Design and Implementation of Detailed Router Software for Segmented-Architecture Field-Programmable Gate Arrays

by

G. G. Lemieux

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
BACHELOR OF APPLIED SCIENCE

DIVISION OF ENGINEERING SCIENCE
FACULTY OF APPLIED SCIENCE AND ENGINEERING
UNIVERSITY OF TORONTO

Supervisor: S.D. Brown

April 1993

Abstract

Segmentation is a modification to the Field-Programmable Gate Array (FPGA) routing architecture that reduces the propagation delay of interconnect signals by using longer wire segments to decrease series resistance and parasitic capacitance. Detailed routers are used to assign segments and routing switches to specific connections in way which limits resource wastage and decreases propagation delay. This thesis describes a software organization which separates the FPGA routing architecture from the routing algorithm. Two key results arise from this organization: an easily configurable FPGA routing architecture which enables further research into the issues of segmentation and switch placement, and a modifiable code base for implementing and exploring different routing algorithms.

To demonstrate use of the code and provide a first-generation detailed router for future comparisons, a new routing algorithm, SEGA, is implemented. SEGA yields excellent routing results in terms of minimum track usage and its effective use of segments, as measured by the number and length of segments allocated for each connection.

Acknowledgments

I would like to thank my supervisor, Professor Stephen Brown, for his patience and constant encouragement throughout the development of this thesis.

Contents

Abstract ii

Acknowledgments iii

Abbreviations vi

Figures vii

Tables viii

Chapter 1 Introduction 1

1.1 FPGA Software ... 2
1.2 Motivation ... 2
1.3 Organization ... 3

Chapter 2 SEGA Algorithm and Results 4

2.1 The FPGA Model ... 4
2.1.1 Wire Segments ... 5
2.1.2 The S Block ... 6
2.1.3 The C Block ... 6
2.2 Problem Definition ... 7
2.3 The SEGA Algorithm ... 8
2.3.1 Phase 1: Enumerating the Detailed Routes ... 8
2.3.2 Phase 2: Connection Formation ... 10
2.3.3 The Cost Function ... 10
2.4 Results ... 12
2.4.1 Routing Completion Results ... 12
2.4.2 Wire Segment Allocation Results ... 13
2.4.3 Segment Allocation Impact on Routing Completion ... 14
2.5 Conclusions ... 15

Chapter 3 Software Design 16

3.1 Organizational Overview ... 16
3.1.1 General Implementation Details ... 17
3.2 FPGA Module ... 18
3.2.1 Creating and Initializing an FPGA Instance ... 18
3.2.2 Defining the FPGA Routing Architecture ... 19
3.2.3 Routing Architecture Data Structures ... 21
3.2.4 Using the FPGA ... 25
3.3 Path Module ... 27
3.3.1 Creating and Initializing a Netlist Instance ... 27
3.3.2 Enumerating Graphs and Paths ... 29

3.3.3 Cost Function 30
3.3.4 Data Structures 31
3.3.5 Operations on Graphs and Paths 33
3.3.6 Correctness 33
3.4 Route Module 34
3.5 Main Module 35
3.6 Plot Module 36

Chapter 4 Conclusions and Future Work **37**

References **38**

Appendix A Route Module Code **39**

Abbreviations

SEGA	Segment Allocator, a Detailed Routing Algorithm
CGE	Coarse Graph Expansion, another Detailed Routing Algorithm
FPGA	Field-Programmable Gate Array
CAD	Computer Aided Design

Figures

Figure

2.1. An M x N FPGA. 5
2.2. Examples of S and C blocks. 6
2.3. An example of an FPGA routing problem. 8
2.4. Expansion of a coarse graph. 9
2.5. Pseudo code for Phase 2 of SEGA. 10
2.6. Segment allocation results. 14
3.1. Code to create a simple 10 x 12 FPGA with 8 tracks per channel. 18
3.2. FPGA instance data structure. 21
3.3. FPGA routing channel data structures. 22
3.4. Wire segment data structure. 23
3.5. Wire segment connectivity in a small FPGA. 25
3.6. A portion of the netlist structure. 28
3.7. (a) A coarse graph and (b) the coarse graph with express edges. 29
3.8. Example elements in the cross referencing arrays. 32
3.9. Pseudo code used to prove correctness of the detailed route. 34
3.10. Route Module pseudo-code implementing the SEGA algorithm. 35

Tables

Table
2.1. Benchmark circuit characteristics. 13
2.2. Impact of Cα and Cβ on routing completion. 15

Chapter 1
Introduction

An important objective of a digital system developer is to minimize cost while maximizing performance. Although high performance would be realized by producing customized integrated circuits, such systems can be prohibitively expensive for low volume units due to the high setup cost associated with the fabrication process. Cost-sensitive developers must seek alternative ways to implement their digital system, compromising performance for the sake of cost.

One viable alternative to customized IC design has been the use of Field-Programmable Gate Arrays (FPGAs). FPGAs are capable of implementing digital circuits with fairly high logic densities, yet they can be programmed by the user. Additionally, rather than sending a design to a fabrication facility where custom silicon would be formed in a few weeks to months, it can be programmed into an FPGA in minutes. Should a design error be found, the fault can be fixed and changes tested almost immediately.

The flexibility offered by the FPGA makes it invaluable for prototyping or small-scale production, for example. At the same time, however, it also hinders speed performance. While custom gate array connections benefit from using metal wires tailored to the length of the connection, the programmability of the FPGA requires many short *wire segments* to be connected together through *routing switches*. These routing switches add a distributed resistive-capacitive load to the connection, increasing the propagation delay of signal transitions. This ultimately limits the speed of the FPGA circuit and makes FPGAs unattractive for high-speed applications.

To improve the speed performance of FPGAs, and thus increase their usefulness, the architecture of FPGAs has evolved to include wire segments of varied lengths and fewer routing switches. This typically improves the speed of connections between possibly distant circuit elements by decreasing the number of routing switches through which the signal must propagate, and by decreasing the number of unused routing switches forming an unnecessary load. It is possible, however, to decrease performance by using long wire segments for short connections, because the long segment could probably be better utilized by a long connection and it represents an larger capacitive load than what a shorter wire segment would incur. Thus, the assignment of

wire segments to connections can influence the speed performance of a circuit realized in an FPGA.

1.1 FPGA Software

The use of good Computer-Aided Design (CAD) tools for programming FPGAs is essential for effective use of the FPGA resources because the vast possibilities can quickly overwhelm any human designer. FPGA CAD tools must do to digital circuits what a compiler does to source code in a high-level programming language; it must translate the source specification into a binary code which utilizes the device resources in an efficient way.

Like a compiler, the process of converting the source specification into a binary code is broken into a number of sequential stages to make the problem more tractable. FPGA programming with CAD tools typically involves the following steps: initial design entry, logic optimization, technology mapping, placement, and routing [1]. While the first two steps perform obvious functions, technology mapping partitions the logic functions of the circuit into blocks which can be implemented by logic elements of the FPGA, placement finds ideal locations for the blocks in the organization of the FPGA and routing connects the appropriate FPGA logic blocks together so the desired circuit can be rendered.

The routing stage can be broken into two sequential processes. The first, called *global routing*, must assign each to connection an imprecise pathway, a *coarse graph*, through the FPGA interconnect. These coarse graphs are used by the second stage, *detailed routing*, as a basis for determining a precise path. The set of these *detailed paths* form the detailed route which can be used to program the interconnect medium of the FPGA. Quite obviously, the detailed graphs are dependent upon the organization of the interconnect — the placement of wire segments and routing switches.

1.2 Motivation

Although FPGAs are capable of high logic capacities, speed performance has become a critical issue for expanding their role in digital design. The generalized interconnect structure of the FPGA forms the largest speed bottleneck. To improve performance, current research focuses on determining the best organization and layout of an FPGA, known as its *architecture*.

Many approaches are being used to improve upon the speed of FPGAs. One such approach attempts to keep signals out of the routing interconnect as much as possible by improving upon the

logic functionality present at a local level. Inevitably, however, signals must travel through the interconnect to form a large circuit. Thus, speeding up the interconnect itself is an important goal. This is done by modifying the routing architecture so that minimal delay is imposed upon each connection formed therein.

To aid current research in the development of good FPGA architectures, it is important to have CAD tools which can effectively utilize the FPGA's resources. In particular, routing architecture research needs good routing software to fairly evaluate the capabilities of each FPGA design. This thesis describes a software organization which facilitates research of routing architectures and their algorithms. To aid both research areas, clear distinction is made between code dependent upon the FPGA architecture and architecturally-independent code. The abilities of the software are demonstrated with the implementation of a new detailed routing algorithm, called SEGA, which considers wire segmentation as part of the routing problem.

1.3 Organization

This thesis is organized as follows. Chapter 2 provides information about the algorithm of the detailed router implemented, SEGA, and associated results. Chapter 3 describes the design and organization of the software used to separate the architecture and the algorithm. The last chapter draws conclusions from the work, and outlines future work that can be done with the design.

Chapter 2

SEGA Algorithm and Results

The detailed routing algorithm called SEGA is first described in [2]. Here, the FPGA model used by SEGA is introduced and the SEGA algorithm is explained. It is pertinent to describe the SEGA algorithm to give necessary background information and insight into the operation of a detailed router. In this way, the software underlying SEGA can be better understood. The chapter concludes by presenting routing results produced by SEGA, showing it to be an effective routing algorithm.

2.1 The FPGA Model

SEGA is designed as a successor to the Coarse Graph Expansion (CGE) algorithm originally presented in [3][4]. Both of these tools are designed to route FPGAs which fit the symmetrical model, described below.

The model represents an FPGA as a rectangular array of *Logic (L) blocks*, depicted in Figure 2.1. Often, logic blocks contain programmable elements of random and sequential logic, but no assumption about that organization is made in this software. More importantly, the space between the logic blocks forms an interconnection medium composed of horizontal and vertical routing channels. Each channel is broken up into alternating *Connection (C) blocks* and *Switch (S) blocks* as well as metal *wire segments* and logic block *pins*. The C blocks and S blocks contain programmable switches which can connect wire segments to pins and to other wire segments, respectively. The channel width, W, denotes the number of *tracks*, or rows (columns), of wire segments contained within. Wire segments can span one or more C blocks, passing through zero or more S blocks in the process. Figure 2.1 shows an FPGA with channel width of three, and each wire segment spans a single C block.

