faults in the middle of the matrix.

6. Conclusions

All Phase-I objectives have been met:

- The proposed supercell has been designed, implemented in a simulated Cell Matrix, debugged, and shown to work.
- Collections of supercells have been successfully tested and shown to work as expected.
- The system has been shown to be able to detect simulated faults in the Cell Matrix substrate, and to organize assemblies of cells to isolate faulty cells from non-faulty ones. Moreover, this testing has been shown to operate in parallel.
- Collections of supercells have also been shown to be able to synthesize new supercells, also in parallel. Moreover, this parallel synthesis occurs in response to a fixed configuration string, independent of the location of faults in the Cell Matrix substrate.
- Collections of supercells have been shown to be able to analyze a genome for a target circuit, differentiate into pieces of that circuit, discover pathways among themselves, and synthesize connections accordingly to implement the target circuit. The supercell collection needs no global knowledge of where the faulty regions are located. Rather, they collectively explore the supercell collection, to "discover" possible pathways for interconnecting themselves. Thus, the entire system operates autonomously.
- Finally, it has been demonstrated that a number of target circuits can be implemented successfully, under a number of different fault patterns. In all cases, the system successfully self-assembles a working version of the target circuit.

This successful completion of all objectives demonstrates the feasibility of the proposed approach.

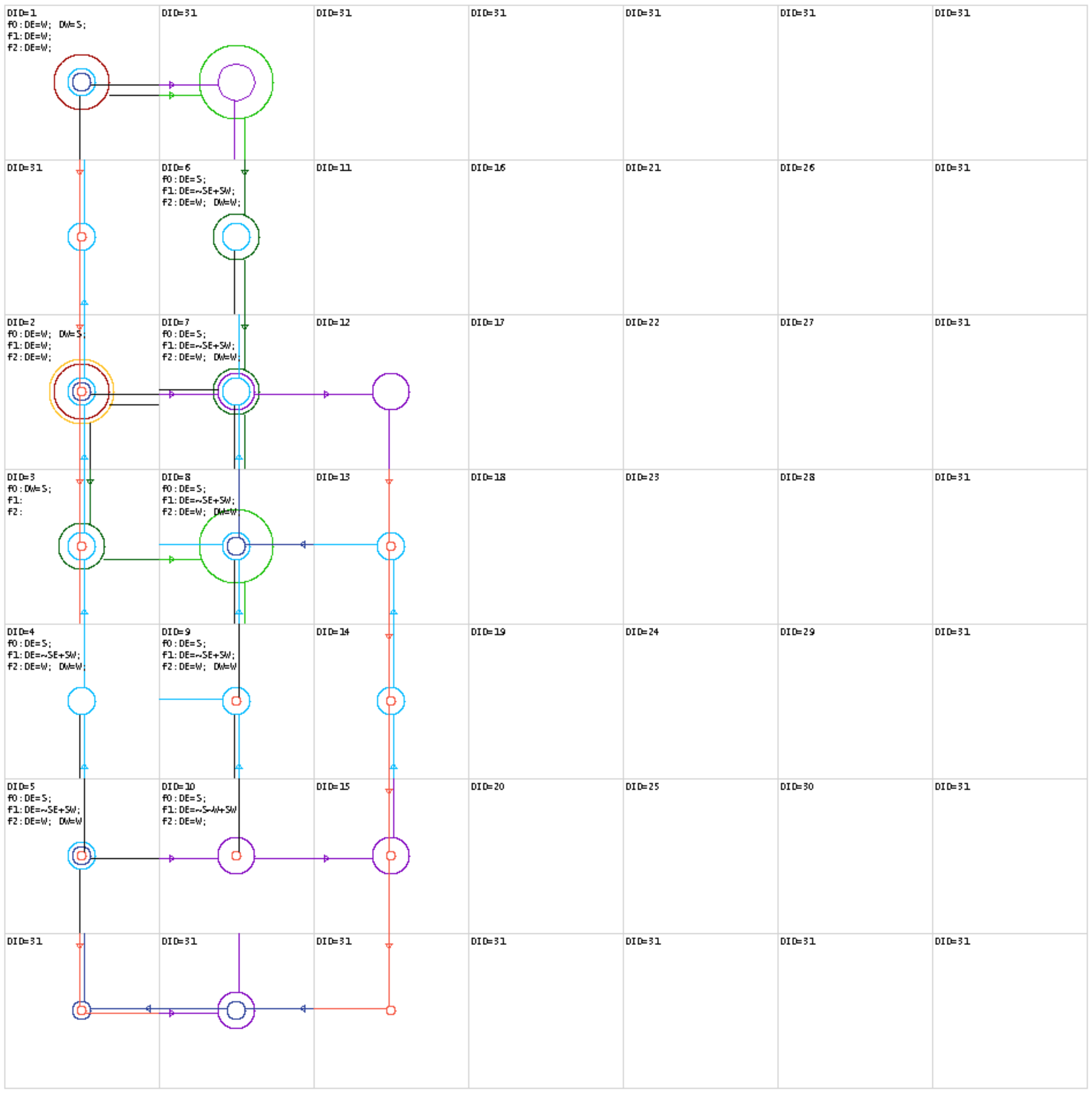


Figure 5.31
LFSR Implementation with Edge Sense Enabled
Underlying hardware contains no faults

