A Genetic Algorithm-Based Place-and-Route Compiler For A Run-time Reconfigurable Computing System

By

Brian C. Kahne

Thesis submitted to the faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science

in

Electrical Engineering

APPROVED:

_____
Peter M. Athanas, Chairman

_____
James R. Armstrong

_____
Charles E. Nunnally

May, 1997

Blacksburg, Virginia

Keywords: Genetic Algorithm, Configurable Computing Wormhole, Run-time Reconfiguration, Routing, Placement

A Genetic Algorithm-Based Place-and-Route Compiler For A Run-time Reconfigurable Computing System

By

Brian C. Kahne

Committee Chairman:

Peter M. Athanas

Bradley Department of Electrical Engineering

(Abstract)

Configurable Computing is a technology which attempts to increase computational power by customizing the computational platform to the specific problem at hand. An experimental computing model known as *wormhole run-time reconfiguration* allows for partial reconfiguration and is highly scalable. In this approach, configuration information and data are grouped together in a computing unit called a stream, which can tunnel through the chip creating a series of interconnected pipelines.

The *Colt/Stallion* project at Virginia Tech implements this computing model into integrated circuits. In order to create applications for this platform, a compiler is needed which can convert a human readable description of an algorithm into the sequences of configuration information understood by the chip itself. This thesis covers two compilers which perform this task.

The first compiler, *Tier1*, requires a programmer to explicitly describe placement and routing inside of the chip. This could be considered equivalent to an assembler for a traditional microprocessor. The second compiler, *Tier2*, allows the user to express a problem as a dataflow graph. Actual placing and routing of this graph onto the physical hardware is taken care of through the use of a genetic algorithm.

A description of the two languages is presented, followed by example applications. In addition, experimental results are included which examine the behavior of the genetic algorithm and how alterations to various genetic operator probabilities affects performance.

Table of Contents

List of Figures

List of Charts

Chapter 1: Introduction

Reconfigurable computing systems have been the subject of intense research as a means of solving numerically intensive computations in hardware without sacrificing applicability to a wide variety of problems [Bit97a]. The traditional design approach has been to use RAM-based FPGAs as the basic building block. Such architectures allow for extremely flexible systems which have been shown to be able to operate at high clock speeds [Her96] and to solve problems with inherently high bandwidths [Her97]. However, these devices were not designed for computation. These devices were primarily designed for glue-logic applications and were optimized for slowly reconfiguring bit-level operations. Thus for economic reasons, there are some significant costs: either the device must be placed into a programming mode and completely reconfigured, or else selective reprogramming is allowed and random access to configuration cells uses up a large amount of silicon routing resources. The switching of these signals can create a heavy power demand. In addition, Most FPGAs are reconfigurable at the level of a one-bit data path. Since many algorithms require a much larger word size in order to store numerical information, this results in a significant redundancy of configuration logic.

An experimental computing model offers an alternative called *wormhole run-time reconfiguration (RTR)* [Bit97a]. In this model, configuration data and information to be processed are grouped together into what are called streams- sequences of words which enter the chip through multiple, parallel data ports. At the start of the stream is the configuration information which is grouped into a series of packets. Each packet configures a single hardware resource within the chip and is simply a collection of words. Following this are the data words which are then operated upon. This information is not divided into packets and can be as long as desired.

All of the hardware resources within the reconfigurable computing platform form a pool of configurable units which can be operated upon by these streams. The streams are injected into this pool, one word per clock cycle, steering themselves and configuring various elements as they are encountered. Thus, it may take multiple clock cycles to program a hardware resource which is configured using a multi-word packet. However, there is never any delay in the system, just the initial latency required to move the configuration header through the chip. After words, the data will stream through continuously. Streams can split and broadcast their information to multiple targets, or join together. Therefore, traditional pipelines, that of a series of linearly connected stages, are only a small subset of the possible configurations which can be present within the system.

The configuration information essentially constructs a deep computational pipelines which then operates upon the data. This concept of programmable systolic arrays corresponds to that of pipenets as discussed in Hwang et al. [Hwa93].

One of the main advantages of this system is that it is efficient in terms of power and space. The only shared resource amongst the pool of resources is the clock line. Configuration cells are physically located close to the structures that they control, so routing overhead is kept to a

minimum. In addition, multiple streams can tunnel their way through the chip simultaneously, allowing for the use of multiple algorithms computing simultaneously with no overhead. There is no explicit programming mode, versus a data processing mode. Instead, any subset of the resources may be programmed by any number of streams while the remaining resources may be processing data. A disadvantage, though, is a loss of flexibility. Since *wormhole routing* is designed to handle data of a certain word size, a problem which requires some other word size may map poorly.

The *Colt/Stallion* architecture, developed at Virginia Tech, is an experimental embodiment of this concept. A first generation concept version, called the *Colt* chip, has been successfully designed and fabricated. It is now being used for the development of digital signal processing algorithms using a run-time reconfigurable paradigm.

In general, a program for the *Colt/Stallion* architecture corresponds to a dataflow graph. Each vertex in the graph represents a computational resource and edges correspond to routing resources. In order to convert an abstract algorithm into a set of streams suitable for configuring the chip, some way must be found in which to map this graph onto the physical resources contained within the device.

The purpose of this thesis is to present a programming approach for wormhole RTR which can transform an abstract dataflow graph into a set of streams capable of programming a *Colt/Stallion* chip. The *Tier1* compiler presented here transforms an input structural specification of an application into one or more streams which can be used to completely or partially configure the distributed resources in a wormhole RTR platform. *Tier2* uses much of the same syntax as the *Tier1* compiler and adds automatic place-and-route capability. The streams synthesized by these compilers can be used for computing on a platform consisting of *Colt/Stallion* configurable computing integrated circuits [Bit97a].

The contributions made by this project include:

- The development of a textual language which natively supports the stream concepts inherent to *wormhole run-time reconfiguration*.

- The development of an assembler which allows a programmer explicit control over all aspects of the *Colt* chip.

- The automation of the place-and-route process, which allows a programmer to describe an algorithm in the form of an abstract dataflow graph. The compiler takes care of mapping this graph to hardware resources and generating the necessary programming streams.

This thesis begins with a brief introduction to stream-based configurable computing in Chapter 2. The formation of the stream headers is performed by the structural synthesis "compilers" called *Tier1* and *Tier2*. Chapter 3 covers the compiling process, an overview of the *Tier1* language, and a discussion of the underlying C++ class structure. Chapter 4 discusses genetic algorithms and how one can be applied to solve the problem of placing and routing an algorithm in the *Colt/Stallion* architecture. Chapter 5 provides an overview of the *Tier2* compiler, including

syntax and a discussion of the class hierarchy. An example application is presented in Chapter 6, illustrating these concepts. Chapter 7 provides experimental results of the *Tier2* compiler. Future directions for research are discussed in Chapter 8 and conclusions can be found in Chapter 9. Finally, the appendices offer a reference guide to the use of the compilers, including a language reference and usage guide.

Chapter 2: An Overview of the Colt Architecture

In order to place the theme of this thesis into proper context, this section provides a brief overview of the stream processing paradigm and how *wormhole RTR* is used. The purpose of this chapter is to introduce the basic architecture of the *Colt* chip. This is necessary because many of the design decisions made in the development of the *Colt* chip greatly affected the development of the *Tier* compilers. A more detailed account can be found in Ray Bittner's dissertation [Bit97a].

2.1 Wormhole Run-time Reconfiguration

Wormhole run-time reconfiguration provides a framework for implementing large-scale rapid run-time reconfigurable CCM platforms. It is intended as a method for rapidly creating and modifying custom computational pathways using a distributed control scheme (*data-driven* partial run-time reconfiguration) [Bit96]. The basic element is the stream, a concatenation of programming header and operand data (refer to Figure 1). The programming header is used to configure a computational pathway through the system, including both the functional units which perform the computations and the routing resources. The stream is self-steering and, as it propagates through the system, configuration information is stripped from the front of the header and is used to program the unit at the head of the stream; thus, the size of the header diminishes as the stream propagates through the system. The stream header is composed of an arbitrary number of packets of programming information. Each packet contains all of the information needed to configure a designated unit in the system. The composition and length of the packets are variable so that different packet types may coexist within the same stream header and hence heterogeneous unit types may be traversed by a given stream.

Figure 1: An illustration of the stream format. A stream is composed of a header segment (expanded on the right) and a data segment.

The stream data section can contain zero to an infinite number of data words for subsequent processing. Internally, the data words carry with themselves a one-bit tag which identifies the

4

data as valid or invalid. This tag is generated by each of the functional units as the streams of data wind through the chip. For instance, if two streams are to enter a functional unit and the result is to be a pair-wise summation of the values, the output valid bit can be set to be a logical-AND of the two input valid bits. In this way, valid data is produced only if both of the input words are valid.

In addition to the data flowing through the chip, three other bit-wide signals can be configured. These are discussed in depth in [Bit97a]. The conditional bit, *cond*, can be generated from a variety of sources within each functional unit (FU) and can be used to modify the behavior of various FU operations. The *carry* bit is generated as a result of ALU operations and the *shift* bit is generated as a result of barrel-shifting operations. The architecture of each FU is discussed later in this section.

Example stream sources are video cameras, A/D converters, and antennas. Intermediate streams may contain, for example, filter weight updates, partial computational results, or new computing

Figure 2: Graphical depiction of the Colt architecture [Bit97a].
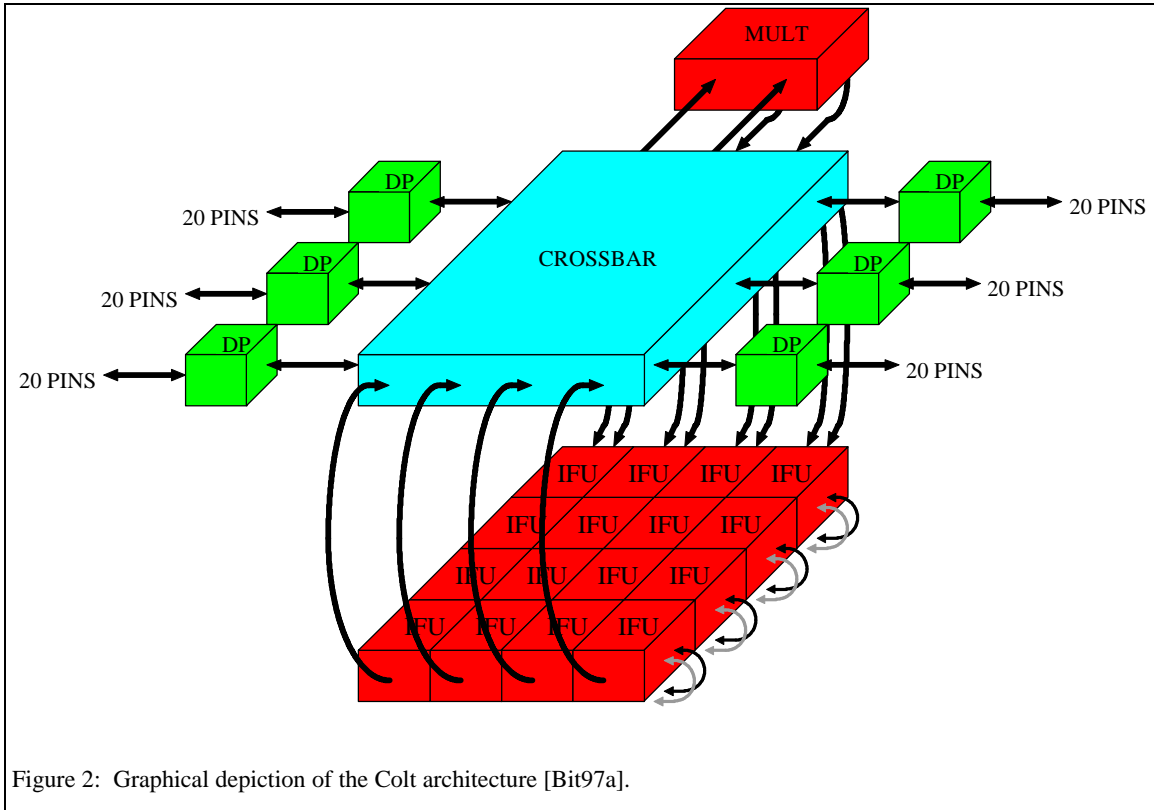
contexts. Details on how wormhole RTR and streams are supported at the physical level can be found in [Bit97a].

Figure 2 gives a graphical depiction of the underlying structure of the *Colt* chip, the first generation device of this architecture. As can be seen, it consists of essentially four parts: the data ports, the crossbar network, the integer multiplier, and the IFU mesh.

Streams, sixteen bits in width, enter the chip through the six data ports; each is bi-directional and can thus either accept a stream or output a stream. Once the stream has passed through a port, it enters the crossbar. This device allows for almost complete connectivity between any attached element and any other element, although data ports are not allowed to communicate directly with other data ports.. A stream programs a pathway through the network; the pathway will persist until another stream changes it. An example of a stream tunneling through the chip is shown in Figure 3. The stream shown is based upon that shown in Figure 1.

The integer multiplier passes all programming information. Data words are multiplied together and produce two output streams: a high byte and a low byte. More complex processing is handled within the mesh. This consists of sixteen interconnected functional units (IFU) which each contain routing resources and a functional unit (FU). The functional unit, diagrammed in Figure 4, can accept two operands, a left word and a right word, and perform various operations on them. The device contains a propagate-generate-result ALU, a barrel shifter, and some simple conditional logic. The four flag bit signals, discussed earlier, are generated by and affect the operation of, the functional unit; they cannot travel outside of the mesh. Inside of the mesh, the bit-wide signals' routing resources are equivalent to that of the data path.

Each IFU contains two sets of routing resources: local connections and the skip bus. The local connections allow an IFU to communicate with the four closest neighbors, i.e. to the top, bottom, left, and right. Generally, they are referred to by ordinal direction, where north points towards the entrance of the data, from the crossbar, into the mesh. Each of the operands and data bits can accept data from any of the four directions without interfering with the operation of any other IFU. The local connections are connected together at the east and west extremes. From the north, local connections direct data into the mesh from the crossbar; the south connects the mesh back to the crossbar.

The skip bus, on the other hand, utilizes shared resources. It consists of two paths: a north-south bus and an east-west bus, each of which passes directly over an IFU. Thus, data can be routed over a device without affecting its operation. An IFU can place data onto the bus, or direct data from one of the buses into the functional unit. This

Figure 3: An illustration of a stream tunneling its way through the Colt chip.

connectivity is not limited to only the data path: A skip bus set exists for each of the bit flags as well.

The skip buses, as mentioned above, can route data over an IFU. They can also accept data from their compass right. Thus, data can travel north, having its source come from the east. Since there are only two buses, data cannot travel north and south simultaneously. In addition, if an operand requires data from the north over the skip bus, for example, this uses up the north/south skip bus resource for that IFU.

The skip bus wraps from east to west in a similar fashion to the local connections. The crossbar can direct data directly to the skip bus from the north, but the skip bus does not connect to the crossbar in the south. Finally, only data may traverse the skip bus. Programming of the interconnections occurs within each IFU and this information may traverse local connections only.

Figure 4: Simplified functional unit (FU) schematic [Bit96].
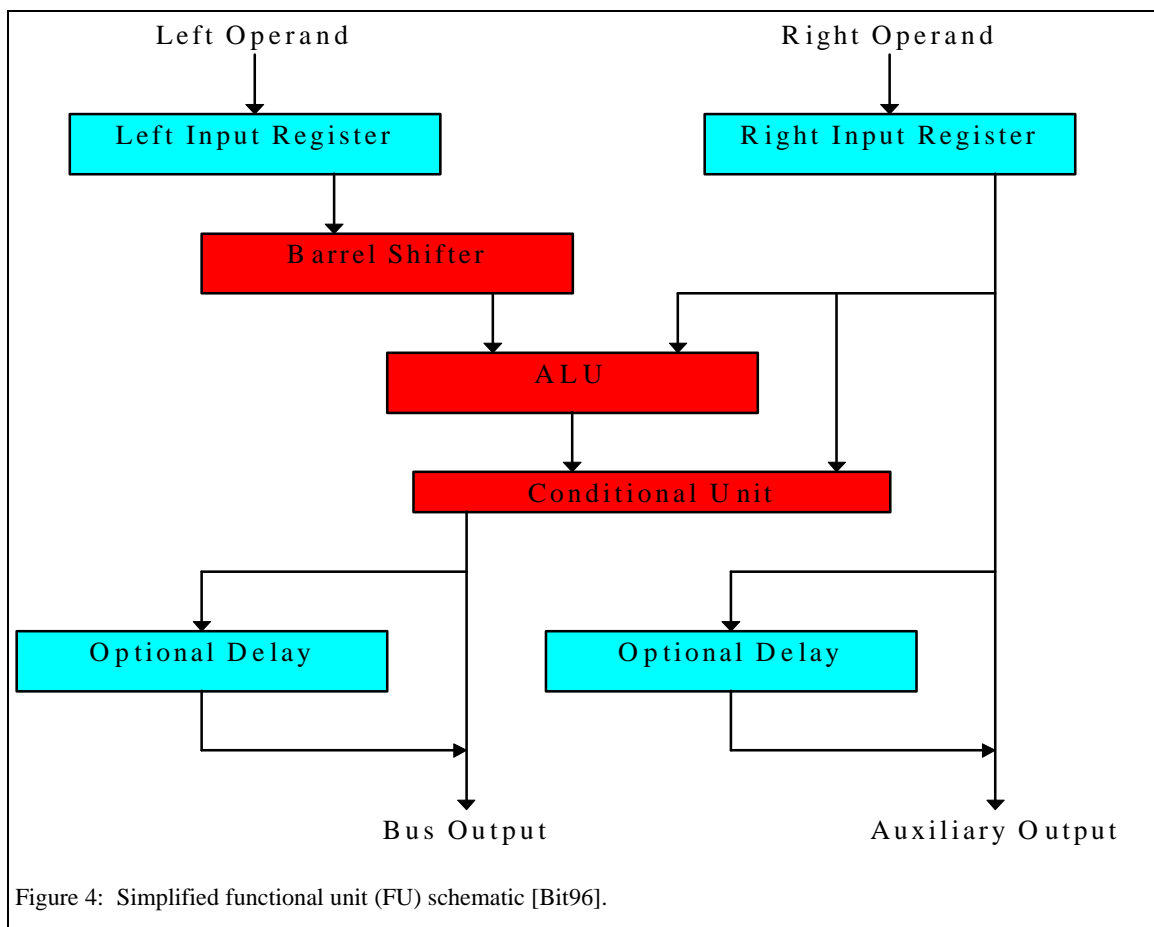
Thus, in the current design iteration, data streams do not precisely mirror the flow of programming information through the chip. This is likely to be rectified in future versions of the architecture.

In general, the *Colt* architecture demonstrates the potential use of RTR concepts. As it is a prototype chip, many limitations are present in its design which will be eliminated in future

generations. Some of the changes, as detailed in Bittner's dissertation, may include such items as increasing the orthogonality between data stream and programming stream handling, increasing the number of functional units within the mesh, and making several improvements to the crossbar and data ports. However, the basic concepts and general architecture will remain the same. Thus, the work presented herein is applicable to future chip versions and thus has been designed in a fairly scalable manner.

Chapter 3: A Tiered Approach Towards Stream Synthesis

This chapter introduces the design philosophy behind the multi-tiered approach of this compiler effort. It discusses the *Tier1* language, a custom syntax which natively supports the stream concept. The language grammar is detailed and an example application is presented.

In order for an applications programmer to map an algorithm to the *Colt/Stallion* architecture, some method must exist by which he or she can describe the flow of data through the chip, and from that, generate the streams which will then be used to program the device. Two compilers have been developed, and a third one is planned, to handle this task. Each level of compiler development, or *tier*, adds a level of abstraction to the process of mapping an algorithm to the final platform.

The decision to create a custom language, rather than use a standard language, such as C or VHDL, or use standard FPGA tools, revolved around several issues. First, the architecture of this chip is very different from traditional FPGAs: rather than being bit-oriented, it is word-oriented. Therefore, traditional FPGA design tools are really not suitable for the task of programming this chip. Second, the current *Colt* chip is resource limited, having only sixteen functional units and one hardware multiplier. Although this will increase in the future, it was decided that it was unacceptable for a compiler to potentially waste resources when mapping an algorithm. Therefore, both of the compilers require explicit control of the functional units; only routing and placement are automated within the *Tier2* compiler. Finally, it was decided that it was better to proceed in smaller steps in order to create tools which could be completed in a timely fashion. A custom syntax, while requiring a steeper learning curve, allowed for more rapid compiler development. In the future, a standard language might be mapped to this custom syntax, as described below.

In the first case, the *Tier1* "compiler" acts as an assembler, allowing the user to have explicit control over all aspects of *Colt*. Certain feature are automated in order to make the overall design process easier. For instance, the compiler makes assumptions about the usage of data within an FU and modifies the handling of the valid bit tag accordingly. As an example, suppose that the user programmed an FU to store a value originating from the north into the left register, and a value from the south into the right register. The result is to be added together and sent to the east. Normally, the valid bit would have to be explicitly set to be the logical-AND of the valid bits from the left and right registers. However, the compiler assumes that since both registers have had assignments made to them, they will form the basis for what data is considered to be valid, and so the valid bit is automatically set to be the logical-AND of the left and right valid bits.

The *Tier2* compiler only requires the user to construct a data-flow graph of the algorithm, where each vertex represents a functional unit in the *Colt* chip. The compiler then finds a valid placement for the algorithm and routes each of the data flow edges. In general, the syntax for the two *tier* languages are almost identical: explicit control of the internal workings of the FU is still required. The main difference is that the *Tier2* language adds a level of abstraction. Rather than

requiring the user to specify specific directions that data should travel in (north, south, etc.), the program simply describes the source of the data, i.e. which functional unit produces a result. The place-and-route capability of the compiler performs the task of routing these data streams inside of the chip.

When using the final compiler, *Tier3*, it is foreseen that the user will write in a higher level language which is not constrained to dealing with individual resources on the chip. This is where a language such as C may be used. The compiler might convert C into an abstract dataflow graph, described using the *Tier2* language, by partitioning the program into tasks to be performed by individual functional units. The *Tier2* compiler would then perform the task of placing and routing the resulting graph. This would allow for a tremendous amount of code-reuse, and would tend to isolate the top compiler level from the physical hardware of the chip. Thus, architecture changes would not require changes to the *Tier3* compiler, only to the *Tier2* compiler. Neither the *Tier3* compiler nor the language has yet been designed, and is a topic for future research.

3.1 Underlying Programming Structure

A programming stream for the *Colt/Stallion* architecture consists of a sequence of packets, where each packet programs an entity on the chip. These devices include crossbar connections, data ports, and functional units. Each packet is made up of a series of words. These words themselves may be further subdivided into various bit fields which are used store values in registers or to program multiplexers for steering data. Figure 5 shows an example of such a multi-field word.

Figure 5: An example configuration register word containing two multiplexer fields and one data register field.
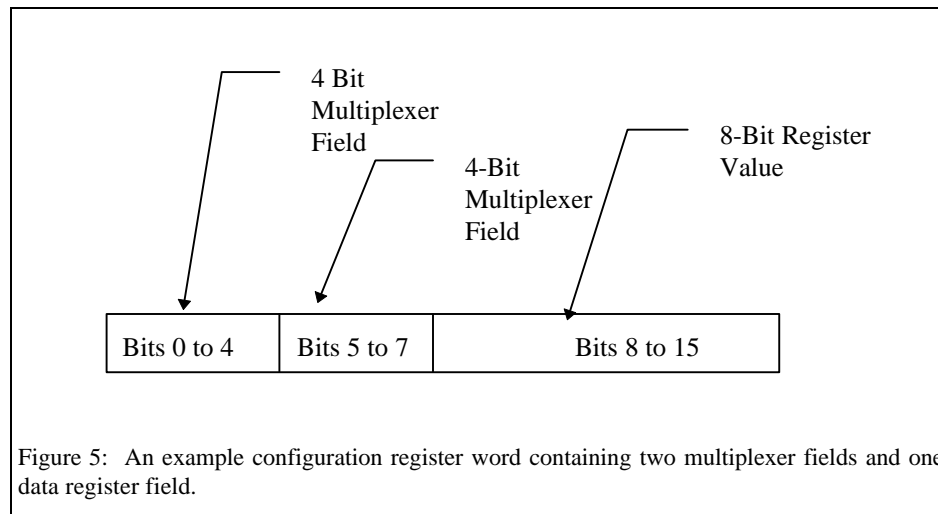
The final result of both the *Tier1* and *Tier2* compilers is a list of these fields, divided up into packets. For instance, the following is a sample description of a packet for programming a dataport.

```
DataPort                    // Declaration describes type of packet
  DPRWBIT = 0               // Read/write field
  DPSYNCBIT = 1             // Selects synchronous mode- described later
  DPLOOPBIT = 0             // Selects loop mode
  DPSYNCMP1 = 1             // One field in the synchronization mask
  DPSYNCMP2 = 1
```

```
      DPSYNCMP3 = 1
      DPSYNCMP4 = 1
      DPSYNCMP5 = 1
      DPSYNCMP6 = 1
      DPADDRESS = 24          // Hardware resource address
   EndDataPort
```

As can be seen, a total of ten fields exist. In this case, all of these fields are packed into a single configuration word. This syntax for describing configuration registers is called `dfc` and is further described in [Bit97a]. The `dfc` code completely describes the values of configuration registers needed to program elements of the *Colt/Stallion* chip, and software exists to transform this format into a sequence of sixteen bit words which can be supplied to the chip or to a simulator.

3.2 Tier1 Language

The *Tier1* compiler uses a custom syntax for describing the functionality of each of the FUs within the *Colt* chip. In order to support the data-flow oriented nature of the *Colt/Stallion* architecture, the focus of the language is on the stream, as defined in the prior chapter. Each stream defines a pathway that data will take as it moves through the chip. This may include traveling from an input data port, through the crossbar, to either the hardware multiplier or to functional units in the mesh. The language itself does not describe any of the data, but rather the manner in which the data will be handled. Once this programming stream has been fed through the *Colt* chip, data may follow.