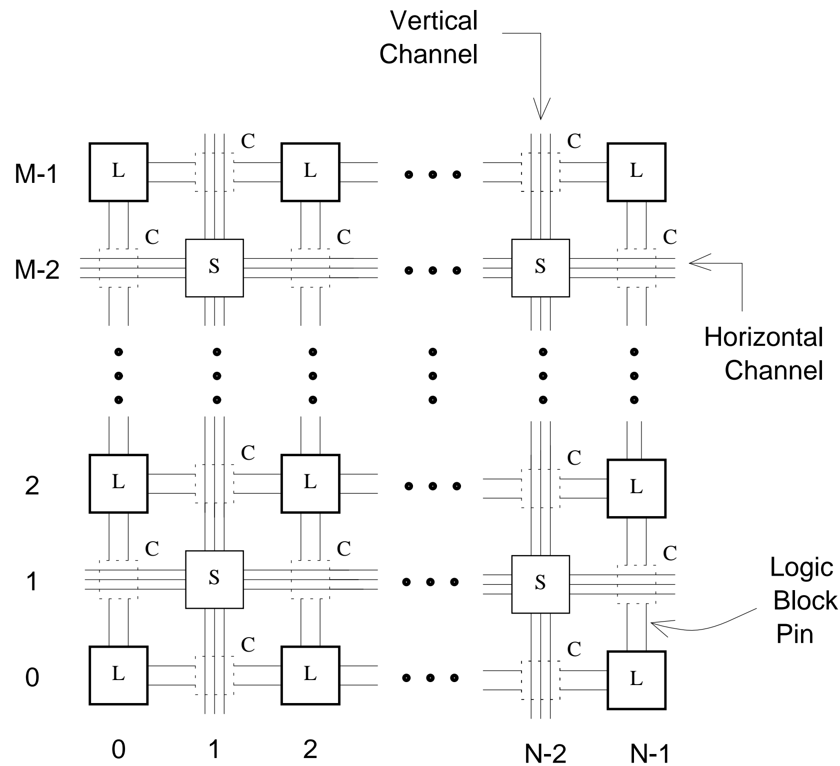

Figure 2.1. An M x N FPGA.

2.1.1 Wire Segments

Wire segments form the backbone of the routing network. They can be thought of as bus wires which can be connected together via S block routing switches to form a longer bus wire.

The length of a wire segment is defined as the number of C blocks through which it passes. When a long wire segment is used for part of a long connection, speed performance can be greater. This is because the signal does not have to pass through as many routing switches at the S blocks, each of which can cause significant delay. However, longer wire segments have inherently greater parasitic capacitance, potentially increasing delay for short connections. Additionally, using a long wire segment for a short connection wastes resources, preventing other connections from being formed. This can cause a circuit to be unroutable, meaning not all of the desired connections can be formed. Thus, only segments appropriate for the connection length should be chosen.

2.1.2 The S Block

Frequently, wire segments must be connected together to form longer connections. The S block holds routing switches that enable a wire segment to connect to other wire segments. An S block is represented as a general four-sided switch box where horizontal and vertical wire segments meet. The maximum number of other wire segments any segment can connect to within an S block is determined by the flexibility of the S block, F_S. In Figure 2.1a, switches attached to long wire segments passing through the S block are represented with an X, and switches which connect the ends of segments are represented with a dashed line. Research has shown that a good value for F_S is three [5][6]—this is reflected in Figure 2.1a. This value of F_S is an important assumption used throughout most of the thesis, since it clearly limits the fanout which can occur at an S block to one.

2.1.3 The C Block

When a signal enters or leaves the interconnect, it must do so through a C block connection switch. The switch, when set, connects a pin to a wire segment. The flexibility of a C block, parameter F_C, is defined as the number of wire segments to which a particular pin may connect. Figure 2.2b shows an example C block where F_C is two, i.e. each pin can connect to two of the three wire segments illustrated. For example, pin two of the left logic block can connect to wire segments in tracks two and three.

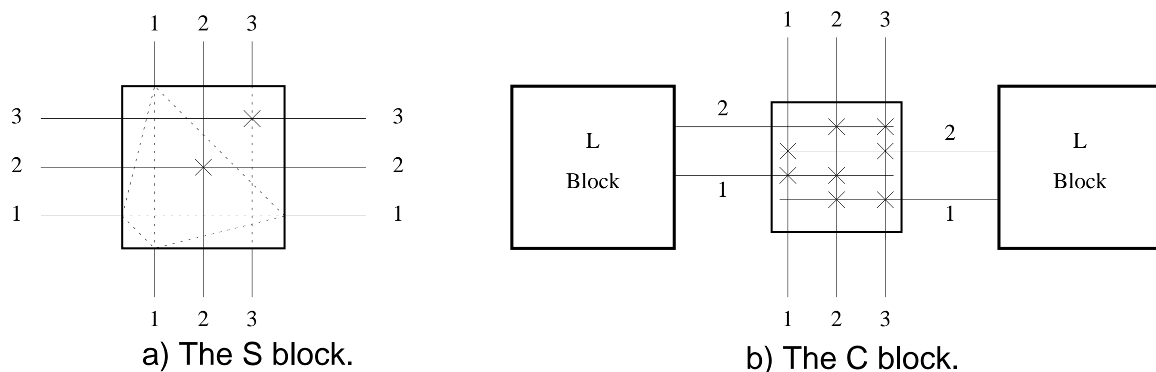

Figure 2.2. Examples of S and C blocks.

2.2 Problem Definition

As described above, routing is divided into global routing and detailed routing. Global routing is done separately as preparation for detailed routing. The global router used divides connections into two-point connections and routes each separately. While routing, it attempts to distribute the density of connections evenly throughout the channels as well as minimize the number of *turns* a connection makes. These are important concepts because the routing resources present in each channel segment are limited, and straight connections can better utilize long segments if they do not turn corners.

The global router assigns a coarse graph to each two-point connection. The coarse graph defines a route through the channels of the FPGA, specified by a sequence of coordinates representing channel segments, which connects logic block pins together. This information forms the input netlist, used by the detailed router as a template for allocating wire segments to connections.

Since multi-point connections are divided into a number of two-point ones, some connections that are part of the same net will overlap within a routing channel. Special consideration should be made to allow these connections to be recombined and reduce possible wastage of routing resources.

For each coarse graph, the detailed router must analyze the numerous paths allowed by the routing architecture and choose the "best" one. The best path is defined as one which would present minimal delay to the signal and have the least impact upon the routability of other coarse graphs. These ideals are not always simultaneously satisfiable, so tradeoffs must occur.

In order to illustrate the possible problems involved, consider the routing example in Figure 2.3. As indicated by thicker wire segments, the figure shows the possibilities for three different connections. Connection A has two possible detailed paths, one on track two and another on track three. Should the router use track two as the detailed path, both Connection B and Connection C are left to rely upon only one possible path (track one). Since Connection B and Connection C carry different signals, they cannot both be routed on track one.

The above example does have a solution if Connection A is formed using track three. Then, Connection B and Connection C can be formed by choosing track one for one of them, and track two for the other. If segment lengths are matched to the connection lengths, the best choice for Connection B is track two and the best choice for connection C is track three. Unfortunately, the

best choices are not always so obvious, so a detailed router must sometimes make guesses for difficult decisions.

One final note should be made about optimizing the speed of each connection. It is well known that the greatest amount of effort to improve performance should be spent upon those connections which lie in the critical path of the circuit. Unfortunately, it is very difficult to automatically determine where the critical path lies so that it can be optimized. Thus, it is assumed that all connections should be optimized in order to have the greatest chance of speed improvement.

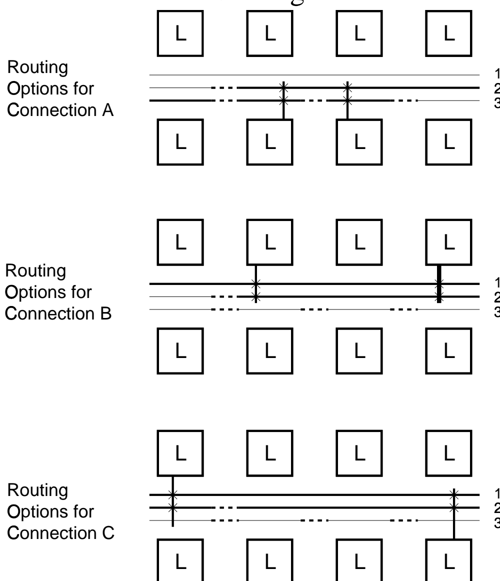

Figure 2.3. An example of an FPGA routing problem.

2.3 The SEGA Algorithm

After inputting the coarse graphs from the global router, the SEGA algorithm proceeds in two phases. In the first phase, each coarse graph is assigned a number of alternative detailed paths. The second phase follows by selecting which detailed path is to be used for each coarse graph. By enumerating the alternatives in the first phase, SEGA can consider the side effect of choosing one alternative over another in the second phase. This enables SEGA to match connection lengths to segment lengths without unnecessarily compromising the desired result of forming all connections.

2.3.1 Phase 1: Enumerating the Detailed Routes

In the first phase, SEGA expands each coarse graph into all detailed paths permitted by the routing architecture. By limiting F_S to three, no fanout can occur at an S block and each coarse graph will

expand into a maximum of F_c detailed paths. Figure 2.4 illustrates a coarse graph being expanded into its detailed paths.