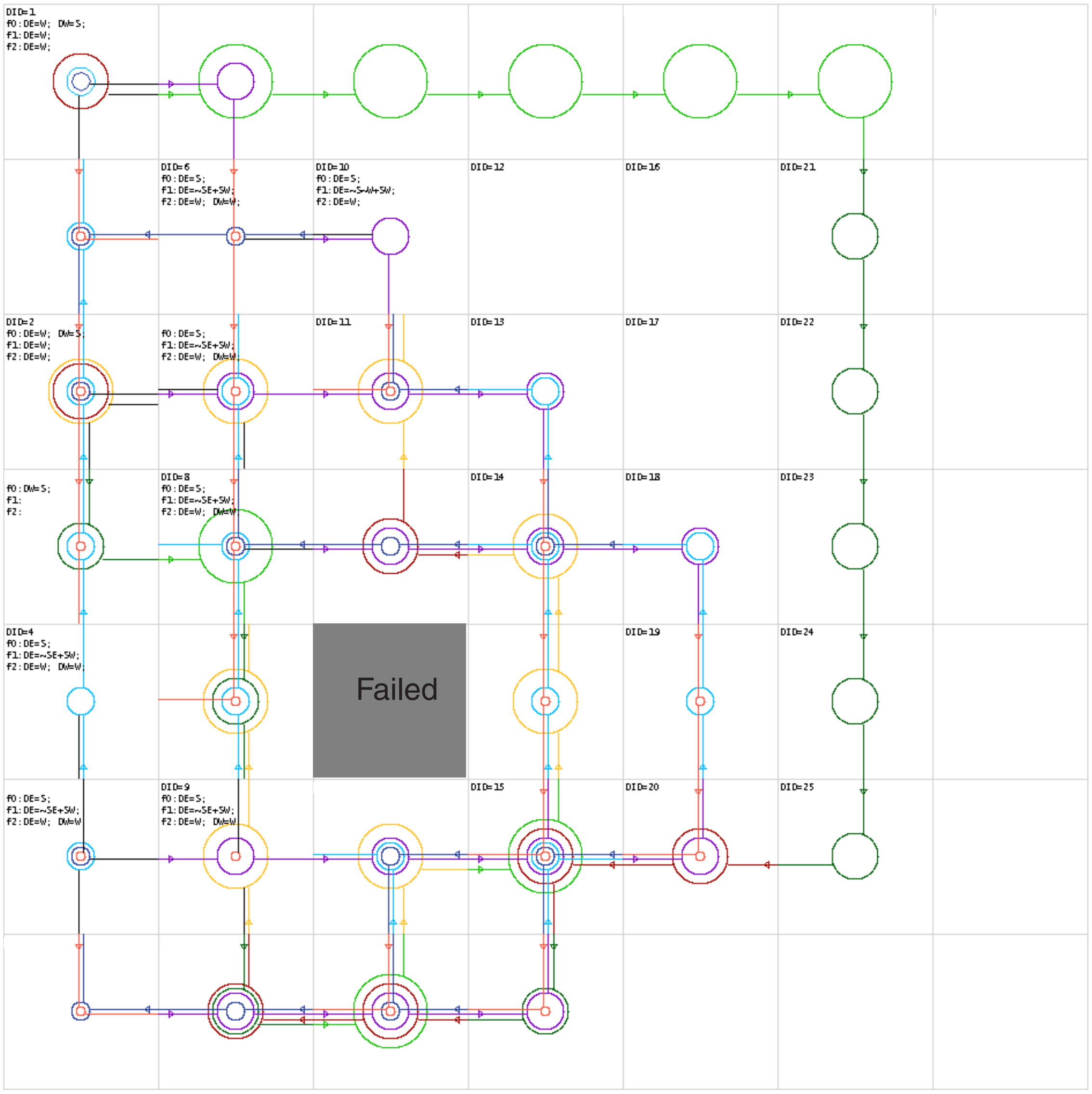


Figure 5.32
LFSR Implementation With One Faulty Region

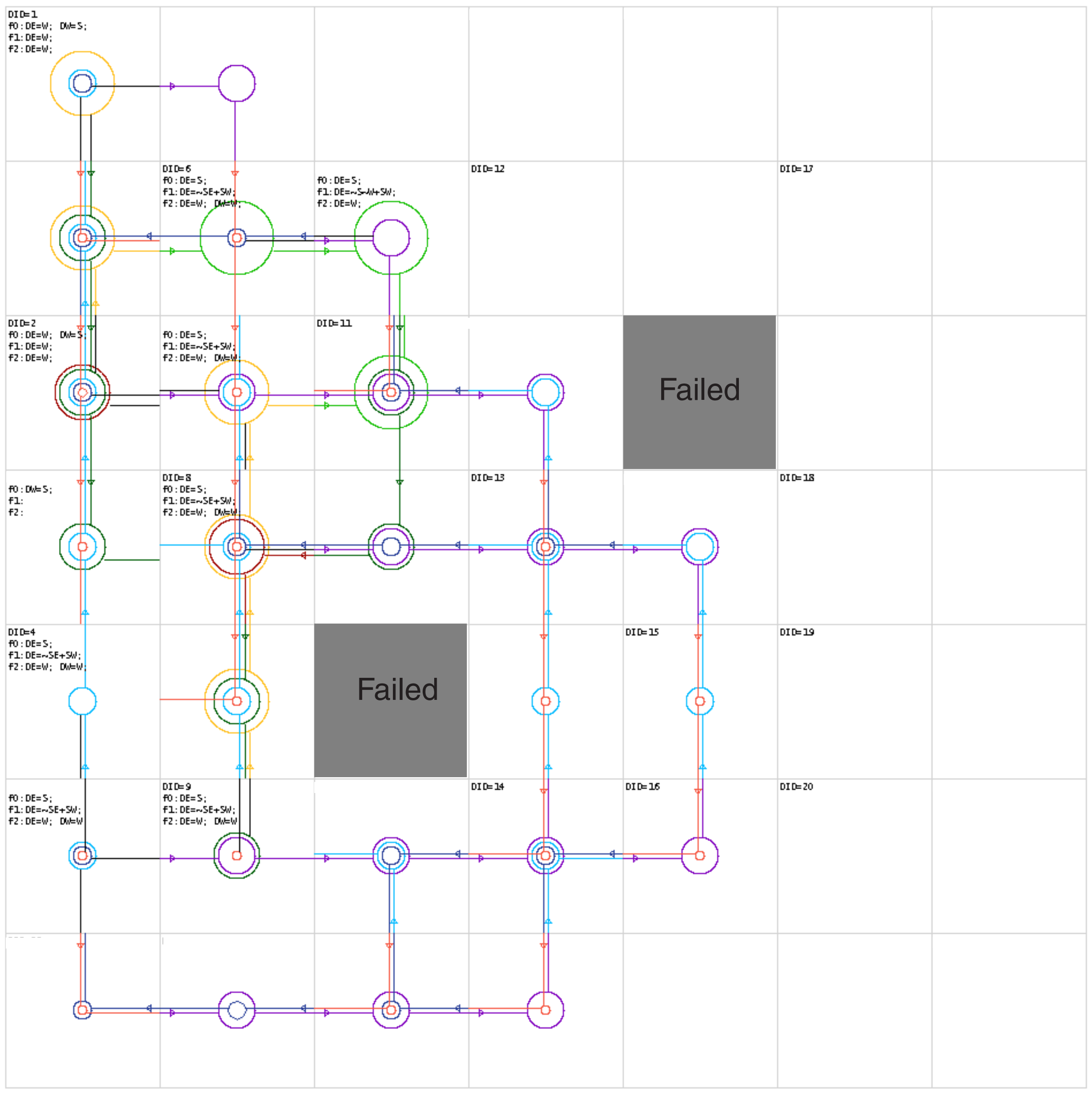