3.1.1   Tier1 Syntax

Contained within each program are a variety of constructs, such as port definitions, which map physical data ports to logical names, components, which allows for the easy reuse of commonly used tasks, and macros, which are used to directly modify the behavior of dfc fields. Refer to Figure 6 for the general structure of a *Tier1* program. The program is first fed through the C preprocessor before it is parsed. This allows for the use of include statements and constants. In addition, a standard library is parsed before the program is processed, which contains commonly used macros and components.

The port definition maps data port numbers (currently one through six) to symbolic names which are then used throughout the stream definitions. For example, the following port definition statement specifies that physical Data Port One will be mapped to the name of "indata" and that physical data port two will be mapped to "outdata".

```
   ports
         indata = 1;
         outdata = 2;
   end ports;
```

In addition, one other important task is taken care of through this construct: all of the ports listed in this construct are used to form the port synchronization map. In order to understand what this is, it is first necessary to understand the various modes that the data ports may assume. Three modes are supported: *raw mode*, *synchronous mode*, and *loop mode*. The first is not supported by either of the compilers. Synchronous mode provides for a means of flow-control between
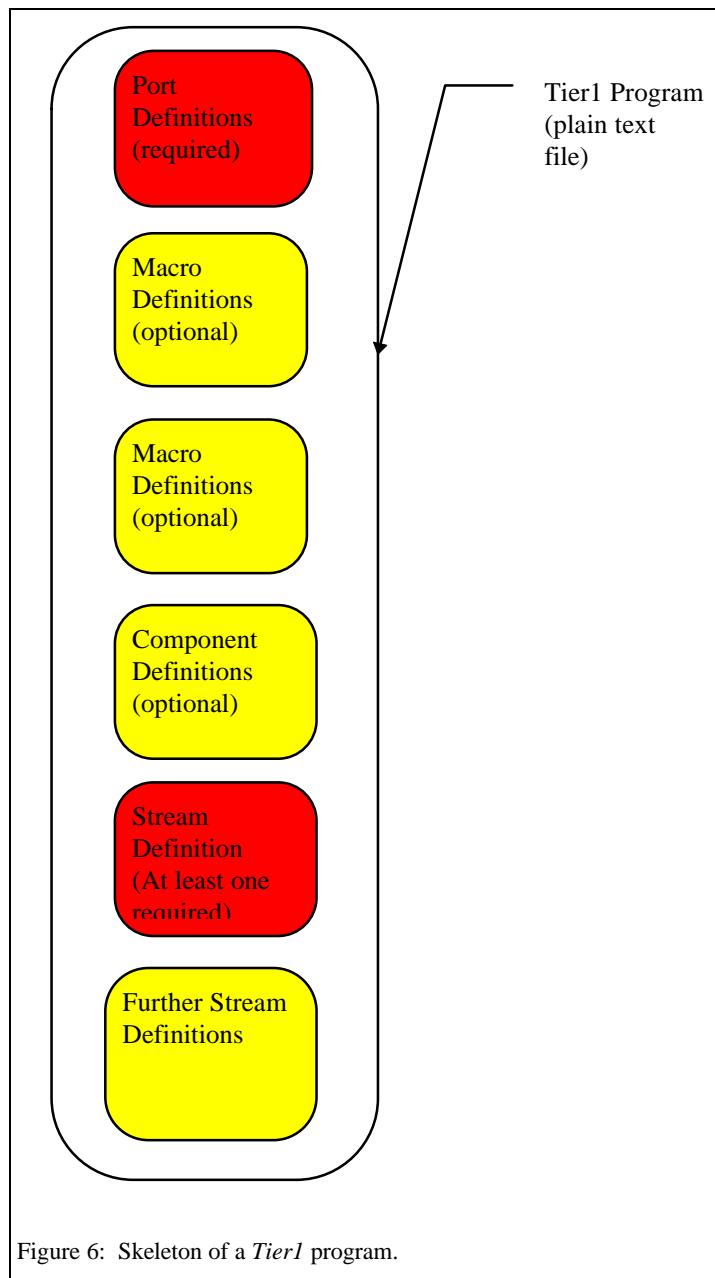
Figure 6: Skeleton of a *Tier1* program.

various ports. Each port contains a synchronization mask, where each of six fields corresponds to a data port on the chip. If the field is set for a particular port, then that port is able to influence the behavior of the other ports specified in the mask. This allows any port in the mask to cause all other ports to either cease sending data into the chip or cease receiving data from the chip. However, the effects are not immediate: the chip will continue to output data until its internal pipelines are empty.

The final data port mode, *loop mode*, provides for the ability to process a single set of operands at a time. Data ports will submit a single set of data to the chip and will not provide any more until the data has been fully processed and has exited the chip. This mode is rarely used, since it is extremely wasteful of clock-cycles, but if the user wishes to force the chip into this mode, then he or she can do so by inserting the keyword `loop` after the keyword `port` in the port definition construct. For an in depth discussion of the data ports, refer to [Bit97a].

Macros allow the user to directly manipulate configuration fields of functional units. They are used in various assignment statements, such as to configure the ALU, to modify valid-bit handling, etc. A macro is evaluated at compile time and allows for basic parameter passing, limited flow control, and simple bit-oriented operations.

Another feature of the macros is that of namespaces. A namespace designates all of the macros which can be called in a particular assignment statement within an FU definition, which are discussed later in this section. This allows the user to group together macros of similar function and prevents macros from being called in situations where their functionality would make no sense. It also allows for name overloading: macros of similar function, but which act upon different elements of a functional unit, can share the same name. Namespaces exist for defining

ALU operations, modifying the functionality of the *shift* bit, *carry* bit, valid bit, conditional bit, and changing the behavior of the left register.

Macros can take an arbitrary number of numerical parameters which can be used within assignment statements or within flow control statements. Assignment statements have three forms:

```
<field> = <parameter or constant>
<field> != <parameter or constant>
<field> &= <parameter or constant>
```

The first form simply assigns the right hand side to the field on the left. The second form performs a logical-OR of the right hand side with the left hand side and stores the result in the field designated by the left hand side. Finally, the third form performs a logical-AND.

Flow control can take two forms: the first is the `assert` statement which is used to check the validity of parameter data. If the parameter does not fit within the prescribed bounds, compilation of the program stops and an error message is reported. For example, the following statement checks to make sure that the parameter called "bits" falls within the range of 0 to 4. If not, then the error message is reported.

```
assert (bits = [0,4]) "Invalid right operand bit specified!";
```

Case statements make up the other kind of flow control. They allow a macro to conditionally execute statements based upon the values of a parameter. The following is an example case statement which examines the value of "bits" and conditionally executes statements based upon that value.

```
case bits is
  when 0:  FNOutSel = 2;
  when 1:  FNOutSel = 3;
  default: FNOutSel = 0;
end case;
```

The statements after the default keyword are executed if the value of the parameter does not equal any of the values listed in the case statement.

All of the statements within a macro can be nested to an arbitrary depth within case statements. This provides for an extremely flexible method of modifying `dfc` fields.

This example macro, called `donothing` takes one parameter, named `bits`, and checks to make sure that its value lies between 0 and 4 inclusive. If not, then the error message is displayed and execution of the macro stops. If it is a valid value, then the configuration field `FNOutSel` receives the value of "bits".

```
macro cond donothing (bits)
  assert (bits = [0,4]) "Invalid value for bits specified!";
  FNOutSel = bits;
end macro;
```

Stream definitions describe the flow of information through the computing platform. The outer definition defines what data port is to be used for input and what data ports are for output. Only

one port can be an input port, but the stream can be split within the stream body and can flow to multiple output ports. An example stream definition is:

```
stream MULT (in hibyte, out multdata)

  <stream body statements>

end stream;
```

Within the body of the statement lie crossbar routing instructions and declarations for functional-unit behavior. The crossbar statements are used to route data from an IFU at the bottom of the mesh, a data port, or a multiplier output port, to any one of the following: an IFU at the top of the mesh, a data port, or a multiplier input port. The only restriction is that data ports cannot route directly to other data ports, nor can multiplier ports route directly to other multiplier ports.

The functional unit definitions constitute the bulk of the code within a *Tier1* program. The construct is called a `block` and each has a unique name within the stream. In addition, the outer block construct contains programming routing information at its end. This programming information directs the flow of the stream to one or more directions (north, south, east, west) or to the crossbar, if the functional unit is at the bottom of the mesh. The following is an example block definition.

```
block RENORMALIZE
  <assignment statements>
  <macro calls>
  .
  .
  .
end, go south to DELAY, go east to PROPAGATE;
```

This definition defines a functional unit called `renormalize` which contains within it a series of statements. Once this functional unit has been programmed, further programming stream packets will flow south to the functional unit `delay` and east to the functional unit `propagate`.

Within the block definition are a series of statements: General assignment statements, ALU specific assignment statements, macro statements, component statements, and skip bus programming statements. The ALU specific statement is rather complex, owing to the fact that a variety of options exist for the flow of data through this portion of the functional unit. An operation can be specified which can take as operands either the left, right or both of the input registers. A shift can be performed, conditionally if desired, upon the data stored in the left register, a constant can be specified for the left register, an optional delay can be designated, and finally, the actual output value can be made conditional. Depending upon the conditional bit, either the ALU result or the value of the right register can propagate to the output.

An example statement is as follows:

```
out = delay ifcond
       left << 2, add
     else right;
```

14

This takes the value of the left register, shifts it left twice, and adds it to the right register. If the conditional bit is true, then this value propagates to the output, otherwise the right register propagates. In either case, a one clock cycle delay is added in.

The addition operator, called add, is actually a macro declared elsewhere in the program. This macro has a namespace of `operator` and is thus allowed to be called within this statement. The body of this macro configures the multiplexers within the ALU itself to perform the addition operation. Refer to [Bit97a] for more information on the structure of the ALU.

General assignment statements are used to adjust the behavior of the left operand register, right operand register, valid bit, conditional bit, conditional output bit, *shift* bit, and *carry* bit. Depending upon the source, an optional inversion can be specified. Also, a macro for that namespace can be called, and a source can be specified. For instance, the following statement modifies the behavior of the conditional bit:

```
cond != condout from local north;
```

This implements the optional inversion and uses the conditional bit produced by the functional unit directly to the north of the functional unit containing this statement. The `condout` identifier is a call to a macro named `condout` with a namespace of `cond`. Macros can also be called directly within the body of the block statement, provided that the called macro has a blank namespace.

Since it is likely that a series of functional units within a program might perform identical functions, except for the routing of information to and from the unit, a method exists for defining a template which can be used over and over again. These are called components. Their syntax is identical to that of the block statement, except that the keyword `component` is used instead of the keyword `block`. Within the body of a block, a component can be called. This causes the compiler to copy all of the `dfc` fields modified by the component over to the block being parsed. For instance, the statement

```
use adder;
```

causes the compiler to search for the component named `adder` and copy over all of its modified `dfc` fields. The rest of the block body would probably consist of routing statements designed to get data from the appropriate source and deliver it to the correct destination.

The final type of construct within the block body is the skip bus statement. This specifies how the skip bus is to route information over a functional unit. A skip bus block can be defined for each of the four types of skip buses: data, conditional bit, *shift* bit, and *carry* bit. The syntax simply describes a direction in which the data is to flow and from where the data is to come. This example routes data to the north from the opposite direction (the south) and routes data to the east from the data output of this functional unit..

```
skip data
 north from opposite;
 east from out;
end skip;
```

This constitutes all of the grammar constructs of the *Tier1* language. For general grammar forms and a complete listing of pre-written macros, refer to Appendix A.

3.1.2   Example Tier1 Program

A small example program is presented here. Its purpose is to take a value from an input register, have two constants added to it, and pass the data to an output register.

```
Ports                               // Port definition construct
        indata = 1;
        outdata = 2;
end ports;

#define ADDVAL  3                    // conditional compilation
#if !defined(COL)                    // through the use of the C
preprocessor
#define COL      1
#endif


// This is the stream which defines all of the actions
stream COLUMN (in indata, out outdata)

        // crossbar connection to skip bus, column 1
 port indata => ifu ifu1 at COL;

        // First ifu in the column
   block ifu1
        rightreg = from local north;   // Accept data from the north
        out = ADDVAL,add;              // Program the ALU to add  a constant
        skip data                      // Send the data out onto the skip
bus
                south from out;
        end skip;
 end, go south to ifu2;

        // Second ifu in the column - program skip bus to go south
 block ifu2
        skip data
                south from opposite;
        end skip;
 end, go south to ifu3;

        // Third ifu in the column - program skip bus to go south
 block ifu3
        skip data
                south from opposite;
        end skip;
 end, go south to ifu4;

        // Fourth ifu in the column
 block ifu4
        rightreg = from skip north;  // Right operand takes from local north
        out = ADDVAL, add;           // left operand set to ADDVAL, ALU set
                                     // to add operator
                                     // add operator is specified in std.tlb
 end, go to crossbar;

 ifu ifu4 => port outdata;           // direct data out of mesh to the
                                     //output data port
```

```
        end stream;
```

The program begins with a port definition section which defines an input port called `indata` and an output port called `outdata`. Following that are some C preprocessor statements which illustrate the use of conditional compilation and constant definitions. Next comes the actual stream definition. It defines an input port and output port in its header and a series of crossbar statements and functional unit definitions within its body.

The first crossbar statement directs data from the input port to the first functional unit, called `iful` in the mesh column specified by the constant COL. When data enters this FU, it travels to the right register, through the ALU where a value of ADDVAL is added on, then out through the skip bus. The data is processed again by `ifu4`, where another constant is added on to it. The final result is passed to the output data port. Definitions for `ifu2` and `ifu3` exist because programming data cannot presently be passed through the skip bus.

3.1.3   Software Design

The *Tier1* language can be divided into two main parts: the parser which converts the text-file program into a set of `dfc` field data structures, and the listing phase which takes these data structures of `dfc` fields and creates the final output streams. The first part of the process, the parser, was constructed using the tool *Visual Parse++* by Sandstone Technologies [San94]. It takes as input a grammar description of the language, in the form of modified BNF. The output is a set of class libraries which can be modified to perform the desired actions. In essence, this tool combines the functionality of the traditional tools *lex* and *yacc* into a single object oriented system. *Visual Parse++* can handle a subset of the LR-Regular languages, which includes all LALR(k) languages. The design time for this compiler was greatly reduced by using this tool and the ability to modify the language without a great deal of effort was greatly increased.

The first step in the design was to create the grammar rules and verify that they worked using *Visual Parse++*. The result can be found in Appendix C. Once these had been created, the actual software design began. The code was developed in C++ using *Visual C++* by Microsoft. Refer to Figure 7 for a graphical view of the class structure of the compiler.

As can be seen by the diagram, the main program instantiates a compiler class, which then performs the bulk of the work. The main program simply deals with opening up required files, reading in command-line parameters, etc.

17
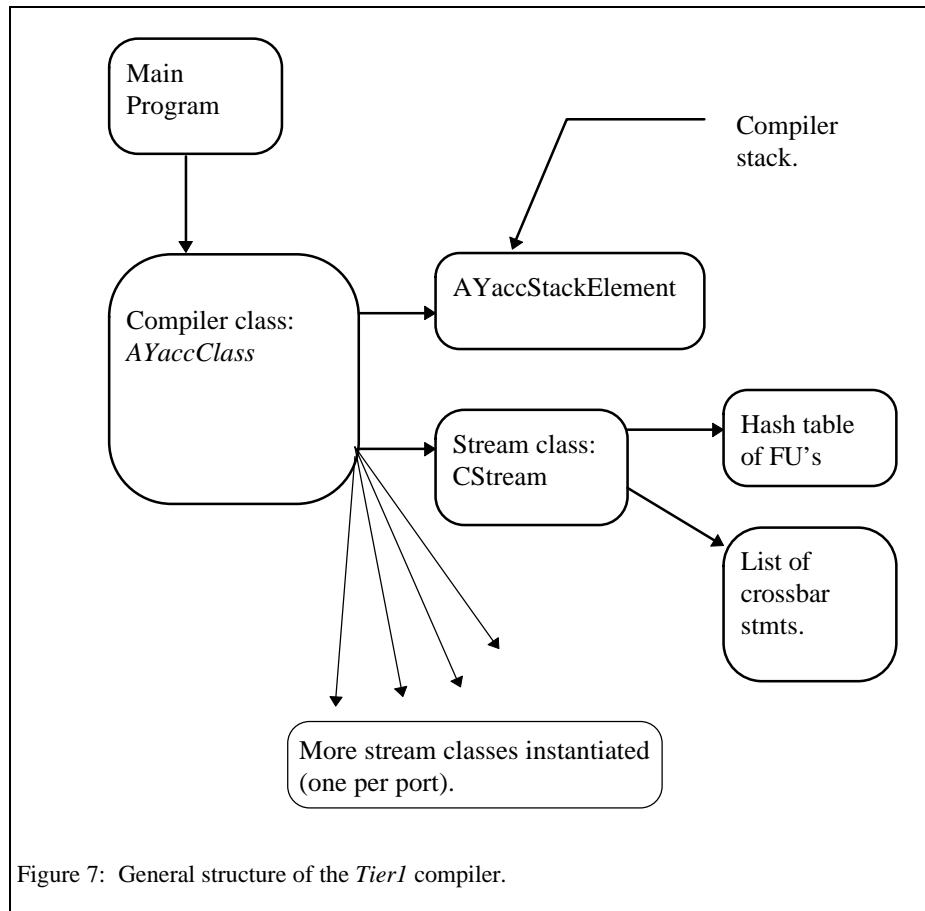
Figure 7:   General structure of the *Tier1* compiler.

The compiler class parses the program file and performs appropriate actions. Each time that it encounters a stream definition, it creates a new `CStream` class. The definitions within the stream are then stored within this class. At the end of the compiling phase, the `CStream` object contains a hash table, indexed by name, of functional unit definitions, and a list of crossbar statements. At this point, the listing phase commences.

Since each stream can have only one input port, but multiple output ports, the programming stream can be viewed as a tree structure.   The listing routine recursively evaluates this tree, writing the `dfc` blocks to an output file. Currently, this output file is named "portN.dfc", where N represents the number of the input port.

The process of listing continues for each stream until all streams have been evaluated. During the listing phase, the compiler checks to make sure that all output ports listed in the stream header have been reached in the stream traversal, and that the tree can map to the topology of the *Colt* chip. For instance, functional units can only reach the crossbar if they are on the bottom row of the mesh. The compiler checks for this and will issue an error message if it detects a violation.

3.1.4   Tier1 Summary

The *Tier1* compiler represents a step forward in ease of programming over the initial tools described in [Bit97a]. It provides a straight-forward method for creating programming streams and also automates certain aspects of the configuration process, thereby simplifying the overall task. The result is that development time is much shorter using this method versus the first generation of tools; however, the problem still exists that the user must explicitly map an algorithm onto the *Colt* chip. This is a tedious process, but is sometimes necessary for when the user wants to do something unconventional.

The main purpose of this compiler, however, was to act as a stepping stone. Once this compiler had been developed, the road to the *Tier2* compiler involved developing the place-and-route algorithms and then modifying the compiler grammar, rather than completely writing the compiler from scratch. Only the data routing aspects of the *Tier2* compiler had to be verified; the programming of the functional units was never modified. Thus, a great deal of code-reuse was made possible and development time was greatly reduced.

Chapter 4: Genetic Algorithms For Placement

The purpose of this chapter is to describe the method by which an abstract dataflow graph can be mapped to the physical hardware of the *Colt/Stallion* architecture. The problem of placement, the mapping of the graph vertices to resources on the chip, is a difficult one. Various algorithms for performing this task are discussed, with the final solution being described in detail. The task of mapping dataflow graph edges to routing resources is also discussed. In addition, an analysis of the final approach is included in order to show that the solution chosen is in fact a scalable method for solving the placement problem.

4.1   The placement problem

The problem of mapping an algorithm to the Colt chip is a difficult one. It is assumed that a program can be represented as a dataflow graph, where vertices represent functional units, data ports, or multiplier ports and edges represent physical connections between these resources (Figure 8). The *Tier2* language fits this description. In fact, a *Tier2* program actually contains four dataflow graphs: one for the data, one for *carry* bits, one for *shift* bits, and one for conditional bits.

The task then becomes that of mapping the vertices of the graph to physical units in the *Colt* chip. Many factors come into play at this point: due to limited routing resources, some configurations may not allow all edges to be mapped to physical connections. Even if all edges can be connected, some placements will obviously be superior to others. For example, a potential solution in which all edges are implemented as local connections will provide far lower propagation times than one in which the edges are implemented as long skip-bus routes.

Another difficulty is the shear magnitude of the problem. Suppose that a particular program utilizes all sixteen of a *Colt* chip's functional units.   Then an exhaustive placement program has sixteen choices when selecting a vertex to occupy the first functional unit. The second FU has fifteen choices, etc. The end result is a total of 16 factorial combinations, or 2.092×10^{13} possible solutions to check. Thus, the order of the problem, using an exhaustive search, is *O(n!)*, where *n* represents the number of functional units within the chip. Of
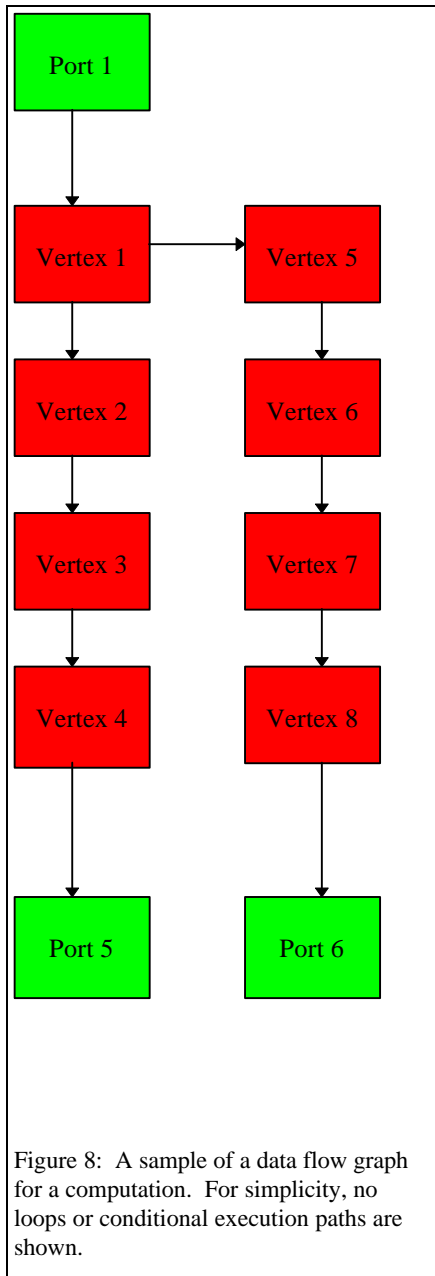
Figure 8: A sample of a data flow graph for a computation. For simplicity, no loops or conditional execution paths are shown.

course, this is something of a simplification, since some solutions will obviously be invalid; a functional unit which accepts data from the crossbar must be on the top row. However, the possible number of solutions is still extremely large even discounting obviously incorrect ones.

Except for very simple cases, the layout problem will be NP-Hard [Sar96]. This means that it is probably only solvable in exponential time, as described above. The alternative is to use a sub-optimal algorithm which will produce a solution of good quality.

4.2   Deterministic Searches

In order to solve this problem in a practical amount of time, an algorithm must be developed which can search through the solution space without trying all possible solutions and produce a good result. In order to determine how good a solution is, it is necessary to have a cost function, or metric, which produces a larger value for worse solutions and a smaller value for better solutions, or vice versa. In this case, the number of skip-bus routing resources required to implement a particular placement was chosen as a cost function. Local bus connections count as a value of zero, since in most instances they represent the shortest possible propagation delay.

Two important categories for such search algorithms are deterministic and non-deterministic. The first class searches for a solution using a fixed set of rules; the search will always be the same, given an identical set of starting conditions. Such traditional techniques as gradient hill climbing fall into this category [Gold89]. This algorithm attempts to find a minimum by always moving in the steepest permissible direction. There are two main problems with this method: First, a differentiable cost function is required. In many algorithms, such as the placement problem being examined, the function is not differentiable. Second of all, this method can easily become trapped within local minima. The algorithm follows a slope, so once any minimum has been reached, no matter how poor, the algorithm cannot find another.

A greedy search method is similar to the hill climbing algorithm, except that it does not require a differentiable cost function. Instead, it works by successively laying down vertices which contribute the smallest amount towards the total cost. Greedy searches suffer from the same problem as hill climbers in that they can easily get stuck in local minima [Sar96]. In addition, a poor initial choice for placement is not correctable; the algorithm generally can only progress forward, attempting to minimize the cost as it proceeds. However, greedy algorithms are frequently very fast and are generally simple to implement [Sar96].

For the placement problem at hand, exhaustive searches are too slow due to the large number of possible solutions, and deterministic solutions are extremely limiting, frequently becoming caught in local minima. What is needed is a way by which the solution space can be searched quickly and without becoming entangled by solutions which appear to be good, but are in fact far short of the global minimum. Deterministic algorithms can be very useful, especially when the solution space is well understood and contains underlying structures which can be exploited by special purpose algorithms [For93]. This is not the case, though, for the placement problem.

4.3   Non-Deterministic Searches

If one randomly searches the possible list of solutions, then it is possible that the global minimum could be discovered eventually. However, a completely random search is no better than an exhaustive search. However, one can search in a directed manner: jump around within the possible solutions until a relatively good placement has been found. At this point, search in the vicinity in the hopes that a good solution is nearby. In case there is not one, always provide for the possibility of performing another wild jump into a far-removed portion of the potential solutions. Such non-deterministic, or stochastic, searches can frequently outperform classical methods in real-word problems [Fog94].

In general then, an initial starting point, or group of starting points, is randomly chosen. The algorithm generally chooses the better ones, but maintains a few poor ones as a means of breaking out of local minima. The solutions are then randomly modified in order to hopefully produce better ones; the magnitude of the alterations decreases as time progresses, or as the overall quality of the solutions increase. Although there is no guarantee of finding the absolute best solution, a good one is generally found in a relatively short amount of time.