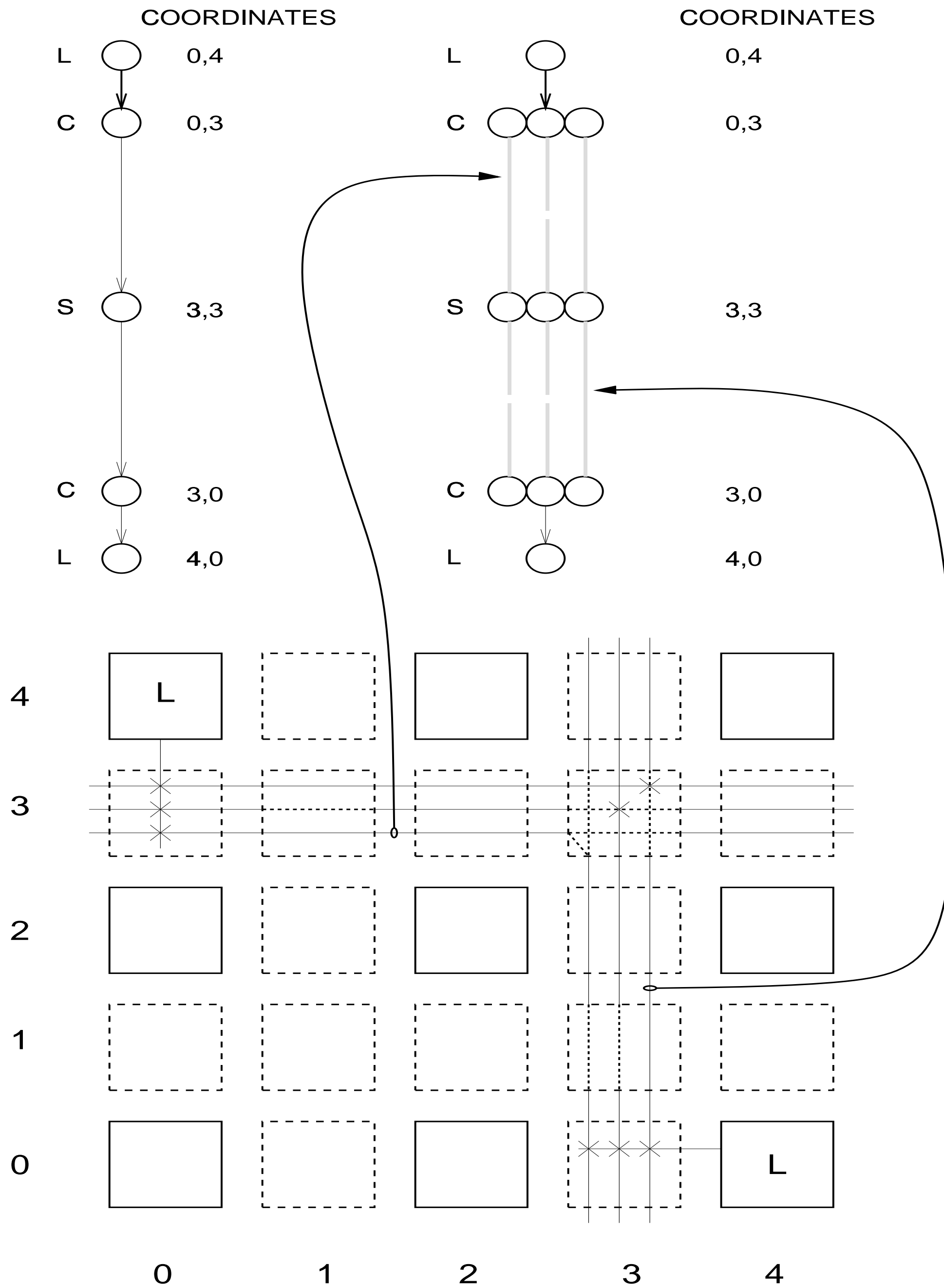

Figure 2.4. Expansion of a coarse graph.

2.3.2 Phase 2: Connection Formation

Once the coarse graphs have been fully expanded, the second phase proceeds to select one detailed path for each coarse graph. The selection decision is made by choosing the lowest cost path (the cost of a path will be described below), however, any connections which are deemed *essential* are always considered first. An essential connection is defined as a coarse graph which has only one possible path remaining; other alternatives (if any) have been exhausted by the formation of other connections. Pseudo code representing this part of the algorithm is presented in Figure 2.5.

```
place all detailed paths into the path list

while the path list is not empty

    if any essential paths exist in the path list
        select the essential path with the lowest cost
    else
        select path with the lowest cost
    end if

    mark the graph corresponding to the selected path as routed,
        and remove all paths in this graph from the path list.

    find all paths that would conflict with the selected path
        (i.e. all paths that are part of different nets, but
        rely upon a wire segment just allocated to the selected
        path) and remove them from the path list.  If a graph
        loses its last remaining path, it is deemed unroutable.

    update the cost of all affected paths

end while
```

Figure 2.5. Pseudo code for Phase 2 of SEGA.

2.3.3 The Cost Function

The selection mechanism in Phase 2 of SEGA is driven by a multi-part cost function defined for each path, p, as:

$$Cost(p) = w_{\alpha}C_{\alpha}(p) + w_{\beta}C_{\beta}(p) + w_{c}C_{c}(p) + w_{f}C_{f}(p). \quad 2.1$$

The first two terms in the cost function reflect an effort to efficiently assign segments of appropriate lengths to a connection, and the last two are concerned with successfully routing all connections. Each cost term is further described below:

$$C_\alpha(p) = \frac{\text{total length of all segments} - \text{minimum possible segment length}}{\text{total length of all segments}} \qquad 2.2$$

$$C_\beta(p) = \frac{\text{number of segments - minimum possible number of segments}}{\text{number of segments}} \qquad 2.3$$

$$C_c(p) = \frac{\text{number of remaining paths}}{F_c} \qquad 2.4$$

C_f:

```
/* pseudo code to compute Cf */
for each path p
    Cf(p) = 0.0
    for each wire segment w in p
        for each path q which also contains w

            /* find the number of alternatives */
            alt = # paths in expanded graph of q
                  which don't conflict with p

            Cf(p) += 1.0 / alt

        end for
    end for

    if Cf(p) < minimum then minimum = Cf(p)
    if Cf(p) > maximum then maximum = Cf(p)

end for

/* rescale */
for each path p
    Cf(p) = (Cf(p) - minimum) / (maximum - minimum)
end for
```

The C_α cost is similar to the one described in [7]. Its purpose is to encourage using paths which do not use overly long wire segments for a connection. The C_β cost, similar to the one used in [7] and [8], encourages the use of paths using as few wire segments as possible. The third cost, C_c, gives priority to paths of graphs which are running out of remaining choices. Finally, C_f defers the use of paths containing wire segments in high demand. C_f is the same cost function as

used in [3][4], except the result is linearly rescaled so that the minimum C_f found maps to zero and the maximum C_f found maps to one. This way, all of the cost terms are in the range [0,1] and no single term can dominate by becoming very large. If the user wishes to emphasize a certain routing objective, the w coefficients can be chosen to form an appropriate weighting scheme. For circuits which need to run faster, w_α and w_β can be increased, while circuits which are difficult to route can have w_c and w_f increased.

When choosing the path with the minimum cost results in a tie, the cost comparator favours the path with the lowest C_f cost in order to minimize the negative impact on others.

2.4 Results

To test the effectiveness of the SEGA routing algorithm, the program was run on a set of circuits, most of which are derived from the MCNC benchmark suite. For all experiments, F_S is three and F_C is equal to W. Additionally, the C_f term is not added to the cost function (i.e. its weight is zero), but it is used to break ties. The results are divided into three sections: the first shows that the algorithm obtains excellent results in terms of routing completion, the second shows how wire segment allocation is improved using C_α and C_β, and the third illustrates the impact of segment allocation upon routing completion.

2.4.1 Routing Completion Results

To demonstrate SEGA's routing ability, the FPGA was configured to consist of only unit length wire segments. This allows SEGA to be directly compared with CGE, a detailed router specifically designed for this configuration. Also included are results from a "maze" router, obtained by disabling CGE's cost function. This turns it into a sequential router, similar to a classical maze router.

Routing ability is measured by the number of tracks required to route 100% of all connections. The results in Table 2.1 indicate the minimum channel width required by each router to route 100% of all connections. These results are bounded by a minimum value, the channel density, since that indicates the maximum number of connections placed in a single channel segment by the global router. As shown, the SEGA results are consistently very close to the minimum possible—in total, it requires only seven extra tracks. Also, the table indicates that SEGA performs slightly better than CGE by using six fewer tracks, and much better than the maze router.

The maze results show how important it is to consider side effects when forming connections. Obviously, SEGA addresses this issue well.

Circuit Name	Channel Density	Minimum *W* for SEGA	Minimum *W* for CGE	Minimum *W* for Maze
9symml	10	10	10	12
alu2	10	11	12	17
alu4	13	15	15	20
apex7	13	13	13	15
example2	17	17	18	21
k2	15	17	19	27
term1	9	10	10	13
too_large	11	12	13	17
vda	13	13	14	19
z03	14	14	14	21
Total	**125**	**132**	**138**	**182**

Table 2.1. Benchmark circuit characteristics.

2.4.2 Wire Segment Allocation Results

The ability to effectively allocate wire segments is shown by configuring the FPGA with segments of various lengths. The FPGA used for these experiments has a channel width of 20 and its segment lengths determined by a Poisson distribution with mean 0.5. This resulted in a wire segment length distribution with approximately 60% of length one, 30% length two, 8% length three, and 2% longer than three.

Of the benchmark suite, only those circuits which could be 100% routed in the segmented FPGA are included. The length and number of segments used in the routing of all the circuits was summed and normalized against the minimum possible. The minimum total length is equal to the number of single length segments routing all the connections would require, and the minimum number of segments is equal to the total number of straight sections present in the coarse graph. After normalization, the minimum value possible for either metric is one.

The effect C_α and C_β have upon segment allocation is shown in Figure 2.6. When both costs are disabled by setting their weights to zero, no concern is given to segment allocation. Enabling either cost shows a marked improvement in the segment allocation quality: the number of segments

used drops from 49% in excess to 28% in excess of the minimum, and the length of segments used drops from 21% in excess to 6%. When both costs are enabled, both aspects of segment allocation deteriorate only slightly compared to their individual minima, to 29% and 8% in excess. This shows that the two costs do not unduly compete with each other when used together, and the overall result is better segment allocation.

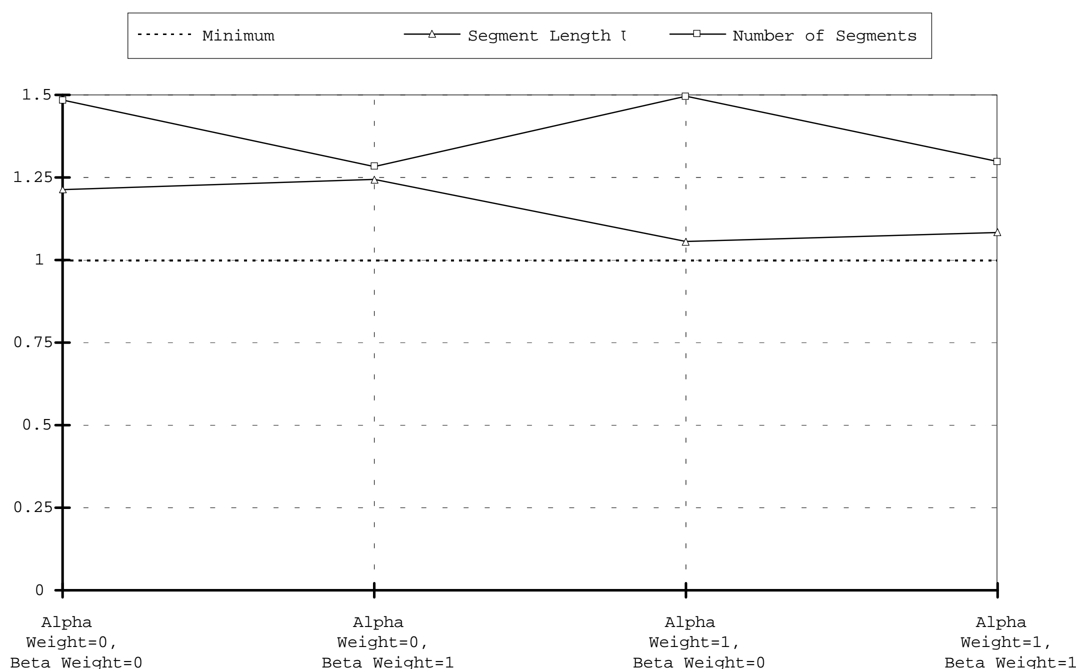

Figure 2.6. Segment allocation results.