Figure 5.33
LFSR With Two Faulty Regions

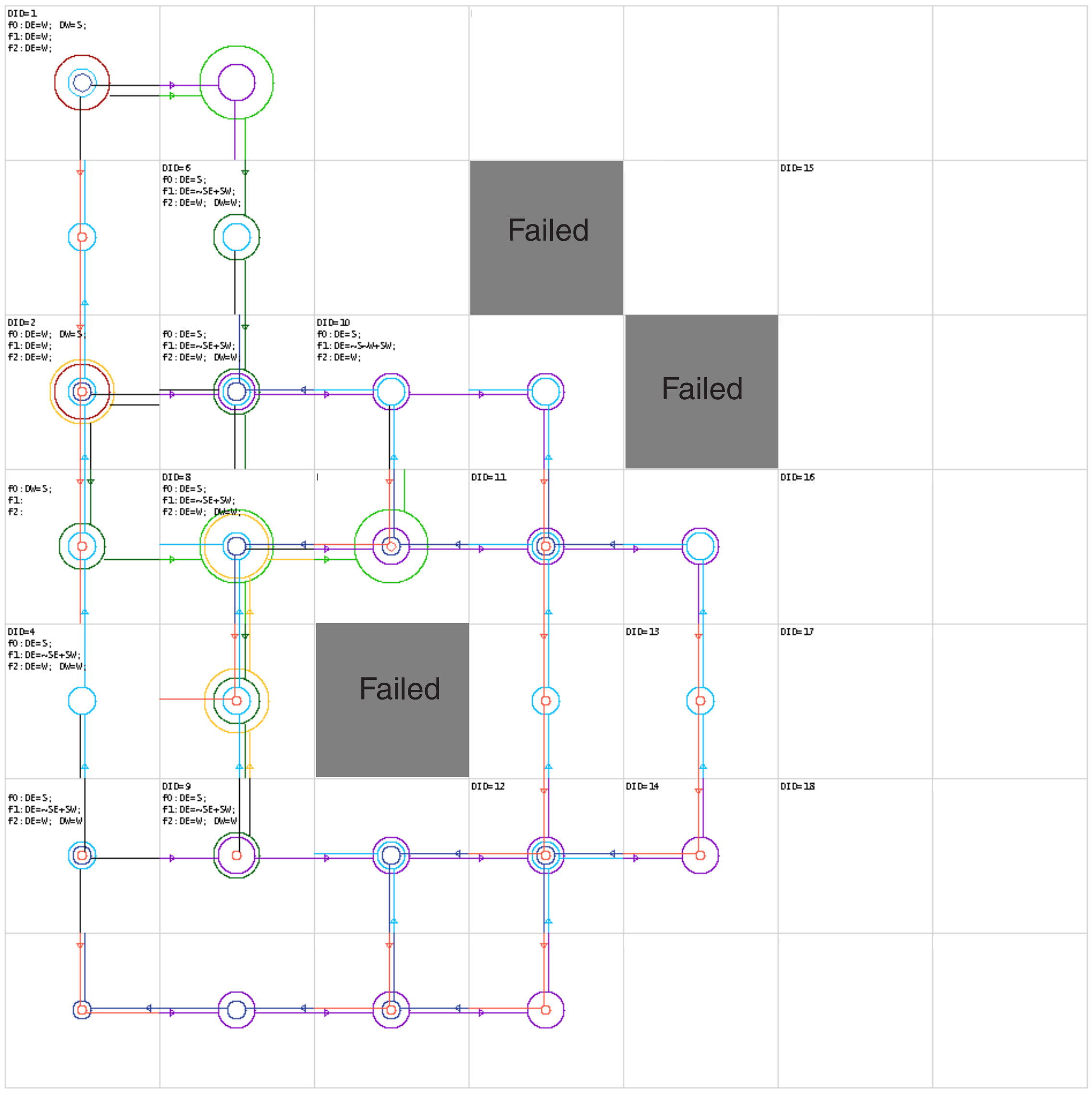


Figure 5.34
LFSR With Three Faulty Regions