Two commonly used non-deterministic search algorithms are simulated annealing and genetic algorithms. The first type was developed in 1953 by Nicholas Metroplis and mimics the process by which the crystal lattices of a glass or metal relax when heated, or similarly, to the behavior of crystal growth [Car97]. Initially, molecules wander around randomly, but as a solution cools, they become immobilized. If the cooling is slow enough, then the molecules settle into a crystal structure in which each molecule is at its lowest energy level. During the cooling, many molecules become trapped in states where their energy level is not at the minimum. However, a few remaining higher energy, still mobile, molecules can bump into the immobilized ones, knocking them into lower energy states. Solving a problem using this technique is analogous to minimizing the energy level of these molecules.

The process works by calculating the energy level, or fitness, of each new solution. If the fitness is better, then the solution is taken. If worse, then it is not immediately rejected; there is a probability that it will be accepted, just as in a crystal, higher energy molecules exist while many others have been immobilized. The probability function used has an exponential decay which mimics the cooling of a crystal solution. As time advances, the solution cools further, and there is less of a chance that higher energy molecules exist. Likewise, as time progresses, there is less of a chance that worse solutions will be chosen.

Genetic algorithms, on the other hand, mimic populations of living creatures. Just as evolution adapts living organisms to deal with the environment, so to do genetic algorithms adapt solutions to best solve a problem, based upon the values of a cost function. Evolutionary strategies have proven to be an extremely robust method for optimization [Fog94]. Considering how well organisms in the real-world have adapted themselves to harsh landscapes, this does not seem so surprising. In fact, genetic algorithms are being recognized as extremely good problem solvers in a diverse set of applications [Gold95].

Simulated
Roulette Wheel

Most Fit Gene-
Large Weighting

Least
Fit
Gene

Complete
Population
of
Placements

Figure 9: An illustration of the weighted selection process.
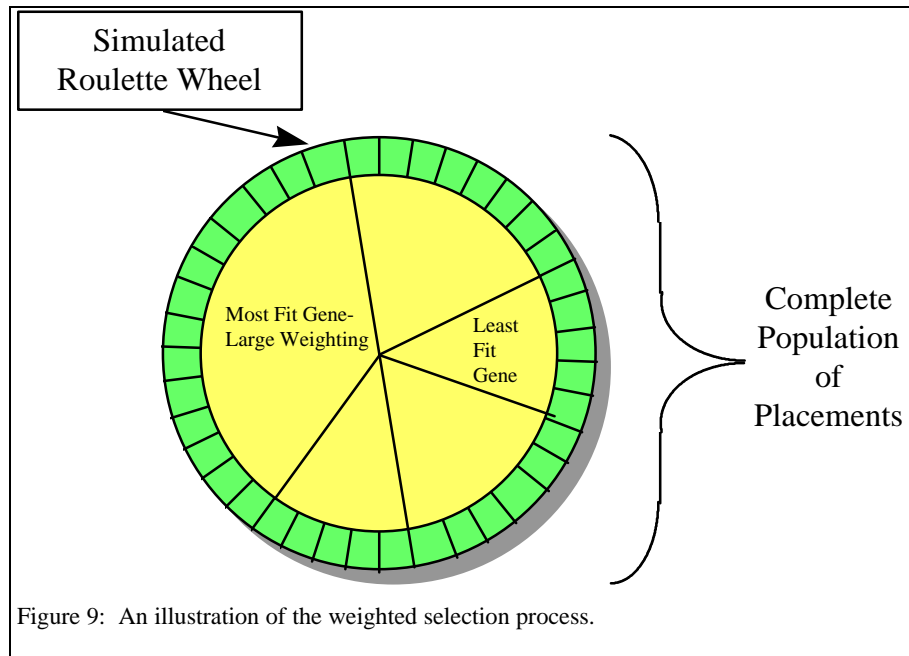
It is first necessary to be able to represent a potential solution to the problem at hand as an ordered list of a finite set of symbols. The set of symbols is known as the allele set and the ordered list is known as a gene. Typically, both the position of the alleles, and their values contribute towards expressing a potential solution [Fog94]. The genetic algorithm constantly modifies a population of genes, testing them for their fitness, until an acceptable one has been found.

The initial population of genes is randomly chosen. For a population of size P, each gene x_i , $i = 1, \ldots P$, is assigned a fitness score, $f(x_i)$, according to the fitness function. Each gene is then assigned a probability of reproduction, p_i, $i = 1 \ldots P$, where the probability is proportional to the gene's fitness relative to all of the other genes in the population.

Selection of the new generation of genes is done in a directed random fashion: in general, the better solutions survive. However, there is always a chance that poor solutions can propagate to the next generation. This provides a means for breaking out of local minima. In the case of the placement problem, a solution in which none of the functional units which communicate with the crossbar are able to do so would yield an extremely poor cost value. However, other aspects of the placement may be superior to all other members of the current generation. Thus, it is kept around in the hope that its positive attributes may be spread to other solutions.

One example of selection is the roulette wheel approached: the new set of genes is chosen randomly, but the selection is weighted towards the better solutions. Refer to Figure 9 for a graphical depiction of this. Once this has been performed, a new set of offspring are generated by two main operators [Fog94]: mutation and crossover. Mutation modifies an existing gene by swapping items, flipping bits, or making some other sort of random change. Crossover chooses two genes (parents), then splices portions of the parents in order to create two children. The result, ideally, are new population elements with higher fitness values. An example of the simplest type of crossover, single-point crossover, is shown Figure 10.

The crossover operator, a distinguishing feature of genetic algorithms, serves as the primary means for searching through the solution space [Fog94]. The presence of this operator is what distinguishes genetic algorithms from all other types of optimization algorithms [Dav91]. The

primary purpose of mutation is to act as a background operator to ensure that all possible alleles can enter a population. In other words, the crossover operator seeks to optimize, while the mutation operator acts as a safety mechanism to escape from local minima.

The type of selection mentioned above, generally termed proportional selection, is well matched with a weaker mutation operator. Proportional selection enforces a relatively low amount of selective pressure upon the population, i.e. the degree to which undesirable elements of the population are excluded, compared with other popular types of selection mechanisms [Bac94]. This means that there is a stronger degree of genetic diversity, and thus the algorithm should theoretically explore the solution space more thoroughly. If the selection operator were stronger, then a much stronger mutation operator would be required.

The general steps for a genetic algorithm are [Rao94], [Dav91]:

```
Generate an initial population of size PSIZE;
while the stopping criteria has not been met do
    Calculate fitness statistics for each individual in the population;
    Select PSIZE parents probabilistically, based upon fitness;
    for I = 1 to PSIZE/2 do
        Pair two parents randomly without replacement;
        Crossover two parents, based on crossover probability, produce two
        offspring;
        Mutate each offspring based upon mutation rate;
    endfor
endwhile
```

Although the genetic operators are rather simply defined, their effects are subtly powerful. Rather than moving through the solution space randomly, the various population elements store a large amount of information about prior fitness values. This is exploited by the operators in order to converge on a satisfactory solution [Hol92].
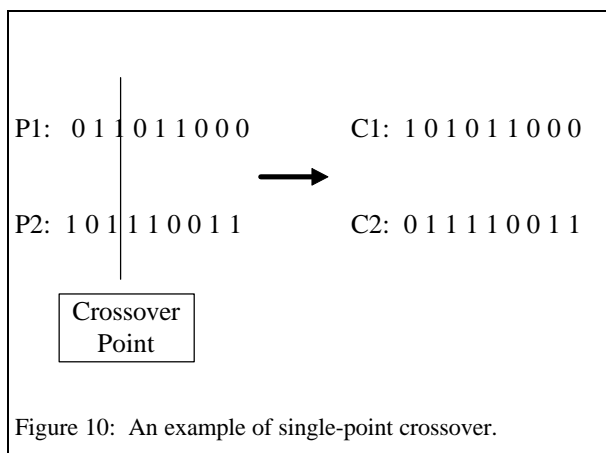
Figure 10: An example of single-point crossover.

One of the advantages of these non-deterministic search algorithms is that the general approach works for any kind of problem whose solutions can be represented as a gene. Lots of analysis of the problem, in the hopes of finding a solution algorithm, need not be done. Instead, all that is required is a good fitness function which can evaluate how good a particular gene is.

This fact is both a strength and a weakness. On the one hand, genetic algorithms can be easily adapted to a problem; all that is needed is an encoding scheme and a fitness function. However, it is difficult to study a GA rigorously, so when such an algorithm fails to perform well, it is frequently difficult to understand why that is the case [For93]. Fortunately, as will be shown in this thesis, a GA performed quite well for this particular problem. In order to significantly improve performance, though, it may be necessary to perform a more rigorous study of the behavior of the operators and fitness function used.

4.4   Genetic Algorithms For The Placement Problem

A genetic algorithm was chosen over simulated annealing to solve the placement problem because of two main advantages: the crossover operation and implicit parallelism. The implicit parallelism allows a genetic algorithm to explore the solution space from numerous starting points. Since parents are able to combine to form new offspring, crossover theoretically allows two placements which have certain positive aspects to share them in order to produce an extremely good offspring. Although simulated annealing would have been a valid choice for dealing with this problem, in practice genetic algorithms have been shown to outperform simulated annealing in some applications [Kwo94], [Rao94]. Future work might involve comparing the results of a simulated annealing approach with the results of this genetic algorithm compiler

4.4.1   Encoding and Genetic Operators

Originally, genetic algorithms were designed with the idea of using binary strings as a means of encoding [Fog94]. However, in a combinatorial problem such as this, binary encoding is not a natural means for representing the problem. Instead, what is desired is an encoding based upon ordinal values, where each value is unique in a particular population element.

For the *Tier2* compiler, the allele set is a range of integers from one to the maximum number of functional units on the chip -- currently sixteen. Each item in the allele set corresponds to a vertex in the dataflow graph representing the algorithm. The gene is an ordered list of the items in the allele set. Each item is unique and its position within the gene represents its position in the actual *Colt* chip. Mapping the gene to the functional unit mesh is very simple: starting from the upper-left corner, place the dataflow node (allele set element) into the mesh from left to right, top to bottom. If the algorithm does not require all of the available FUs, the gene still contains values representing them; they are simply empty placeholders.

For instance, the gene,

1	5	9	10	2	6	11	12	3	7	13	14	4	8	15	16

describes the data flow graph in Figure 8 as having Vertex 1 placed in the Colt FU at row 1, column 1. Vertex 5 is located in row 1, column 3, and Vertex 3 is located at row 3, column 1.

A problem arises with this implementation: using traditional genetic operators, illegal values would appear. For instance, the single-point crossover operator, as described above, would produce children which had multiple instances of ordinal values. A great deal of research has gone into solving this problem [Poon95], and some of the results have been used in this project.

The genetic operators are implemented in the following way: mutation is performed by randomly swapping elements. Selection is done using the roulette wheel approach. Finally, crossover is performed using the partial matching, or partially mapped (PMX) technique [Gold89], and maintains the uniqueness of the elements for the children. The result is that given a population of valid placements, i.e. one in which every data flow vertex is represented exactly once, these operators produce a new population of valid placements.

The PMX crossover operator has been shown to work well with various types of permutation problems [Cro95], [Poon95], [Wil95]. It attempts to preserve the absolute position in ordering in the offspring [Pos93]. Since the genes convey information through the ordering of the alleles set members, this is critical to the performance of the operator. Although this technique is very disruptive to the genes, it does manage to exchange useful information between the two parents [Wil95], and performs fairly well when compared against other permutation based operators [Poon94], [Oliv87]. In addition, it was a built-in operator with the genetic algorithm library used in this project, thereby reducing the overall implementation time.
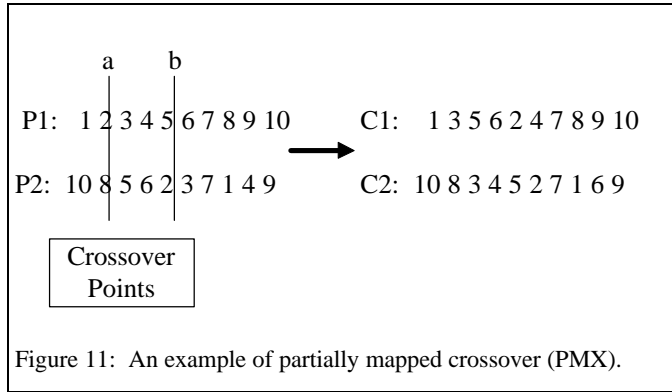
Figure 11: An example of partially mapped crossover (PMX).

An example of PMX is shown in Figure 11. The algorithm works by copying one of the parents to each of the genes. The two points, *a* and *b*, are chosen at random. Then, elements lying between points *a* and *b* are swapped, based upon the location of that item in the parent gene that the child was not copied from. For instance, *C2* is copied from *P2*. Next, the swaps are made. On the second swap, *P1[4] = 4*, so the value of *4* will be swapped into *C2[4]*. *C2[9] = 4*, so a swap is made between *C2[4]* and *C2[9],* resulting in the values shown in Figure 11. The swapping continues until the second crossover point is reached.

In order to support run-time reconfiguration, the compiler has the ability to exclude particular function units from being used for both placement and routing. The exact method by which exclusion is performed is discussed in more detail in Appendix B. In terms of gene expression, the method is very simple: the gene is simply shortened to the number of functional units available. Thus, if four units were excluded from placement, the allele set would contain only twelve items and the gene would consist of twelve unique items from the allele set. The fitness function, when it tiles the data flow vertices onto the physical graph just prior to routing, takes care of ensuring that the proper functional units are skipped. In the future, the exclusion capability will allow designers to create a library of pre-compiled algorithm modules which can be moved into the mesh as necessary.

These pre-compiled modules mesh well with the concept of run-time reconfigurability. As discussed in [Bit97c], an external device, known as an stream controller, might hold all of the aforementioned modules within its own memory. Input streams to this device would contain simple op-codes describing functions to be implemented, rather than the actual configuration words required to configure the array of *Colt/Stallion* chips. As each op-code is encountered, the stream controller would find the corresponding pre-compiled module, search for an empty space within the processing array, and configure the necessary functional units. A user wishing to create this library of modules would use the exclusion features of the *Tier2* compiler to restrict the place-and-route options when compiling each of the individual components. The modules would then be grouped together within the stream controller device for later use.

4.4.2   The Fitness Function

The fitness calculation, as mentioned earlier, is a measure of how good a particular solution to the problem is.   Since local connections between functional units are short in length and thus extremely fast, they are assigned a score of zero. Thus, a placement which contains only local connections would yield a score of zero- the best possible score.

Long skip bus connections are slow, since they involve passing electrical signals across long wires and through transmission gates. In this particular cost metric, each link in an edge implemented through the skip bus adds one to the score. The total score is a sum of the length of all of the edges in all four of the dataflow graphs associated with the algorithm.

As mentioned earlier, invalid placements generate a numerical score just as valid ones do. In order to differentiate between invalid and valid, an invalid score is equal to the total number of unroutable edges multiplied by a threshold value of 1000, which is far greater than the total number of routing resources within the chip. Thus, any score greater than 1000 represents an invalid placement.

A count of connections which are not adjacent to the crossbar, but need to be, is multiplied by a bias value of two, then added to the count of unroutable edges. This total is then multiplied by the threshold value. Functional units which must be adjacent to the crossbar are ones which must output data to multiplier ports or data ports. Even though the crossbar is capable of directly communicating with the skip bus, at least one data port must connect to a functional unit locally, or through a multiplier port. This is due to the fact that the skip bus cannot carry programming information. Thus, if all streams entered the *Colt* chip and were directed towards the crossbar, then the programming would stop and the chip would never be fully configured. Therefore, a value of at least 1000 will be produced if no functional unit communicates with a data port through a local connection.

4.4.3   Stopping Criteria

Every genetic algorithm must have some sort of stopping criteria which indicates when a good solution has been reached. This implementation uses three: if a score of 0 is found, then the GA stops immediately, since the score represents a solution using local connections exclusively. This is the best possible solution in terms of propagation delays.

The second criteria is convergence; it determines when a "good" solution has been reached. The best (minimum) score is divided by the average score of the entire current generation. If the value is greater than the convergence percentage, a parameter set by the compiler, then the algorithm stops. This is used to detect when scores have ceased jumping around wildly and have settled down to similar values. Although there is no guarantee that this is a particularly good solution, it generally indicates that the algorithm has found the best solution it will find for this particular evolution. Of course, the minimum solution must be below the threshold value in order for the algorithm to stop.

Finally, there always exists the possibility that no valid solution exists, or that the genetic algorithm is just incapable of finding one. For this reason, an upper bound is placed upon the

number of generations that the genetic algorithm will create. If no valid solution has been found by the time that this bound has been reached, the genetic algorithm stops and the compiler issues a warning message.

4.4.4   Routing

Since the cost function score represents the number of routing resources used, the fitness function must actually route a given placement in order to determine the final score. For each gene generated by the genetic algorithm, the placement is overlaid onto the chip. Each item in the mesh stores the source of the data it needs to receive (IFU coordinates, data port number, or multiplier number). Thus, four graphs exist, one for each type of data: bus output data,

Figure 12: An illustration of the maze-router attempting to avoid obstacles. An attempt to reach the destination through the shortest possible path failed, so the router tried the only other available path, then tried again to reach the destination.

conditional bit, *shift* bit, and *carry* bit. Once the functional units have been placed, the compiler attempts to route each of the data edges.

The local connections are attempted first. If two FUs are adjacent, and there is a dataflow edge between them, the local connection is set, and the data edge is marked has having been implemented. Next comes the non-local routing. Each FU is checked to see whether it is the destination of a dataflow edge. If it is, then a maze-router attempts to implement the connection from the specified source to the destination FU.

The maze-router starts at the source FU. Based upon the location of the destination, relative to the current location, the maze-router computes a direction in which to go (north, south, east, west) and then moves to that new FU. However, before the move is made, the router makes sure

that it is possible to implement a skip bus connection between the current functional unit that it is at and the new functional unit that it wishes to move to. It may be impossible to make the connection due to the fact that the skip bus resource might already have been used to implement another edge.

If the router is unable to travel in another direction, then it tries all other directions, then backs up to the previous functional unit in the path that it has traced out, and attempts to go in a different direction. Refer to Figure 12 for an graphical depiction of this process. In the worst case, all possible routes to the destination are tried. Generally, though, the algorithm quickly finds a solution.

Due to the fact that the routing problem has been shown to be NP-Complete [Rao94], an

The data is then routed North, from compass-right (East).

A functional unit within the *Colt* chip.

Data is routed across the bus from West to East.

Figure 13: Implementation of a left-turn on the Colt skip bus. Data comes from the west and is routed to the north.
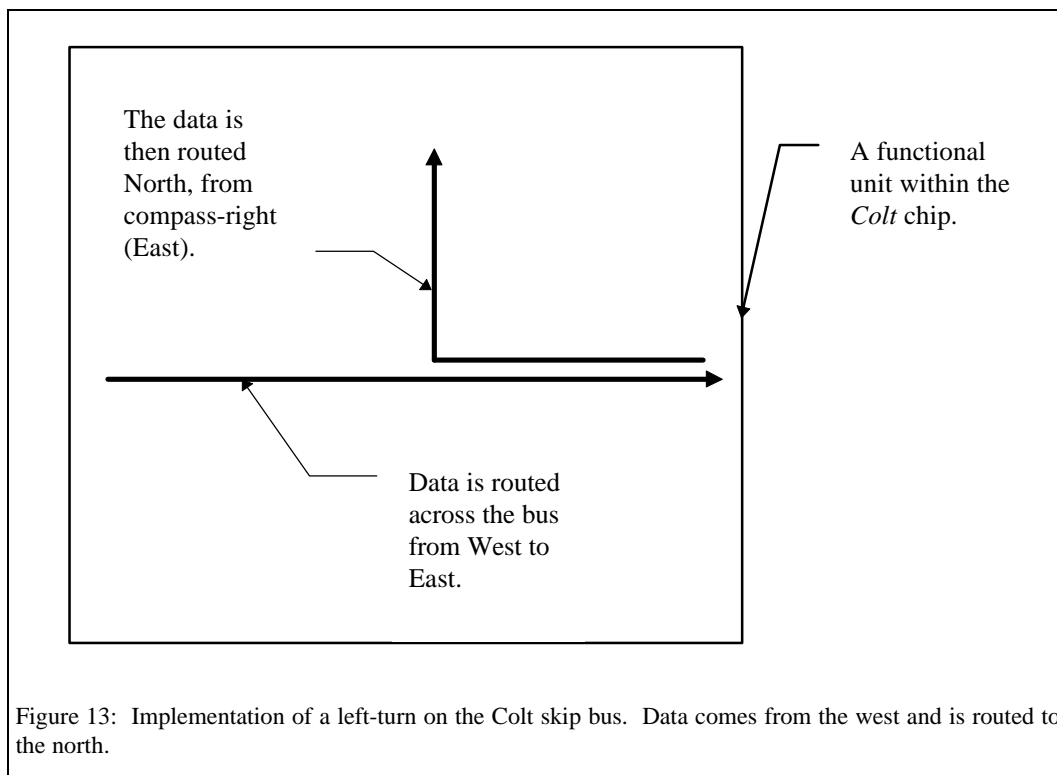
exhaustive approach is not a practical solution. Instead, the router does not attempt to actually solve the routing problem, but rather attempts to find a relatively good solution. Since the router attempts to minimize resource usage each step of the way, it is a greedy algorithm. Therefore, it suffers from the ability to fall into local minima. However, for this problem it performed well, as can be seen in the Experimental Results section. It also benefits from the fact that if it fails to obtain an acceptable routing solution, the genetic algorithm operating above it will discard that particular placement. Thus, the GA can help out the router by breeding solutions which the router is better able to handle.

Certain complications to the basic routing system arose due to the nature of the skip bus. As described earlier, two pathways exist: An east-west bus and a north-south bus. For a given

direction, north for example, the signal source can come from the opposite direction, south, or from the compass-right direction, east. However, it is not possible to directly route data from the compass-left. In order to do so, it is necessary to use one bus to route data across a bus, then use the other bus to route from compass-right. Refer to Figure 13 for an illustration of this process. The end result is that left turns are costly and should be avoided. The maze-router takes this into account by attempting to perform right-turns whenever possible. Note that the fitness calculation takes this bias against left-hand turns into account by adding a value of two to the total score for each left turn, versus a value of one for a right turn.

The main disadvantage of this algorithm is that it is greedy in nature. Therefore, a poor initial choice will rarely be corrected, but will instead force the router to use up extra resources towards the end of the path. Only if all subsequent paths fail will the early paths be modified. The advantage of this algorithm, though, is that it is very quick. Only one path is routed, and the router attempts to go to the destination using as few skip bus resources as possible. In addition, its recursive nature allows the maze-router to cover all possible paths, so it is a very complete search method. Further research might be performed on finding better ways to route signals, but for this first attempt at a compiler, the performance was quite satisfactory.

4.4.5   Analysis of the Fitness Function

By using a count of the number of routing resources for a fitness function, the genetic algorithm is able to accurately determine how good a particular placement is. This also eliminates the need for a later attempt at routing, since the job has already been performed. For a relatively small number of functional units, as in the *Colt* chip, performance is quite satisfactory. Please refer to the Experimental Results chapter for further analysis. The main question that arises is whether or not this method is scalable to larger architectures, such as the future *Stallion* chip. Since a genetic algorithm must try many potential placements, and evaluate the fitness of all of them, it is crucial that the routing algorithm be able to quickly implement all of the dataflow graph edges. Thus, an analysis of the routing algorithm is necessary.

Of the two routing phases, implementing local connections and implementing non-local connections, the prior can be neglected as contributing very little towards the overall computational time. It requires searching through each of the functional units, so it is an $O(n)$ algorithm, where n represents the number of functional units, but it can be done very quickly. The bulk of the processing time is spent routing non-local connections.

If every functional unit were to have a data edge which could not be implemented through a local connection, then there would be a total of n edges. In the absolute worst case, each functional unit would have edges for each type of data, for a total of *4n* edges. This represents an upper bound and is extremely unlikely to ever actually occur. The maze-router must traverse through nodes from source to destination. The mesh wraps around from east to west, but not from north to south, so the worst-case path length is equal to one length of the mesh vertically, plus half of the mesh horizontally. Since the mesh is a square, one length is equal to $\ddot{O}n$. Thus, the worst case path length is:

$$\sqrt{n} + \frac{\sqrt{n}}{2} = \frac{3(\sqrt{n})}{2}$$

Since each edge must be traversed by the maze-router, the total effort required has an upper bound of:

$$(4n)\left(\frac{3\sqrt{n}}{2}\right) = 6n\sqrt{n} = O(n\sqrt{n}) = O\left(n^{\frac{3}{2}}\right)$$

Therefore the algorithm scales fairly well. For instance, if a particular placing effort for the Colt chip, with *n=16*, requires a total of sixty seconds, then a Stallion chip with *n=100* would require a total of 15 minutes and 37 seconds. While not exactly fast, it is still within acceptable limits.

For future versions of this architecture, it is reasonable to assume that the number of functional units will stay within one order of magnitude of the current Colt chip. Therefore, the prior example is close to an upper bound on the limit to which the algorithm might have to scale. However, propagation delays for routing a signal from one end of a Stallion chip to another might be prohibitively high. As a consequence, not every functional unit will be able to reach every other functional unit within one clock cycle. Instead, each FU will have a radius of contact; a maximum path length that a signal may propagate within one clock cycle.

This fact improves the performance of the router in several ways. First of all, a pre-screening trial will be able to eliminate placements which violate the radii requirements. Such a process would require scanning through all of the functional units once- an *O(n)* algorithm. After that, the normal routing process would take place. However, each functional unit would now have a neighborhood within which signals could be routed to. Therefore, rather than the maze-router being an *O(Ön)* algorithm, it becomes an *O(r)* algorithm, where *r* equals the radius of connectivity. Thus, as the number of functional units scales up, the radius will stay constant, and the algorithm begins to approximate an *O(n)* algorithm, rather than an *O(n³ᐟ²)*.