2.4.3 Segment Allocation Impact on Routing Completion

For this experiment, the FPGA was configured with a Poisson distribution of segments as before, but rather than fixing the channel width to 20 the minimum channel width required to route 100% of the connections was measured.

The effect of wisely allocating segments upon minimum channel width is shown in Table 2.2. When either of the segment allocation costs are enabled, the number of tracks required generally increases as expected. Of particular interest, however, is the fact that C_β causes a marked increase of 22% in channel width requirements but C_α only causes a mild increase of 7%. This can be explained by understanding that C_α reduces the distance which a signal can depart from its global route at each turn, thereby helping to reduce the inflated channel density inherit in segmented channels. Additionally, when both costs are used 28% more tracks are needed to route the circuits.

This indicates that effective segment allocation with SEGA does not come without incurring a penalty on routability.

Circuit Name	w_α=0, w_β=0	w_α=1, w_β=0	w_α=0, w_β=1	w_α=1, w_β=1
9symml	11	12	12	13
alu2	14	13	15	16
alu4	15	16	20	21
apex7	13	13	15	17
bus_cntlD4	11	11	14	14
C880	15	14	17	18
cht	17	17	20	19
dmaD4	12	13	15	15
dram_fsmD4	12	14	14	15
EbnrD4	13	17	16	18
example2	17	18	22	23
k2	18	20	24	26
term1	11	12	12	13
too_large	12	15	16	16
vda	15	16	19	20
z03	15	16	19	19
TOTAL	**221**	**237**	**270**	**283**

Table 2.2. Impact of C_α and C_β on routing completion.

2.5 Conclusions

The evolution of FPGA routing architectures to include wire segments of various lengths presents a new and challenging issue to the detailed routing problem. The SEGA algorithm presented in this chapter is a simple yet proficient method capable of achieving excellent routing completion and intelligent segment allocation. Additionally, the tradeoff between good segment allocation and routability is shown to be roughly 20% better segment allocation at the expense of 30% more routing tracks. It would be prudent to explore SEGA's capabilities with these issues at a more in-depth level in the future by varying the channel segmentation and routing switch placement.

Chapter 3
Software Design

The software supporting the implementation of SEGA is a flexible, modular design intended to assist the development of routing architectures and algorithms. Code involving the FPGA architecture is kept distinctly separate from code implementing the routing algorithm.

The various modules will be described in detail below. The lowest level, the FPGA Module, will be described first, followed in layer order by the Path Module, the Route Module, and the Main Module. An additional Plotting Module, which aids visualization and debugging by displaying a graphical representation of the FPGA and its routing, is also briefly discussed.

3.1 Organizational Overview

The software system is organized into numerous layers, similar to a multi-layered operating system. The layered concept permits outer layers to access procedures of inner layers, either directly or indirectly, but it does not allow inner layers to access outer ones. This model allows the layers to run simultaneously in a concurrent environment without fear of internal deadlock; the resource allocation graph is guaranteed to be cycle-free because although a higher level can be waiting for a lower level, a lower level can never be waiting upon a higher level.

To further support concurrency, no global variables are permitted within a layer. Thus, all data structures must be *instantiated*. An instance of a structure is requested by a higher software layer, and successive references to the structure require an instance pointer. Since the lower layer does not contain any state information, it can act as a server for multiple higher level clients. The motivation for providing this level distinction is twofold: it clearly separates data structures to reduce module interdependencies and it enables the software layers to run in a distributed environment, on multiple networked workstations with separate memory spaces for example.

3.1.1 General Implementation Details

Layers communicate using a message passing interface which funnels all procedures through a single function with a large `switch{}` statement. By concentrating the interface to one location, it is easier to change the method of communication. Currently, the software runs as a single process and communication is done with a function call, but it could be easily modified to use multiple processes and remote procedure calls, pipes, or some other form of interprocess communication. To help simplify notation, macros have been defined which make the code more functional. For example, a normal message would be passed by:

```
rc = Fpga( pFpga, FPGA_SET_X, 42, NULL );
```

This has been simplified with a macro to just:

```
rc = FPGA_SET_X( pFpga, 42 );
```

Layers are encouraged to have its data types internalized so they cannot be seen by any other layer. This is implemented using two types of header files: a .h file and a _i.h file. The regular .h file is exported to other layers, so it contains message definitions and such. The _i.h file is considered an “internal” header file which defines local data structures not seen externally.

As often as possible, variable names follow the Hungarian notation style standard. This is a form of prefix notation which encodes the data type within the variable name. It is useful for keeping mental notes about the range of a value which can be stored, how many times a pointer can be dereferenced, and so forth. For example, some of the more common prefixes are “a” for array, “p” for pointer, and “i” for integer.

The lowest layer, the FPGA Module, manages all data structures dependent upon the FPGA architecture. Above that is the Path Module which houses cost functions, the coarse graphs, and detailed paths. Driving the Path Module is the Route Module where high-level routing decisions are made. Finally, at the highest level, is the Main Module. The Main Module acts an interface routine which translates user requirements from the UNIX command line into appropriate messages to the underlying software. Each of these modules and its related data structures is explained in detail below.

3.2 FPGA Module

The FPGA Module forms the kernel of this software. This module builds a data structure representation of the FPGA model depicted in Chapter 2, and provides access to that information through a programming interface. Conceptually, it is important to encapsulate all FPGA architecturally-dependent features within this module so that other modules need only consider the task of routing.

3.2.1 Creating and Initializing an FPGA Instance

To use the FPGA Module, an instance of an FPGA must first be created. Higher layers always refer to the instance using an instance pointer declared with the `PFPGA` data type. To create a usable FPGA, the following steps are required:

1. Call FPGA_CREATE to allocate memory for an instance structure.
2. Prepare the FPGA by sending relevant architectural information (more on this below).
3. Call FPGA_INIT to construct a representation of the FPGA.

An example of these steps can be found in Figure 3.1. After initialization, it is important that the architectural parameters **never** change; a more robust implementation should lock these parameters as read-only at this point.

```
#include "fpga.h"
#include "fpgamacros.h"

PFPGA   pFpga = NULL;           /* instance pointer */
RC      rc;                     /* return code */

pFpga = FPGA_CREATE( pFpga );   /* create the instance */

/* architectural features */
rc = FPGA_SET_X( pFpga, 12 );   /* set largest FPGA coordinates */
rc = FPGA_SET_Y( pFpga, 10 );
rc = FPGA_SET_W( pFpga,  8 );   /* set number of tracks per channel */

rc = FPGA_INIT( pFpga );        /* initialize the instance */

/* The instance is ready for use */
```

Figure 3.1. Code to create a simple 10 x 12 FPGA with 8 tracks per channel.

3.2.2 Defining the FPGA Routing Architecture

To analyze different routing architectures, control over the placement of switches and wire segments is important. The architecture can be summarized by three major features:

1. S block topology,
2. C block topology, and
3. Length and placement of wire segments.

Previous research has focused on determining good S and C block topologies [5][6], while wire segmentation is largely unexplored. This software allows customization of each these features by placing them in independent procedures. The current implementation constructs simple S and C blocks, yet provides a large degree of flexibility for wire segment length and placement. Each of these features will be discussed in detail below.

3.2.2.1 S Block Topology

The S block is implemented with a flexibility parameter of three. This means that each wire segment touching the S block can connect to a maximum of three other wire segments. Since the S block is four-sided, it is natural for a wire segment ending at one edge to connect to a wire segment at each of the other three edges. This allows a connection to pass through the S block or turn up or down.

A long wire segment which passes through the S block is only permitted to connect to one (long) or two other wire segments at the top and bottom edges. This limits fanout of the expanded graph near the root of the graph only (where the logic block pin connects to a C block), and is important to maintain control over the number of detailed paths examined by SEGA. Although this is a limitation of SEGA, the FPGA Module is written to support fanout at the S block. One must exercise extreme caution when allowing fanout, however, lest the large number of possibilities overwhelm the practical limitations of the hardware (growth is exponential!).

Exploration into S block topologies is a simple matter of modifying one routine, `FpgaConnectSegments` in `s_block.c`. The procedure loops through all S blocks in the FPGA and connects the appropriate wire segments together in each block. The data structures representing this are discussed further below.

Currently, the only option available to modify the S block is the ability to depopulate segments. This option prevents S block switches from being placed in the middle of the wire segment, i.e.

only the two S blocks at the ends of a wire segment contain switches. This option is hazardous because it can result in very few available turns at each S block.

3.2.2.2 C Block Topology

The design of an efficient C block is a difficult problem beyond the scope of this thesis. If a logic block pin is to have less than full connectivity to the channel (i.e. $F_C < W$), care must be taken to ensure that all logic block pins it might connect with have connectivity switches on as many of the same tracks as possible. Furthermore, the notion of pin equivalence[1] increases the complexity of this problem.

To eliminate problems which may occur due to inefficient C blocks, F_C is assumed to equal the channel width. Future work should implement a more practical alternative, possibly using this software to help determine a good topology.

The C blocks are built in a similar way S blocks are: `FpgaConnectLogicBlocks` in `c_block.c` loops through all C blocks and connects the pins available at each logic block to the C block. Also like the S block, switch depopulation is supported. This means after all the C blocks are connected, another routine can be used to disconnect all logic block pins which connect to the middle of a wire segment.

3.2.2.3 Wire Segmentation

The primary concern of the FPGA module is with the construction of the wire segment layout. An exact description of the segment layout is very difficult to achieve, so an approximate method is used.

FPGA wire segments are placed using a segment length generator to determine the next segment length. Each segment is placed end-over-end in a track until the track is full, then the next track is begun. When the current channel is filled, the next channel is started just like the first. This means that the segmentation is ultimately determined by the segment length generator.

The segment length generator is capable of generating lengths from a variety of probability distributions, including but not limited to Poisson, geometric, and binomial. Additionally, a channel can be divided into a number of groups of tracks, and each group can have its own

[1]When two pins are inputs to the same look-up table in a logic block, they are said to be *equivalent* because either pin may be selected for use provided the contents of the look-up table reflect the desired logic function.

distribution parameters. This degree of flexibility makes it possible to define a wide range of segmentation types.

3.2.3 Routing Architecture Data Structures

The S blocks, C blocks, and wire segments are all represented by a complex data construct, rooted by the instance structure given in Figure 3.2. The instance structure, `_fpga`, also stores all the architectural parameters used to build the FPGA, such as F_C and F_S. The data structures representing the FPGA are presented below.

```
struct _fpga {
    BOOL     bInit;            /* flag indicating an initialized FPGA */
    COORD    coordMax;         /* maximum size of the cell array */
    u_int    uiWh, uiWv;       /* tracks per [hv]channel (channel width) */
    u_int    uiFc, uiFs;       /* connectivity characteristics */
    u_int    uiPinsPerL;       /* pins per L block */

    u_int    uiCurrTrackGrp;   /* state info for segment generator */
    u_int    uiNumTrackGrps;   /* number of track groups below */
    TRACKGRP *aTrackGrps;      /* array of track groups, one elem per grp */
    POPINFO  PopInfo;          /* population info */

    CHANNELS hChannels;        /* the channel layouts themselves */
    CHANNELS vChannels;

    WIRESEGID wsIDNext;        /* the next available wire segment ID */
    PWIRESEG  pwsArray;        /* array holding *all* WIRESEG structures */
    int       iWSArraySize;

    BOOL      bConserveMemory;
};
typedef struct _fpga FPGA;
```

Figure 3.2. FPGA instance data structure.

3.2.3.1 FPGA Channels

The data structure representation of the FPGA is hierarchical, like the model. The FPGA is divided into channels, blocks (S or C), and tracks by a three-dimensional array. This is illustrated in Figure 3.3, where pointers are displayed as solid lines and array indices as dashed lines. To understand how the data structure is used, consider the following example.

Suppose it is necessary to determine which wire segments pass through the C block located at coordinates (2,1). Since the abscissa is even and the ordinate is odd, it can easily be seen that the C block lies within a horizontal channel. The horizontal channel number is then derived from the Y

component[2], and the X component can be used directly as a block offset into the channel. These values form the first two indices into the `hChannels` array; by forming a third index with a track number, a specific wire segment pointer can be obtained. The wire segment pointer, described in the next section, uniquely identifies a wire segment.

Although the above example references a C block, referencing an S block is similar. The key difference is that a single S block is accessible from both the horizontal and vertical routing channels, but only horizontal wire segments are present in the horizontal channels (the situation is similar for vertical segments and channels). Also, note that a track which has wire segments ending in the adjacent C blocks does not reference either wire segment, so the entry is denoted with a nil pointer. This implies that only long wire segments can be referenced at S blocks, and only at those S blocks which it passes directly through.

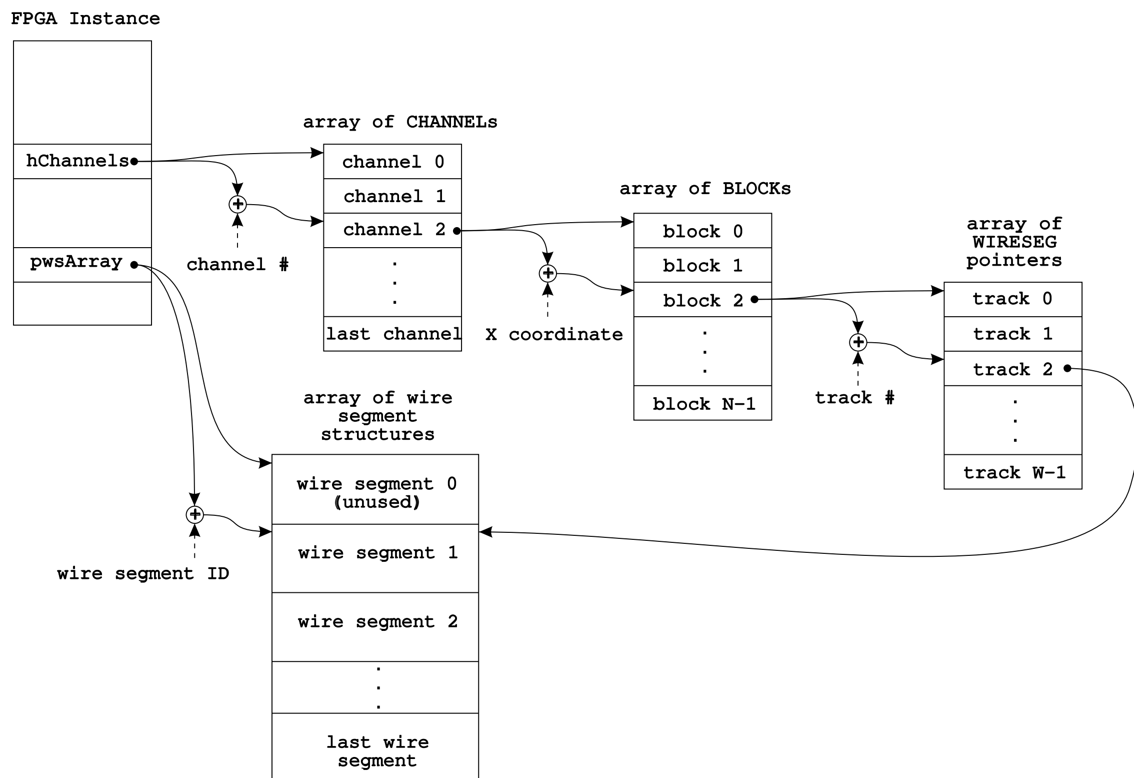

Figure 3.3. FPGA routing channel data structures.

[2]The macro `COORDtoCHAN` is provided for this purpose.

3.2.3.2 Wire Segments

As seen so far, the S and C blocks are merely represented as groups of wire segments without routing switches. This is because routing switches are associated with each wire segment, rather than the S or C block in which it lies. In so doing, the wire segment becomes an object that knows about which other objects it can connect to. Therefore, the S or C block topology is only needed to build the interrelationships between wire segments and pins (and not for routing).

Details about a wire segment are stored in a `_wireseg` structure, shown in Figure 3.4. The first field, `wsID`, stores an identifying integer unique to the wire segment. This number is important because other layers refer to wire segments by their wire segment ID[3]. To avoid the possible confusion of associating a wire segment ID of zero with a NULL pointer or a FALSE Boolean value, wire segment IDs are assigned starting at one. Also, increments of one are used so that a wire segment ID can be used directly as an array index. In particular, the wire segment structures themselves are stored in an array which can be indexed by a wire segment ID.

```
struct _wireseg {

    WIRESEGID wsID;          /* wire segment ID number */

    BOOL      bHoriz;        /* vert. or horiz. running segment */
    COORD     coordOrigin;   /* smallest possible coords */
    u_char    ucLength;      /* length of segment, in C blocks */
    u_int     uiTrack;       /* track # of segment */

    /* a 2-d array of connections
     * first index is block (S or C block)
     * second index cycles through all possible
     * connections at that block
     */
    CONNECTION    **ppConnection;

};
typedef struct _wireseg   WIRESEG;
```

Figure 3.4. Wire segment data structure.

The next few fields of the `_wireseg` structure are fairly self-explanatory. If the wire segment lies in a horizontal channel, `bHoriz` is true. The coordinates of the leftmost/lowermost end of the wire segment (always a C block) are stored in `coordOrigin`, and its length (in units

[3]In particular, the Path Module uses the wire segment ID to index an array.

of C blocks) is in `ucLength`. Finally, the track number in which the wire segment lies can be found in `uiTrack`.

3.2.3.3 Wire Segment Connections

The most important field of the wire segment structure is `ppConnection`. This purpose of this field is to describe which pins and wire segments can be connected to this wire segment. It consists of a ragged two-dimensional array of `CONNECTION` constructs, where the data type `CONNECTION` identifies either a pin or a wire segment using a C language union.

The first dimension of `ppConnection` refers to which C or S block the wire segment connects in, and the second dimension forms the array of connections that can be made within that block. The size of the first dimension is always equal to $2 \times ucLength + 1$, since a wire segment goes through *ucLength* C blocks and touches upon *ucLength* + 1 S blocks (including those at the segment's endpoints). The blocks are ordered consecutively in the array such that the smallest coordinate is at the zeroth index, and S blocks are always at an even index. For example, the bold wire segment in Figure 3.5 can have connections in the S blocks at (1,1) and (3,1) as well as the C block at (2,1). Here, `ppConnection[0]` would refer to connections possible in the S block at (1,1), `ppConnection[1]` would refer to connections at (2,1), and `ppConnection[2]` would refer to connections at (3,1). If no connection is possible at a block, or if the block does not exist, the `ppConnection[i]` value may be NULL. This always occurs for wire segments which reach the edge of the FPGA, since the end of such a wire segment does not have an S block to connect in. For example, the horizontal wire segment immediately left of the bold one in Figure 3.5 reaches the left edge of the FPGA, so its `ppConnection[0]` is NULL and its `ppConnection[1]` refers to connections in the C block at (0,1). Because of this, it is prudent to check the result of the first index for a non-NULL value before forming the second index.

The second `ppConnection` array dimension indexes each connection which can be made within the block. It can be of arbitrary size — thus, the `ppConnection` array is ragged — so the last element is indicated by a NULL entry. Furthermore, there is no particular order given to the entries in this part of the array.

When a particular connection result is obtained via `ppConnection[i][j]`, it can be either a *wire segment pointer* or a *terminal*. A wire segment pointer is returned when `i` is even (i.e. at an S block), and it points directly to the `_wireseg` structure of a connectable wire segment. Otherwise, when `i` is odd, a terminal is returned.

Terminals are specially computed integers which uniquely identify logic block pins. While two different logic blocks may both have a pin zero, the terminal numbers of each pin is different. Thus, a single terminal number can indicate a particular logic block and pin. For ease of debugging only, all terminal numbers are negative. This helps differentiate them from wire segment pointers, which are most often located at lower memory addresses and so appear as positive integers to most computers.

Rather than using terminal numbers to indicate a logic block pin, a pointer to some data type could be used. This would allow pins to be treated as objects, like wire segments, and permit greater flexibility in their use. This may be useful if a router is programmed to take advantage of pin equivalence, where alternative pins would be useful information stored in such a structure.

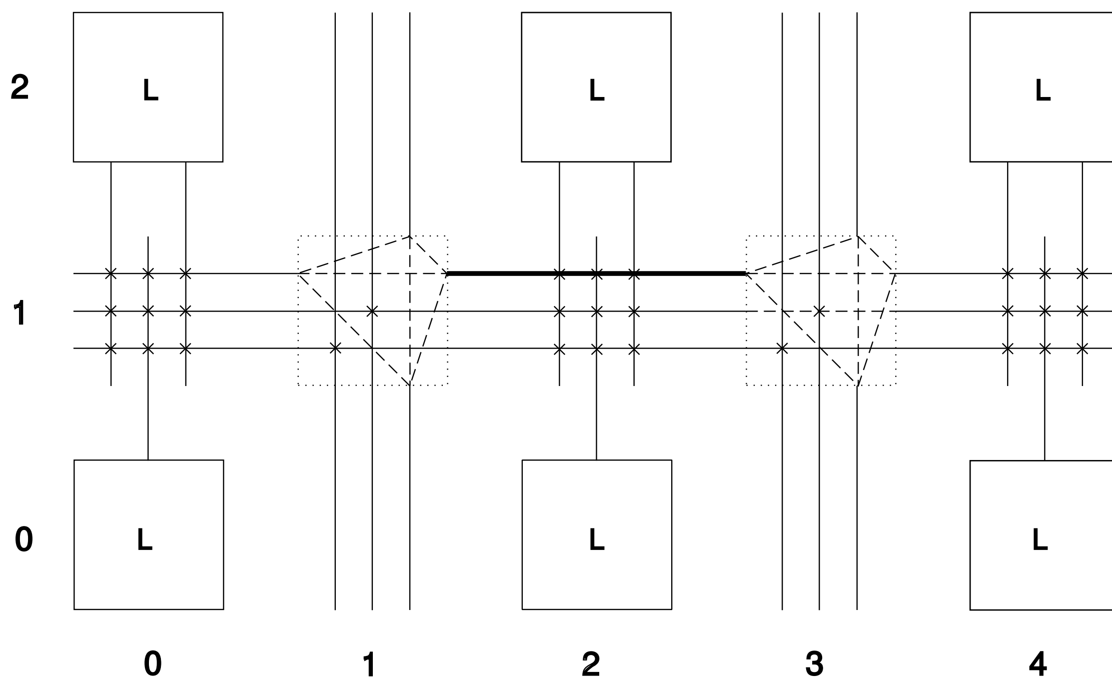

Figure 3.5. Wire segment connectivity in a small FPGA.

3.2.4 Using the FPGA

The FPGA data structures described above are not visible to other software layers, so a programming interface for routing is required.

3.2.4.1 FPGA Channel Expansion Using Lines

The FPGA is modeled as a number of horizontal and vertical channels through which a signal may travel. Upper software layers attempts to create detailed routes through the FPGA from a coarse graph, i.e. a sequence of channel segments. These ideas are merged here by the intermediary concept of *lines*. A line is defined to be a partial detailed route, consisting of a list of wire segments within a single channel. Therefore, a detailed route would consist of one or more lines which are connectable at S blocks.

Lines are created by breaking down a coarse graph into a number of straight sections and having the FPGA Module consider each straight section independently. For each section, the FPGA Module enumerates all possible paths through the channel and assigns each to a line. If there is no S block fanout, there are always W lines in each straight section. Note that S block fanout can cause a great number of lines to be created for any section consisting of more than 2 S blocks, so caution should be used under these circumstances. Normally, however, the problem is not present because fanout is not used.

3.2.4.2 Connecting Lines Together With Reductions

Once a coarse graph is subdivided into straight sections and each section is expanded into its respective lines, lines from different sections must be connected together to form each detailed route. While the process of actually connecting the lines is primarily the responsibility of the Path Module, the FPGA Module must determine whether two lines can be connected together. This is done by reducing the set of lines created during expansion into a subset; a line becomes a member of the subset if it is capable of connecting to a user-specified wire segment or terminal.

For efficiency, upper layers are permitted to “peek” into the reduced line set. The message FPGA_GET_LINE_ARRAY returns a null-terminated array. Each element of this array is a reduced line set consisting of a null-terminated array of wire segment IDs. The wire segment IDs at the ends of a line can be used to reduce another set of lines. This process continues until a sequence of lines has been found which connects to the logic block pins, and a path can be formed.

3.3 Path Module

The Path Module makes only a few basic assumptions about the FPGA architecture. It assumes the FPGA conforms to the symmetrical model, and that straight wire segments of various lengths fill the channels.

The purpose of the Path Module is to manage all details pertaining to the coarse graphs and detailed paths. The objective is to provide a flexible and clean programming interface which enables different detailed routing algorithms to be implemented by changing the upper software layers. It is recognized that the first implementation of this module cannot anticipate every possible feature desired for different algorithms, but it is hoped that over time this module will develop a full-featured interface capable of serving a variety of algorithms. Therefore, this implementation focuses upon the features necessary to implement the SEGA algorithm, but with general routing algorithms in mind.

3.3.1 Creating and Initializing a Netlist Instance

Because this module maintains both paths and graphs forming a type of netlist, a Path Module instance is also called a netlist instance. A portion of the netlist structure is shown in Figure 3.6; trivial fields have been deleted for clarity. Preparing the instance is similar to that for an FPGA: an instance is created, parameters are set, and then an initialization message is sent.

Before the initialization message, two parameters must be set. First, a file pointer to the input netlist must be set. Second, the FPGA instance pointer being used must be sent to the path instance **before** the FPGA instance is initialized. This is because the Path Module must be able set the FPGA dimensions required by the input netlist. Once these parameters are fixed, the netlist instance can be initialized.

Initializing the netlist instance inputs a coarse graph, *G*, from the file pointer previously set. The format of the input file consists of one or more nets outlining a coarse graph. A single net looks like:

```
<net_name> <net_number>
<x_1> <y_1> 1 <pin1>
<x_2> <y_2> 1
...
<x_n-1> <y_n-1> 1
<x_n> <y_n> 0 <pin2>
```

where <net_name> is a whitespace-free text string describing the net, <net_number> is an integer identifying the electrically equivalent nets, the <x_i> and <y_i> values are integers indicating FPGA coordinates[4], and <pin> gives the logic block pin number. The 1 or 0 in the third column indicates whether this is the last <x_n> <y_n> pair and the last value, <pin2>, should be read.

```
typedef struct _netlist {

    PFPGA      pFpga;               /* the architecture for this netlist */
    FILE      *fp_netlist;

    u_int      uiSizeOfGraphArray; /* size of graphs array */
    GRAPH   **ppGraphArray;        /* the graphs in the netlist */

    u_int      uiSizeOfPathArray;  /* size of paths array */
    PATH    **ppPathArray;         /* the paths in the netlist */

    PPATH    **pppPath_WS;     /* Special arrays:  zeroth element is size
*/
    GRAPH_WS **ppGraph_WS;     /* of array, other elements are data */

    u_int      uiNumWS;        /* number of wire segments in fpga */
    u_char    *pucWSLength;    /* array, zeroth elem garbage */

    STACK      lineStack;      /* stack of lines along a path */
    STACK      pathStack;      /* stack of lines along a path */

} NETLIST;
```

Figure 3.6. A portion of the netlist structure.

As mentioned in section 3.2.4.1, the G must be divided into a sequence of straight sections to expand into lines. These straight sections are computed at initialization time while the netlist is being read from the input file. The result is called a coarse graph with express edges, G_X, illustrated in Figure 3.7. Express edges refer to the long straight sections which quickly jump from turn to turn in the coarse route. Obviously, express edges allow quick identification of sections which can be given to the FPGA Module for expansion.

[4]The sequence of coordinates must describe a valid path of adjacent blocks. Minimally, it will be an L-C-L block sequence. It will always be of the form L-C-(S-C)-L, where (S-C) is repeated 0 or more times.

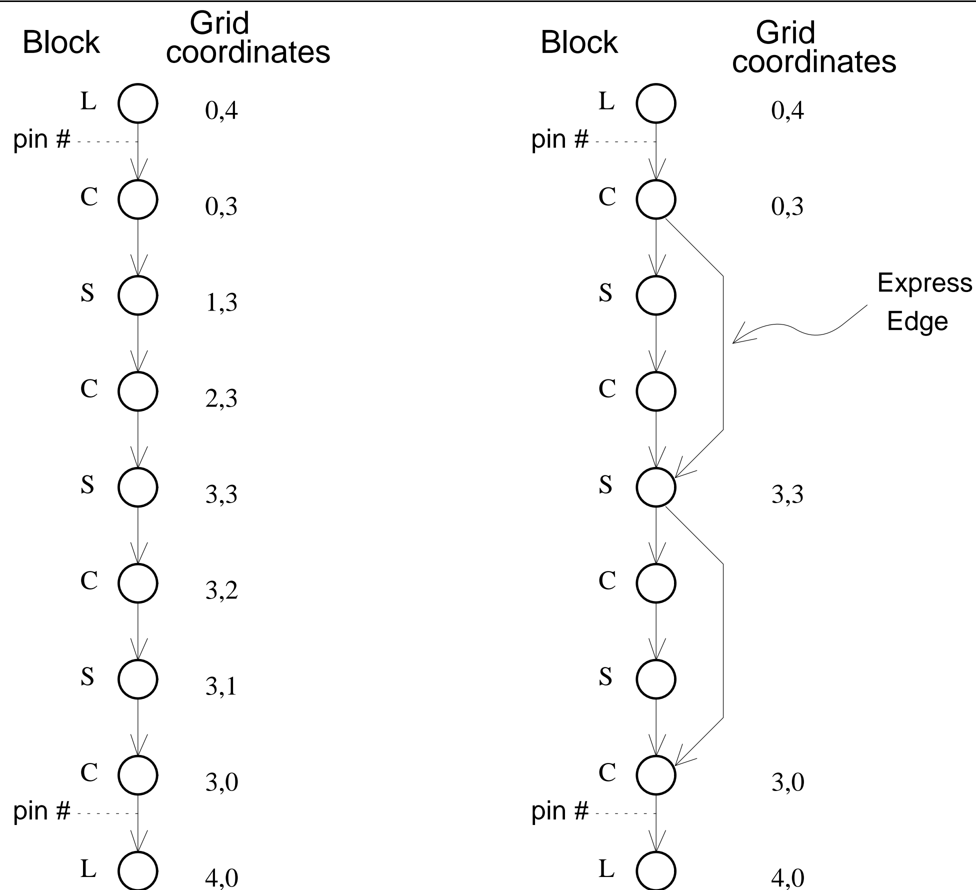

Figure 3.7. (a) A coarse graph and (b) the coarse graph with express edges.

3.3.2 Enumerating Graphs and Paths

Once all of the graphs have been built, access to them is provided through the use of enumerations. An enumeration forms an ordered set of unique items, such as graphs, which can be cycled through item by item. Each item can have a number of operations performed upon it, but these will be described in a later section. For now, the use of enumerations is described.

An enumeration is started with the message PATH_ENUM_BEGIN and a parameter indicating the types of objects to be enumerated. Currently, the objects which can be enumerated include graphs, paths, essential paths, and non-essential paths. When an enumeration is no longer needed, the PATH_ENUM_END message is used to free up allocated memory.

The next (possibly first) item in an enumeration can be obtained with the PATH_ENUM_GET_NEXT message. If there are no more items left, a NULL is returned.

When a enumeration of paths is begun, an additional message can be sent to it. This message, PATH_ENUM_REORDER, finds the lowest cost path (costs are discussed in the next section) and places it first in the enumeration. The search for the lowest cost path is necessary to implement most types of routing algorithms, where the lowest cost path is chosen in preference to others.

Enumerations form a consistent method of hiding the Path Module data structures from other modules, but it is an inefficient method for practical use in a commercial router. Optimizations, such as the use of a heap structures, can be implemented if execution time of the algorithm gets too large. However, such optimizations can complicate the organization of the Path Module significantly, since the dynamic nature of the cost functions and weights force many heap updates.

3.3.3 Cost Function

The reordering of enumerated paths is based upon a cost function. Ideally, the cost function would be completely specifiable by a routing algorithm developer at the Route Module level. Unfortunately, this is not feasible because most cost functions require information only present in the Path Module. To compromise, multiple cost functions are summed in the Path Module using an assignable weighting scheme.

Currently, four different cost functions have been defined corresponding to the four costs used by SEGA. They can be used individually by fixing all but one weight to zero, or collectively by providing more than one non-zero weight. The message-passing organization made it difficult to pass floating point numbers as a parameter, so weights are currently assigned positive integer values. This does not impose any serious limitations since weights are purely relative to each other and equivalent integers can usually be found by multiplying all weights by one sufficiently large integer. For example, rather than using the weights {1.1, 0.05, 0, 0}, the equivalent integer weights of {110, 5, 0, 0} can be used.

It is important that each individual cost function only return a value in the range of [0,1]. If it returns a value outside of that range, then that cost function may have an unfair advantage (or disadvantage) relative to the other cost functions. The C_f cost used by SEGA does not naturally fall within the [0,1] range, so it is necessary to continuously rescale all C_f costs. The rescaling is done by linearly mapping all C_f costs such that the smallest maps to zero and the largest maps to one. By observing this fairness policy, the importance of weight relativity is maintained for the user.

3.3.4 Data Structures

3.3.4.1 Graph Structure

The origin of all of the coarse graphs is contained in the netlist by `ppGraphArray`, an array of graph pointers. A graph pointer dereferences a structure containing a number of properties, such as the name and number of the net it belongs to, a linked list of channel segments (with express edges), and an array of path structure pointers. This last field lists all valid paths remaining which can be used to form the required connection.

3.3.4.2 Path Structure

A path structure identifies a NULL-terminated list of wire segment IDs which can be connected together to route its parent graph. Additionally, it contains all of the computed costs, a pointer to its parent graph, and results from the cost functions. Although storing results from each of the costs separately consumes more memory, it is necessary so that the costs can be considered separately as needed.

3.3.4.3 Wire Segment Lengths

To assist the computation of costs, the netlist contains an array of wire segment lengths, `pucWSLength`. Rather than query for a segment's length each time it is referenced in a path, all segments are queried once and the length is stored in an array. This array, indexed by a wire segment ID, forms a type of cache for the segment length information already obtainable from the FPGA Module.

3.3.4.4 Path Interdependencies

A single wire segment can be considered for use by many paths, but it can only carry one signal. The `ppGraph_WS` and `pppPath_WS` fields of the netlist structure cross-reference the numerous paths which compete for wire segments. Both of the fields are arrays long enough to be indexed by a wire segment ID. By doing so, all paths dependent upon a single wire segment can be found.

Although both fields are first indexed by a wire segment ID, their second index is slightly different. The second index of pppPath_WS cycles through a list of pointers to path structures which depend upon the current wire segment. To allow for dynamic sizing, the zeroth element of the second dimension indicates the length of the rest of the array (excluding the zeroth one) because some of the array elements themselves may be nil. Like the second array dimension of pppPath_WS, the zeroth element of ppGraph_WS stores the length of the rest of the array so it may be resized as needed[5]. The difference is that the second index to ppGraph_WS references a small structure rather than a path pointer. The structure stores a graph structure pointer and a count indicating how many of the graph's paths depend upon the current wire segment. The use of this information will become apparent with the following example.

Suppose there are two graphs to be routed, G_1 and G_2. The first graph has three possible paths, $P_{1,1}$, $P_{1,2}$ and $P_{1,3}$, while the second graph has two paths, $P_{2,1}$ and $P_{2,2}$. Furthermore, assume the first two paths in G_1 and the second path in G_2 require use of the same wire segment. For this configuration, the entries in pppPath_WS and ppGraph_WS at that wire segment ID may appear as in Figure 3.8. If the router was considering assigning $P_{2,2}$ to G_2, it could observe from ppGraph_WS that two of G_1's paths rely upon the same wire segment, implying G_1 would be left with only $3 - 2 = 1$ remaining alternative. If it proceeded to assign $P_{2,2}$ to G_2, it could delete $P_{1,1}$ and $P_{1,2}$ from G_1's possible paths by checking pppPath_WS. Of course, deleting also requires keeping these cross reference arrays current by clearing the appropriate entries.

	Path
0	7
1	NULL
2	P(2,2)
3	P(1,1)
4	NULL
5	NULL
6	P(1,2)
7	NULL

pppPath_WS array element

	Graph	Count
0	4	4
1	G(1)	2
2	G(2)	1
3	NULL	0
4	NULL	0

ppGraph_WS array element

Figure 3.8. Example elements in the cross referencing arrays.

[5]Because elements of the array are structures and not just an integer, both elements of the structure hold this length value.

The example can be slightly more complicated if G_1 and G_2 can share segments because they belong to the same net. In this case, G_1's alternatives would still be three and the two paths should not be removed from the pppPath_WS array when $P_{2,2}$ is allocated to G_2. This special scenario is automatically handled by the cross-referencing code.

While the primary purpose of the cross referencing arrays is to identify paths which can no longer be considered due to conflicts, the above example shows how the number of alternative paths for a graph can be found and one potential difficulty. Another pitfall to avoid is that paths listed together in the same pppPath_WS array element is a necessary, but not sufficient, condition for conflict. That is, if two paths are not listed together in one element, they may still conflict at another wire segment. This is important when computing the number of alternatives, because the method shown above (and used in the code) always makes optimistic assumptions, namely that fewer conflicts occur and more alternatives exist.

3.3.5 Operations on Graphs and Paths

Any path or graph obtained via an enumeration can have an operator applied to it. Current operators allow for a graph to be expanded into its detailed paths, a path cost to be recomputed, and a path to be marked. These operations enable upper software layers to have control over the algorithm used for routing, while the gory details are embedded within the Path Module.

While the expanding a graph and computing a cost is straight-forward, marking a path is much more difficult. It involves two major steps: removing sibling paths of the same graph, since they are redundant, and removing any other *conflicting* paths which depend upon a wire segment just allocated to the path being marked. Sibling paths are easily obtained and removed, since each path knows its parent graph, and the parent graph knows all of its children. However, conflicting paths are more difficult to detect. Fortunately, the data structure used to compute the C_f cost is exactly what is needed to find conflicting paths. Whenever a path which is removed, the costs of its siblings is recomputed since both the C_f and C_c costs have changed. Thus, the marking of a path requires updating a number of data structures and related costs.

3.3.6 Correctness

After all paths have been assigned route to a graph or discarded, it is important to assert that the assigned routes are all credible and different electrical nets are disjoint. A routine called PathSanity is written for this purpose. This provides a certain level of confidence that the

chosen detailed routes can be programmed into a real FPGA and work correctly. The PathSanity procedure is also used to collect some routing statistics, such as total wire segment length used, to provide a summary of the routers segment allocation performance.

The `PathSanity` procedure follows the pseudo code presented in Figure 3.9. The sanity check is partitioned into three smaller checks, which collectively imply that the router achieved its purpose. The first check ensures every routed graph is assigned only one path, since more would be wasteful. The second check guarantees only one electrical net is present on every allocated wire segment. Lastly, and most importantly, the paths are double-checked to assert that the relevant wire segments and logic block pins can be connected together via routing switches.

Once the sanity check is complete, there is a high degree of confidence that a interconnect of the FPGA would be programmed correctly to meet the input specification.

```
for each graph
    if the graph could not be routed
        assert no valid paths remain
    else
        assert only one routed path exists
    end if
end for

for each wire segment in the FPGA
    if the wire segment is allocated
        assert wire segment is used only by graphs of same net
    end if
end for

for each routed path
    assert that the path is valid
end for
```

Figure 3.9. Pseudo code used to prove correctness of the detailed route.

3.4 Route Module

The purpose of the Route Module is to drive the Path and FPGA Modules to route a circuit with a routing algorithm. It is a good example of how to use the lower software layers at an application level. The actual code should read like pseudo code, placing the onus on the lower levels to implement the functionality required for each pseudo code statement.

Because the Route Module is designed to be a fully user-programmed layer, there are no actual requirements. Rather, the Route Module represents an intended level of organization for the programmer to follow.

Currently, the Route Module implements the SEGA routing algorithm described in Chapter 2. The pseudo code which implements SEGA is shown in Figure 3.10. In the pseudo code, the ⇐ symbol is used to denote the use of an enumeration provided by the Path Module. For comparative purposes, the actual source code for the main Route Module routine is listed in Appendix A.

```
/* Phase 1 */
F ⇐ set of all coarse graphs

for each graph in F
    expand the graph into its detailed paths, adding
        the paths to the path list
end for

/* Phase 2 */
while path list is not empty

    if essential paths exist
        F ⇐ all essential paths
    else
        F ⇐ all other paths
    end if

    sort F
    P ⇐ first path from F (i.e. lowest cost path)

    mark P as detailed route, removing conflicting and
        redundant paths from path list
    draw P on screen

end while
```

Figure 3.10. Route Module pseudo-code implementing the SEGA algorithm.

3.5 Main Module

Like the Route Module, the Main Module is only a suggested level of organization for a router programmer to follow. This layer performs all of the initialization and setup required for routing.

For the SEGA implementation, the Main Module interprets command line switches given by the user and sends messages with their intended meaning to the lower modules. The input netlist is opened, Path and FPGA instances are initialized and created, and control is passed to the Route Module. This module also prints error messages corresponding to bad return codes from the lower layers.

3.6 Plot Module

The Plot Module serves as an invaluable debugging tool for this software. By providing an interactive X windows display of the FPGA, it gives instant visualization to the FPGA architecture, detailed paths, and even the routing process.

Unfortunately, the Plot Module does not conform to the same philosophy with which the FPGA and Path Modules were written; it is not yet a formal part of the software. Future work could involve rewriting parts of the plotting code to separate it from its current FPGA Module data structure dependencies and expanding upon the interactive capabilities of the interface. For example, highlighting an entire net would immediately show the amount of segment sharing taking place, indicating the effectiveness of an algorithm to recombine two-point connections.

Chapter 4
Conclusions and Future Work

Recent FPGA research has focused upon making FPGAs faster and capable of holding more logic. However, the exploration of good routing architectures is being done with outdated CAD tools which are incapable of fully utilizing the routing resources. The need for new, flexible routing software to support FPGA architectural research is apparent, as well as the need for new routing algorithms to take advantage of new features.

In support of these needs, the SEGA algorithm was developed and a modular, adaptable software base was designed. Experimental results reveal that SEGA is an algorithm proficient at both routing completion and segment allocation. The implementation of SEGA is designed to be generic so the routing algorithm can be modified and the FPGA architecture enhanced. Moreover, the modular design of the software clearly differentiates FPGA architecture-dependent code from architecture-independent code.

The possibilities for future work are nearly endless. The C_α and C_β costs could be unified into a single cost which more accurately represents the delay imposed upon a connection. As well, two-point nets should be recombined into complete multi-point nets so delays can be better modeled. FPGA architectural research can relentlessly improve upon the S and C block topologies, and algorithm development will no doubt expand upon the abilities of the Path Module. The speed of the Path Module needs to be improved; this can be done by providing better enumeration data structures and algorithms, such as heaps and up-trees. The Plot Module should be adapted to conform with the ideology present in the rest of the software, and the capabilities of the user interface could be improved to further aid diagnostics. Global routing may even be added to the list of responsibilities. The software described in this thesis forms a basis for a whole realm of new FPGA research.

References

[1] Stephen D. Brown, Robert J. Francis, Jonathan Rose and Zvonko G. Vranesic, "Field-Programmable Gate Arrays," Kluwer Academic Publishers, 222 pages, 1992.

[2] G. Lemieux and S. Brown, "A Detailed Routing Algorithm for Allocating Wire Segments in Field-Programmable Gate Arrays," *Proc. 1993 ACM Physical Design Workshop*, April 1993.

[3] S. Brown, J. Rose and Z. Vranesic, "A Detailed Router for Field-Programmable Gate Arrays," *Proc. IEEE International Conference on Computer Aided Design*, pp. 382-385, Nov. 1990.

[4] S. Brown, J. Rose and Z. Vranesic, "A Detailed Router for Field-Programmable Gate Arrays," *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, Vol. 11, No. 5, pp. 620-628, May 1992.

[5] J. Rose and S. Brown, "The Effect of Switch Box Flexibility on Routability of Field-Programmable Gate Arrays," *Proc. 1990 Custom Integrated Circuits Conference*, pp. 27.5.1-27.5.4, May 1990.

[6] J. Rose and S. Brown, "Flexibility of Interconnection Structures in Field-Programmable Gate Arrays," *IEEE Journal of Solid State Circuits*, Vol. 26 No. 3, pp. 277-282, March 1991.

[7] K. Roy and M. Mehendale, "Optimization of Channel Segmentation for Channeled Architecture FPGAs," *Proc. 1992 Custom Integrated Circuits Conference*, pp.4.4.1-4.4.4, May 1992.

[8] J. Greene, V. Roychowdhury, S. Kaptanoglu, and A. El Gamal, "Segmented Channel Routing," *Proc. 27th Design Automation Conference*, pp. 567-572, June 1990.

Appendix A

Route Module Code

```
#include <stdio.h>

#include "mytypes.h"
#include "error.h"

#include "stack.h"
#include "fpga.h"
#include "path.h"
#include "route.h"

#include "path_i.h"

extern char PlotFlag;    /* program globals */
extern int verboselevel;


/*
 * At this point, we assume the following has occurred:
 *
 *    1) all coarse graphs are defined (read in from netlist)
 *    2) an FPGA architecture of appropriate dims has been instantiated
 *
 * What remains to be done is:
 *
 *    1) for each coarse graph Gi:
 *        a) expand the graph
 *        b) for each path in Gi:
 *            i) find Alpha, Beta
 *
 *    2) for each path Pi:
 *        a) initialize cost Cf
 *        b) place Pi in "to be routed" bucket B
 *
 *    3) while B is not empty
 *        a) sort B:
 *            i) paths in an essential graph are given priority
 *           ii) sort by cost as secondary key
 *
 */

RC RouteCct( PNETLIST pNetList )
{
           BOOL    bEssential;
           PPATH   P,Q;
           PGRAPH  G;
           PFOREST F;
           int    numgraphs, numpaths, totalpaths, numessential;

           RC rc = RC_OK;

           numgraphs = numpaths = totalpaths = numessential = 0;

           Printf( 1, "Routing...\n" );
```

```
        /* F is a forest; a collection of graphs G
         * Expand the forest
         */
        Printf( 1, "Expanding graphs...\n" );
        F = (PFOREST)Path( pNetList, PATH_ENUM_BEGIN, ENUM_GRAPHS, NULL );
        if( F == NULL ) {
                rc = RC_ENUM_ERROR;
                goto cleanup;
        }

        while( (G = (PGRAPH)Path( pNetList, PATH_ENUM_GET_NEXT, (MP)F, NULL )) != NULL ) {
                numgraphs++;
                Printf( 3, "Expanding graph %8s (0x%x) ", G->pucNetName, (MP)G ); fflush( stdout );

                rc = Path( pNetList, PATH_EXPAND, (MP)G, (MP)(&numpaths) );
                if( rc != RC_OK ) goto cleanup;

                Printf( 3, "into %d paths.\n", numpaths ); fflush( stdout );
                totalpaths += numpaths;

                if( numpaths == 0 ); /* graph has no alternatives in architecture */
        }
        Path( pNetList, PATH_ENUM_END, (MP)F, NULL );

        Printf( 1, "Expanded %d graphs into %d paths, average %f paths per graph.\n",
                numgraphs, totalpaths, (float)totalpaths/(float)numgraphs );


#if 0
        /* Inform the path module of the weight assignments being used */
        Path( pNetList, PATH_SET_W1, (MP)W1, NULL );
        Path( pNetList, PATH_SET_W2, (MP)W2, NULL );
        Path( pNetList, PATH_SET_W3, (MP)W3, NULL );
        Path( pNetList, PATH_SET_W4, (MP)W4, NULL );
#endif

        /* Inform the path module of the method to be used for scaling
         * the Cf cost to within [0,1]
         */

        /* Initialize all of the costs correctly */
        Printf( 1, "Initializing costs...\n" );
        Path( pNetList, PATH_INIT_COSTS, NULL, NULL );



        /* F is a forest; a collection of paths P
         * Route the darn things!
         */

        Printf( 1, "Selecting paths...\n" );
        while( 1 ) {

                if(  F = (PFOREST)Path( pNetList, PATH_ENUM_BEGIN,
                                        ENUM_PATHS_ESSENTIAL, NULL ) ) {
                        bEssential = TRUE;
                } else {
                        F = (PFOREST)Path( pNetList, PATH_ENUM_BEGIN,
                                ENUM_PATHS_OTHER, NULL     );
                        bEssential = FALSE;
                        if( F == NULL ) {
                                Printf( 1, "\nNo more paths!\n" );
                                break;
                        }
                }

                /* Find and mark the lowest cost connection.
                 * Marking automatically updates all of
                 * the changed costs, and removes unnecessary
                 * paths from memory.
                 */
                Path( pNetList, PATH_ENUM_REORDER, (MP)F, NULL );
```

```
			P = (PPATH)Path( pNetList, PATH_ENUM_GET_NEXT, (MP)F, NULL );
			if( P == NULL ) {
				rc = RC_ENUM_ERROR;
				goto cleanup;
			}

			if( verboselevel > 2 ) {
				while( Q = (PPATH)Path( pNetList, PATH_ENUM_GET_NEXT, (MP)F, NULL ) ) {
					Printf( 5, "Considering graph %8s with path 0x%x, cost %f "
						"(a=%f b=%f c=%f sc=%f)...\n",
						Q->pGraph->pucNetName, Q, Q->cost.fC, Q->cost.fAlpha,
						Q->cost.fBeta, Q->cost.fCf, Q->cost.fSCf );
				}
			}

			if( bEssential ) {
				Printf( 2, "Marking essential graph %s with path 0x%x, cost %f "
						"(a=%f b=%f c=%f sc=%f)...\n",
						P->pGraph->pucNetName, P, P->cost.fC,
						P->cost.fAlpha, P->cost.fBeta, P->cost.fCf, P->cost.fSCf );
				numessential++;
			} else {
				Printf( 3, "Marking graph %s with path 0x%x, cost %f "
					"(a=%f b=%f c=%f sc=%f)...\n",
					P->pGraph->pucNetName, P, P->cost.fC,
					P->cost.fAlpha, P->cost.fBeta, P->cost.fCf, P->cost.fSCf );
			}

			if( PlotFlag ) {
				DrawPath( P->pwsID, P->pGraph->termHead, P->pGraph->termTail );
			}

			Path( pNetList, PATH_MARK, (MP)P, NULL );

			Path( pNetList, PATH_ENUM_END, (MP)F, NULL );
		}

		/* No more paths, routing done */
		/* Print out some statistics */
		Printf( 1, "Routed %d essential graphs.\n", numessential );
		Printf( 1, "Routed %d out of %d graphs, %f percent.\n",
			pNetList->uiGraphsRouted, pNetList->uiNumberOfGraphs,
			100.0 * (float)pNetList->uiGraphsRouted /
			(float)pNetList->uiNumberOfGraphs );
		if( pNetList->uiGraphsNotRouted ) {
			Printf( 1, "Failed to route %d out of %d graphs, %f percent.\n",
				pNetList->uiGraphsNotRouted, pNetList->uiNumberOfGraphs,
				100.0 * (float)pNetList->uiGraphsNotRouted /
				(float)pNetList->uiNumberOfGraphs );
			rc = RC_FAILED_TO_ROUTE;
		}
		Printf( 1, "Done routing.\n\n" );

cleanup:
		return rc;
}
```