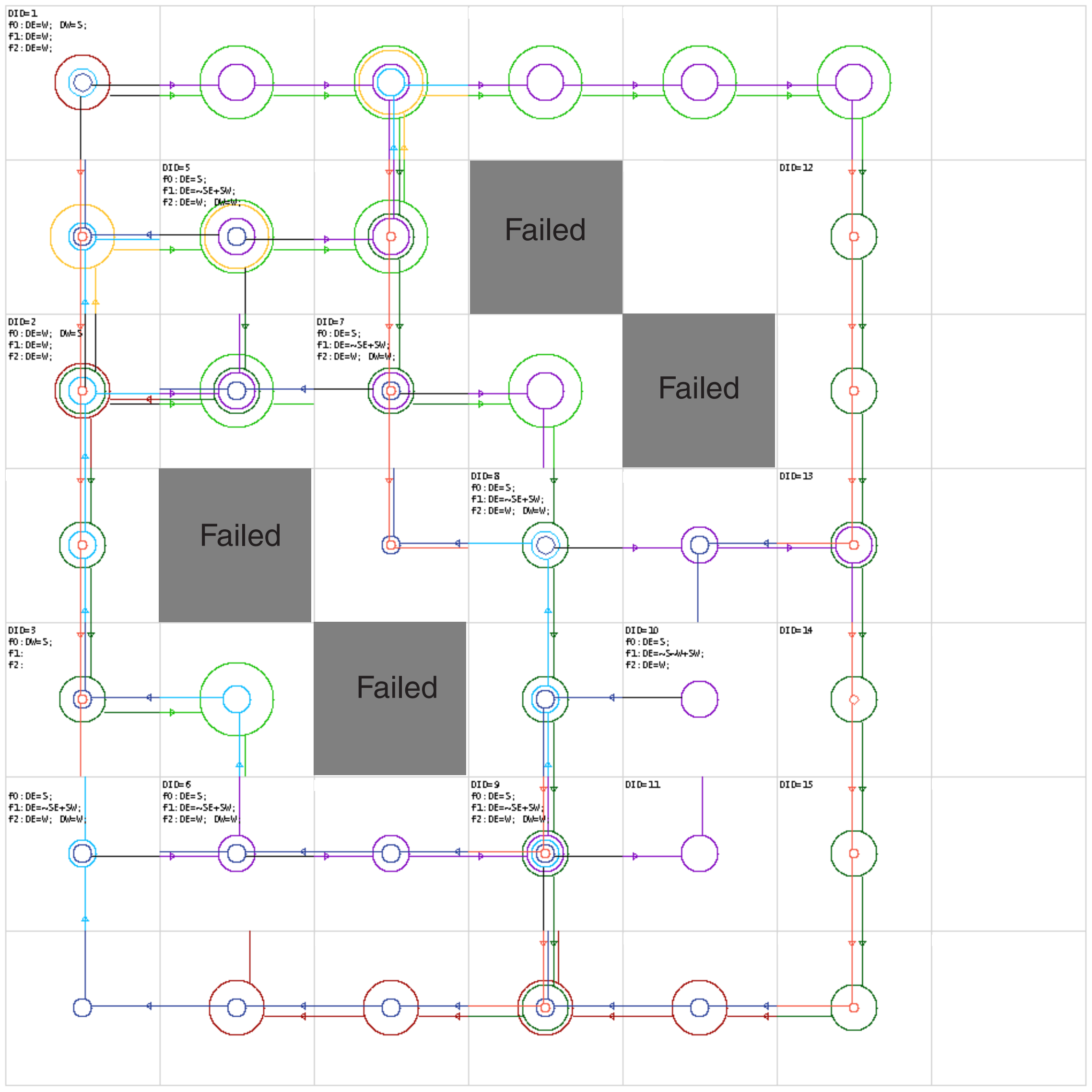


Figure 5.35
LFSR With Four Faulty Regions

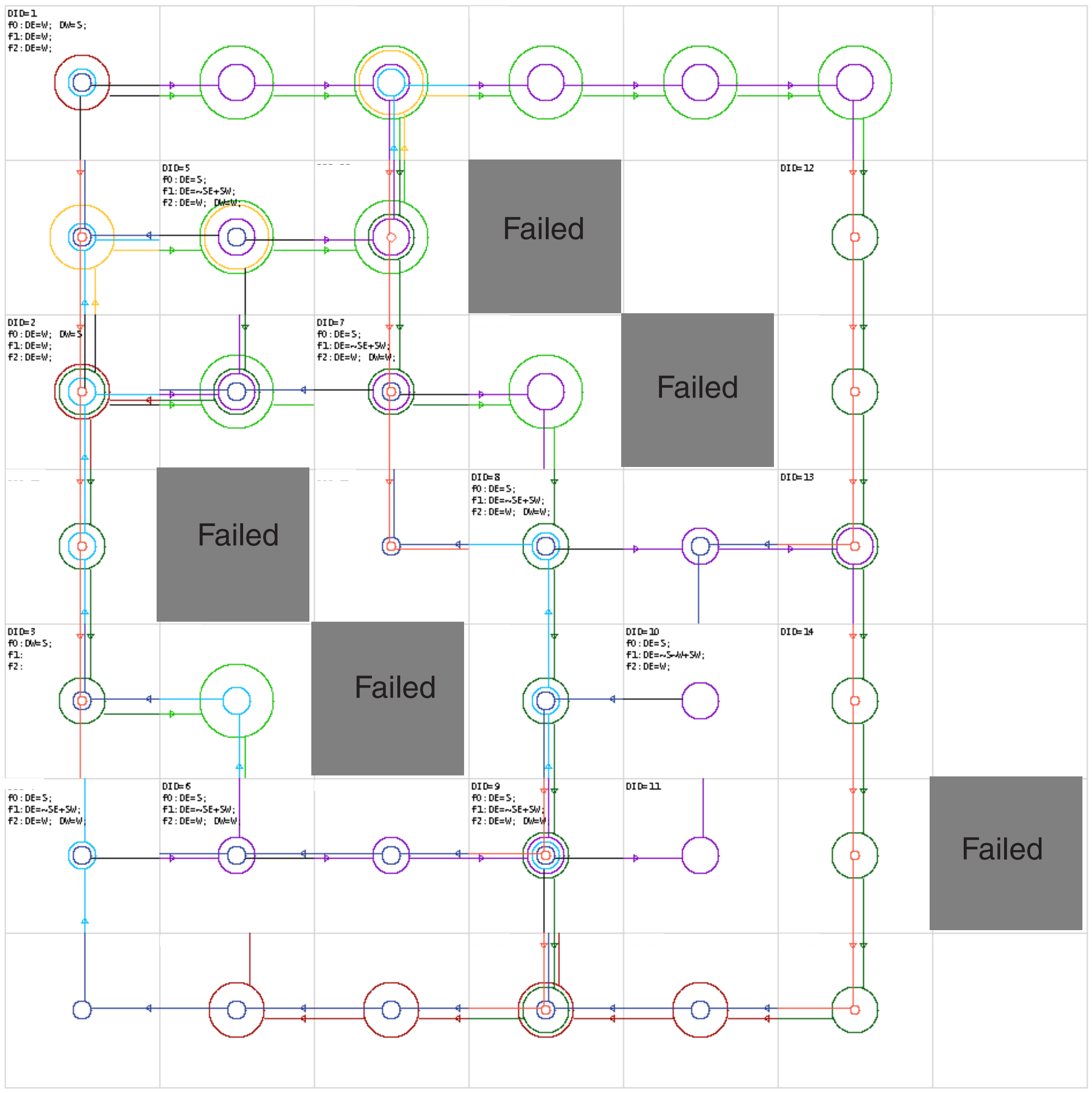


Figure 5.36
LFSR With Five Faulty Regions