The importance of having a computationally efficient cost function cannot be overstated. Since every population element requires a cost value in order for the genetic algorithm to operate, this function is executed numerous times. The bulk of the processing time is in fact spent evaluating this function. As a consequence, any increases in efficiency result in a large performance gain.

Although the results of the algorithmic analysis are encouraging, in the end optimizations will most likely be required in order to boost up the performance of the router in a larger chip. This analysis does show, though, that physical constraints on propagation delays place sufficient restrictions upon routing solutions so as to make this algorithm fairly scalable to larger architectures.

Chapter 5: *Tier2* Language Overview and Compiler Design

5.1 Syntax Overview

As previously discussed, the advantage of the *Tier2* language over the *Tier1* language is to remove the need for the programmer to explicitly map an algorithm onto the *Colt* chip. Whereas the *Tier1* language requires the user to manually implement data routing and program stream creation, the *Tier2* language has the programmer create an abstract dataflow graph, as in Figure 8. Gone is the need for explicit skip bus programming, explicit directions for data directions within assignment statements, and explicit directions for the flow of programming streams.

The overall syntax is very similar to the *Tier1* language. The only difference is that the skip bus programming constructs have been removed, crossbar statements have been modified, programming flow information is no longer specified at the end of a block definition, assignment statements have been modified, and stream definitions have changed.

A *Tier2* program now contains only a single stream definition. No port headers are required; any port described in the port definition section can be used within the stream body. If ports are used for both input and output, a compiler error will be issued. The general format for a Tier2 program is shown in Figure 14.

Crossbar statements have been modified slightly: the source for a crossbar statement can be either a functional unit, a multiplier port, or a data port. This remains the same as in *Tier1*. However, the allowed destinations have changed: only data ports and multiplier ports are valid. This is due to the fact that assignment statements within a functional unit definition will specify data sources. Thus, it is unnecessary to use a crossbar statement to perform the same task.

The basic block definition remains largely the same, except that the construct terminates with simply an `end`, rather than any sort of programming direction data. The general structure looks as follows:

```
block IFU_NAME
  <assignment statements>
  <macro calls>
  <component definitions>
end;
```

The ALU assignment statement remains identical, but the other assignment statements have been modified slightly: The `from <direction>` portion has been replaced with syntax identical to the source portion of a crossbar statement. For instance, a valid assignment statement is now:

```
rightreg = ifu propsouth;
```

In this case, the right register is set to receive data from a functional unit named `propsouth`. The compiler performs the task of making sure that the routing resources within the chip direct the data properly. Please refer to Appendix A for a more complete syntax reference.

5.2   Genetic Algorithm Code

The purpose of this project was to create a compiler, not to engineer a genetic algorithm class library.   Therefore, a public domain class library was used in order to reduce the overall development time. The system chosen was the GAlib 2.4.2 genetic algorithm package, written by Matthew Wall at the Massachusetts Institute of Technology [Wall97]. It is a very extensible, easy to use class library, and worked without requiring any modifications. Use of this tool greatly reduced the effort in bringing this compiler to fruition.

One of the main features of this package which made it perfect for the place-and-route application was its inclusion of operators designed to work with genes in which allele set members had to remain unique and all members had to be present in a gene. The genetic operators described earlier in this work were included in GAlib and therefore did not have to be separately designed and tested.

5.3   Software Design

In keeping with the idea of using the *Tier1* compiler as a stepping stone, the overall shell of the first project was used almost completely intact.   The grammar was modified using *Visual Parse++* and new C++ code was created. However, *Visual Parse++* is able to modify existing project files without destroying existing code. This feature is one of the greatest assets of the tool. Once the compiler had been modified to use only a single CStream class, and the tree-traversal listing system had been removed, the new class hierarchy for solving the place-and-route problem was moved in, and a new listing mechanism was added.

5.3.1   Place-and-Route Class Hierarchy

A great emphasis was placed on creating a class hierarchy which would be extendible to future *Stallion* architectures. Since the design of the future chips has not yet been set, it was decided to hard-code the routing abilities of the *Colt* chip, but in such as way as to make it easy for future software
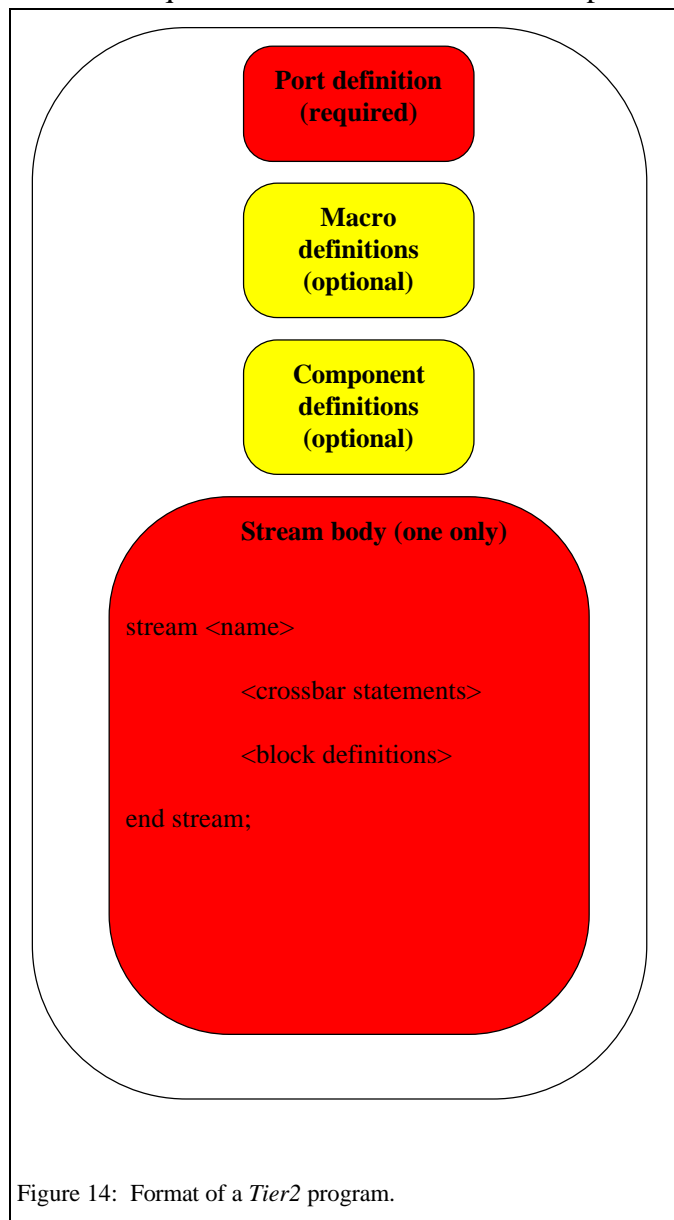
Figure 14: Format of a *Tier2* program.

designers to modify the classes to work with the *Stallion* chip.

This was done by layering the various aspects of the place-and-route algorithm. At the base layer is the `CGraph` class. Its function is to specify the mesh topology, such as the number of rows and columns, and to store dataflow edge information for a specific placement. As an example, it stores the fact that a functional unit at Row 2, Column 2 requires data from a functional unit at Row 1, Column 4. This class is capable of storing two source requirements per functional unit. This allows it to work with the fact that each FU can accept data into a left and right operand. The same class is used to work with the bit flags; only the left side is used in such a case. Assuming that a future *Stallion* architecture contains a mesh of functional units, and that each functional unit can receive data from at most two sources, then this class will require few, if any, changes.

The `CRoute` class is derived from the `CGraph` class. It uses the underlying edge information to perform the routing task. It contains a list of routing resources available to the chip. This is used by the maze-router, a method within this class, to route all of the edges. In addition, it performs the important task of creating programming streams. These streams are simply ordered lists of resource names, such as "data port six", "functional unit at row two and column 1", etc. The task of converting a tag

Figure 15: The Tier2 place-and-route class hierarchy.

Figure 16: General structure of the *Tier2* compiler.
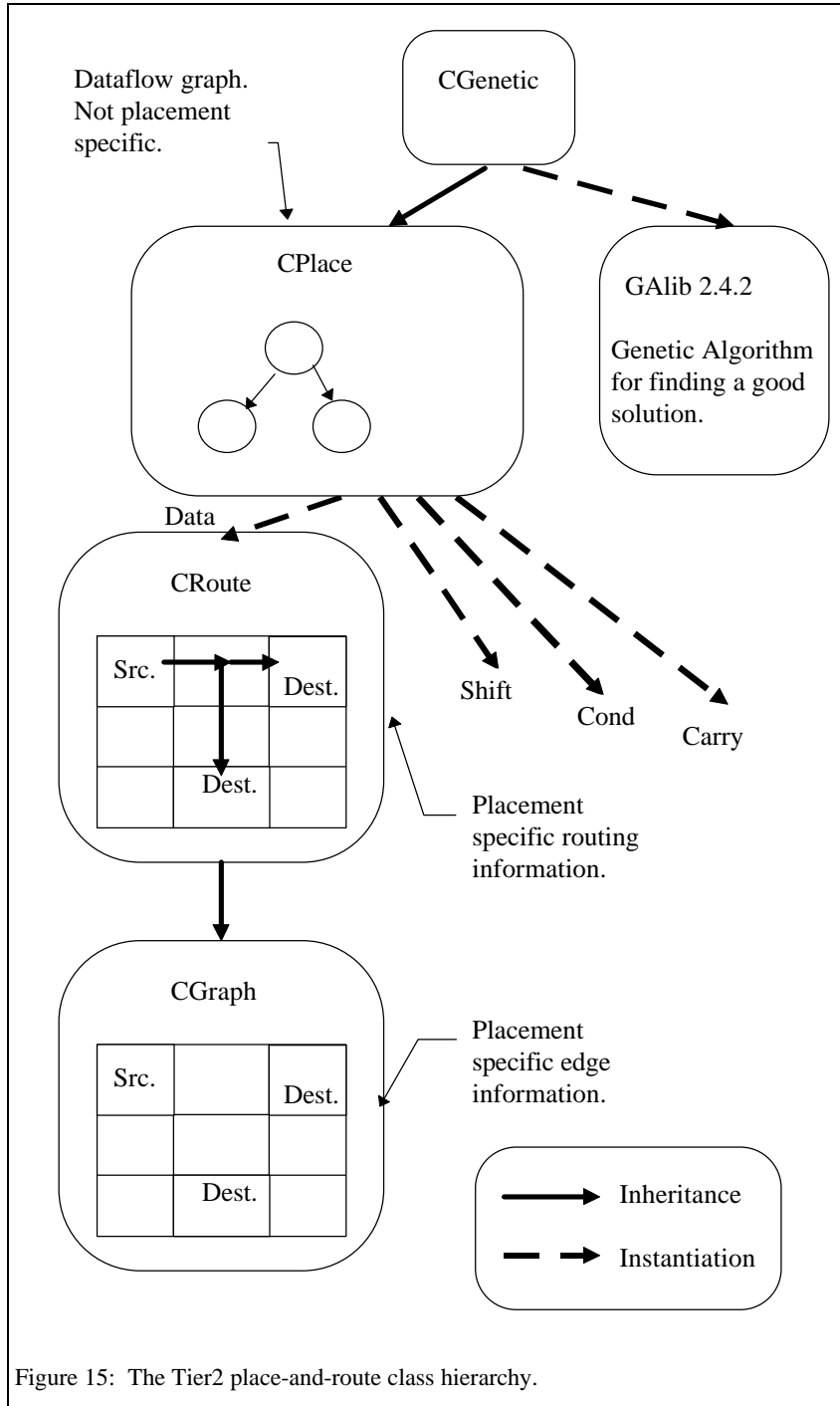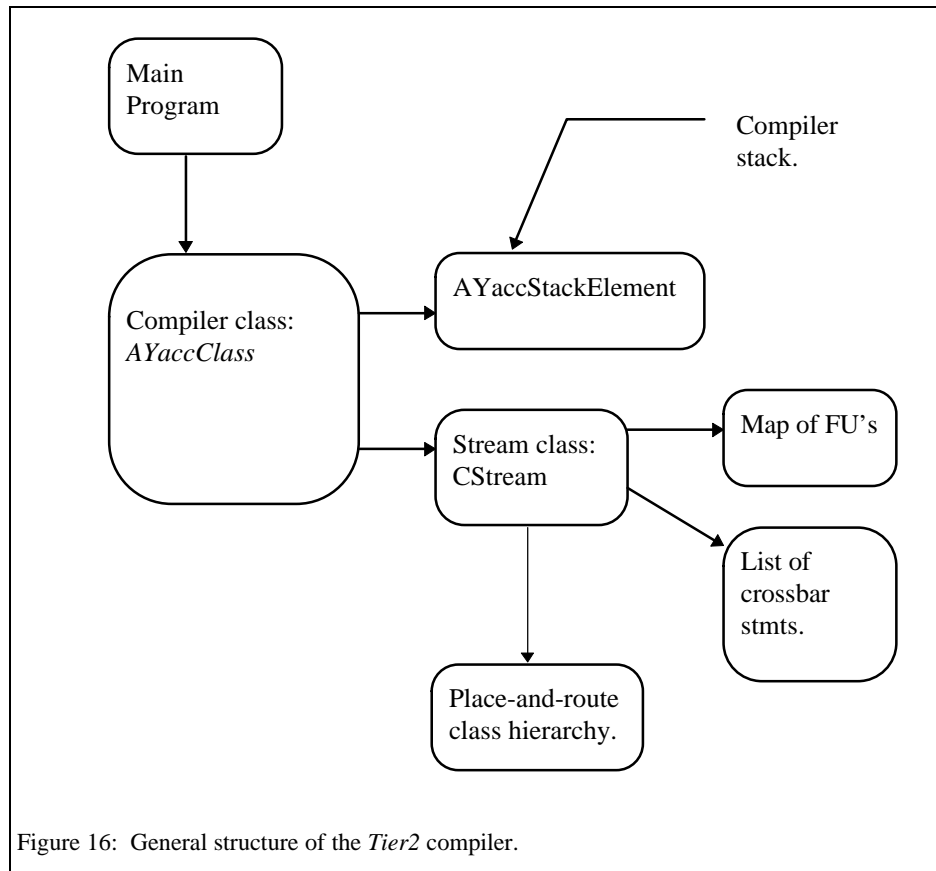
to actual `dfc` code is handled at a higher level.

This class is tied specifically to the *Colt* architecture, in that the list of routing resources available matches the chip's capabilities. Future architecture changes will require modification of some of the data structures and changes to decisions that the maze-router makes. It is difficult to predict how severe the changes will be, but the general structure and architecture of the maze-router will probably not change by much.

The prior classes worked with a specific placement and a single data type. The `CPlace` class instantiates four `CRoute` classes to handle the four types of data: bus output, *carry* bit, *shift* bit, and conditional bit. The class itself stores the abstract dataflow graph generated by parsing the *Tier2* program file. It is able to accept a placement description, i.e. a gene from the genetic algorithm, map the dataflow graph onto a specific placement using the gene, then call the routing routines in order to generate a fitness score. If the *Stallion* architecture adds future data types, little will need to be changed in this class. More `CRoute` class will have to be instantiated, but the dataflow graph has been designed to handle an arbitrary number of edges between vertices, so expansion of this data structure will be relatively easy.

Finally, there is the CGenetic class. It is derived from the CPlace class and instantiates the GAlib genetic algorithm class. The underlying CPlace class is transparent so that classes which instantiate the CGenetic class can directly communicate with the CPlace class in order to add edges and vertices to the dataflow graph. Once the graph has been specified by the compiler by parsing the *Tier2* code, a method is called which instantiates a genetic algorithm and attempts to evolve a solution. During the evolutionary process, the routing functions within the CRoute classes are called by the fitness function contained within the CPlace class. The result is that the best score and best placement are stored for later use. Programming streams based upon this placement can then be generated by the CRoute class and accessed by helper methods.

Figure 17: Depiction of the synthesized programming stream for the sample data flow graph in Figure 8. The left-hand side depicts the stream pathway through the configurable resources, while the right-hand side shows the actual stream structure for this example. Note that in this example, the stream must split at Vertex (1,1).
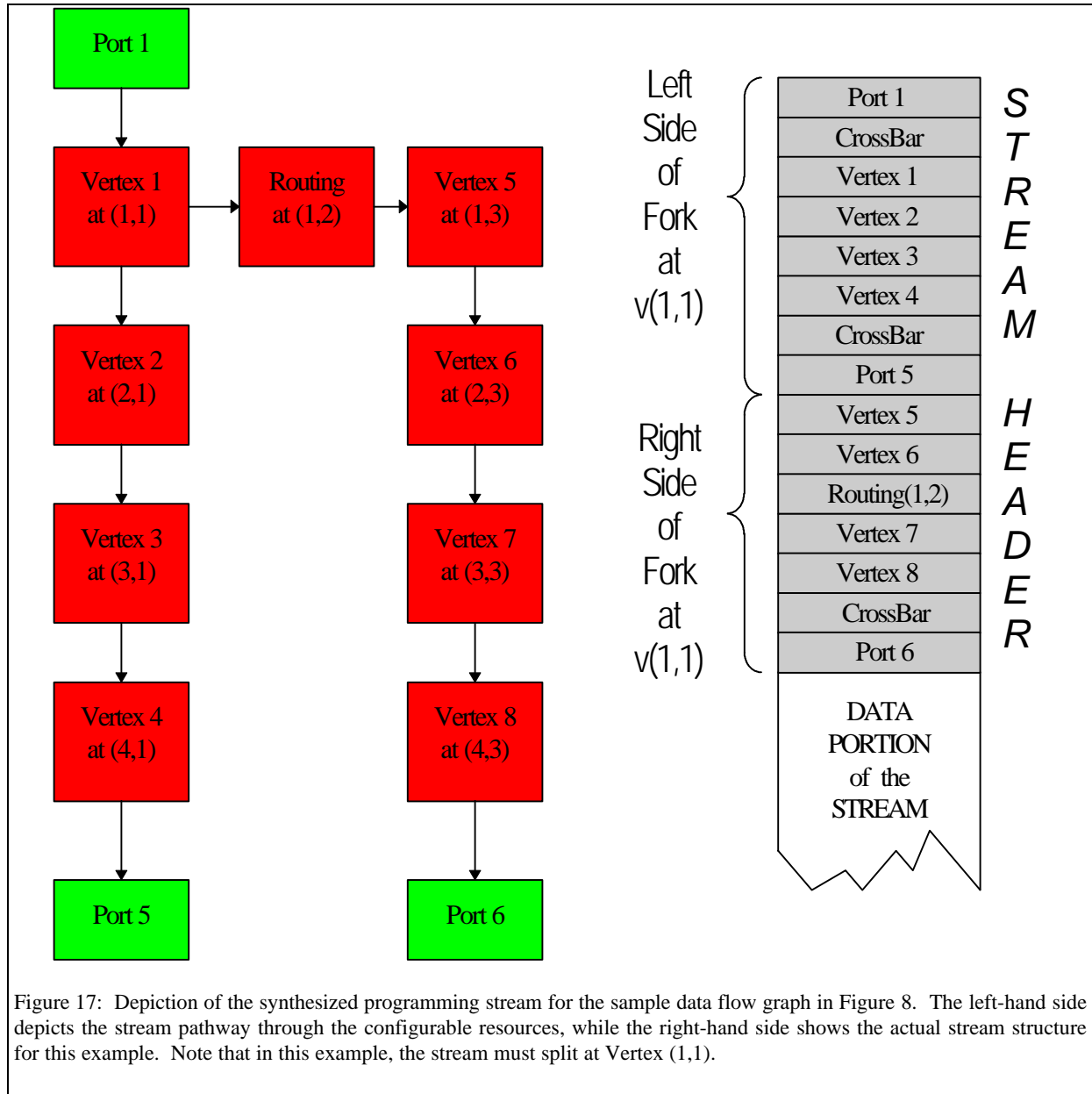
Architectural changes to the hardware will necessitate few changes to this class.

The main compiler class, `AYaccClass` converts the programming list into actual `dfc` code by accessing its stored database of functional units and crossbar statements. The process is simple: once the lists have been generated by `CRoute`, the list for each data port is retrieved. The list is traversed in order and each item in the list is searched for in the hash tables stored within the `AYaccClass`. The item's routing resources for each type of data are then configured, as specified by the various `CRoute` classes, and the object is written to the correct output port file. Please refer to Figure 15 for a graphical description of the place-and-route class hierarchy. Figure 16 shows the place-and-route hierarchy in relationship to the entire project.

The listing phase for *Tier2* is similar to that in *Tier1* in that streams may split within the chip. In some cases, the programming stream barely follows the path of actual data. This is caused when data streams from ports enter the crossbar and then enter the mesh through the skip bus. In such as instance, programming data from that stream can proceed no further. Instead, a programming stream from some other local connection must configure these functional units. Figure 17 demonstrates the programming sequence for the dataflow graph shown in Figure 8 and the placement described in the following gene:

1	9	5	10	2	11	6	12	3	13	7	14	4	15	8	16

Notice that the functional unit at Row 1, Column 2, identified as Vertex 9 in the gene, is not really used within the actual dataflow graph. It is simply used to route information, through the skip bus, from Vertex 1 to Vertex 5. Note also that the stream splits at Vertex 1. The programming stream is routed both south and east by this functional unit. However, the information for the southern route is ignored by the eastern route because none of the packets match the address for the FU at (1,2). When that packet does arrive, the data proceeds east, then south. Following that, the data streams through the chip and the actual processing commences.

Chapter 6: An Example Application

In this chapter, a relatively complex example algorithm is mapped to the Colt chip using the *Tier2* language and compiler. The algorithm itself is first introduced, then the abstract dataflow graph is presented. Next, a potential mapping is shown, which was produced by the *Tier2* compiler. Finally, the streams required to configure the chip are graphically depicted.

The application which was used throughout the development process of this compiler was the floating point multiplier developed in [Bit97b]. This program uses all of the IFUs within the *Colt* chip and requires a large number of connections, including the use of flag bits. The multiplier is discussed in-depth in [Bit97b], but a brief overview of its operation will be discussed here.

Two 32-bit floating point numbers are multiplied together to produce a 32-bit result. The format of the data is as follows:
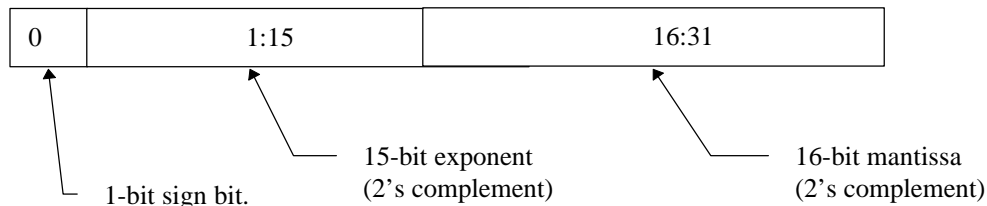
The exponents of the two operands enter the chip through data ports one and three. The mantissas enter through data ports two and four. The mantissas are multiplied together using the hardware multiplier. The high word is kept as the result, but is conditionally shifted to the left if normalization is required. In such a case, the high bit of the low word of the multiplication result is shifted into the low bit of the high word. If normalization takes place, then the exponent of the result is decremented by one.

The exponents are handled by first shifting out the sign bits using a left shift. These sign bits are XOR'd together and shifted back in at a later time. Addition of the exponents occurs next, followed by a check for overflow. If overflow occurred, the smallest possible exponent is loaded in. Next, the exponent is decremented if the mantissa was normalized. A final check for overflow occurs, and finally, the sign bit is shifted into the exponent word using a right shift. For a more detailed explanation, please refer to [Bit97b].

The initial placement by the original creator, Ray Bittner, required approximately two weeks and resulted in a score of 20. As will be shown, the *Tier2* compiler is able to accomplish the same task in far less time, generally around one minute, and is able to obtain a score roughly as good, and frequently better.

The following is the *Tier2* version of the floating point multiplier.

```
//
// tier2 implementation of the floating point multiplier.
//
```

```
// Floating point multiplier, as described in Ray Bittner's
// dissertation.  This is the "improved" implementation-
// the mantissa and exponent outputs are synchronized.
//

ports
 exp1 = 1;        // Exponent 1
 man1 = 2;        // Mantissa 1
 exp2 = 3;        // Exponent 2
 man2 = 4;        // Mantissa 2
 eout = 5;        // Exponent output
 mout = 6;        // Mantissa output
end ports;


stream FPMULT

// First mantissa just goes through the multiplier
// and ends at the skip bus entrance to the rightmost
// column of IFUs
 port man1 => mult at 1;

// Second mantissa goes through the multiplier,
// then programs the rightmost columns of IFUs to
// perform the renormalization task.
 port man2 => mult at 2;

 // shifts the low word to the left by one,
 // passes the highword of the multiplier through
 // the skip bus.
 block shiftlow
         leftreg = mult at 2;
         out = left << 1;          // bus output is actually ignored

         // Take cond from left, pass it thru ALU, then send it south
         // so that it can be used to check for overflow from the
         // exponent addition
         cond = ifu addexp;
         carry != cond;
         out = grabcarry;
         // We've moved the carry into the data word.  Now we'll pass it south
         // and extract it from the data word.
 end;

 // passes shift and bus output south
 block pass_shift_and_out
         // Now we have to extract the carry bit from above
         // First, load in the data word
         rightreg = ifu shiftlow;
         // Extract it to cond
         condout = rightop(0);
 end;

 // Here's where the renormalization is done.
 // The highorder word is latched into both operands, and
 // the highorder bit determines whether a shifted or
 // non-shifted version should be used.
 block renormalize
         // clock in the highorder bit
         leftreg = mult at 1;
         rightreg = mult at 1;
         shift != ifu shiftlow;
```

```
        condout = rightsign;    // conditional bit = bit 16
                                // of right operand
        cond != condout;

        // If the conditional bit is true, then we take
        // the left-shifted-by-one version, else just
        // take the unshifted version
        out = left << 1 ifcond, passleft;
end;

// Double delay of the bus output
block delay2
        rightreg = ifu renormalize;

        cond = zero;
        // activate the delay- force the ALU to be bypassed
        out = delay ifcond 0 else right;
end;

ifu delay2 => port mout;

    // Left exponent stream (goes straight down column 1)

// This block extracts the sign bit from the first exponent word
block sign1
        // Left word comes from crossbar
        leftreg = port exp1;
        rightreg = port exp1;   // simulator fix only
        // Shift once to the left
        out = left << 1, passleft;
end;

// Shifts in the sign bit for the right exponent
block insertsign1
        shift != ifu sign2;
        out = delay 0 << 1, passleft;
end;

// Computes the new sign bit
block signcalc
        rightreg = ifu insertsign1;
        leftreg = ifu insertsign2;
        out = delay add;
end;

// Shifts in the sign bit for the left exponent
block insertsign2
        shift != ifu sign1;
        out = delay 0 << 1, passleft;
end;

    // Right exponent (goes down column 2, then down column 3)

// Shifts to the left once to remove sign bit
block sign2
        leftreg = port exp2;
        rightreg = port exp2;
        out = left << 1, passleft;
end;

// Southern stream (column 2)
```

```
        // Decrementing the exponent is required due to a mantissa change
        block decrexp
                rightreg = ifu checkoverflow;
                cond = ifu renormalize;
                out =   ifcond
                                65534, add      // Add -2
                        else right;

                // Save the overflow result
                condout = aluoverflow;
        end;

        // Propagate data south, add on a delay to the cond bit
        block propsouth
                // The cond bit from above (alu overflow) propagates south and
                // is moved into the ALU
                cond = ifu decrexp;
                carry = cond;
                out = grabcarry;
        end;

        // checks overflow from decrement operation
        block checkovr2
                rightreg = ifu decrexp;
                cond = ifu prop2;
                out =   ifcond
                                32768, passleft
                        else right;
        end;

        // Eastern stream (column 3)

        // This adds the exponents together
        block addexp
                rightreg = ifu sign2;
                leftreg = ifu sign1;

                out = add;
                // send condout to the east in order to put in a delay
                condout = aluoverflow;
        end;

        // check for overflow.  If it has occurred, then we set the value to
        // the smallest possible exponent.
        block checkoverflow
                rightreg = ifu addexp;
                cond = ifu pass_shift_and_out;
                out =   ifcond
                                32768, passleft
                        else right;
        end;

        // The left register extracts the carry bit (which is actually the
        // overflow bit from the decrexp ifu)
        // This also adds a one clock-cycle delay to the sign bit data
        block prop2

                // Get the lowest bit from the register and put it onto the skip bus
                rightreg = ifu propsouth;
                condout = rightop(0);

                // The sign bit has to be converted into data here
                leftreg = ifu signcalc;
```

```
            out = passleft;

            // We're not really using data from the right side,
            // so the valid bit should only be tied to the data
            // on the left.
            validleftonly;
    end;

    // Combines the sign bit and exponent data together
    block prop3
            // Exponent data
            leftreg = ifu checkovr2;
            // Sign data
            rightreg = ifu prop2;

            // Grab the low-bit from the right register
            condout = rightop(0);
            // Move it over to the shift bit
            shift = condout;

            // Shift in sign bit, then pass straight thru ALU
            out = left >> 1,passleft;
        end;

    ifu prop3 => port eout; // direct the exponent to the output port.

end stream;
```

After this file is parsed by the compiler, the `CPlace` class contains a dataflow graph of this algorithm. This graph is presented in Figure 18. Inside of the compiler, the data flow for the four different data types are stored separately. In this graph, however, they have been included in one figure for clarity's sake.

The names of the block definitions in the floating point multiplier program are identical to those in the illustration. The overall functionality of the algorithm has already been explained, but a few additional notes are needed to explain some peculiarities of the *Colt* architecture. First, the signs bits are shifted out of the exponent words in vertices `Sign1` and `Sign2`. The sign bits are then shifting into the low bits of data path in vertices `InsertSign1` and `InsertSign2`, then XOR'd together in vertex `SignCalc` using the ALU. The result is shifted into the final exponent in vertex `Prop3`. The `Prop2` block serves two purposes: it acts as a one-clock cycle delay for path equalization purposes, and it acts to delay by one clock cycle the conditional bit produced by the vertex `DecrExp`. `Pass_Shift_And_Out` serves the same purpose for the conditional bit produced by `AddExp`. These conditional bits are high if the ALU detects an overflow. However, this bit is produced immediately even though it is actually needed one clock cycle later. In order to delay the signal, it is shifted into a functional unit, passed to another functional unit, and shifted out again. This is a significant waste of resources and will be removed in the next version of the *Colt*/*Stallion* architecture.

The rest of the dataflow graph is fairly straightforward: the exponents are passed from `Sign1` and `Sign2` to `AddExp`, where they are added, then passed to `CheckOverflow` to check for an overflow condition. After that, the exponent is decremented if `Renormalize` signals that it

Figure 18: Data flow graph for the floating point multiplier. Names of vertices are the same as those used in the floating point multiplier program.
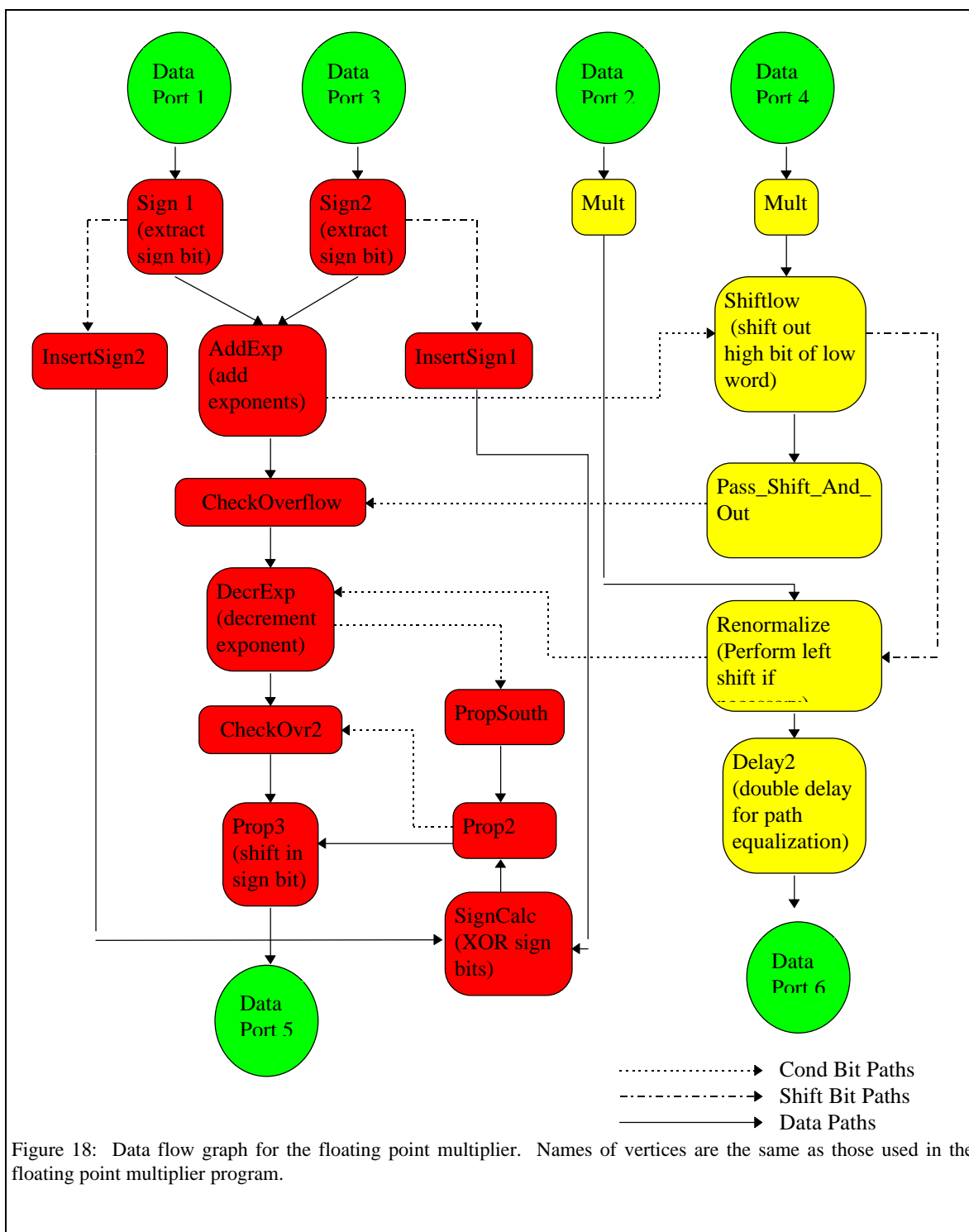
performed a shift operation. Overflow is checked for once again in `CheckOvr2`, then the sign bit is shifted back in in `Prop3`. The mantissa side is equally straightforward: the two mantissas are multiplied together and the high word is passed to `Renormalize`. The low word's high bit is shifted out in `Shiftlow`. `Renormalize` performs a shift if necessary, then passes the result to `Delay2` for a two clock cycle delay. Finally, the resulting values are passed through the crossbar to the output data ports.

Once this dataflow graph has been extracted from the program file, and the all of the various constructs within the file have been parsed correctly and stored within the compiler's internal data structures, the genetic algorithm attempts to find a suitable placement. One such result is the placement shown in Figure 19. The particular solution had a score of 21 and had an execution
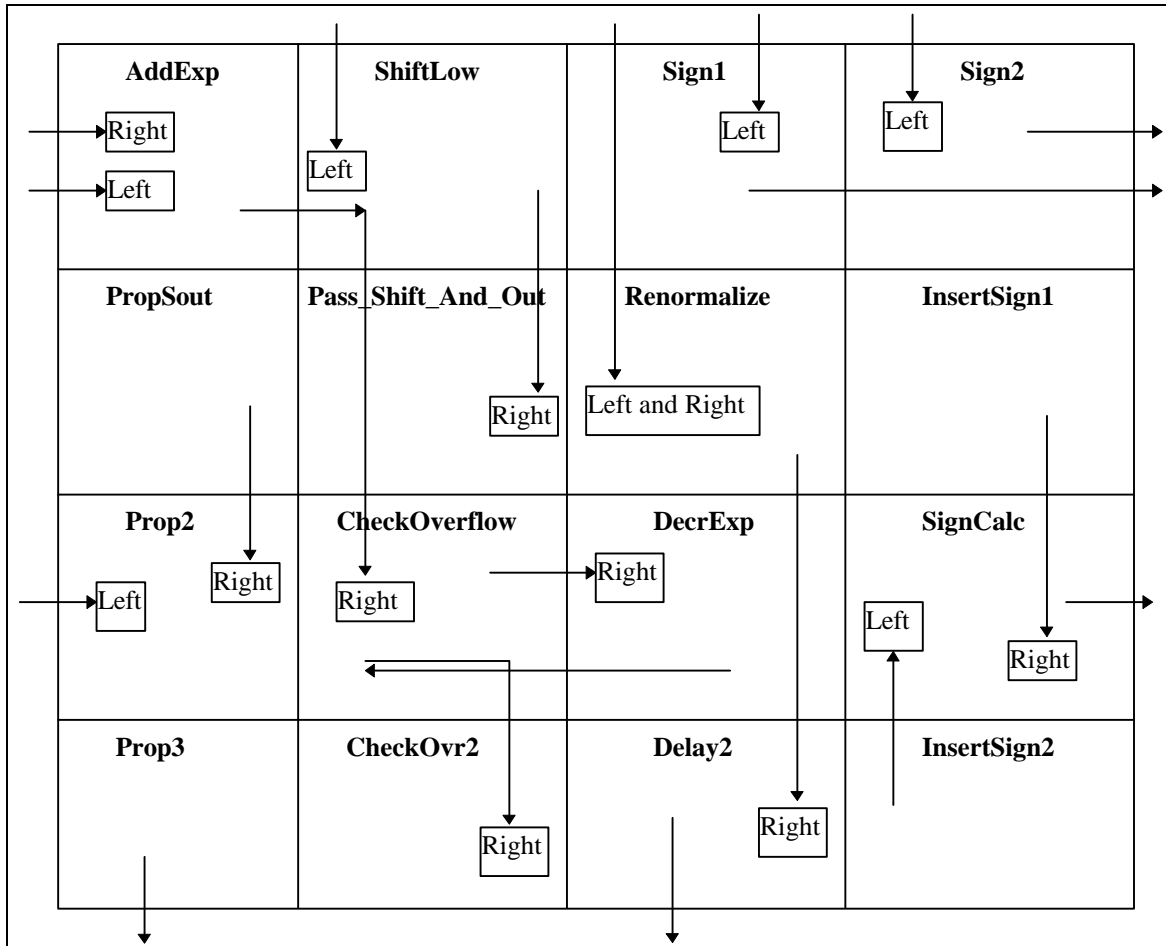
Figure 19: A sample placement for the floating point multiplier (score is 21). The routing resources are shown for the data bus. *Shift* and conditional bit paths have been excluded for clarity. `Left` and `Right` refer to the left operand and right operand data registers, respectively.

Arrows emanating from the top of the mesh are data paths from the crossbar. Likewise, arrows pointing out of the mesh from the bottom are data paths going into the crossbar. The east and west sides of the mesh are connected together, so arrows extending off of the right-hand side connect directly to the arrows entering from the left side.

Figure 20: Programming streams for the placement shown in Figure 19.
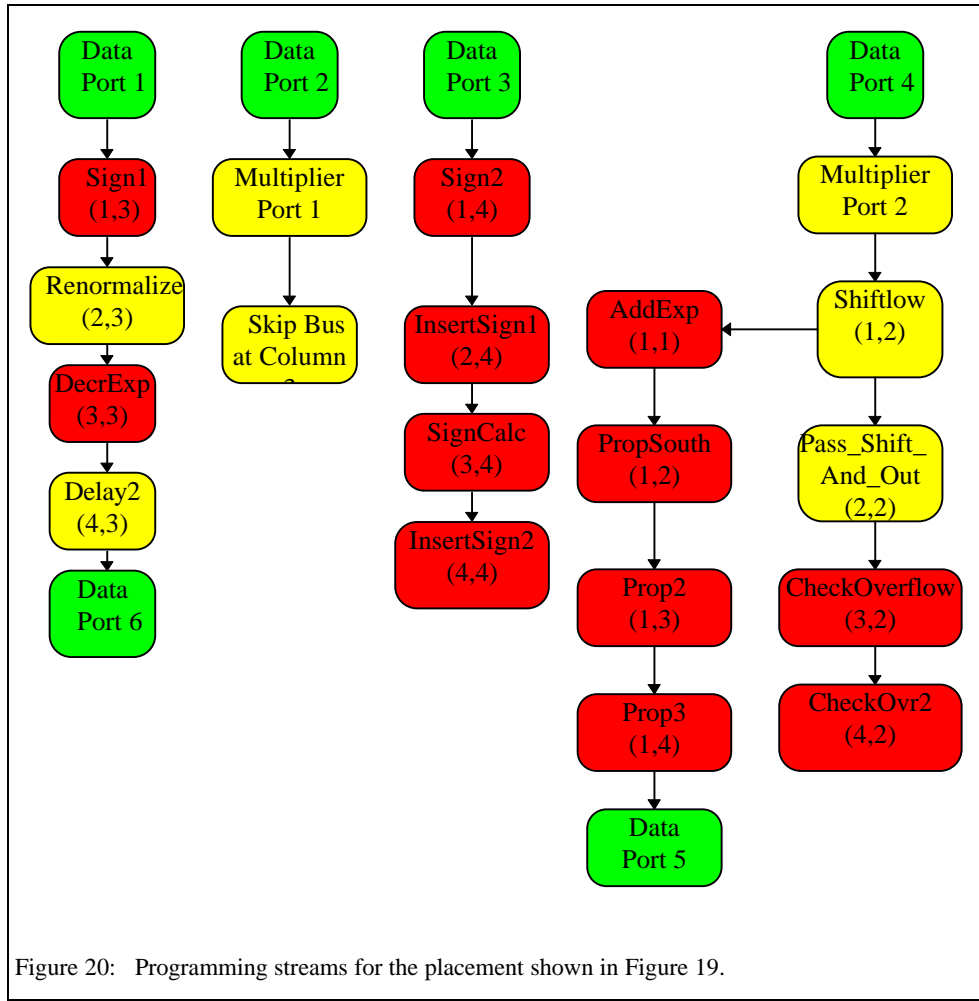
time of 19 seconds. Thus, the result required one more skip bus resource than the original, manually laid-out attempt, but required far less time to complete. Other solutions have been found which required even fewer resources.

The routing resources required to implement the data bus connections for this particular placement are also shown in Figure 19. Multiplier and data ports are not shown, only the functional unit mesh. The programming streams are shown in Figure 20. As can be seen, the flow of programming information does not exactly match the flow of data through the chip due to the restriction on programming data being carried over the skip bus.

The next section, Experimental Results, discusses the overall performance of this compiler in more depth. However, this example illustrates how effectively a genetic algorithm can solve a difficult problem without actually knowing anything about the problem itself.

Chapter 7: Experimental Results

In order to judge the efficacy of the compiler, two programs were selected for study: the floating point multiplier and a simple two-column summation program. The dataflow graph for the column program is shown in Figure 8, where every vertex in the graph adds a constant value to the input data. The reason for choosing the floating point multiplier was to rigorously test the compiler using a complex algorithm, while the columns program was chosen because it possesses an obviously optimal placement solution: every edge can be implemented using only local connections.

Both *Tier2* programs were run with a repetition value of five hundred, i.e. five hundred independent placement trials. Unless noted otherwise, the percentage chance of crossover was set at 75%, mutation was set at 30%, convergence was set at 70%, and the population size was set at twenty. A Pentium Pro 180 with 96 MB of memory running Windows 95 was used as the computing platform.

The results show that on average, performance of the compiler is not as good as a human manually performing placement. However, the minimum score is equal to, in the case of the columns program, or superior to, in the case of the floating point multiplier program, the manual placement score. For example, in Chart 1, the results of fifty trials are shown. As can be seen, the minimum score is 19, one routing resource less than the manual score. However, for the full five hundred trials, the minimum score was a low of 14, which is 30% lower than the best manual placement of 20. The cumulative average and minimum, after every fifty attempts, is shown in Chart 2. Since the lowest score produced is the solution used for the final placement, this means that the compiler ends up outperforming manual placement.

These results suggest that the more trials are made, the more likely an extremely low score will be discovered. This makes sense, considering as how the search is non-deterministic. It would be optimal if the compiler could find the best solution every time, but it appears that the genetic algorithm can become trapped in local minima. On average, the local minima is slightly worse than what a manual placement could produce. However, over enough trials, the minimum score is much better.

The problem of premature convergence, before the optimal solution has been found, is frequently observed in genetic algorithms due to the exponential reproduction of the best chromosomes combined with the functioning of the crossover operator. Once a population has converged to the point that crossover operations simply cycle through a reoccurring set of genes, and a very large mutation would be required to break out of this predicament, some literature recommends that the run be stopped [Fog94]. This is the technique used with *Tier2*: convergence is detected and the algorithm stops. Another run is begun in the hopes that it will convergence in an even better solution. Another method for handling this is to use a hill-climbing method, or greedy method, to search for a minima which lies in close proximity [Fog94]. This might be a promising area of research for future compiler versions.

Chart 1: Floating point multiplier placement results for fifty attempts.
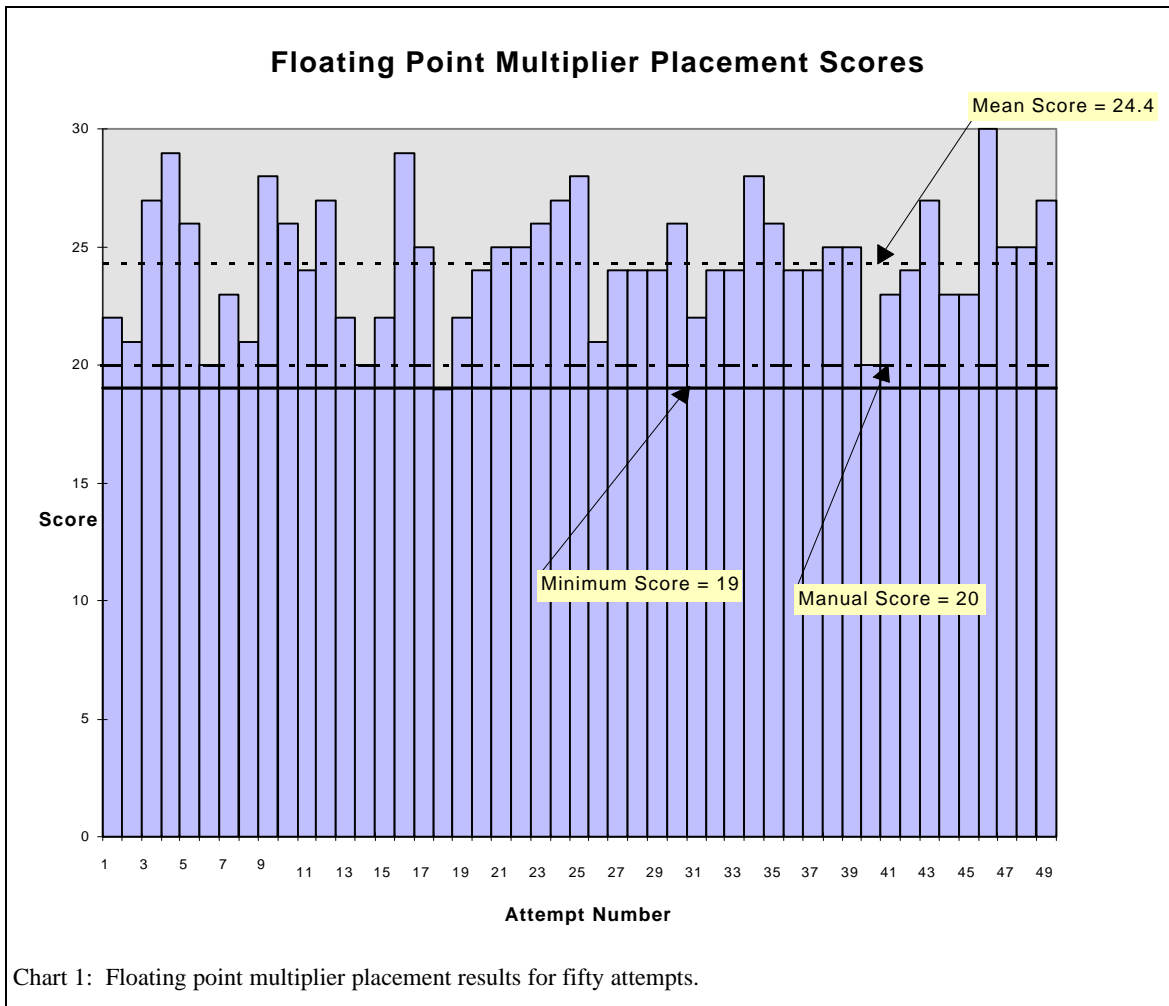
Chart 3 shows the execution times required for each placement attempt. As the arrows show, the best placement time took the longest, while the worst placement time took the shortest amount of time. Since the stopping criteria for the genetic algorithm, in this case, is convergence, the results indicate that the best placement occurs when the algorithm wanders around within the solution space for the longest time, avoiding relatively poor minima. When a minima is encountered, the scores of the population drift towards a common value and eventually converge.

Since mutation seems to be the primary means for keeping a population from easily slipping into a local minima, an experiment was run to see what the effects are of changing the mutation chance. The columns program was used for this experiment in order to keep the runtimes low and also to allow for the possibility of placement even for situations in which the genetic operator probabilities made it unlikely for a good placement to take place. At first, the experiment was run using the floating point multiplier, but it was soon discovered that for certain combinations of probabilities, no solutions were found. The columns program is much less restrictive in its placement requirements, and so contains many more potential solutions than does the floating

point multiplier. For this test, the crossover percentage was kept at 75% and convergence was set at 70%. Each mutation value was run for fifty independent placement attempts.

Chart 4 shows the average placement score and runtime for the mutation experiment. Notice how zero mutation produces extremely poor results: not all populations converged, even for such a simple case as the columns test. This is because the genetic algorithm's ability to jump out of local minima was effectively shut off. The optimal mutation rate was 10%; subsequently, the average score rose and the average execution time also rose dramatically. This demonstrates that mutation is really not the primary operator involved with finding good solutions. Rather, it is simply a safety mechanism which allows for escaping from local minima. If the mutation rate is too high, then the algorithm degenerates from a directed search into an inefficient random search.

Chart 5 is a similar experiment, except that the convergence criteria was changed, while the mutation probability was held to a constant value of 30% and the crossover probability was held at 75%. As can be seen, the higher the convergence requirement, the better the score. However, the higher the placement time. This is a fairly obvious finding: a stricter convergence requirement means that more generations will be required in order for the scores of the genes within the populations to approach a common value. For more complex problems, the convergence value will probably have to be kept low, at least at first, in order to find out whether a solution actually exists. Later, once this has been ascertained, a large value can be used in order to attempt to find

Chart 2: Average and minimum score (cumulative) for five hundred placement attempts.
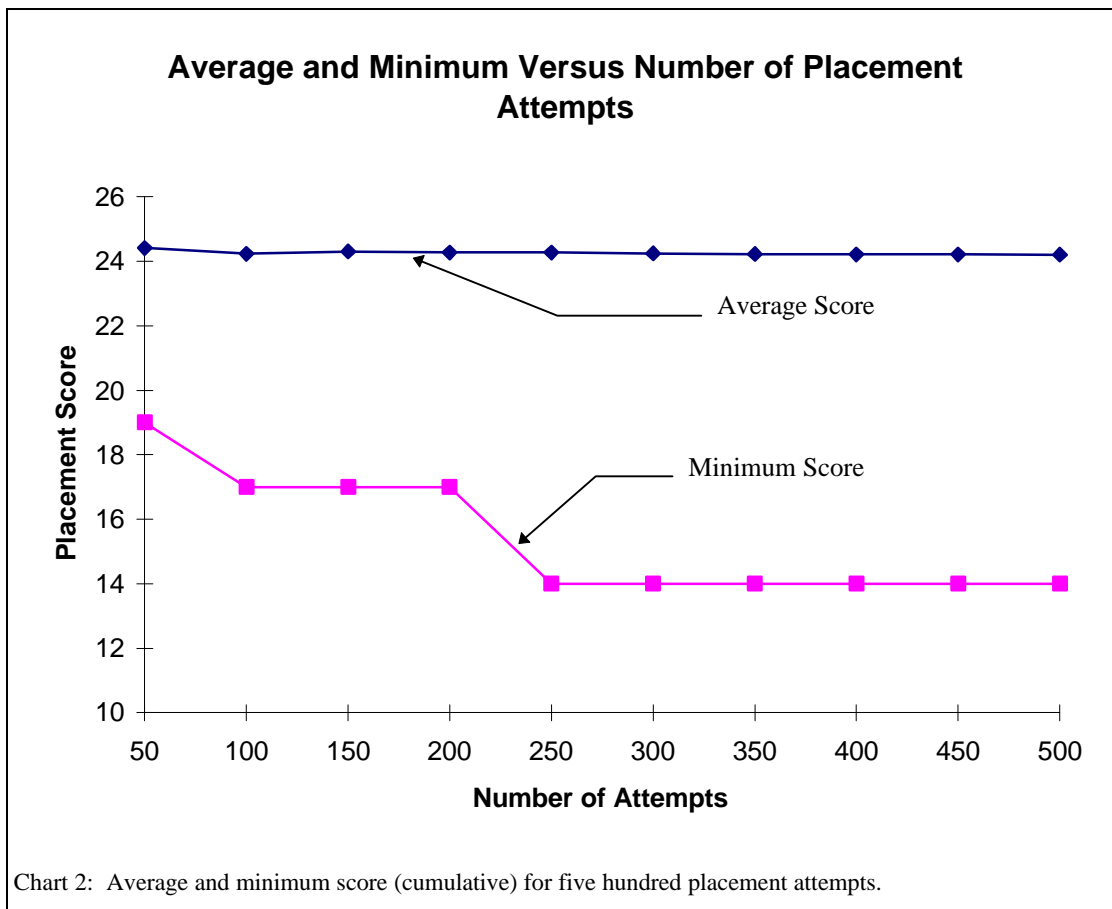
the best possible placement.

Of course, many other possible experiments could be run, including modifying the crossover probability, the method for selecting new population elements, etc.   The ones illustrated here simply demonstrate a few of the important aspects of the genetic algorithm.
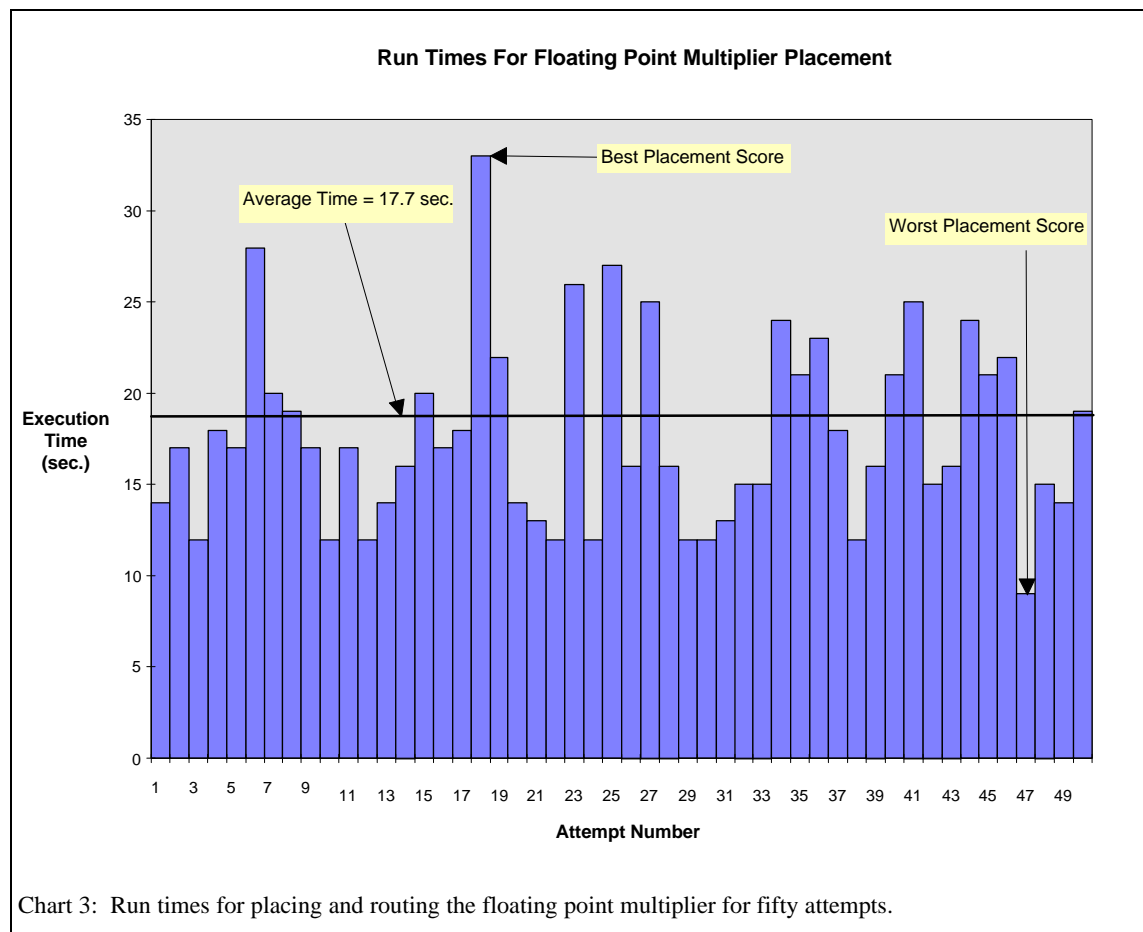
Chart 3: Run times for placing and routing the floating point multiplier for fifty attempts.
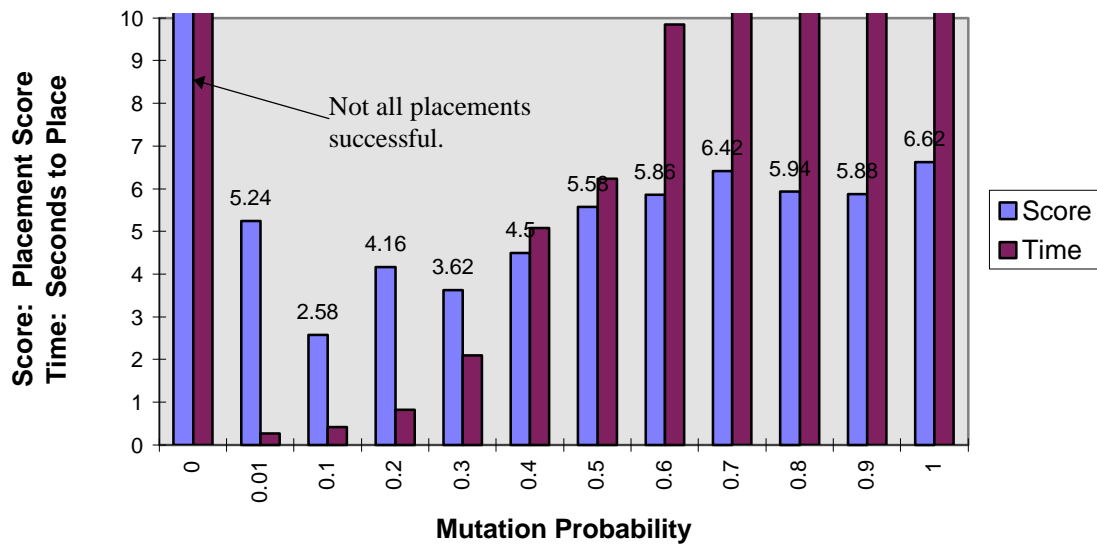
Chart 4: Mutation experiment results.
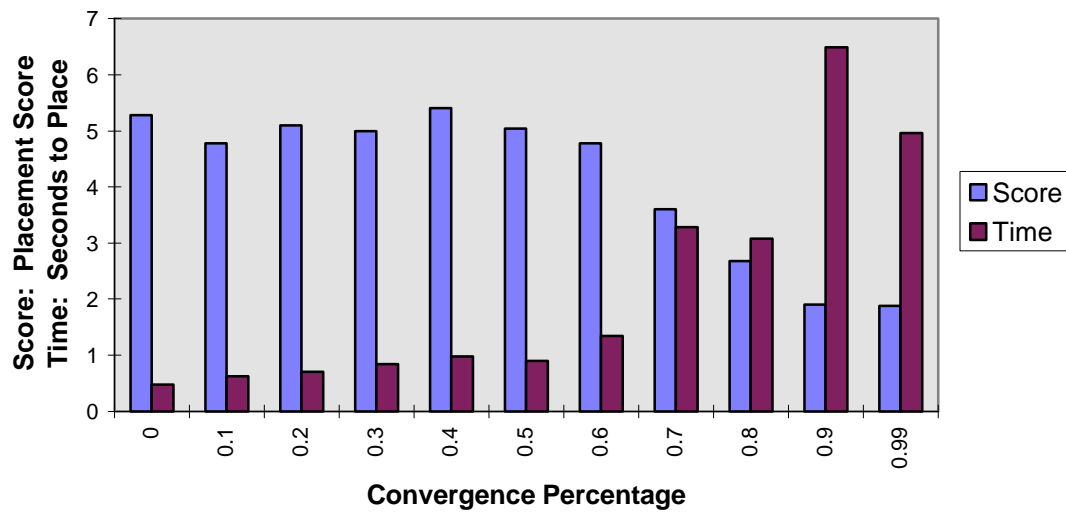
Chart 5: Convergence experiment results.

Chapter 8: Future Work

While the *Tier2* compilers has accomplished its goal of being able to efficiently place and route an algorithm within the *Colt* chip, much work remains for increasing the ease with which applications can be developed. In addition, there are many aspects of the *Tier2* place-and-route algorithm which could be experimented with in the hopes of optimizing the system.

8.1   Compiler Modifications

The *Tier2* language would lend itself nicely to being created via a graphical shell. In the simplest case, a directed graph could be drawn in a graphical window. The vertices of the graph would correspond to hardware resources such as data ports and functional units; the user would double-click on a node and would then be able to enter *Tier2* code directly or make selections from a menu. The main advantage of such a system would be to give the user a better feel for how information is flowing through the chip. This shell would then list out a *Tier2* file which would then be parsed by the compiler.

Such a shell would be simple to write, and if done in a language such as Java, would be cross-platform compatible. This raises another point: the current *Tier2* compiler uses data structures from the *Microsoft Foundation Class Library*. Therefore, it will currently only compile under *Visual C++*. This situation could be rectified by removing the *MFC* data structures from the code and replacing them with *Standard Template Library* constructs.

This would be fairly simple to accomplish because the items used from *MFC* are all present in *STL*: Lists, arrays, and unique-key hash tables. Unfortunately, *STL* does not yet contain a string library. Since *MFC* string classes are used extensively throughout the project, a non-standard string library would have to be used. Other than that, though, the process of conversion would be very straight forward: change the variable declarations from *MFC* to *STL* throughout the program, then modify the methods called within the bodies of the functions.

8.2   Language Modifications

Currently, the programmer must ensure that pathways through the Colt chip are properly synchronized. For instance, in the floating point multiplier example, the purpose of the functional unit named `Delay2` is to provide a two clock cycle delay for the mantissa so that it is synchronized with the exponent data.

A language construct could be added to the system which would instruct the compiler to synchronize two or more data streams. The compiler would then add in optional delays, and potentially extra functional units, in order to implement the requisite path lengthening. One of the main reasons for not doing this for the current *Tier2* version was that bit flag delays are extremely difficult to implement; the use of multiple functional units is required, as in the floating point multiplier. However, this will change with the next architectural version of the chip, so it will soon be easy for a compiler to insert delays, where needed, for all four data types.

A feature which will be absolutely vital for practical applications is the expansion of the *Tier2* system to work with multiple *Colt/Stallion* chips. Currently, if a *Tier2* program requires more than the maximum number of functional units present on a single chip, the compiler issues a warning message. Modifications could be made to the system which would act to partition a large algorithm into dataflow sub-graphs, each of which could fit on a single chip. The sub-graphs would be placed and routed using the current system.

One of the main issues involved in such a change would be ensuring that all of the *Colt/Stallion* chips were correctly configured- modifications to the listing phase would be required. Second of all, whenever a partition took place, several clock cycles of delay would necessarily be inserted into a data pathway due to the need for the pathway to pass through two data ports. Finally, flag bit signals cannot propagate between chips, so the compiler would have to take that into account when attempting to partition the design. The delay problem could be handled by using the synchronization constructs discussed earlier in this section. The partitioning issue would be a more difficult problem. However, language constructs to aid the compiler in making a good decision could be added. For instance, it might be wise to add some sort of a blocking construct which would surround a series of functional units which the programmer desired to have contained within one chip.

Finally, the *Tier3* language and compiler have yet to be designed. Whether it is worth undergoing the effort of development is somewhat in debate: the *Colt/Stallion* chips have a limited number of hardware resources available to them. Wasting them due to inefficiencies in compiling would be a poor decision. It might be determined that a graphical shell surrounding an enhanced *Tier2* compiler provides sufficient usability for the application programmers. On the other hand, certain aspects of the chip are fairly difficult to implement; looping is one example [Bit97a]. A higher-level language could make the implementation of this much easier. Overall, it seems that the utility of a *Tier3* compiler is somewhat limited. Development efforts would be better spent upon making the suggested changes as described in the previous sections, and enhancing the place-and-route abilities, as will be discussed in the next section.

8.3   Place-And-Route Experimentation

The focus of the development effort for this project was on creating a usable piece of software which would greatly ease *Colt/Stallion* programming efforts. Therefore, little attempt was made at trying multiple approaches in order to discover an optimal one. Many aspects of the place-and-route system could be examined in order to find potentially better solutions.

For instance, the choice between using genetic algorithms and simulated annealing was fairly arbitrary. As previously mentioned, the ability of genetic algorithms to combine solutions to produce a potentially better solution is very useful. However, simulated annealing is a powerful tool and might work well for this problem. Future research might involve substituting a simulated annealing package for GAlib and exploring the tradeoffs.

The router is perhaps one of the most critical areas of the entire class hierarchy: it is called repeatedly by the fitness function for the genetic algorithm. Thus, its efficiency greatly affects the

execution speed of the compiler. better heuristics might decrease the number of routing resources required for the implementation of an average path, thus allowing more connections on a chip. Future researchers might also attempt to create a more efficient system, perhaps by using multiple steps, as in gate array routing. Recent work has shown that a two-step routing system which utilized a separate global router and detailed router produced competitive results with traditional combined routers, but required less computing time [Lem97].

The actual cost value returned by the fitness function is currently a count of the number of routing resources used. While this is an indication of the quality of a particular placement, it does allow for placements with long skip bus routes to potentially score better than placements with many small skip bus routes. This might be a problem if the chip is intended to run at a very high clock speed. The solution is fairly trivial, but indicates a path for new research. Currently, the router calculates the length of each path as it implements them; all path lengths are summed to produce a final score. Thus, to minimize path lengths, the cost function could simply return the maximum value returned by the router. The genetic algorithm would naturally attempt to minimize this score, thus producing a final solution with short skip bus path lengths. More research could be done on developing various cost metrics. Ultimately, it may be desirable to be able to select a particular cost metric based upon the problem at hand.

Finally, work could be spent on optimizing the genetic algorithm itself. The GAlib library allows for a great deal of configuring, including convergence percentages, upper bounds on the number of generations allowed, etc. Modifications of these parameters might yield better results overall. In addition, different initial populations or genetic operators might also improve performance.

Overall, there is a great deal more work to be done on this project before it could ever be deemed complete. In terms of priorities, it is probably best to expand the *Tier2* language to add synchronization features, then multi-chip capabilities. Experimenting with the place-and-route system might yield some useful insights, but it would probably be more useful to add a graphical shell in order to aid visualization of the flow of data through the chip.

Chapter 9: Conclusions

The results of the *Tier1* and *Tier2* development process are highly encouraging. Using modern software tools, *Visual C++* and *Visual Parse++,* a custom language and supporting compiler were rapidly developed and implemented. In addition, a difficult problem, that of placing and routing a data flow graph within the *Colt/Stallion* architecture, was efficiently solved through the use of a genetic algorithm.

Due to the nature of genetic algorithms, their generality across a broad array of problems, it was relatively simple to use an existing package towards solving the problems presented herein. The tool used to do so was GAlib by Mathew Wall of the Massachusetts Institute of Technology, a freely distributed class library of genetic algorithms [Wall97]. Although the compiler is rarely able to obtain an optimal solution, it is able to obtain a relatively good solution, such as one which requires only a few additional hardware resources. In terms of the time saved by the placing of a *Colt/Stallion* program automatically, versus manually, the tradeoff is fully justified.

Although the *Tier2* compiler represents a fully usable program, there remain many features which should be added. One of the most important of these is the ability to automatically work with multiple chips, rather than issuing an error if a program requires more hardware resources than a single chip possesses.

As the *Colt/Stallion* architecture matures, the complexity of these chips will increase dramatically, as will the algorithms intended for use. The *Tier2* approach, that of using a custom language combined with a genetic algorithm for placement purposes, represents a scalable solution for the programming of run-time reconfigurable software.

References

[Bac94]   T. Back, "Selective Pressure in Evolutionary Algorithms: A Characterization of Selection Mechanisms", *Proceedings*, *IEEE Conference on Evolutionary Computation Proceedings,* vol. 2, pp. 57-62, June, 1994.

[Bit96]   R. Bittner, M. Musgrove, P. Athanas, "Colt: An Experiment in Wormhole Run-Time Reconfiguration," *High-Speed Computing, Digital Signal Processing, and Filtering Using Reconfigurable Logic*, (SPIE), pp. 187-194, November, 1996.

[Bit97a]   R. Bittner, "Wormhole Run-time Reconfiguration: Conceptualization and VLSI Design of a High Performance Computing System," Ph.D. Dissertation, Bradley Department of Electrical and Computer Engineering, Virginia Tech, January, 1997.

[Bit97b]   R. Bittner, P. Athanas, "Computing Kernels Implemented With A Wormhole RTR CCM," *Field-Programmable Custom Computing Machines*, pp. 120-129, April, 1997.

[Bit97-3] R. Bittner, P. Athanas, "Wormhole Run-time Reconfiguration," *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 79-85, February, 1997.

[Car97]   S. Carlson, "Algorithm of the Gods," in *Scientific American,* pp. 121-123, March 1997.

[Cro95]   F. D. Croce, R. Tadei, G. Volta, "A Genetic Algorithm For The Job Shop Problem," *Computers Operational Research*, vol. 22, no. 1, pp. 15-24, 1995.

[Dav91]   L. Davis, *Handbook of Genetic Algorithms*, Van Nostrand Reinhold, 1991.

[Fog94]   D. Fogel, "An Introduction to Simulated Evolutionary Optimization", *IEEE Transactions on Neural Networks,* vol. 5, no. 1, pp. 3-8, January, 1994.

[For93]   S. Forrest and M. Mitchell, "What Makes a Problem hard for a Genetic Algorithm? Some Anomalous Results and Their Explanation", *Machine Learning*, vol. 13, no.2-3, pp. 129-161, November-December, 1993.

[Gold89]   D. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, 1989.

[Gold95]   D. Goldberg, "The Existential Pleasures of Genetic Algorithms," *Genetic Algorithms in Engineering and Computer Science*, Wiley, 1995.

[Her96]   B. Von Herzen, "250 MHz Correlation Using High-Performance Reconfigurable Computing Engines," *High-Speed Computing, Digital Signal Processing, and Filtering Using Reconfigurable Logic*, (SPIE), pp. 187-194, November, 1996.

[Her97]   J. Woodfill, B. Von Herzen, "Real-Time Stereo Vision on the PARTS Reconfigurable Computer," *Field-Programmable Custom Computing Machines*, pp. 34-43, April, 1997.

[Hol92]   J. Holland, *Adaptation in Natural and Artificial Systems*, MIT Press, 1992.

[Hwa93]   K. Hwang, *Advanced Computer Architecture: Parallelism, Scalability, Programmability*, McGraw-Hill, 1993.

[Kwo94]   D. P. Kwok, F. Sheng, "Genetic Algorithm and Simulated Annealing for Optimal Robot Arm PID Control," *Proceedings*, *IEEE Conference on Evolutionary Computation*, vol. 2, pp. 707-712, June, 1994.

[Lem97]   G. G. Lemieux, S. D. Brown, D. Vranesic, "On Two-Step Routing For FPGAs," *International symposium on Physical Design*, (ACM), pp. 60-66, April, 1997.

[Oliv87] I. M. Oliver, D. J. Smith, J. R. Holland, "A Study of Permutation Crossover Operators on the Traveling Salesman Problem," *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms*, July, 1987.

[Poon95] P. W. Poon, J. N. Carter, "Genetic Algorithm Crossover Operators For Ordering Applications," *Computers Operational Research*, vol. 22, no 1., pp. 135-147, 1995.

[Pos93]   P. Poshyanonda, C. H. Dagli, "Genetic Neuro-Nester for Irregular Patterns," *Intelligent Engineering Systems Through Artificial Neural Networks*, ASME Press, vol. 3, pp. 825-830, 1993.

[Rao94]   B. B. Rao, L. M. Patnaik, R. C. Hansdah, "A Genetic Algorithm for Channel Routing using Inter-Cluster Mutation," *Proceedings*, *IEEE Conference on Evolutionary Computations*, vol 1, pp. 97-103, June, 1994.

[San94]   Sandstone Technology, Inc., *Visual Parse++ Version 2.0 Guide and Reference*, Carlsbad, CA, Sandstone Technology, 1994.

[Sar96]   M. Sarrafzadeh, C. K. Wong, *An Introduction to VLSI Physical Design*, pp. 21-24, McGraw-Hill, 1996.

[Wal97]   M. Wall, *Matthew's GAlib: A C++ Library of Genetic Algorithm Components*, http://lancet.mit.edu/ga/.

[Wil95]   A. G. Williamson, K. Watson, "Optimizing Flexible Manufacturing System Layout With Genetic Algorithms," *Fourth Annual Conference on Factor 2000, (IEE), pp. 12-18, October, 1994.*

Appendix A: *Tier1* and *Tier2* Language Reference

A.1   Grammar conventions

When describing the language grammar, the following syntax is used:

```
<token>                 Represents a token that is required in the construct
[token]                 An optional token in a construct
<token1 | token2>            Either token1 or token2 may be chosen
<token1 || token2>      Either token1 or token2 or both may be chosen
[ … ]                   Optional repeat of prior definition
```

Neither language is case-sensitive. Identifiers (stream names, block names, etc.) may be any alphanumeric string, starting with an alphabetical character. However, reserve words, i.e. those used to form a construct, cannot be used as identifiers.

A.2   Program Structure

The Tier1 and Tier2 languages are divided up into several top-level constructs which can occur in any order, but are not allowed to be nested. These are: stream declarations, port binding declarations, macro definitions, and component definitions. Each of these will be discussed within this document.

A.3   Program Comments   (*Tier1* and *Tier2*)

Two types of comments are provided for:

```
Multiple-line comments:        /* ….  */
Single-line comments:          // ….
```

A.4   Port Declaration Construct (*Tier1* and *Tier2*)

The port declaration section is used to bind physical port addresses to logical names. By default, all ports listed in a port declaration section will be bound together in synchronous mode, unless the loop mode keyword is specified. In this mode, data items are processed individually by the chip. This is a required construct and must precede any stream declarations. Only one of these constructs is allowed per program file. Currently, the range of data ports is one through six.

```
 /* Port binding section */
ports [loop]
<name> =  <port number>;
        .
        .
        .
end ports;
```

A.5   Stream Definitions (*Tier1*)

The basic construct which defines a programming stream is the `stream` block.. It declares the input and output ports used by the stream. The number of allowed input ports is currently limited to one. However, the stream can have zero or more output ports. Multiple stream constructs are allowed, one for each input port, up to the maximum number of ports for the chip.

IFU programming information is placed within the begin-end statements. If data ports are referenced, only the ports listed in the stream header may be used.

```
/* The stream declaration */
stream <name> (<in> <port name>, [<out> <port name> [, …]])
 <cross bar routing statements>
 <block definitions>
end stream;
```

A.6   Stream Definitions (*Tier2*)

The basic construct of the Tier2 language is also the `stream` block, but it takes a slightly different form. One stream construct is allowed and the header does not list input or output ports. Within the main body, any data port listed in the port construct section may be referenced.

```
/* The stream declaration */
stream <name>
 <cross bar routing statements>
 <block definitions>
end stream;
```

A.7   Crossbar Statements (Tier1)

The crossbar routing statements provide a means for programming the flow of data through the crossbar. They take the following form:

```
< port <port name> | ifu <FU name> | mult at <port number>> =>
<ifu <FU name> at <column> | skip at <column> | mult at <port number> | port
<port name>>;
```

The `mult` keyword specifies the hardware multiplier within the *Colt* chip. The range of port numbers is currently 1 to 2. The column required for specifying an FU destination or skip bus destination ranges in value from 1 to 4, which is the maximum number of columns in the *Colt* mesh. `Port` refers to data ports; the `<port name>` for a destination may be any output port listed in the stream header and for a source, may be any input port listed in the stream header. `<FU name>` may be the name of any block construct within the stream, including those occurring after the occurrence of the crossbar statement.

A.8   Crossbar Statements (*Tier2*)

The *Tier2* version is almost identical to the *Tier1* version, except for one significant change: no FU or skip bus may be specified as a destination. This is due to the fact that the *Tier2* language operates on the idea of an abstract dataflow graph, rather than a fixed placement within a Colt chip. Therefore, column numbers have not yet been set.

```
< port <port name> | ifu <FU name> | mult at <port number>> => mult at <port
number> | port <port name>>;
```

A.9   Block Definitions (*Tier1*)

Each stream contains a series of block constructs which specify the functionality of the individual functional units inside of the Colt chip. The block contains a name so that it can be referenced by other blocks, and a series of assignment statements. At the end of the block, in the *Tier1* language, is where the programmer specifies the direction of the programming stream.

From zero to four directions can be specified; each direction and destination block name given must be unique. If the block is empty and only programming direction information is specified, then by default, the single-word-programming flag is set. This allows a programmer to create a stream solely designed to pipe configuration information through a functional unit.

To connect to the crossbar, the statement `go to crossbar` is used. The compiler will issue an error if the ifu cannot be physically connected to the crossbar.

```
/* Defines the functioning of a single FU */
block <name>
 [assignment statements]
 [macro statements]
 [component statements]
 [skip bus blocks]
/* End the block, specify where programming information is to go next */
end [[go <direction> to <name>] | [go to crossbar]][,…]];
```

The statements within the block can be placed in any order and may be repeated. In such a case, the latter statements may override the effects of earlier statements.

A.10   Block Definitions (*Tier2*)

Tier2 block definitions are almost identical to *Tier1* versions, except for the fact that no programming direction information is specified at the end of the block. In addition, no skip bus constructs are allowed within the block.

```
/* Defines the functioning of a single FU */
block <name>
 [assignment statements]
 [macro statements]
 [component statements]
/* End the block, specify where programming information is to go next */
end;
```

A.11   Component/Macro Calls Within Block Constructs (*Tier1 and Tier2*)

Both languages allow macros to be called directly and for components to be instantiated directly within a block construct. Macro calls can be used to make modifications to general aspects of the functional unit and component calls allow a functional unit to be configured using a predefined template.

The macro call takes the form of:

```
        macro [(<integer> [,<integer>. . . ])];
```

If a macro requires parameters, then the exact number must be supplied as a comma-delimited list within parentheses. If no parameters are required, then the macro is called using simply its name with no parentheses. In addition, the macro must be defined before it can be called.

The component instantiation statement is simply:

```
        use <component name>;
```

As with the macro call, the component must already have been defined. When a component is used, the `dfc` fields modified by the component overwrite the values in the block making the call. Any subsequent writes to `dfc` fields within the block (macro calls, assignment statements, etc.) overwrite the values specified by the component.

A.12   ALU Assignment Statement (*Tier1 and Tier2*)

To control the data bus output, the following syntax is used:

```
        out = [delay] [ifcond] <[<left | integer> << | >> <integer>[!][ifcond] <,> ]
        || <op> > [else right];
```

The delay keyword specifies that the optional delay register is to be used. The first `ifcond` keyword is paired with the trailing `else right` keywords; they denote the fact that the conditional unit is being used to select between the ALU output and the right operand register. The next set of tokens controls the barrel shifter. The operator $\ll$ denotes a left shift and $\gg$ denotes a right shift. If the reserved word `left` is included, then the left operand register is used as-is. If an integer is specified, then the left operand register is programmed to use a constant value and the integer specified is used as the constant. The `ifcond` token indicates a conditional shift. Inclusion of the exclamation point negates the condition- the shifting occurs if the conditional bit is false. The `<integer>` represents an integer which specifies the number of shifts to perform. Finally, `<op>` specifies an operator for the ALU to use. The operators are defined outside of the stream as macros; they will be discussed later in the document.

If no operator is included, then the ALU is programmed to pass the left operand straight through. This means that if only a shift is included, the shifted data will be passed without problem. If no shifting information is given, then the IFU's shift control bit is disabled and data passes through the barrel shifter without modification.

An example bus output control statement might be:

```
        out = ifcond
                left << 2 ifcond,
                add
            else right;
```

This statement performs a conditional left shift of two bits, then adds that value to the right register using the operator add. Finally, depending upon the conditional bit, the FU selects between the ALU output and the right operand.

In order to simplify the handling of the valid bit tag, the compiler makes several assumptions when processing the out assignment statement. If there has been an explicit assignment statement to leftreg or the out statement contains a constant or left keyword, then the left register is considered to have been "used". If there is an explicit assignment statement to the rightreg, then the right register is considered to have been "used". If both registers have been used, then the valid bit is set to be the logical-AND of the valid bits from the left and right registers. If only the left register has been used, then the valid bit is set to be the logical-AND of the left register's valid bit and the conditional bit, which is set to a constant value of one. In other words, the left register's valid bit becomes the output valid bit. The same holds true if only the right register has been used. This was done in order to make life easier for the programmer. However, it will be overridden should the user make any explicit assignment statements to *cond* or to valid, either before or after the out assignment statement.

This is an example of how this rule simplifies programming. If the programmer wants to add the right register, directed from local north, to a constant and send the result to the bus output, then the following code would suffice:

```
// Tier1 code, but the out assignment statement is valid for Tier2 also
block ifu1
  rightreg = from local north;    // Right operand takes from local north
  out = ADDVAL, add;              // left operand set to ADDVAL,
                                  // ALU set to add operator
end, go south to ifu2;
```

If the special rules did not exist, then an extra statement specifying the valid bit control would have to be included:

```
valid = leftandright16;
```

A.13   Flag Bit Assignment Statements (*Tier1 and Tier2*)

Both languages use assignment statements to alter the functionality of the three bit flags, conditional, *shift*, and *carry*, and the two operand registers. The *Tier1* language requires the user to explicitly control the source direction of data, should bit information from other functional units be specified. The *Tier2* language, on the other hand, has the user specify only the source functional unit; the compiler performs the routing itself. In the *Tier2* language format given below, the term <xbar source> refers to any valid crossbar statement source: a functional unit, a data port, or a multiplier port. Of course, for the three bit flags, only FU sources may be specified.

The conditional bit is set in the following way:

```
Tier1:
  cond [!]= < <macro> || from <local|skip> <direction> >;

Tier2:
```

```
cond [!]= < <macro> || <xbar source>;
```

Macro operations:

zero
      Conditional bit will be a constant zero (before inversion operator).

condin
      Conditional bit will come from an external source. This is not really
      ever needed, since specifying an external direction/source (internally
      known as condin) automatically specifies this value.

shiftout
      Conditional bit will come from the output of the barrel shifter.

condout
      Conditional bit will be the value set by the condout statement.

Either or both portions may be placed in the statement. The first part, a macro name, is used to specify the value that the conditional bit will take. The second part specifies what external source the conditional input bit is taken from. Finally, the exclamation point signifies that the optional inversion is to be used.

The conditional output bit has a similar format:

Tier1:
```
  condout = < <macro> || from <local|skip> <direction>>;
```

Tier2:
```
  condout = < <macro> || <xbar source>;
```

Macro operations:

rightop (<integer>)
      Conditional output will be the specified bit of the right operand.
      Range <integer> is 0 through 4, inclusive.

ALUsign
      Conditional output will be the sign bit from the ALU output.

carryout
      Conditional output will be the carry bit.

rightnor (<integer>)
      Conditional output will be the logical-NOR of the right operand, from
      the specified bit through bit 15. Range of <integer> is 12 through
      14, inclusive.

rightsign
      Conditional output will be the sign bit of the right operand.

rightvalid
      Conditional output will be the valid bit tag for the right operand.

ALUoverflow
      Conditional output will indicate whether an overflow occurred with the
      ALU.

The direction specification portion is included in case the programmer wishes to have `condout` use the value of `condin`, but the `cond` statement does not specify a from direction for `condin`.

The *shift* bit is controlled in the following way:

```
Tier1:
  shift [!]= < <macro> || from <local|skip> <direction> >;

Tier2:
  shift [!]= < <macro> || <xbar source>;

Macro operations:

zero
      Shift bit will be a constant zero (before inversion operator)
condsign
      Shift bit will be the sign bit from the conditional output.
condout
      Shift bit will be the value set by the condout statement.
```

If a source specification is used, then the compiler directs the shift-in bit to the shift flag. Otherwise, the specified macro is executed.

The *carry* bit is controlled by:

```
Tier1:
  carry [!]= < <macro> || from <local|skip> <direction> >;

Tier2:
  carry [!]= < <macro> || <xbar source>;

Macro operations:
zero
      Carry bit will be a constant zero (before inversion operator)
shiftout
      Carry bit will be the output of the barrel shifter.
cond
      Carry bit will be the conditional bit.
```

As with *shift* and *cond*, the use of the source specification means that the user wishes the value to come from an external source. Otherwise, the value specified by the macro is used.

The valid bit is controlled by:

```
Tier1 and Tier2:
  valid = < <macro> >;

Macro operations:
cond
      Valid bit will be the conditional bit

condandleft16
      Valid bit will be the conditional bit logical-ANDed with the valid bit
      of the left operand.

condandright16
      Valid bit will be the conditional bit logical-ANDed with the valid bit
      of the right operand.

leftandright16
```

```
            Valid bit will be the logical-AND of the valid bits for the left and
            right operands.
```

The left operand has a slightly different format:

```
    Tier1:
      leftreg = [loop] < <macro> || from <local|skip> <direction> >;

    Tier2
      leftreg = [loop] < <macro> || <xbar source>;

    Macro operations:
    all
          Left register will accept any data, even invalid words.

    validzero
          Left register will be set to a constant zero with the valid bit set as
          true.

    validdata (<integer>)
          Left register will accept only valid data.  The integer parameter
          allows an initial value to be specified.  This will be overwritten
          whenever new valid data is encountered.

    constant(<integer>)
          Left register will be set to a valid constant value as specified by
          <integer>.
```

The `loop` keyword is used to set the loop-back bit for the left operand. This forces the left register to act as an accumulator by automatically storing the bus output value of the functional unit.

By default, the constant loaded by the constant macro and the constant loaded by the bus output assignment statement has its valid bit set. If this is not desired, then a call must be made to the `clearvalid()` macro which will turn off the constant's valid bit.

The right operand has a very simple syntax: only a source can be specified:

```
    Tier1:
      rightreg = from <local|skip> <direction>;

    Tier2:
      rightreg = <xbar source>;
```

A.14   Skip Bus Construct (*Tier1*)

The skip bus is programmed using the skip construct. It consists of the following:

```
    skip <data | shift | carry | cond>
      <direction blocks>
    end skip;
```

The direction blocks specify the signals that are being directed. The following blocks are allowed:

```
    skip data
          [<direction> from <right|opposite|aux|out>;]
             .
```

```
              .
              .
     end skip;

     skip shift
              [<direction> from <right|opposite|shiftout>;]
              .
              .
              .
     end skip;

     skip carry
              [<direction> from <right|opposite|carryout>;]
              .
              .
              .
     end skip;

     skip cond
              [<direction> from <right|opposite|condout>;]
              .
              .
              .
     end skip;
```

The `<direction> from` statements control what data may be placed onto the skip bus. The actual direction of the bus is determined by both these statements and by the assignment statements within the block definitions. Since each functional unit controls the skip bus on its eastern and southern side, the compiler is able to ascertain the desired bus direction by examining both the skip constructs and assignment statements. For instance, an assignment statement which specifies that the left register should obtain data from the east indicates that the skip bus for the functional unit lying to the east will pass data in a westerly direction.

The number of direction statements placed within the construct is not limited. However, later statements will override earlier statements. Thus, it only makes sense to have a maximum of two statements: one for the east/west skip bus and one for the north/south skip bus.

A.15   Macro Definitions (*Tier1 and Tier2*)

The next top-level statement to be discussed is the macro language, which is the same for both *Tier1* and *Tier2* languages. This provides a direct interface to the `dfc` level. By providing this interface, future changes to the architecture of the chip will require fewer changes to the compiler or language syntax than if everything were hard-coded into the language grammar. Each macro can be thought of as a subroutine that can be used to directly modify `dfc` fields. Simple parameter passing and conditional statement support is also included.

The basic definition is:
```
     macro [LHS operand] <name> ([parm-name [,…]])
       [assert statements]
       [assignment statements]
       [case statements]
     end macro;
```

The LHS operand provides for overloading of macros that have the same name, but are used in different assignment statements. For example, the control bits *cond*, *shift*, and *carry* can all be set to zero. The overloading feature allows the same name to be used for all three of these macros, even though the actual macros are different. If it is not used, then the macro may not be used in an assignment statement, only as a statement in a block construct.

```
Allowed LHS identifiers:
cond
      Macros for the cond assignment statement

condout
      Macros for the condout assignment statement

shift
      Macros for the shift assignment statement

carry
      Macros for the carry assignment statement

valid
      Macros for the valid assignment statement

leftreg
      Macros for the leftreg assignment statement

operator
      Operator macros for the bus output assignment statement.

<none>
      Macros to be called directly within a block body.
```

The name of the macro must be unique within an LHS namespace for the scope of the files visible to the compiler. The parameter list is optional; it allows a variable number of integer parameters to be passed into the macro in order to control its functionality.

In a macro definition, parentheses are required after the macro name. In the macro call, however, no parentheses are required if the macro takes no arguments. It is required, though, that if a macro accepts arguments, the exact number of arguments must be supplied in a call to that macro.

Since the ALU in each functional unit uses an extremely flexible propagate-generate-result design, the language has been designed to have no predefined operators. Instead, macro definitions allow the programmer to define any possible operation by directly controlling the propagate (P), generate (G) and result (R) dfc fields of the ALU. For common operations such as addition, predefined operators exist in a standard library (discussed in The Standard Library section). To define an operator, which is used within an out statement, declare a macro with an LHS value of operator.

For instance, the add operator is defined as:

```
macro operator add ()
      P = 6;
      G = 8;
      R = 6;
end macro;
```

So that the macro can verify the correctness of the parameters, a range checking mechanism is provided. This takes the following form:

```
assert (<range stmt>) <string>;

<range stmt>:   parm = <range>
<range>:        '[' <integer >,<integer >']'[, …]
```

An example is:

```
assert (n = [1,4]) "Bit value out of bounds!";
```

The range statement simply allows the programmer to specify a list of valid integer values for a given parameter. If the parameter does not fall within the specified ranges, the error string is displayed by the compiler and the compilation process stops. Note that the ranges, specified by pair of values within brackets, is inclusive.

Assignment statements within macros take the following form:

```
<dfc field>   <'=' | '|=' | '&=' >   <integer | parm>;
```

<Dfc field> is any valid dfc field. The '=' is an assignment, the '|=' performs a logical-OR with the field on the left and the value on the right, and the '&=' performs a logical-AND operation. In order to simplify the compiler, compound expressions are not supported, such as '(var1 or var2) or var3'. A possible future enhancement might be to add this functionality.

Finally, the case statement provides the ability to use parameters to modify macro behavior. It takes the following form:

```
case <parm> is
      when <integer > : <macro statements>;
       .
       .
       .
      [default : <macro statements>];
end case;
```

The parameter specified in the first line is successively compared to integer values specified in the body of the case construct. If a match is found, then the macro statements associated with that value are executed and evaluation of the case statement ceases. If the default clause is present, then that body of statements will execute if no matching value was found. Nesting to an arbitrary depth is supported, so nested case statements, assert statements, etc. are allowed.

A.16   Component Definitions (*Tier1 and Tier*)

The component construct provides a simple way to reuse IFU descriptions. The syntax is almost identical to that of a block statement within a stream definition, except that it provides a template for an IFU description, rather than actually generating programming stream information. The component's name may be placed within a block definition using a use <component name>

statement, in which case the IFU is modified to take on the new values specified. Since latter statements override former statements within a block definition, the information specified by a component can be easily overridden.

The basic component definition is:

```
Tier1:
  component <name>
          <assignment statements>
          <macro statements>
          <component statements>
          <skip bus blocks>
  end component;

Tier2:
  component <name>
          <assignment statements>
          <macro statements>
          <component statements>
  end component;
```

Note that program flow information cannot be specified, even for *Tier1* definitions. However, everything else about the component definition is identical to that of the block definition.

One of the main uses of the component definition is to create a template for a commonly used function, such as an FU which adds values from the left and right registers. The actual block definition is used to provides the information about operand sources.

A.17   Include Statements and Constant Definitions (*Tier1 and Tier2*)

In order to allow for code reuse and hierarchical design, the C preprocessor is run on every supplied file before the compiler parses it. This means that `include` statements, conditional compilation, and constant definitions are all legal within a *Tier1* and *Tier2* programs.

For a complete guide to preprocessor syntax, please consult an appropriate C language reference. Commonly used statements are:

```
#include "<filename>";
```

The preprocessor inserts the specified filename into the original file at the point of the include statement. Now, all constructs in the included file will be compiled together with the original file.

Constants can be defined with:

```
#define <token> <replacement>
```

The preprocessor will scan the program file for occurrences of <token> and substitute in <replacement>.

A.18   The Standard Library (*Tier1 and Tier2*)

Any useful program written in these languge will require a large number of macros and operators in order to configure the various bit flags and the ALU. For that reason, an implied include

statement at the beginning of each program exists. In other words, the compiler automatically searches for the standard library file, named `std.tlb`, parses it first, then proceeds with the parsing of the original program file. This way, a program will not have to bother including the same standard library for every single program that contains macro definitions for conditional bit operations, etc.

Note that currently, the preprocessor is not run on the standard library. This may be changed in the future, should need arise.

Appendix B: Compiler Usage

The purpose of this appendix is to instruct the user on the functioning of the compilers themselves, such as command-line options, accessory files, etc.

B.1   The Tier1 Compiler (*Tier1*)

The executable for this compiler is called `tier1.exe`. The usage format is as follows:

```
tier1 [<optional parameters>] <input file name> [<optional parameters>]
```

The input file name specifies a valid *Tier1* program file. There is no default extension, though programs generally use a ".clt" naming convention.

The optional parameters are:

```
-h
     Displays command-line options

-w
     Disables warning messages.  These are generated whenever a dfc field is
     set to a value and then overwritten.

<valid cpp options>
     C preprocessor options are passed to the C preprocessor.  This allows
     for control of conditional compilation by using the define option -D.
```

The compiler does not actually check to make sure that an option is a valid preprocessor argument. Instead, anything not recognized as a compiler option is passed to the preprocessor.

B.2   The Tier2 Compiler (*Tier2*)

The executable for this compiler is called `tier2.exe`. The usage format is as follows:

```
tier2 [<optional parameters>] <input file name> [<optional parameters>]
```

The input file name specifies a valid *Tier2* program file. There is no default extension, though programs generally use a ".clt" naming convention.

The optional parameters are:

```
-h
     Displays command-line options

-w
     Disables warning messages.  These are generated whenever a dfc field is
     set to a value and then overwritten.

-l<filename>
     Specify the log file name for routing and placement information.  The
     default is <input file>.log.
```

```
-t
     Try local routing only.  This disables the usage of the skip bus as a
     means of routing.  Use this option if the program is relatively simple
     and a planar solution is obviously possible.

-r<integer>
     Specify a repeat count.  The compiler will attempt to perform the
     place-and-route action for the given number of times, where each trial
     is independent of the others.

-g<filename>
     Specify a genetic algorithm parameter file.  This allows the user to
     customize such features as convergence percentage, maximum number of
     generations, etc.  Please refer to GAlib documentation for a complete
     listing of options.

-x<filename>
     Specify an exclusion file.  This allows the user to preclude the use of
     certain functional units from placement.  Use this in order to allow
     more than one algorithm to coexist on a single chip.

 <valid cpp options>
     C preprocessor options are passed to the C preprocessor.  This allows
     for control of conditional compilation by using the define option -D.
```

The compiler does not actually check to make sure that an option is a valid preprocessor argument. Instead, anything not recognized as a compiler option is passed to the preprocessor

B.3   Files Required/Produced (*Tier1*)

The *Tier1* compiler requires certain files to operate. These are:

```
tier1.exe
     The compiler executable

tier1.rc
     Resource file.  Specifies the location of necessary files.  Searched
     for first in the directory from where the compiler was executed.  If
     not found there, it is searched for in the directory where the compiler
     is located.

address.map
     Specifies addresses of hardware resources within the chip.

tier1.llr
     Lexing table file.

tier1.dfa
     Yaccing table file.

std.tlb
     Standard library file.

dfc.exe
     Converts .dfc files into .pwl files.  Refer to [Bit97a]

cpp.exe
     C preprocessor.

fielddef.h
     Header file used by the dfc program.

port(n).dfc
```

```
        Output dfc files produced by the compiler.  (n) represents the port
        number and ranges from 1 to the maximum port number, currently 6.

    port(n).pwl
        Converted dfc file- generated by the dfc.exe program.  Refer to
        [Bit97a].
```

B.4 Files Required/Produced (*Tier2*)

The *Tier2* compiler requires certain files to operate. These are:

```
    tier2.exe
        The compiler executable

    tier2.rc
        Resource file.  Specifies the location of necessary files.  Searched
        for first in the directory from where the compiler was executed.  If
        not found there, it is searched for in the directory where the compiler
        is located.

    address.map
        Specifies addresses of hardware resources within the chip.

    tier2.llr
        Lexing table file.

    tier2.dfa
        Yaccing table file.

    std.tlb
        Standard library file.

    dfc.exe
        Converts .dfc files into .pwl files.  Refer to [Bit97a]

    cpp.exe
        C preprocessor.

    fielddef.h
        Header file used by the dfc program.

    port(n).dfc
        Output dfc files produced by the compiler.  (n) represents the port
        number and ranges from 1 to the maximum port number, currently 6.

    port(n).pwl
        Converted dfc file- generated by the dfc.exe program.  Refer to
        [Bit97a].

    <input file>.log
        The log file produced by the compiler.  It contains placement and
        routing information.  The name may be overridden by a command-line
        option.
```

B.5   The Resource File (*Tier1 and Tier2*)

The resource file is used to specify the location of key files needed by the compiler in order to
operate. It is searched for first in the directory from which the compiler was executed, and if not
found there, in the directory containing the executable. This allows a user to override the values
of system default settings.

The general format is:

```
; <comment>              Comment line
<token> = <pathname>     Data line
```

The file is not case sensitive and comment lines may be interspersed with data lines. The pathname must be the full pathname of the file, including the filename itself. Allowed tokens are as follows.

std
> Specify the location of the standard library file std.tlb.

dfc
> Specify the location of the dfc.exe program.

Fieldheader
> Specify the location of the header file fielddef.h.

cpp
> Specify the location of the C preprocessor program cpp.exe.

addr
> Specify the location of the address file address.map.

dfa
> Specify the location of the yaccing table file.

llr
> Specify the location of the lexing table file.

B.6   The Address Map File (*Tier1 and Tier2*)

The address map file, `address.map`, specifies the address of hardware resources within the *Colt* chip. Its location is specified in the resource file.

This file should never have to be modified by the user. The format is as follows:

Comments:
```
  ;   <comment>
```

Data port specifications:
```
 p <number of data ports> <beginning address> <beginning crossbar port>
```

 <number of data ports>:
> Currently 6. Value range is 1 through 6.

 <beginning address>:
> Currently 24. Address range is 24 to 29.

 <beginning crossbar port>:
> Currently 11. Port range is 11 to 16.

Multiplier port specifications:
```
 m <number of ports> <beginning crossbar port>
```

 <number of ports>
> Currently 2. Value range is 1 through 2.

 <beginning crossbar port>:
> Currently 9. Port range is 9 through 11.

FU mesh specifications:

```
         f <number of rows> <number of columns> <beginning FU address>

         <number of rows>:
              Currently 4.  Value range is 1 through 4.

         <number of columns>:
              Currently 4.  Value range is 1 through 4.

         <beginning FU address>:
              Currently 32.  Ranges from a value of 32 to 47, starting in the upper
              left corner and going horizontally across, to the lower right corner.

     Local bus port specifications:
       l <first port address> <second port address> . ..   <max port address>

       <first port address>
            Currently 1.

       <second port address>
            Currently 3.

       <third port address>
            Currently 5.

       <fourth port address>
            Currently 7.  The total number of ports listed here must be equal to
            the number of columns in the FU mesh.

     Skip bus port specifications:
       s <first port address> <second port address> … <max port address>

       <first port address>
            Currently 1.

       <second port address>
            Currently 3.

       <third port address>
            Currently 5.

       <fourth port address>
            Currently 7.  The total number of ports listed here must be equal to
            the number of columns in the FU mesh.
```

The file is not case sensitive. Since the map file was not intended to be readily modified by the user, it contains little error checking capabilities. Therefore, be very careful about making changes to this file.

B.7   The Exclusion File (*Tier2*)

The exclusion file allows the user to force the compiler to not use certain functional units for either routing purposes or placement purposes. The main use of this is to allow multiple algorithms to coexist on a single chip.

The format is very simple. Note that this file is not required for operation.

```
     ; <comment>            Comment line
     <row>,<column>         FU to exclude
```

Any functional unit, specified by row and column, listed in this file, will be excluded from placement or routing.

Appendix C: Grammar Reference

C.1   *Tier1* Grammar Reference

The following is the language grammar for the *Tier1* language, as specified using Visual Parse++.
This is the definitive reference for program structure, since the compiler itself is created using this
file. Refer to [San94] for a complete guide to the format of this file.

```
// Rule file for the tier-1 compiler
//
%expression Main
'[ \n\t\r]+'                        %ignore;
'[a-zA-Z][a-zA-Z0-9_]*'             idstring;
'\".*\"'                            qstring;
'/\*'                               %ignore, %push MultiLineComment;
'//'                                %ignore, %push SingleLineComment;
';'                                 Semicolon,';';
':'                                 Colon,':';
'='                                 Equals,'=';
'!='                                InvEquals,'!=';
'=>'                                Arrow,'=>';
'\('                                LeftParen,'(';
'\)'                                RightParen,')';
','                                 Comma,',';
'\['                                LeftBracket,'[';
'\]'                                RightBracket,']';
'\|='                               OrEqual,'|=';
'\&='                               AndEqual,'&=';
'<<'                                ShiftLeft,'<<';
'>>'                                ShiftRight,'>>';
'!'                                 Not,'!';
'#[ \n\t\r]*[a-zA-Z]+.*\n'          HASHIGNORE;
'#[ \n\t\r]*[0-9]+'                 HASHINT;
'[\-+]?[0-9]+'                      Dec,'dec';
'[0-9A-Fa-f]+[hH]'                  Hex,'hex';

'[bB][eE][gG][iI][nN]'              BEGIN;
'[eE][nN][dD]'                      END;
'[pP][oO][rR][tT][sS]'              PORTS;
'[pP][oO][rR][tT]'                  PORT;
'[lL][oO][oO][pP]'                  LOOP;
'[sS][tT][rR][eE][aA][mM]'          STREAM;
//'[oO][pP][eE][rR][aA][tT][oO][rR]'    OPERATOR;
'[iI][nN]'                          IN;
'[oO][uU][tT]'                      OUT;
'[iI][fF][uU]'                      IFU;
'[aA][tT]'                          AT;
'[mM][uU][lL][tT]'                  MULT;
'[lL][oO][cC][aA][lL]'              LOCAL;
'[sS][kK][iI][pP]'                  SKIP;
'[mM][aA][cC][rR][oO]'              MACRO;
'[aA][sS][sS][eE][rR][tT]'          ASSERT;
'[cC][aA][sS][eE]'                  CASE;
'[iI][sS]'                          IS;
'[wW][hH][eE][nN]'                  WHEN;
'[dD][eE][fF][aA][uU][lL][tT]'      DEFAULT;
'[bB][lL][oO][cC][kK]'              BLOCK;
'[gG][oO]'                          GO;
'[tT][oO]'                          TO;
```

```
'[fF][rR][oO][mM]'                    FROM;
'[cC][rR][oO][sS][sS][bB][aA][rR]'    CROSSBAR;
'[nN][oO][rR][tT][hH]'                NORTH;
'[sS][oO][uU][tT][hH]'                SOUTH;
'[eE][aA][sS][tT]'                    EAST;
'[wW][eE][sS][tT]'                    WEST;
'[oO][uU][tT]'                        OUT;
'[dD][eE][lL][aA][yY]'                DELAY;
'[iI][fF][cC][oO][nN][dD]'            IFCOND;
'[eE][lL][sS][eE]'                    ELSE;
'[rR][iI][gG][hH][tT]'                RIGHT;
'[lL][eE][fF][tT]'                    LEFT;
'[uU][sS][eE]'                        USE;
'[cC][oO][mM][pP][oO][nN][eE][nN][tT]'  COMPONENT;

%expression MultiLineComment
'.'                                         %ignore;
'\n'                        %ignore;
'\*/'                       %ignore, %pop;

%expression SingleLineComment
  '.'                       %ignore;
  '\n'                      %ignore, %pop;

%prec
1, idstring,                %left;


%production start

// The program consists of an arbitrarily long list of  constructs
Start           start           ->      constr;
StartList       start           ->      start constr;

// This is the list of constructs
ConstrPort      constr          ->      portspec ';';
ConstrStream    constr          ->      streamspec ';';
ConstrMacro     constr          ->      macrospec ';';
//ConstrOper     constr          ->      operspec ';';
ConstrComp      constr          ->      compspec ';';
ConstrCpp       constr          ->      cppline;
ConstrError     constr          ->      %error ';';

// Lines thrown in by the preprocessor should be parsed so that
// we know what file we're currently in

// This is to ignore any statements which begin with a '#' and then have
// an alphabetical character following it, i.e. any normal preprocessor
// declaration which slips through
CppIgnore       cppline         ->      HASHIGNORE;
// This matches the special preprocessor declarations produced by the
// preprocessor to indicate include files
CppStart        cppline         ->      hashint quotedstring;
CppLine         cppline         ->      hashint quotedstring integer;

// Component definition section
  CompSpec        compspec        ->      COMPONENT identifier blockbody END
                                          COMPONENT;

// Macro definition section
MacroSpec       macrospec       ->      MACRO macrostart macroheader
                                        macrobody END MACRO;
// Macro header if an LHS operand is specified for overloading purposes
```

```
MacroLHS          macrospec       ->      MACRO identifier macrostart
macroheader
                                          macrobody END MACRO;
// Specification for the macro parameter list, etc.
MacroHeader       macroheader     ->      '(' macroparms ')';
MacroStart        macrostart              ->      identifier;
MacroParmNull     macroparms      ->      ;
MacroParmOne      macroparms      ->      identifier;
MacroParmList     macroparms      ->      macroparms ',' identifier;
// Body of the macro- statements allowed within a definition
MacroBodyNull     macrobody       ->      ;
MacroBodyList     macrobody       ->      macrobody macrostmts;
// These are the allowed statements
MacroAssertStmt   macrostmts      ->      assertstmt ';';
MacroAssignStmt   macrostmts      ->      assignstmt ';';
MacroCaseStmt     macrostmts      ->      casestmt ';';
MacroError        macrostmts      ->      %error ';';


// Definition for an assert statement
AssertStmt        assertstmt      ->      ASSERT '(' assertrange ')'
                                          quotedstring;
AssertRange       assertrange     ->      assignsrc '=' rangelist;
RangeListOne      rangelist       ->      rangesingle;
RangeListMany     rangelist       ->      rangesingle ',' rangelist;
RangeSingle       rangesingle     ->      '[' integer ',' integer ']';


// Definition for an assignment statement
AssignStmt        assignstmt      ->      identifier assignop assignsrc;
AssignEqual       assignop        ->      '=';
AssignOr          assignop        ->      '|=';
AssignAnd         assignop        ->      '&=';
AssignSrcParm     assignsrc       ->      identifier;
AssignSrcInt      assignsrc       ->      integer;


// Definition for a case statement
CaseStmt          casestmt        ->      CASE assignsrc IS casebody END CASE;
CaseBodyOne       casebody        ->      ; //casecond;
CaseBodyList      casebody        ->      casebody casecond casedefault;
CaseCond          casecond        ->      casewhen ':' casestmts;
CaseWhen          casewhen        ->      WHEN integer;
CaseDefaultNull   casedefault     ->      ;
CaseDefaultList   casedefault     ->      DEFAULT ':' casestmts;
CaseStmtsNull     casestmts       ->      ;
CaseStmtsList     casestmts       ->      casestmts macrostmts;


// Stream declaration section
StreamSpec        streamspec      ->      STREAM streamheader streambody END
                                          STREAM;
// Specifications for the port list
StreamHeader      streamheader    ->      identifier '(' streamports ')';
StreamPorts       streamports     ->      streaminport streamoutlist;
StreamInPort      streaminport    ->      IN identifier;
StreamOutNull     streamoutlist   ->      ;
StreamOutList     streamoutlist   ->      ',' streamoutport streamoutlist;
StreamOutPort     streamoutport   ->      OUT identifier;
// Specifications for the body of the stream
StreamBodyNull    streambody      ->      ;
StreamBodyList    streambody      ->      streambody streamstmts;


// These are the allowed statements within a stream
StreamXBStmt      streamstmts     ->      xbstmt ';';
BlockStmt         streamstmts     ->      blockdef ';';
StreamError       streamstmts     ->      %error ';';
```

```
// This is the section for the definition of IFU blocks
BlockDef          blockdef        ->      BLOCK identifier blockbody END
                                           blockroute;
BlockBodyNull     blockbody       ->      ;
BlockBodyList     blockbody       ->      blockbody blockstmts;
BlockRouteNull    blockroute      ->      ;
// Block program routing info
BlockRouteList    blockroute      ->      blockroute ',' blockdir;
BlockDirIfu       blockdir        ->      GO direction TO identifier;
BlockDirXB        blockdir        ->      GO TO CROSSBAR;
DirectionEast     direction       ->      EAST;
DirectionWest     direction       ->      WEST;
DirectionNorth    direction       ->      NORTH;
DirectionSouth    direction       ->      SOUTH;


// Here's where we define the statements that can go within a block
BlockOut          blockstmts      ->      bassignout ';';
BlockBitOp        blockstmts      ->      bassignbit ';';
BlockMacro        blockstmts      ->      bmacrocall ';';
BlockComp         blockstmts      ->      bcompuse   ';';
BlockSkip         blockstmts      ->      bskipblock ';';
BlockError        blockstmts      ->      %error ';';


// Component usage statement
BCompUse          bcompuse        ->      USE identifier;


// Skip bus programmin block
BSkipBlock        bskipblock      ->      SKIP bskiptype bskipbody END SKIP;
BSkipType         bskiptype       ->      identifier;
BSkipBodyNull     bskipbody       ->      ;
BSkipBodyList     bskipbody       ->      bskipbody bskipstmt ';';
BSkipError        bskipbody       ->      %error ';';
BSkipStmtDir      bskipstmt       ->      GO direction;
BSkipStmtSrc      bskipstmt       ->      direction FROM bskipsrc;
BSkipSrcID        bskipsrc        ->      identifier;
BSkipSrcOUT       bskipsrc        ->      OUT;
BSkipSrcRIGHT     bskipsrc        ->      RIGHT;


// macro call within a block (not in an assignment stmt)
BmacroCall        bmacrocall      ->      bitmacname bitparmlist;


// Generalized flag modification statement.  Compiler handles special cases
BAssignBit        bassignbit      ->      bittype bitop bitloop bitmacro
                                           bitvalsrc;
BitValSrc         bitvalsrc       ->      FROM bitsrc direction;
BitValSrcNull     bitvalsrc       ->      ;
BitType           bittype         ->      identifier;
BitOpEqual        bitop           ->      '=';
BitOpNotEqual     bitop           ->      '!=';
BitLoopYes        bitloop         ->      LOOP;
BitLoopNo         bitloop         ->      ;
BitMacroNull      bitmacro        ->      ;
BitMacro          bitmacro        ->      bitmacname bitparmlist;
BitMacName        bitmacname      ->      identifier;
BitParmNone       bitparmlist     ->      ;
BitParmList       bitparmlist     ->      '(' bitparms ')';
BitParmsNull      bitparms        ->      ;
BitParmOne        bitparms        ->      integer;
BitParmMany       bitparms        ->      bitparms ',' integer;


BitSrcLocal       bitsrc          ->      LOCAL;
BitSrcSkip        bitsrc          ->      SKIP;
```

79

```
// Assignment for the main output
BAssignOut        bassignout    ->    OUT '=' boutdelay;
BOutDelayYes      boutdelay     ->    DELAY bifshiftop;
BOutDelayNo       boutdelay     ->    bifshiftop;
BIfShiftOpCond    bifshiftop    ->    IFCOND bshiftop ELSE RIGHT;
BIfShiftOp        bifshiftop    ->    bshiftop;
BShiftOp          bshiftop      ->    bshift ',' boperator;
BShiftOnly        bshiftop      ->    bshift;
BOpOnly           bshiftop      ->    boperator;
BShift            bshift        ->    bshiftsrc bshiftcomm;
BShiftComm        bshiftcomm    ->    bshiftdir bshiftval bshiftcond;
BShiftCommNull    bshiftcomm    ->    ;
BOperator         boperator     ->    identifier;
BShiftSrcLeft     bshiftsrc     ->    LEFT;
BShiftSrcConst    bshiftsrc     ->    integer;
BShiftDirLeft     bshiftdir     ->    '<<';
BShiftDirRight    bshiftdir     ->    '>>';
BShiftVal         bshiftval     ->    integer;
BShiftCondYes     bshiftcond    ->    IFCOND;
BShiftCondYesN    bshiftcond    ->    '!' IFCOND;
BShiftCondNo      bshiftcond    ->    ;

// Crossbar routing statement
XbStmt            xbstmt        ->    xbsource '=>' xbdest;
// Possible crossbar source statements
XbSourcePort      xbsource      ->    xbportds;
XbSourceIfu       xbsource      ->    xbifusource;
XbSourceMult      xbsource      ->    xbmultds;

// Possible crossbar destination statements
XbDestPort        xbdest        ->    xbportds;
XbDestIfu         xbdest        ->    xbifudest;
XbDestMult        xbdest        ->    xbmultds;
XbDestSkip        xbdest        ->    xbskipds;

// Destination/Source types
XbDSPort          xbportds      ->    PORT identifier;
XbIfuDest         xbifudest     ->    IFU identifier AT integer;
XbIfuSource       xbifusource   ->    IFU identifier;
XbDSMult          xbmultds      ->    MULT AT integer;
XbDSSkip          xbskipds      ->    SKIP AT integer;

// Port declaration section
PortSpec          portspec      ->    PORTS portdecllist END PORTS;
PortSpecLoop      portspec      ->    PORTS LOOP portdecllist END PORTS;
PortDeclListOne   portdecllist  ->    portdecl;
PortDeclList      portdecllist  ->     portdecl portdecllist;
PortDecl          portdecl      ->    identifier '=' integer ';';

IntegerDec        integer       ->    'dec';
IntegerHex        integer       ->    'hex';
HashInt           hashint       ->    HASHINT;

Identifier        identifier    ->    idstring;
QuotedString      quotedstring  ->    qstring;
```

C.2   *Tier2* Grammar Reference

The following is the language grammar for the *Tier2* language, as specified using Visual Parse++.
This is the definitive reference for program structure, since the compiler itself is created using this
file. Refer to [San94] for a complete guide to the format of this file.

```
//
// Rule file for the tier-2 compiler
// This is essentially identical to the tier1 specification, except
// that now the user does not specify programming directions for individual
//blocks,
// and data sources are specified by name, rather than by direction.  The
//place/route
// algorithm takes care of figuring out all of the directions.
//
// Now, only a single stream is allowed, and it takes no port parameters.
This //is simply a means of
// grouping together block defs and crossbar stmts.  The stream represents all
//of the programming // information to be created.
%expression Main

'[ \n\t\r]+'                    %ignore;
'[a-zA-Z][a-zA-Z0-9_]*'         idstring;
'\".*\"'                        qstring;
'/\*'                                   %ignore, %push MultiLineComment;
'//'                            %ignore, %push SingleLineComment;
';'                             Semicolon,';';
':'                             Colon,':';
'='                             Equals,'=';
'!='                            InvEquals,'!=';
'=>'                            Arrow,'=>';
'\('                            LeftParen,'(';
'\)'                            RightParen,')';
','                             Comma,',';
'\['                            LeftBracket,'[';
'\]'                            RightBracket,']';
'\|='                           OrEqual,'|=';
'\&='                           AndEqual,'&=';
'<<'                            ShiftLeft,'<<';
'>>'                            ShiftRight,'>>';
'!'                             Not,'!';
'#[ \n\t\r]*[a-zA-Z]+.*\n'      HASHIGNORE;
'#[ \n\t\r]*[0-9]+'             HASHINT;
'[\-+]?[0-9]+'                  Dec,'dec';
'[0-9A-Fa-f]+[hH]'              Hex,'hex';

'[bB][eE][gG][iI][nN]'          BEGIN;
'[eE][nN][dD]'                  END;
'[pP][oO][rR][tT][sS]'          PORTS;
'[pP][oO][rR][tT]'              PORT;
'[lL][oO][oO][pP]'              LOOP;
'[sS][tT][rR][eE][aA][mM]'      STREAM;
//'[oO][pP][eE][rR][aA][tT][oO][rR]'     OPERATOR;
'[iI][nN]'                      IN;
'[oO][uU][tT]'                  OUT;
'[iI][fF][uU]'                  IFU;
'[aA][tT]'                      AT;
'[mM][uU][lL][tT]'              MULT;
'[lL][oO][cC][aA][lL]'          LOCAL;
'[sS][kK][iI][pP]'              SKIP;
'[mM][aA][cC][rR][oO]'          MACRO;
'[aA][sS][sS][eE][rR][tT]'      ASSERT;
'[cC][aA][sS][eE]'              CASE;
'[iI][sS]'                      IS;
'[wW][hH][eE][nN]'              WHEN;
'[dD][eE][fF][aA][uU][lL][tT]'  DEFAULT;
'[bB][lL][oO][cC][kK]'          BLOCK;
'[gG][oO]'                      GO;
'[tT][oO]'                      TO;
```

81

```
'[fF][rR][oO][mM]'                      FROM;
'[cC][rR][oO][sS][sS][bB][aA][rR]'      CROSSBAR;
'[nN][oO][rR][tT][hH]'                  NORTH;
'[sS][oO][uU][tT][hH]'                  SOUTH;
'[eE][aA][sS][tT]'                      EAST;
'[wW][eE][sS][tT]'                      WEST;
'[oO][uU][tT]'                          OUT;
'[dD][eE][lL][aA][yY]'                  DELAY;
'[iI][fF][cC][oO][nN][dD]'              IFCOND;
'[eE][lL][sS][eE]'                      ELSE;
'[rR][iI][gG][hH][tT]'                  RIGHT;
'[lL][eE][fF][tT]'                      LEFT;
'[uU][sS][eE]'                          USE;
'[cC][oO][mM][pP][oO][nN][eE][nN][tT]'  COMPONENT;

%expression MultiLineComment
'.'                                     %ignore;
'\n'                                    %ignore;
'\*/'                                   %ignore, %pop;

%expression SingleLineComment
  '.'                                   %ignore;
  '\n'                                  %ignore, %pop;

%prec
1, idstring,                            %left;


%production start

// The program consists of an arbitrarily long list of  constructs
Start           start       ->      constr;
StartList       start       ->      start constr;

// This is the list of constructs
ConstrPort      constr      ->      portspec ';';
ConstrStream    constr      ->      streamspec ';';
ConstrMacro     constr      ->      macrospec ';';
//ConstrOper     constr      ->      operspec ';';
ConstrComp      constr      ->      compspec ';';
ConstrCpp       constr      ->      cppline;
ConstrError     constr      ->      %error ';';

// Lines thrown in by the preprocessor should be parsed so that
// we know what file we're currently in

// This is to ignore any statements which begin with a '#' and then have
// an alphabetical character following it, i.e. any normal preprocessor
// declaration which slips through
CppIgnore       cppline         ->      HASHIGNORE;
// This matches the special preprocessor declarations produced by the
// preprocessor to indicate include files
CppStart        cppline         ->      hashint quotedstring;
CppLine         cppline         ->      hashint quotedstring integer;

// Component definition section
CompSpec        compspec        ->      COMPONENT identifier blockbody END
                                        COMPONENT;

// Macro definition section
MacroSpec       macrospec       ->      MACRO macrostart macroheader
                                        macrobody END MACRO;
// Macro header if an LHS operand is specified for overloading purposes
```

```
MacroLHS        macrospec       ->      MACRO identifier macrostart
macroheader
                                        macrobody END MACRO;
// Specification for the macro parameter list, etc.
MacroHeader     macroheader     ->      '(' macroparms ')';
MacroStart      macrostart      ->      identifier;
MacroParmNull   macroparms      ->      ;
MacroParmOne    macroparms      ->      identifier;
MacroParmList   macroparms      ->      macroparms ',' identifier;
// Body of the macro- statements allowed within a definition
MacroBodyNull   macrobody       ->      ;
MacroBodyList   macrobody       ->      macrobody macrostmts;
// These are the allowed statements
MacroAssertStmt macrostmts      ->      assertstmt ';';
MacroAssignStmt macrostmts      ->      assignstmt ';';
MacroCaseStmt   macrostmts      ->      casestmt ';';
MacroError      macrostmts      ->      %error ';';


// Definition for an assert statement
AssertStmt      assertstmt      ->      ASSERT '(' assertrange ')'
                                        quotedstring;
AssertRange     assertrange     ->      assignsrc '=' rangelist;
RangeListOne    rangelist       ->      rangesingle;
RangeListMany   rangelist       ->      rangesingle ',' rangelist;
RangeSingle     rangesingle     ->      '[' integer ',' integer ']';


// Definition for an assignment statement
AssignStmt      assignstmt      ->      identifier assignop assignsrc;
AssignEqual     assignop        ->      '=';
AssignOr        assignop        ->      '|=';
AssignAnd       assignop        ->      '&=';
AssignSrcParm   assignsrc       ->      identifier;
AssignSrcInt    assignsrc       ->      integer;


// Definition for a case statement
CaseStmt        casestmt        ->      CASE assignsrc IS casebody END CASE;
CaseBodyOne     casebody        ->      ; //casecond;
CaseBodyList    casebody        ->      casebody casecond casedefault;
CaseCond        casecond        ->      casewhen ':' casestmts;
CaseWhen        casewhen        ->      WHEN integer;
CaseDefaultNull casedefault     ->      ;
CaseDefaultList casedefault     ->      DEFAULT ':' casestmts;
CaseStmtsNull   casestmts       ->      ;
CaseStmtsList   casestmts       ->      casestmts macrostmts;


// Stream declaration section
StreamSpec      streamspec      ->      STREAM streamheader streambody END
                                        STREAM;
// Specifications for the port list
StreamHeader    streamheader    ->      identifier;
// Specifications for the body of the stream
StreamBodyNull  streambody      ->      ;
StreamBodyList  streambody      ->      streambody streamstmts;


// These are the allowed statements within a stream
StreamXBStmt    streamstmts     ->      xbstmt ';';
BlockStmt       streamstmts     ->      blockdef ';';
StreamError     streamstmts     ->      %error ';';


// This is the section for the definition of IFU blocks
BlockDef        blockdef        ->      BLOCK identifier blockbody END;
BlockBodyNull   blockbody       ->      ;
BlockBodyList   blockbody       ->      blockbody blockstmts;
```

```
//DirectionEast   direction     ->     EAST;
//DirectionWest   direction     ->     WEST;
//DirectionNorth  direction     ->     NORTH;
//DirectionSouth  direction     ->     SOUTH;
// Here's where we define the statements that can go within a block
BlockOut         blockstmts    ->     bassignout ';';
BlockBitOp       blockstmts    ->     bassignbit ';';
BlockMacro       blockstmts    ->     bmacrocall ';';
BlockComp        blockstmts    ->     bcompuse   ';';
BlockError       blockstmts    ->     %error ';';

// Component usage statement
BCompUse         bcompuse      ->     USE identifier;

// macro call within a block (not in an assignment stmt)
BmacroCall       bmacrocall    ->     bitmacname bitparmlist;

// Generalized flag modification statement.  Compiler handles special cases
BAssignBit       bassignbit    ->     bittype bitop bitloop bitmacro
bitvalsrc;
BitValSrc        bitvalsrc     ->     xbsource;
BitValSrcNull    bitvalsrc     ->     ;
BitType          bittype       ->     identifier;
BitOpEqual       bitop         ->     '=';
BitOpNotEqual    bitop         ->     '!=';
BitLoopYes       bitloop       ->     LOOP;
BitLoopNo        bitloop       ->     ;
BitMacroNull     bitmacro      ->     ;
BitMacro         bitmacro      ->     bitmacname bitparmlist;
BitMacName       bitmacname    ->     identifier;
BitParmNone      bitparmlist   ->     ;
BitParmList      bitparmlist   ->     '(' bitparms ')';
BitParmsNull     bitparms      ->     ;
BitParmOne       bitparms      ->     integer;
BitParmMany      bitparms      ->     bitparms ',' integer;

// Assignment for the main output
BAssignOut       bassignout    ->     OUT '=' boutdelay;
BOutDelayYes     boutdelay     ->     DELAY bifshiftop;
BOutDelayNo      boutdelay     ->     bifshiftop;
BIfShiftOpCond   bifshiftop    ->     IFCOND bshiftop ELSE RIGHT;
BIfShiftOp       bifshiftop    ->     bshiftop;
BShiftOp         bshiftop      ->     bshift ',' boperator;
BShiftOnly       bshiftop      ->     bshift;
BOpOnly          bshiftop      ->     boperator;
BShift           bshift        ->     bshiftsrc bshiftcomm;
BShiftComm       bshiftcomm    ->     bshiftdir bshiftval bshiftcond;
BShiftCommNull   bshiftcomm    ->     ;
BOperator        boperator     ->     identifier;
BShiftSrcLeft    bshiftsrc     ->     LEFT;
BShiftSrcConst   bshiftsrc     ->     integer;
BShiftDirLeft    bshiftdir     ->     '<<';
BShiftDirRight   bshiftdir     ->     '>>';
BShiftVal        bshiftval     ->     integer;
BShiftCondYes    bshiftcond    ->     IFCOND;
BShiftCondYesN   bshiftcond    ->     '!' IFCOND;
BShiftCondNo     bshiftcond    ->     ;

// Crossbar routing statement
XbStmt           xbstmt        ->     xbsource '=>' xbdest;
// Possible crossbar source statements
XbSourcePort     xbsource      ->     xbportds;
```

```
XbSourceIfu      xbsource      ->     xbifuds;
XbSourceMult     xbsource      ->     xbmultds;

// Possible crossbar destination statements
XbDestPort       xbdest        ->     xbportds;
XbDestMult       xbdest        ->     xbmultds;

// Destination/Source types
XbDSPort         xbportds      ->     PORT identifier;
XbIfuSource      xbifuds       ->     IFU identifier;
XbDSMult         xbmultds      ->     MULT AT integer;

// Port declaration section
PortSpec         portspec      ->     PORTS portdecllist END PORTS;
PortSpecLoop     portspec      ->     PORTS LOOP portdecllist END PORTS;
PortDeclListOne  portdecllist  ->     portdecl;
PortDeclList     portdecllist  ->     portdecl portdecllist;
PortDecl         portdecl      ->     identifier '=' integer ';';

// Define various constants- numerical values
IntegerDec       integer       ->     'dec';
IntegerHex       integer       ->     'hex';
HashInt          hashint       ->     HASHINT;

// Define identifiers and quoted strings.
Identifier       identifier    ->     idstring;
QuotedString     quotedstring  ->     qstring;
```

VITA

Brian Kahne was born a United States citizen on October 11, 1972 in Long Beach California. He moved to the Washington, D.C. area in 1985 and attended high school at Thomas Jefferson High School for Science and Technology. Brian graduated summa cum laude with a bachelor's degree in Computer Engineering at Virginia Tech in 1995 and his Master of Science in Electrical Engineering in 1997. His focus has been on microprocessor architecture with a strong leaning towards computer networking. Brian will be joining Motorola's PowerPC team in August, 1997, where he will be working on performance modeling of the PowerPC chip and its front-end chip set.