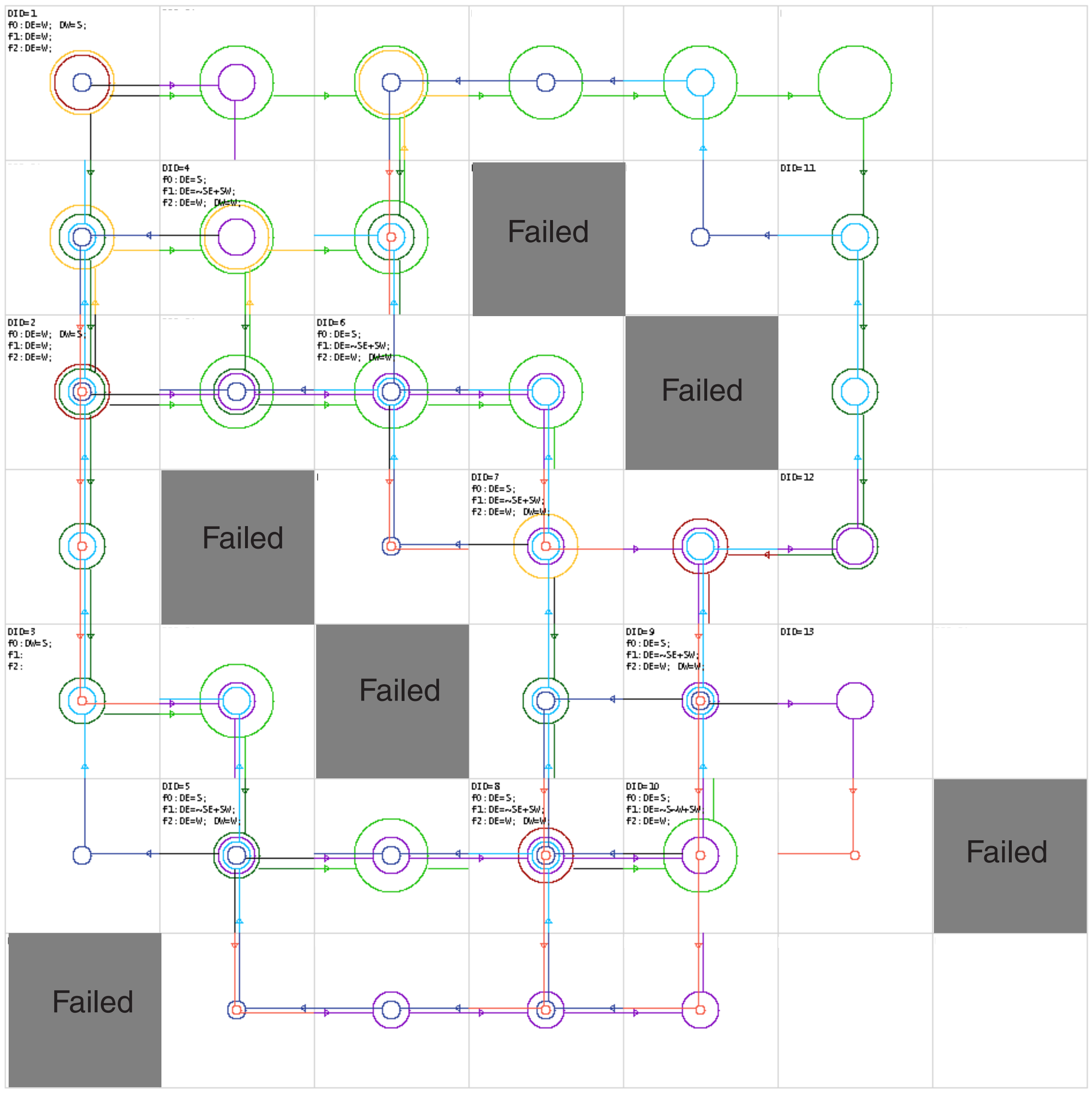


Figure 5.37
LFSR With Six Faulty Regions

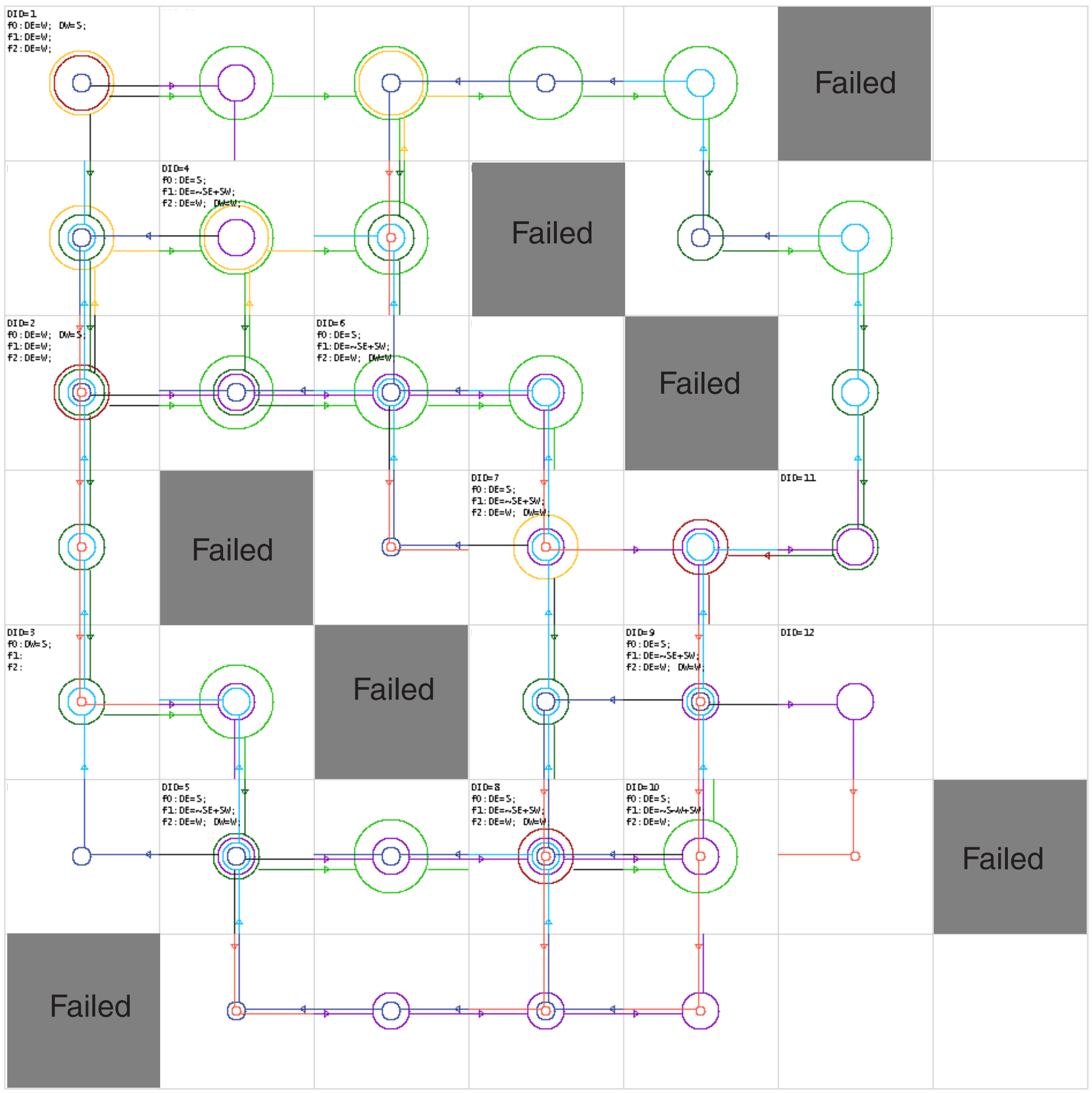


Figure 5.38
LFSR With Seven Faulty Regions
Every row and every column now contains one faulty region.
The assembled LFSR still works perfectly.

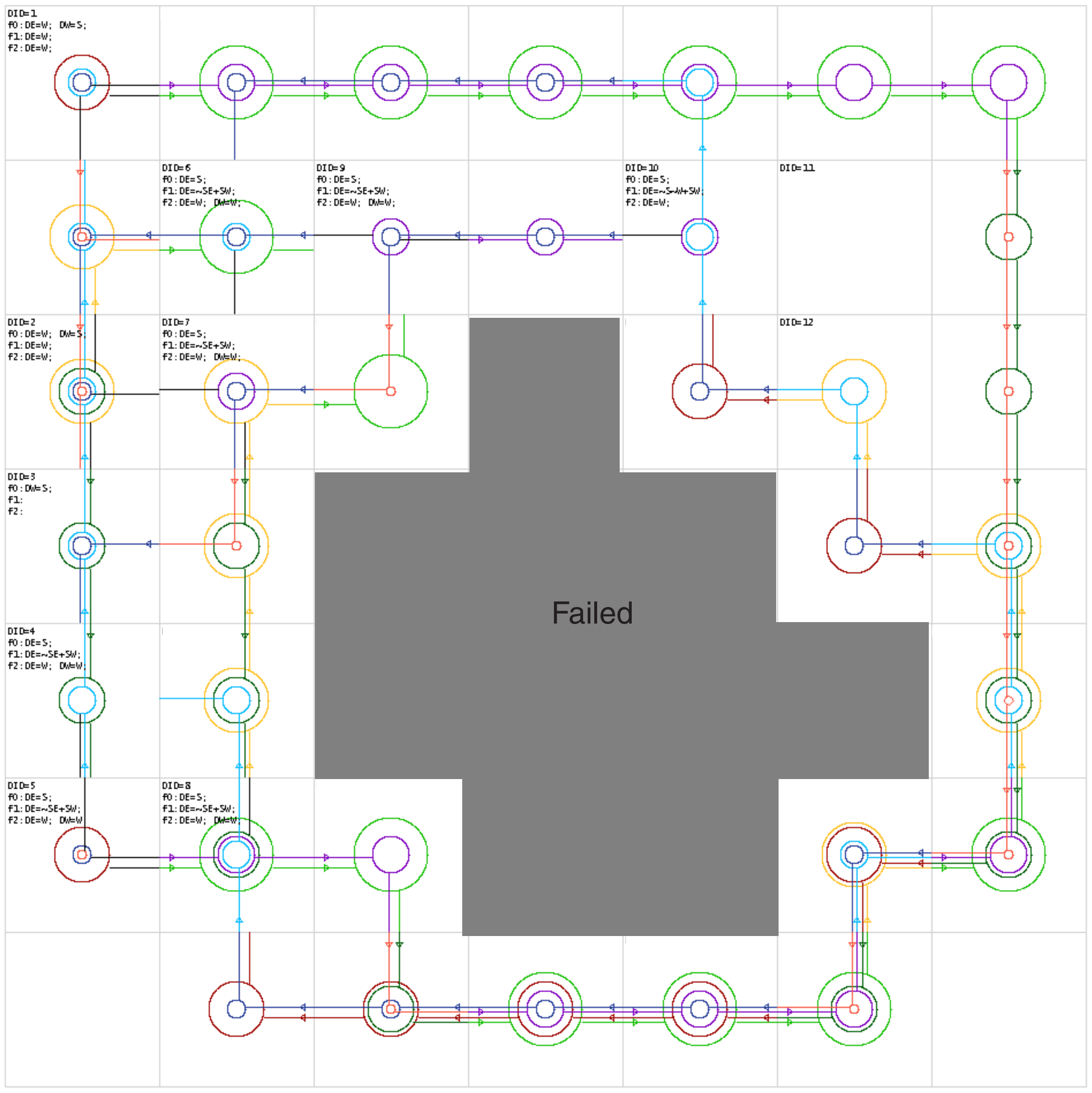


Figure 5.39
Supercell Implementation of LFSR With Edge Sense Enabled and One Large Faulty Region

The results of this work are innovative for a number of reasons:

- The system is implemented from a fixed configuration string, without the need for pre-existing knowledge about fault locations in the hardware. Thus, the entire system requires only an external memory for delivering the configuration string to the Cell Matrix.
- The work is based on a novel, fine-grained reconfigurable platform (the Cell Matrix). This is actually a general-purpose platform, which was used in this research without modification. The Cell Matrix architecture is extremely powerful, yet flexible enough to easily implement the kinds of circuitry found in supercells.
- This work involves the creation of a self-organizing systems, whose members work collectively to achieve a desired global goal.
- The system is extremely parallel, with supercells operating simultaneously and independently in an orchestrated fashion.
- The system uses the notion of distributing processing power, instead of distributing data. Each supercell generates its own local information about global events such as state changes.
- The system employs *self-configurable* elements, such as the path synthesizer which configures Cell Matrix cells to dynamically build routes between supercells. This is not only run-time configuration, it is *autonomous* run-time configuration.

The ability to implement self-repairing circuits such as the ones developed in this research is clearly important, and has applications to systems used in remote or hazardous environments such as deep space. Given the limited bandwidth which may be available in such environments, it is not feasible to perform offline analysis of a failed system, and then upload complex repair instructions. The system we have implemented needs only one command: “Repair yourself.”

The size of the supercells implemented in this work is 270x270 cells. The overhead of the supercell approach is considerable for simple target circuits such as the proof-of-concepts ones studied in Phase-I, but decreases relative to the target circuit’s size as the granularity of the genome is increased. For example, if the genome specifies functional blocks on the order of a floating point unit or a large memory, the supercell payload size may increase dramatically, while the supercell’s own size increase only slightly.

The supercell approach trades a large cell count for an extremely powerful, completely general purpose system, able to implement **any** digital circuit. This is well suited to future atomic-scale manufacturing technologies, where systems with huge switch counts will be possible, but will be prone to numerous fabrication errors.

For Phase-II, we propose focusing on a specific class of problems, for which we will develop a much finer-grained methodology, better suited to today’s manufacturing capabilities. This methodology (called *Medusa Circuitry*) does not rely on a superimposed intermediate layer of supercells between the Cell Matrix hardware and the target circuit. Rather, it performs its parallel tests and configurations, and then moves itself out of the way, freeing up the cells it had previously occupied so that new circuits may be configured on them. In many cases, the overhead completely vanishes, resulting in a cell usage which is effectively the same as the size of the final target circuit. Development of this methodology to work on faulty hardware will be one of the three major goals in our Phase-II proposal.

The power and versatility of the Cell Matrix architecture has been demonstrated by the **variety** of circuits and system elements implemented on the Cell Matrix in this research, including arithmetic processors, parallel search and sort routines, serial data processors, state machines, hardware testers, fault isolators, and path synthesizers. The examples demonstrate that beyond the present work, the Cell Matrix architecture is an extremely useful, **general purpose** reconfigurable platform. Its inherent fault tolerance, combined with fault detection and isolation methodologies, may promote the use of higher density, cutting-edge fabrication techniques well before they are viable for other architectures. Therefore, this work may lead to significant commercial possibilities in the area of general reconfigurable hardware. In Phase-II, we will propose the development of prototypes of such hardware.

The Cell Matrix has previously been shown to be a platform well suited to evolvable hardware (EHW) research. (Mac2 1999, Aro 2001, Sek 2001, Deg 2001). EHW represents another promising approach to implementing robust systems, because the implemented systems can adapt to changes in their environment by evolving competitively. Given the self-configuring capabilities of the Cell Matrix, it should support advanced EHW capabilities, including intrinsic evolution (i.e., evolution without external support) and evolution of dynamic, self-modifying circuits. This will be the third major task in our Phase-II proposal, the development of an EHW platform, both in software and in hardware.

The potential applications of this project's results in Phase-III are numerous, including: implementing of critical circuits for use in remote or hazardous environments, such as deep space; use as a commercial process driver for debugging new chip fabrication technologies; use in analyzing damage incurred in, say, a high-radiation environment; and, as nanotechnology emerges, use in implementing massively parallel digital circuits on substrates manufactured at the atomic scale which are extremely likely to contains defects. Because of its inherent fault tolerance, its extremely simple design, and its perfectly scalable architecture, the Cell Matrix provides a natural means for utilizing the extremely high switch counts which may be possible with atomic-scale manufacturing (Dur 2001). Therefore, continued research into this architecture is useful, not only for the field of autonomous self-repairing circuits, but as an ideal architectural target for future manufacturing technologies.

Commercial applications of Cell Matrix hardware itself include: use for distributed processing, e.g. in a distributed sensor-array processing system; use in architecture tuning, such as a just-in-time hardware compiler; and deployment as a next generation reconfigurable (self-configurable) device.

Commercial applications of an EHW tool set include direct sales of our hardware and software to labs and universities, and, indirectly, marketing of evolvable systems (such as an adaptive digital filter based on our EHW platform).

Finally, taking all these areas together, the long-term potential applications of our research and development include physically self-repairing systems which synthesize new hardware from raw materials in their environment, for use in interplanetary missions or remote terrestrial operations; implementation of smart materials, such as ship hulls which analyze their own stresses and strains; and implementation of intelligent, long-lived, self-reproducing artificial life.

For these reasons, we believe our Phase-I results justify award of a Phase-II contract.

7. Acknowledgements

This work was funded in part by NASA Contract NAS2-01049 under the NASA SBIR program. Cell Matrix Corporation gratefully acknowledges the assistance of personnel from NASA Ames Research Center throughout this contract, including:

Ms. Kelly Garcia (Contract and grant Administrator);
Mr. Jason Lohn (COTR);
Ms. Rosa Tonarelli (SBIR Program Manager); and
Ms. Gail Woll (Contracting Officer).

References

[Aro 2001] Arostegui, J., Sanchez, E. and Cabestany, J., "An In-System Routing Strategy for Evolvable Hardware Platforms," Proceedings of The Third NASA/DOD Workshop on Evolvable Hardware (EH'01), A. Stoica, D. Keymeulen and J. Lohn, eds., pgs. 157-158, 2001.

[CMC] Cell Matrix Corporation Website, http://www.cellmatrix.com

[Deg 1999] de Garis, H., "Review of Proceedings of the First NASA/DoD Workshop on Evolvable Hardware," IEEE Transactions on Evolutionary Computation, Vol. 3, No. 4, Nov. 1999.

[Dur 2001] Durbeck, L. and Macias, N., "The Cell Matrix: An Architecture for Nanocomputing," to appear in Nanotechnology, Institute of Physics Publishing Ltd., 2001.

[Mac 2001] Macias, N., “Circuits and Sequences for Enabling Remote Access to and Control of Non-Adjacent Cells in a Locally Self-Reconfigurable Processing System Composed of Self-Dual Processing Cells,” US Patent Application #09/702,574, allowed July 2001.

[Mac1 1999] Macias, N., Henry, L. and Raju, M., “Self-Reconfigurable Parallel Processor Made From Regularly-Connected Self-Dual Code/Data Processing Cells,” US Patent #5,886,537, issued March 1999.

[Mac2 1999] Macias, N, "Ring Around the PIG: A Parallel GA with Only Local Interactions Coupled with a Self-Reconfigurable Hardware Platform to Implement an O(1) Evolutionary Cycle for Evolvable Hardware," Proceedings of the 1999 Congress on Evolutionary Computation, 1999.

[Mac3 1999] Macias, N., “The PIG Paradigm: The Design and Use of a Massively Parallel Fine Grained Self-Reconfigurable Infinitely Scalable Architecture,” Proceedings of The First NASA/DOD Workshop on Evolvable Hardware (EH’99), A. Stoica, D. Keymeulen and J. Lohn, eds., pgs. 175-180, 1999.

[Sek 2001] Sekanina, L., and Dvoøák, V., “A Totally Distributed Genetic Algorithm: From a Cellular System to the Mesh of Processors,” Proceedings of 15th European Simulation Multiconference 2001, p. 539-543, 2001.

REPORT DOCUMENTATION PAGE		*Form Approved OMB No. 0704-0188*

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE 15 August 2001	3. REPORT TYPE AND DATES COVERED Final: 16 Feb 2001-16 Aug 2001

4. TITLE AND SUBTITLE Autonomously Self-Repairing Circuits	5. FUNDING NUMBERS C
6. AUTHORS Nicholas J. Macias, Lisa J. K. Durbeck	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Cell Matrix Corporation PO Box 510485 Salt Lake City, Utah 84151	8. PERFORMING ORGANIZATION REPORT NUMBER NAS201049F
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) NASA Ames Research Center	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES	
12a. DISTRIBUTION/AVAILABILITY STATEMENT NASA	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

A self-repairing system based on a novel self-configurable architecture is described. The self-repairing system implements a desired target circuit on a virtual intermediate layer composed of “supercells.” These supercells are able to perform all the tasks necessary to implement a target circuit on imperfect hardware, including analysis of the hardware for faults, isolation of faults, synthesis of supercells while avoiding faults, and differentiation of the supercell layer into the target circuit. Moreover, all these operations occur in response to a fixed configuration string that specifies the target circuit. For a given target circuit, the same string is always used, regardless of the location or nature of faults in the hardware. Thus, self-repair is achieved without external fault analysis or configuration string recompilation. This allows the system to autonomously self-repair in response to a single “repair” command.

14. SUBJECT TERMS Self Repairing; Fault Tolerance; Robust Systems; Reconfigurable Hardware; Evolvable Hardware; Nanotechnology; Embryonics; Moletronics; Self Configurable			15. NUMBER OF PAGES 112
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL