Silicon Compilation

Thesis by

David Lawrence Johannsen

in Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

California Institute of Technology

Pasadena, California

1981

(submitted May 13, 1981)

to my wife and son

Acknowledgments

The author expresses his grateful appreciation to Jesus Christ for His patient guidance and many blessings. Special acknowledgments also go to my wife, Janet, for her encouragement and support. I would also like to thank my parents and sisters for their contributions to my development.

I wish to extend my thanks to some of the people who worked with me at Caltech:

to Carver Mead, for his many insights and his sense of humor,

to Ron Ayres, for his friendship and aid, and USC-Information Sciences Institute,

to Jim Tobias, Jeff Sondeen, and John Wawrzynek, for their participation and support,

and to Chuck Seitz, Ivan Sutherland, and the Caltech Silicon Structures Project.

Abstract

Modern integrated circuits are among the most complex systems designed by man. Although we have seen a rapid increase in fabrication technology, traditional design methodologies have not evolved at a rate commensurate with the increasing design complexity potential. These circuit design methodologies fail when applied to Very Large Scale Integrated (VLSI) circuit design. This thesis proposes a new design methodology which manages the complexity VLSI design, allowing economical generation of correctly functioning circuits.

Cost is one measurement of a design methodology's value. A good design methodology rapidly and efficiently translates high level system specifications into working parts. Traditional techniques partition the translation process into many steps; each design tool is focused upon one of these design steps. This partitioning precludes the consideration of global constraints, and introduces a literal explosion of data being transfered between design steps. The design process becomes error-prone and time consuming.

The technique of silicon compilation presented in this thesis automatically translates from high level specifications into correct geometric descriptions. In this approach, the designer interacts at a high level of abstraction, and need not be concerned with lower levels of detail, facilitating exploration of alternate system architectures. Furthermore, since the implementation is algorithmically generated, chip descriptions can be made correct by construction. Finally, the user is given technology independence, because the high level specification need not require knowledge of fabrication details. This flexibility allows the user to take advantage of technology advances.

This thesis explores various aspects of silicon compilation, and presents a prototype compiler, Bristle Blocks. The methodology is demonstrated through the design of several chips. The practicality of the methodology results from the concern for efficiency of the design process and of the chip designs produced by the system.

Table of Contents

List of Illustrations

Introduction

A circuit qualifies as a VLSI circuit when a single designer, using manual design methods, requires more than one lifetime to complete the design.

Using the above criterion, we are currently at the threshold of VLSI: chips are beginning to take more than a person's lifetime to design. Furthermore, industry experts project that things will get much worse [18]. As fabrication technology advances and the density of circuitry increases, so will the functionality of the chips. As the functionality increases, the complexity of the design will increase exponentially due to the interactions among the circuit elements, and therefore the design time will also increase exponentially. If we are going to exploit these density increases, we will have to change the way in which we design circuits.

Not too many years ago, software design engineers (programmers) were faced with a very similar problem. The computer technology was advancing and giving the programmers ever increasing memory space. Exploiting this tremendous increase in machine capability, program complexity grew rapidly, resulting in exponential increases in design and implementation time, due to interaction between pieces of the program. One very successful tool that developed was the higher-level language and compiler.

Software compilers allow programmers to specify their designs more by intent, on a semantic level, rather than on an implementation level, freeing programmers from the concerns of many nitty-gritty implementation details. Programmers also enjoy the ability to rapidly modify their programs. The compiler handles the tedious and error-prone task of translating the high-level semantic definition of the program to its low-level implementation.

If the techniques and concepts used in creating software compilers were used in creating hardware compilers, or silicon compilers, the VLSI designer would be freed from the complexity of low-level implementation details and could spend more time on interesting tasks like product definition and algorithm research.

Similarities between Software Compilers and Silicon Compilers

Both software compilers and silicon compilers have the same basic goal: hiding implementation details from the user, allowing the user to work at an algorithmic or behavioral level. When writing a program in a high-level language, the user does not want to know exact physical addresses where his code is being placed, nor does he care how the compiler allocates registers, nor even what the instruction set is on the target machine. Were he to be bogged down with this incredible volume of implementation details, software costs would be many times higher than they are today. In exactly the same way, a user designing with a silicon compiler language does not need to know the exact physical locations where the compiler is placing his circuitry, the mask layers that the compiler uses, or even the design rules required by the fab line.

Compilers maintain datatype consistency. Most of the modern software languages require data to be typed: the user must specify what kind of data is stored in the program variables. The compiler can then check to make sure that the variables are used correctly. If the user attempts to store an INTEGER value into a REAL variable, the compiler can either notify the user of an improper assignment, or convert the INTEGER data into the REAL format before completing the assignment. In like manner, silicon compilers can verify the usage of cells and their interconnections. If the output of one cell, which is valid during one clock phase, is connected to a second cell which samples that signal during a different clock phase, the compiler can either notify the user of the timing error or take corrective action on its own.

The outputs of compilers are correct by construction: no one takes every output of a FORTRAN compiler and runs a design rule checking program on it to verify that the assembly code correctly implements the source specification. Similarly, the output of a working silicon compiler is guaranteed correct. The resulting designs do not have to be run through DRC programs to verify the correct design of the chip. This is very important as we approach VLSI, where the time and cost to perform a full DRC are astronomical.

Compilers allow for continuous modification of the design. When the specifications for a software program are changed, the program is modified to reflect the change. Rarely is the entire program scrapped and written anew, as

might be the case if the program were written in machine code. It is quite natural in the software world for programs to go through many revisions as new discoveries are made. In hardware design, machine specifications are in a constant state of flux, but radical changes to the design are unaffordable if large portions of the design must be redone, as is usually the case if the design consists of geometric primitives. Compilers make changes affordable, giving the designer freedom to explore many design strategies.

Compilers work with templates. Software compilers use templates for each of the basic language constructs. For instance, the IF-THEN-ELSE statement

```
IF <expression> THEN <statement1> ELSE <statement2> FI
```

always compiles into the machine code

```
        evaluate <expression>
        BRANCH IF FALSE label1
        <statement1>
        GOTO label2
label1: <statement2>
label2: .......
```

In a similar manner, silicon compilers have templates, called 'floorplans', for the constructs in the language. These floorplans explicitly state the wiring management for the chip and describe how the pieces of the chip connect together.

Differences between Software Compilation and Silicon Compilation

The first major difference between the tasks of software compilers and silicon compilers concerns the dimensionality of the result. Software compilers generate a one-dimensional result. The location, or address, of any resulting machine word is a single number. Silicon compilers generate a two-dimensional result. The location of any resulting primitive device is given as both an X and a Y coordinate.

To be completely general, silicon compilers would have to do box-packing which, for large designs, becomes impractical. To avoid this problem, silicon compilers make use of the hierarchy of the specification, equating the physical hierarchy with the logical hierarchy as specified by the user and directed by the floorplan of

the compiler.

The second major difference between software compilation and silicon compilation concerns the communication between various pieces in the design. In software, the GOTO and CALL instructions provide linkage between the various modules. For all practical purposes, the communication costs between any two points in memory is constant, regardless of the relative positions of the two points. In silicon, wires provide the linkage between the modules. The cost to communicate between two points is directly related to the relative positions of the points: the further apart the points are located, the more area and time/power are required to implement the communication. In addition, the wire can not be routed arbitrarily across the chip because it must avoid other modules and wires that might be in its path. In fact, a communication path may be impossible due to obstacles on the chip. A software GOTO has no obstacles; it doesn't have to dodge a certain set of words in memory, and it doesn't modify every word that it has to pass over.

It is interesting that this second problem, the GOTO, can be solved using the same technique as the solution to the dimensionality problem: through the hierarchy. Using a hierarchical description of the chip, communication between modules can only occur in well-defined manners between objects on a single level of the hierarchy or between two adjacent levels of the hierarchy. The floorplan of the compiler can guarantee these two types of communication, and hence the silicon compiler can compile any chip which can be specified in the compiler's language.

Design of a Silicon Compiler

This thesis explores some of the possibilities available using the silicon compilation concept. The first section reviews various design techniques and weighs the potentials of design tools. Some of the newer concepts are dicussed in detail, with practical examples to illustrate the techniques. The second section documents a working silicon compiler and dicusses the design tradeoffs and experience learned using the compiler. The work presented in this thesis was performed by the author except where explicitly stated otherwise.

Part One

Design Methodologies

Chapter 1: The Design Tool Space

The first integrated circuit masks were designed completely 'by hand'. All of the geometric features were drawn directly on the film used to expose the working plates. The designer was completely free to lay out any circuit desired, but every single shape had to be drawn on the film. As digitally controlled film plotters were developed, the design style incorporated computers to drive the plotters. The masks were still designed on mylar, but then the designs were 'digitized': the geometric shapes were described to the computer. The computer then drove the film plotter to make the actual masks [2].

Once the computer-film plotter team was introduced, it was noticed that the final goal of the layout designer was not necessarily to generate the actual masks, but to build the data description in the computer's memory. Methods were developed to enter the geometric shapes directly into the computer, rather than working on mylar then digitizing the result. Interactive graphics systems allowed the user to design the circuits directly on a CRT screen and get instant visual feedback of the computer's perception of the design [1][7][8][33].

The design task using interactive graphics was still formidable. The user had to verify that every geometric shape met all of the process design rules. The fabrication processes were in a constant state of flux, so the design rules frequently changed. Due to the long design cycle for large chips, these chips could not use the state-of-the-art technology. Work then proceeded to divorce the design rules from the design. If the design could be specified independent of the design rules, and if a program could then convert this technology-independent layout description into the actual artwork, chips could make use of the most advanced technology, and as the technology advanced, the new masks could be generated from the old designs. This design technique was called the 'sticks' approach [21][24][27][28][32][34].

These design methodologies are fundamentally 'Geometric' techniques. The basic atoms of the design are graphical objects, and an object in the design, like a register, is a collection of graphical objects. As these graphics techniques were being developed, a totally different approach using 'Procedural' cell design techniques was being explored. In the procedural methodology, the cells are described as a program which, when executed, would generate the description of the layout

directly in the computer memory.

The first step towards procedural cell design was to develop languages for describing artwork. As these special purpose languages were used, it was noticed that there was a need for higher level programming constructs, which led to the development of an 'Imbedded Language'. Rather than design a special purpose language from scratch, routines for generating the geometric shapes were written in a high level programming language. The designer in using an imbedded language has the full power of the high level language to describe the layout [4][19][31].

When chips are designed using the procedural approach, a large collection of subroutines are written to implement each of the low-level units of the design. To complete the design task, these pieces must be glued together. The tedious and error-prone wiring task can be done by a program. An imbedded language system with automatic wiring generators and cell management systems becomes a 'Chip Assembler'. In the chip assembler, the designer is interconnecting a series of macro-modules. The user designs the low level layouts, while the program generates the wiring that puts together the chip [6][29].

Extending this approach one step further, the high-level features of a chip class can be hardwired into the design system. To design a particular chip in this class of chips, the user need only specify the unique features of the chip, allowing the program to automatically generate the remainder of the chip. These systems are called 'Silicon Compilers', reminiscent of software compilers used to write programs [5][12][13].

We have mentioned six basic design approaches here. Each of these systems has advantages for certain design requirements and disadvantages for others. We will discuss some of the design requirements here to put a perspective upon the design style used throughout this thesis.

1.1: Flexibility

One comparison of the design tools that can be made is that of flexibility: How flexible are the design tools? Perhaps the first type of flexibility that comes to

mind has to do with the architectures of chips. Can we design any type of chip with a particular design tool, or does the tool restrict the kinds of designable chips. The most restrictive design tool presented here is the Silicon Compiler. The Silicon Compiler accepts a formal, high-level language specification of the chip to be implemented. Hence, the kinds of chips compilable with a particular compiler are limited to those expressible in the language. If you can express the chip in the input language, you can compile the chip. Chip Assemblers are more flexible than compilers. With an assembler, the user fuses collections of macro-modules to form the chip. The number and complexity of the macro-modules, along with the communication costs, are the primary limiting factors on chip architecture. Still more floorplan-flexible are the Sticks and Interactive Graphics systems. These only limit the geometric primitives available in the design, and restrict the cell boundries to be rectangles. Finally, hand design and imbedded language systems allow the most flexible design system. It is possible to design any designable chip with these two system.

The above discussion talked about the absolute limits of each of the design system. On the other hand, there are practical limitations to each of the tools. Perhaps the biggest limitations are design time and the notorious complexity issues. While it is theoretically possible to design any chip with the hand design methodology, the implementation time may be astronomical. Similarly, the design complexity may be so large that it is virtually impossible to design a provably correct chip in a reasonable time. Systems like silicon compilers, however, may be able to design these chips in a very short time.

Another flexibility measure has to do with technology dependence. The silicon technology is always advancing. Are the tools able to take advantange of technology advancements? It is here that the Sticks design systems really shine. Since the system performs all of the design rule dependent operations, the user designs 'technology free' designs. This does not mean that the user is not aware of the CMOS/NMOS differences, but the user does not need to see differences in various NMOS processes. Each of the other design systems require work to modify an existing design to make use of new technology. For silicon compilers and chip assemblers, the cell libraries have to be redesigned for the new design rules. For the other systems, the entire chip must be redesigned when the technology is modified.

A third flexibility might be called specification flexibility. When the specifications for the chip change, how much work is it to redesign the chip? During the design of virtually every chip, ways are found to make the chip better. Another design team may change the environment of the chip, or the chip designers may discover a hidden cost during the implementation of the chip. In any case, it may be very desirable to 'start all over' and re-implement the chip. In many cases, a redesign of the chip may require starting from scratch, and this cost of scrapping the whole design may be prohibitive. If the company has invested several man-years in the design, the design modifications are not economically feasible. With a silicon compiler, however, a company invests man-days, not man-years, into a design. With this small investment into the design, redesigns are virtually free. The designer may quickly and easily explore many design tradeoffs, which are impossible with the other design techniques.

1.2: Specification

Part of the process of designing a chip may be thought of as specification translations. The design team is given an input specification of a chip, usually a functional specification, and must produce an output specification of the chip. This output chip specification may be a large drawing of the chip, as in hand design, or the specification may be a data structure in a computer's memory, as in graphics and sticks systems, or the specification may be a program in a high-level language, as in the compiler, assembler, and imbedded language systems. The fundamental task of the design team is to translate a specification in the input language to a specification in the output language. This translation process may be accomplished in one step, or in many steps with different design groups performing each step in the translation.

One would like to match the output language as closely as possible to the language used by the design team. If the design team is working with logic equations, one would like the output language to be logic equations; if the design team worked with Register Transfer (RT) equations, the optimal output language would be an RT language. This language match is desirable for two reasons. First, the design specification would be intuitive, so that the designers can easily express their intent in the language. An expression in the language could be easily understood by

the designers. Second, the designers can directly produce the output specification as the chip is being designed.

If the design language is intuitive, a great majority of the design errors can be avoided. If the specification is non-intuitive, it is difficult for the designers to catch design errors in the chip specification. Of the six design tools mentions, the imbedded language systems are perhaps the least intuitive. The user wishes to implement a function. Short of that, the user wishes to describe the picture of a circuit to implement the function. In imbedded language systems, the user writes a program to generate a picture to implement the function. Things are better with the hand design, graphics, and sticks tools. With these tools, the user directly generates the picture to implement the function. The most intuitive systems, however, are the assemblers and compilers. Here, the user describes the function, which is the desired quantity.

When the designers are directly designing in the output language, the chip specification is complete when the designers have finished implementing the function. If the designers' specification has to go through translations or re-specifications, there is a greater probability of errors. Therefore, the output language should closely match the design language used by the designers.

The specification language can enforce design correctness. With a suitable specification language, it is impossible to generate most design errors because the language does not allow for the specification of errors. For example, the Sticks design system does not allow the user to design circuits with dimensional design rule violations because the Sticks language does not permit the specification of dimensional information (except transistor sizes). Since the user can not specify the spacing between two metal features, it is impossible for the user to design circuits with metal-to-metal spacing violations. Similarly, with assemblers and compilers, it may be impossible for the user to generate chips with timing errors or logical interconnection errors simply because the input languages do not permit specification of these errors.

1.3: Composition

The design of the low level cells, which comprise 80% of the chip area, typically takes less than 10% of the design time, with the rest of the design time consumed by the interconnection of these low level cells. Most of the design errors occur in these interconnections, also. Low level cells are small, self-contained units that the designers can completely understand while the cell is being designed, while the interconnection cells are large, global units which are impossible to fully understand. A good design system should aid in the composition of cells.

There are two sides to composition systems. On one hand, the system should aid in the generation of interconnect geometry where needed. If the design tool can automatically generate the interconnection geometry, a great deal of the interconnection design time can be performed by the machine. Secondly, the system should verify that the interconnection was correct. The verifications may assure that electrical and timing constraints are met. At a higher level, the composition system may verify that the logical contraints are met, and that signals are used correctly.

Currently, very few of the design tools have the conception of cell composition. The chip assembler and silicon compiler have squarely faced the issue of chip composition and interconnection verification. In the other systems, it is difficult to see how a composition system can be added.

Closely related to the composition aspect of the chip design system is the hierarchical philosophy of the system. The hierarchy of virtually all design tools is recursive. You are either dealing with a composition cell or with a leaf cell. All composition cells look and act the same. This means that the same design tool can be used to design every composition cell on the chip. Unfortunately, very few hierarchies exhibit a recursive nature.

In human hierarchies, large companies, for instance, one sees several levels of management directing the operation of the companies. Other than the fact that people fill each of the positions in a company tree, there is little similarity in each of the positions. The tasks of the vice presidents are very different from the tasks of the section managers. The chairman-of-the-board's job relates little to a project

manager's job. A person well suited to one of these jobs can not in general fill another person's job.

Similarly, we have a hierarchies in our design systems. At a low level, the user may be dealing with polygons. At higher levels, the user is dealing with flip-flops, registers, ALUs, microprocessors, then complete systems. A microprocessor is not the same sort of object as a register. One does not design an 68000 the way one designs a static D-flip-flop.

Most existing design tools are recursive systems. At the highest level of design, the user is still drawing boxes and polygons. The primitives of the design system are still the graphics primitives, rather than being data buses, registers, or microprocessors. The silicon compiler is a hierarchical system, but not necessarily a recursive system. The system knows the difference between an inverter and an ALU. Any of the other design tools except hand design can make use of non-recursive hierarchies, yet none of these systems currently takes advantage of hierarchies of specification primitives.

1.4: Verification

As VLSI becomes a reality, the verification issue must be squarely faced. In present design systems, verification is done by analysing the graphics primitives which comprise the mask sets. All information regarding the structure of the design has been thrown away. This is like writing a program to analyse the object file produced by a FORTRAN compiler to verify that the compiler is operating correctly. With VLSI chips, it is impractical to perform verification checks by analysing the artwork.

Instead, we must guarantee correctness by construction. If we generate correct layouts, we do not have to verify the artwork. We need only verify our methods for constructing the layouts. The task of verifying our construction methods is much simpler than verifying artwork. Our construction procedures take a well defined input language; we need only verify that every legal input produces correct output. To verify artwork, we must be prepared to accept any input, including tricks-of-the-trade. With the graphics systems and imbedded languages, the input

language is a direct specification of the artwork, so our verification task is by definition the task of verifying the artwork. Hence, it will become impossible to verify VLSI designs produced by the current graphics and imbedded language tools. The assembler and compiler, however, take an input language which is far more concise than an artwork specification. Hence, we have a hope of verifying designs produced by these systems.

Another side of verification has to do with the capturing of intent. When the designer designs a cell, the designer has an intent about how the cell is to be used. To properly use a cell, the user must know this intent and meet the restrictions of the intent. If the user exceeds the limits of the intent, the cell will not function properly. In design systems which consider a cell to be nothing more than artwork, this intent information must be captured in cell documentation, since it can not be captured with the cell. Users of the cell must check the documentation and manually verify that the cell is being used properly. The procedural design systems do not restrict the concept of a cell to just the artwork. The designer writes a program to generate the artwork. The designer can add additional code to the program to capture additional intent. This documentation is kept with the cell. In addition, the cell itself can verify that it is being used properly.

1.5: Efficiency

When one speaks of design efficiency, one usually refers to measures of chip area, chip speed, and chip power consumption. Given a chip specification and infinite time, one would expect hand design and imbedded language systems to produce the most optimal chips, followed by interactive graphics systems, sticks systems, chip assemblers, and finally silicon compilers. These later design systems have area penalties due to fixed floorplans and geometric primitives.

One rarely has infinite time, however, in which to implement a chip. An approximation of the ideal chip must be made. For instance, with hand design methods, one spends a lot of time planning alternate architectures and approximating the design costs for various approaches. Once the range of design candidates is narrowed, detailed design can begin. As the detailed design nears completion, many of the design approximations may be found to have been

erroneous. Due to the large investment in the design, a redesign of the chip is seldom feasible. As a result, the final chip may be non-optimal in the ideal sense, but may be fairly good from a practical standpoint.

With the more inefficient design systems, chips can be implemented much more rapidly than with hand design systems (otherwise these other systems would not exist). Because of this reduced design time, it becomes affordable to iterate the design. When these design approximations are found to be in error, the chip may be redesigned. With the possibility of design iterations, dramatic architecture variations can be explored. Chips resulting from architecture modifications may well have very large performance advantages over the original hand designed chip. In highly complex systems, the system organization has a much greater effect upon performance than the details of low level cells. Thus, even though the resulting chip is known to be less optimal than a hand-design of the same architecture, the chip is more optimal than the hand designed chip since the hand designed chip would not be implemented in the new architecture.

1.6: Conclusions

There are many design techniques in use today. Each of these systems cater to a particular design style. They have various limitations on design capabilities, and they have different aides for the designer. As technology advances towards VLSI, our design requirements are going to change. We will require fundamentally new design principles. Although Silicon Compilers may have undesirable restrictions on the types of chips we design, they provide design capabilities that are impossible to achieve with our present day tools. They have the potential to implement in an hour what current design techniques implement in a lifetime. Machine architecture tradeoffs can be explored in an almost interactive environment. Design verification can be performed at a level previously unattainable. For these reasons, and others, this thesis explores the realm of the Silicon Compiler.

Chapter 2. Hand Design

The fundamental task of designing a VLSI circuit is to manage the complexity of the design. Even modest chips designed today have several million rectangles and hundreds of thousands of devices. Unless the management of the design complexity is squarely faced when the design process is begun, the implementation of the chip may become an impossible task. Fortunately, techniques exist which successfully aid in the management of complexity. In this chapter, we will discuss methods for managing complexity and chip planning.

2.1: Management of Complexity

We can observe complexity management principles being applied in almost every area of life. We can use these same techniques for designing large integrated circuits. Three of these techniques will be examined. One technique is the use of conventions, a second is partitioning of the design, and a third involves abstraction of data.

Examples of the first complexity tool, conventions, are readily observable in daily life. Traffic signals are a successful convention in our modern world. If everyone agrees to abide by the restrictions implied by traffic signals, a much more complex and inefficient system of maintaining road safety can be avoided. Traffic laws and Law Enforcement Officers assure us that (almost) everyone agrees to the convention. In VLSI design systems, conventions can be made with regard to functional partitioning or timing relationships. If the designer faithfully adheres to these conventions, he may feel confident that the design will operate correctly. If there are circumstances where the designer feels that the conventions should not be followed, he will have to take extra steps to verify that the circuit will still operate correctly.

The second design aid is partitioning. Rather than solving a large problem all at once, the problem can be broken into smaller, separable pieces each of which is easier to solve than the original problem. This process may again be repeated for each of these new, smaller problems, until we are left with simple problems that are straightforward to solve. If we have properly partitioned the problem, each of

the solutions can be combined to solve the whole problem. To allow each of these separate solutions to be used together, we must design and specify an interface between the pieces. For example, in software programming, a large program is broken into several subroutines. To assure proper operation of the collection of subroutines, guidelines concerning register and memory usage, data structures, calling conventions, and parameter types are developed and adhered to.

The third aid to handling complexity is data abstraction. There are (at least) two branches of physics dealing with objects in motion. In Classical Mechanics, everything is very deterministic, and we treat objects like air pucks as indivisible, uniform objects. If we look very closely at our air pucks and how they interact, we find that Classical Mechanics does not precisely describe the observable interactions. We use Quantum Mechanics when we need these precise equations. If we look closer still, we find discrepancies between the physically observable events and the calculated Quantum Mechanical events: Quantum Mechanics does not completely define how our air pucks work. In both of these cases, we know that our theories and formulas are wrong, and yet we can still profit by using them. In each of these fields, approximations are made. We do not look at each of the individual subatomic particles which compose an air puck. Instead, we abstract this incredibly large amount of data into a fairly simple model. Similarly, for VLSI design, we do not have to examine every single geometric primitive within a region of the chip when designing the neighboring regions. Almost every function implemented on a chip, certainly every function of reasonable size, requires two areas of silicon: The first is a private area over which this function has exclusive rights, and the second is an interface area, where external signals connect to the function. To use this function, no knowledge of the private area is required. We can abstract the function to an interface and a 'black box', and hence have less information externally required to use the function. By imposing suitable design conventions, the interface area can be a small percentage of the total function area, which greatly reduces the data requirements.

These three techniques of complexity management are used in the cell concept of VLSI design. A cell's layout is defined to be a rectangular area of silicon with the geometric shapes required to implement the cell's function contained completely within the rectangular limits and an interface area limited to the perimeter of the area. Along this perimeter, there are 'ports' or 'terminals', where external signals

may connect to this cell. No external cells or geometric shapes may extend within a cell's rectangular limit, the minimum bounding box (MBB) of the cell.

An important by-product of the cellular design approach concerns data sharing: if a function is replicated on a chip (or across many chips), the layout which implements the function can be shared between the various instances. The function is converted to silicon once, and the resulting pattern can be used many times, thus factoring the design cost of the chip. This layout sharing is identical to the use of subroutines for code sharing in software programming.

The internal structure of a cell's layout consists of combinations of primitive geometric shapes and instances of other cells. This recursive nature of the cell definition allows us to hierarchically design chips. Rowson [25] has defined two types of cells: Leaf cells and Composition cells. Leaf cells contain only geometric primitives, no references to other cells. Composition cells contain only references to other cells, no geometric primitives. We will use this Leaf cell definition, but we will allow our Composition cells' layouts to contain geometric primitives. Adding geometry to composition cells is done for conceptual ease; simple transformations convert from one form to the other.

2.2: Chip Planning: The Floorplan

The arrangement of subcells within a composition cell can have a dramatic effect upon the size and performance of the chip. To aid the user in composing cells, the notion of a 'floorplan' has been developed. A floorplan is the blueprint which indicates topologically how the subcells fit together to form the complete cell. The floorplan also shows the wiring strategy used in the cell. Floorplans are invaluable aids for top-down chip planning. The relative size and placement of the major subdivisions of a cell are quickly visuallized. The communication costs for various arrangements can also be determined.

To illustrate the use of floorplaning, we will discuss the planning for the OM2 datapath chip [15][16]. A functional block diagram of the datapath chip is shown in figure 2-1. At the highest chip level, we needed a chip with three bi-directional Input/Output ports. These were to communicate with the datapath of the chip. One

of these ports was to be mainly a control port, which brought the instruction word into the decoders. The other two ports were data ports, connected to the internal data buses. In addition to the datapath, we also required some flag logic, and additional control input pads. Our primary data flow was to run horizontally through the chip and the primary control flow was to run vertically.

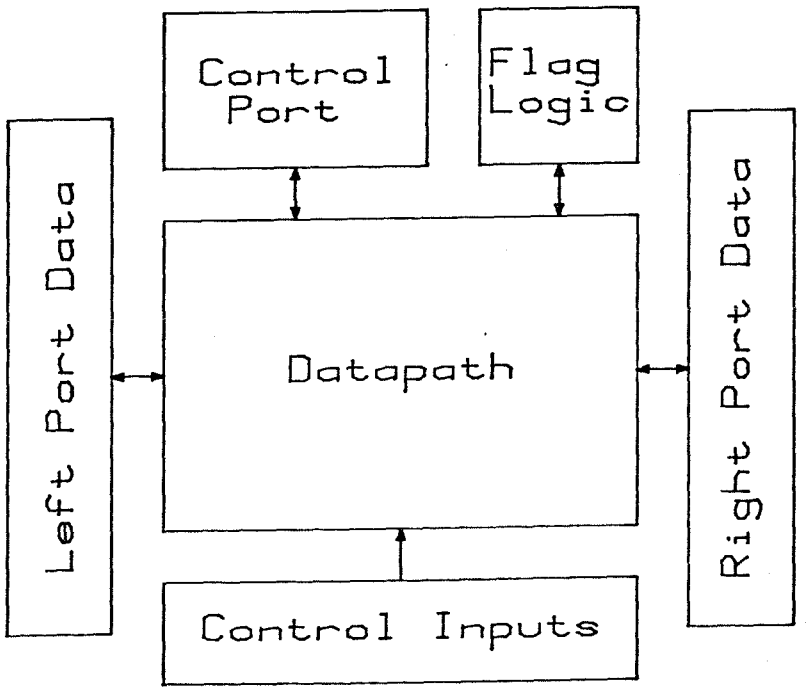


Fig. 2-1: OM2 Datapath Chip Block Diagram

Figure 2-2 shows the high level floorplan for the chip. We have the two data ports on the west and east edges of the chip, the datapath in the center, the literal port and flags to the north, and the control input pads on the south. The sizes of the various boxes were estimated, considering the functions of each element, which completes the planning of the highest level composition cell layout.

We can now decompose the subcells within the global floorplan. The datapath section was to be composed of data processing elements and instruction decoders. The decoders were to take the microcontrol bits entering the chip and drive each of the processing elements' control lines as a function of the input. The instruction decoder was broken into two sections. One section was placed above the processing elements, the other was placed below the elements. This was done because the cell size estimates showed the processing elements to be much narrower than the full

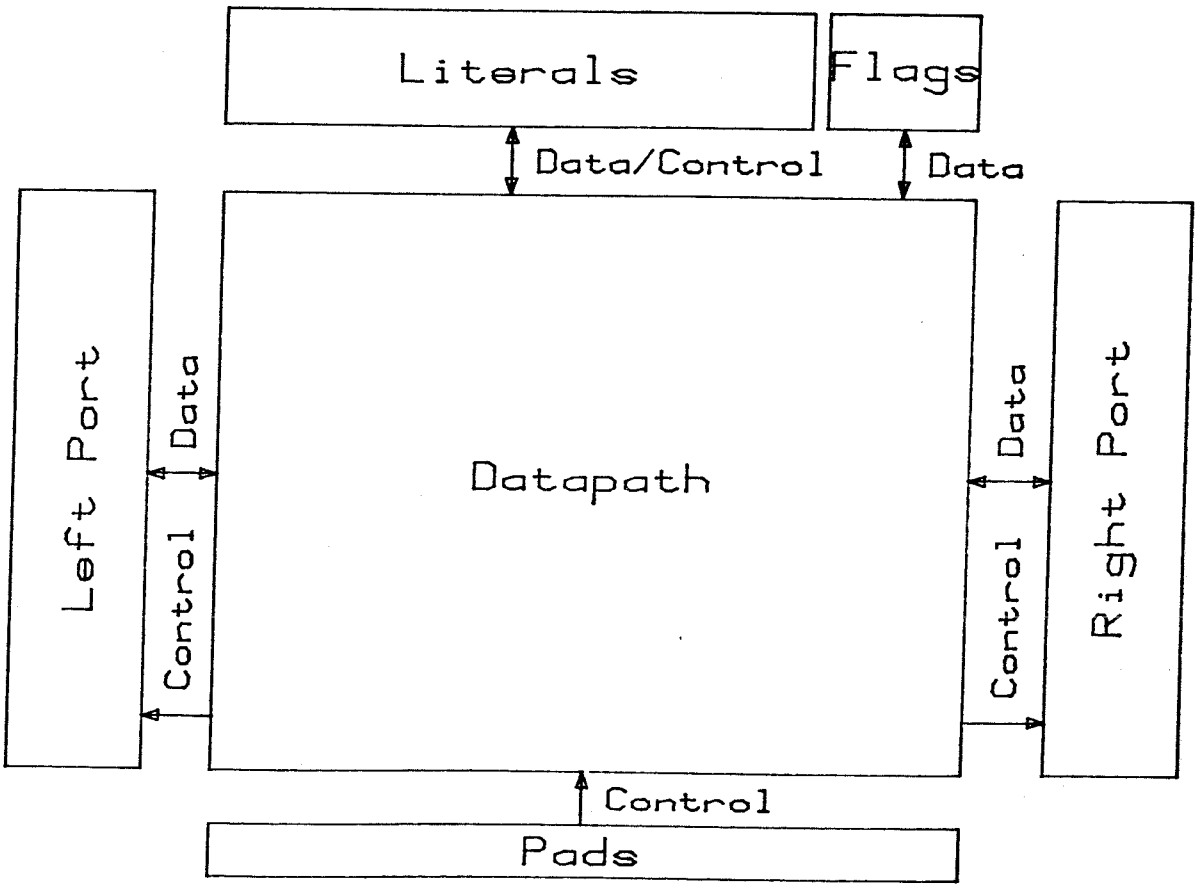


Fig. 2-2: Full Chip Logical Floorplan

decoder. Buffers were placed between the decoder and datapath. These buffers synchronized the decoder's signals, and satisfied the electrical requirements of driving large datapath loads from weak decoder outputs. Figure 2-3 shows the floorplan of the datapath section of the chip.

Decomposing the processing element section, we needed a register array, a barrel shifter, and an ALU. To relieve much of the register bottleneck, we had two data buses running between the individual elements. Each of the registers in the array could read or write to either bus. The shift array read data from the bus and drove the ALU multiplexer. The ALU could read data from either of the buses or from the shifter. The ALU and shifter were chained together to speed up the multiply and divide operations. The ALU's output could drive either bus. The buses also connected to the two data ports. Figure 2-4 is the floorplan for the processing elements.

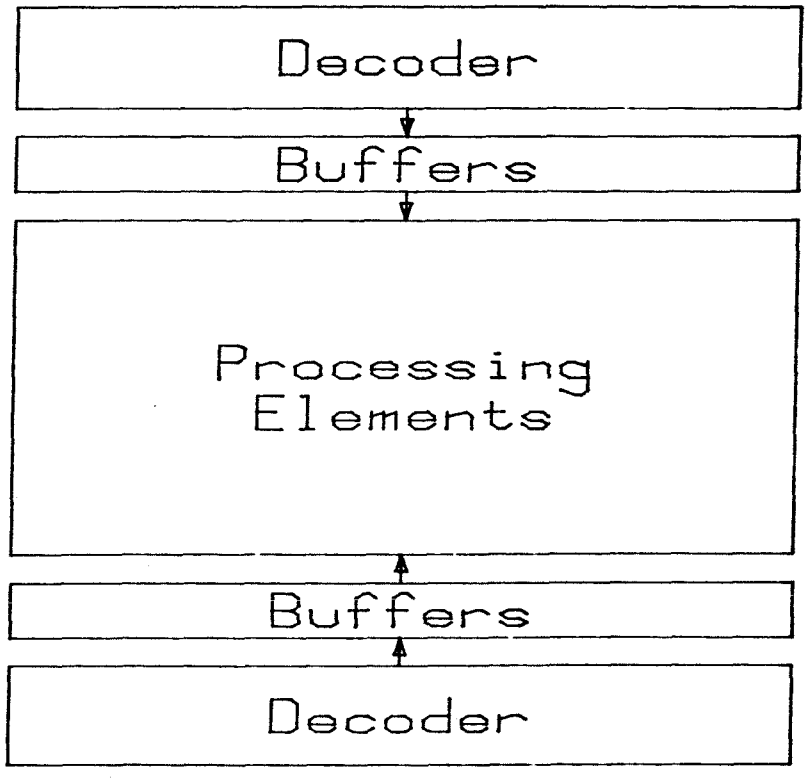


Fig. 2-3: Datapath Logical Floorplan

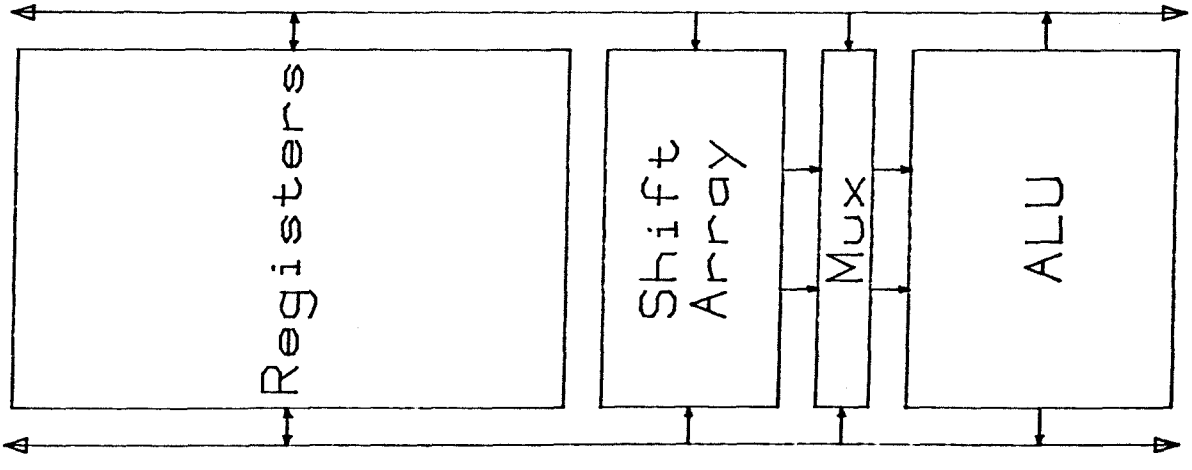


Fig. 2-4: Single Bit-Slice Logical Floorplan

We can further examine the ALU. The ALU was built of five leaf cells. The first section was the two input registers. The second section was the logic for computing carry propagate, carry kill, and carry generate. The third section was the actual carry chain. The fourth section computed the ALU output as a function of the carry input and the carry propagate signals. The final section was the output

registers. Figure 2-5 shows this layout.

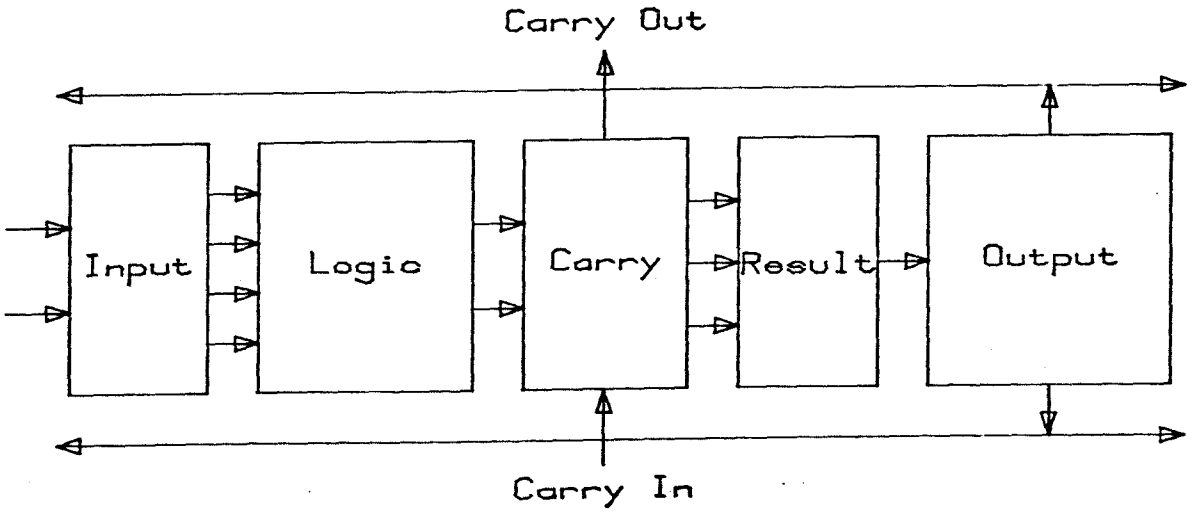


Fig. 2-5: ALU Logical Floorplan

At this point, we began designing the leaf cells. For example, we could see that the input registers received data from the left side of the cell and drove data out the right side of the cell. The two data buses ran through the cell, but were not used by the cell. Control lines for the cell ran vertically through the cell. Once the ALU cells were designed, we could draw the physical floorplan for the ALU. In figure 2-6, we have the subcells shown to scale. We have also shown the layer for each control and data line. The control lines ran in metal, and the data buses were run in polysilicon.

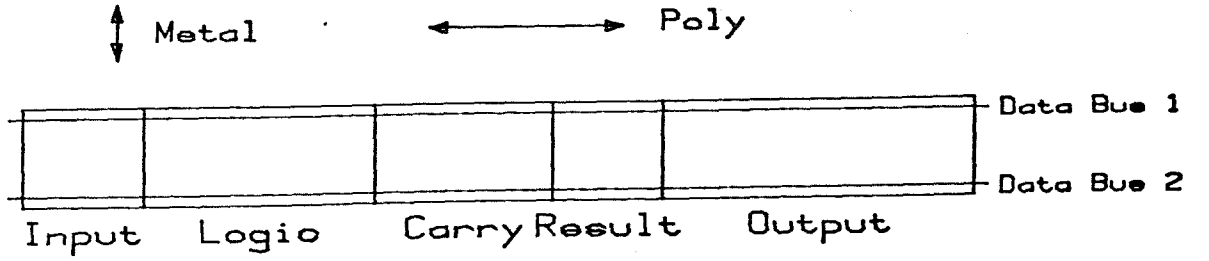


Fig. 2-6: ALU Physical Floorplan

As the remaining datapath elements were designed, the bit-slice physical floorplan took shape (fig. 2-7). The control and data lines in the register array had different layer conventions than the other processing elements, as shown in the figure. Similarly, the datapath floorplan (fig. 2-8) and finally the entire chip floorplan

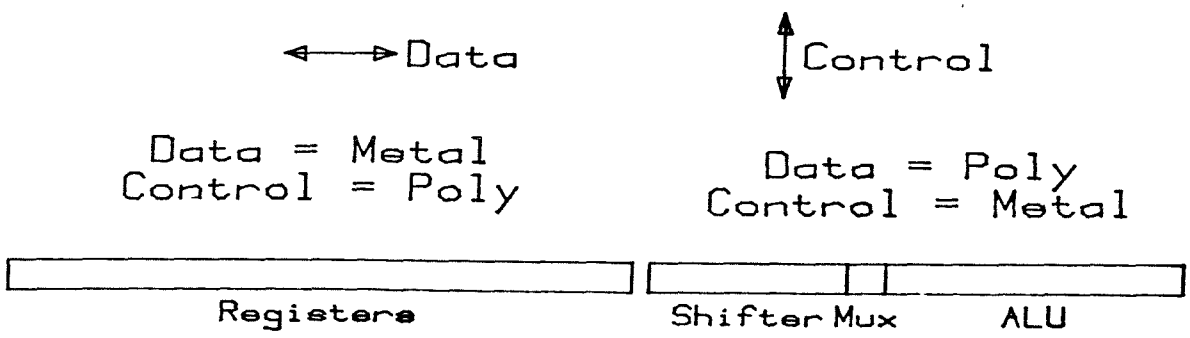


Fig. 2-7: Bit-Slice Physical Floorplan

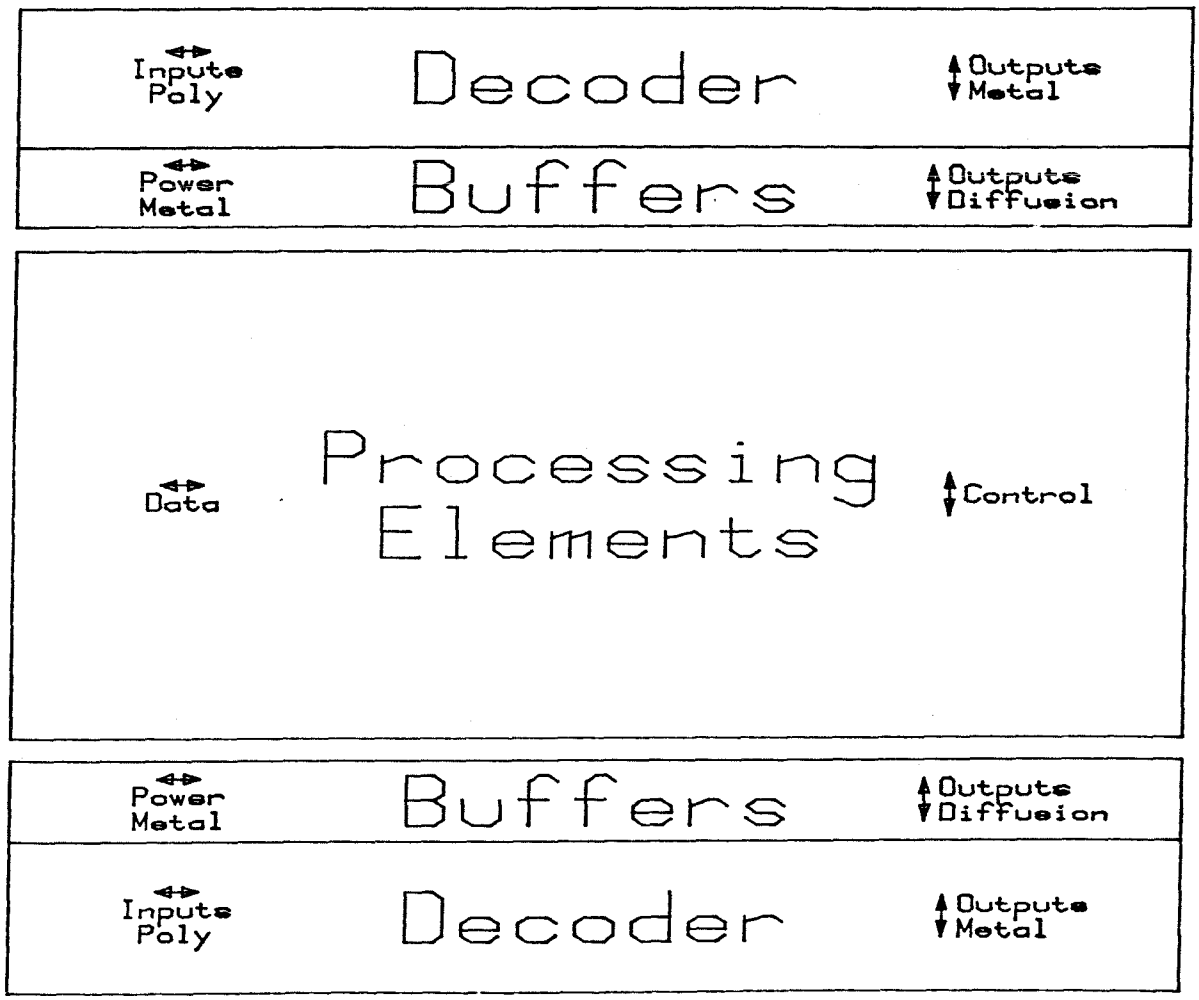


Fig. 2-8: Datapath Physical Floorplan

(fig. 2-9) were completed in the same manner. The final chip layout is shown in figure 2-10. Much of the regularity of the design was due to the use of floorplanning. Due to the regularity of the design and the completeness of the planning, the chip was designed in nine man-months.

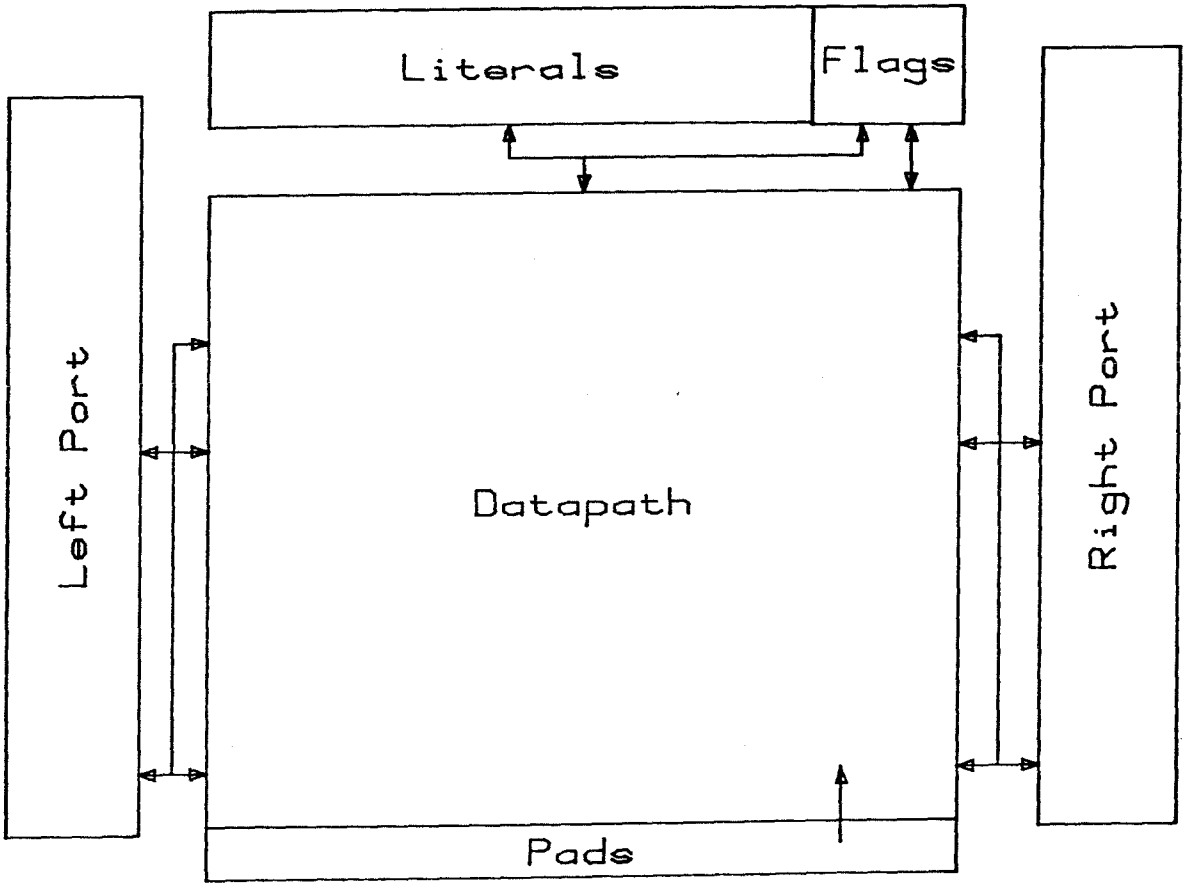


Fig. 2-9: Full Chip Physical Floorplan

2.3: The Slicing Floorplan

Specific chip architectures have related floorplans which are suitable for those particular chip structures. To build general purpose design tools, we would like to have general models for cells and floorplans. We can then build the tools to take advantage of the resulting floorplans.

In top-down design, we take a description of a large unit, and decompose this unit into simpler, smaller units. Each of these units can be similarly decomposed. This decomposition process continues until all of the descriptions an be easily

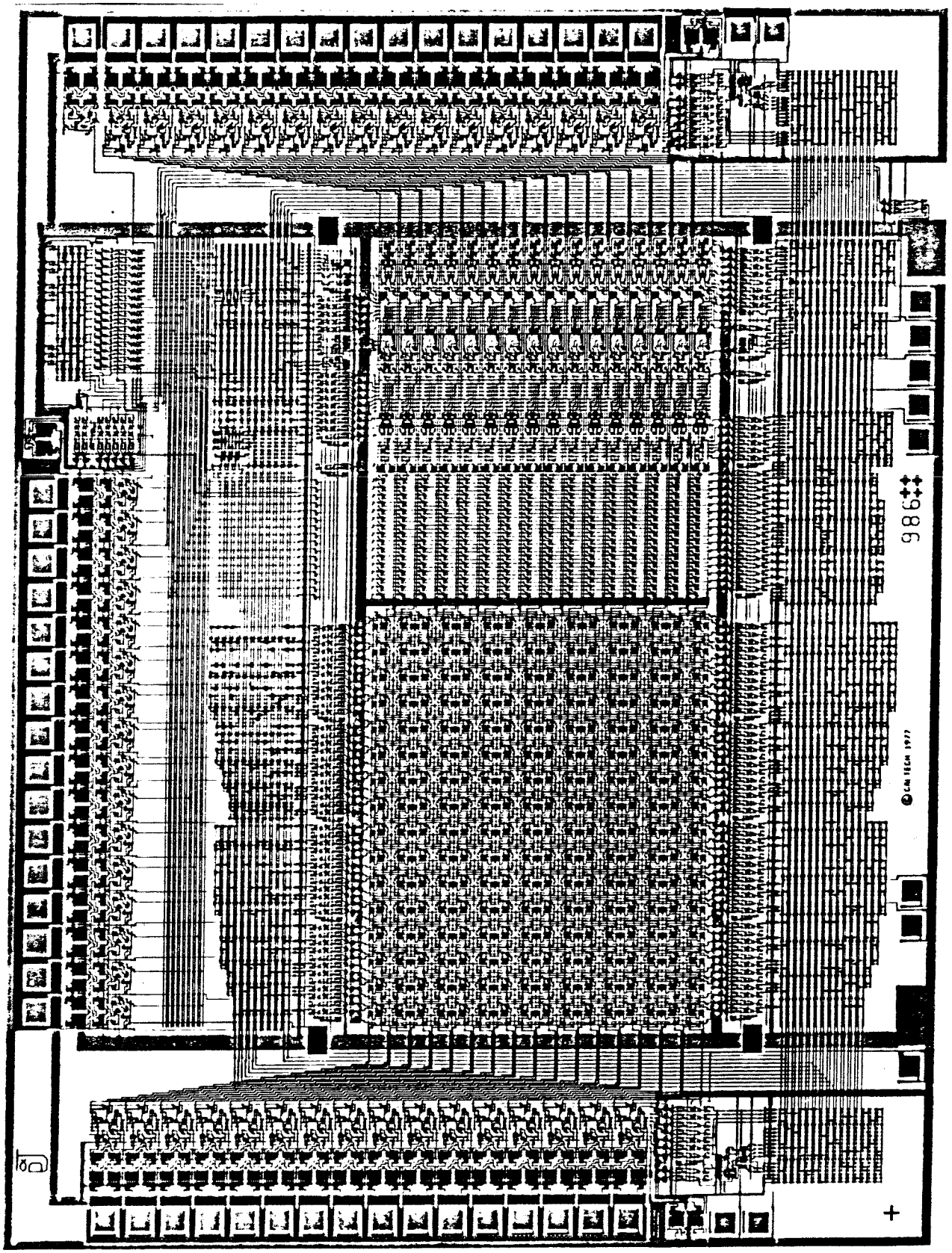


Fig. 2-10: OM2 Datapath Chip Mask Set

implemented. We then work bottom-up, fusing these lower-level implementations to form implementations for each of these larger units. When we reach the highest chip level, we have an implementation of the chip. We want our general purpose floorplan to model the top-down design, bottom-up implementation style of design.

Our complexity management strategy uses rectangular cells. Our general-purpose floorplan will therefore use rectangular cells. To perform top-down design, we need to provide the capability of decomposing cells. To decompose a cell, we will divide the given rectangular cell into smaller rectangular regions. To perform bottom-up fusions of cells, we need to interconnect each of the subcells to form the implementation of the given cell.

Completely general 'glue' between the cells would allow transistors to be added in the interconnections between the cells. Allowing transistors between cells is usually an example of local optimization, rather than global optimization, and the specification and verification of these 'glue' circuits can introduce many errors into the design. Therefore, for our general model, we will restrict all transistors to lie within subcells, and only allow wiring to fuse the subcells together.

If we allow completely general subdivision of a cell into subcells, we may have no preferred order of composition. With preferred composition orders, we can achieve more optimal circuit layouts. Without preferred composition orders, we can not determine the optimum design for a particular cell until every other cell on the chip has been designed. Hence, we can never achieve an optimum design, although we can approach optimal designs by iterating the design many times. Figure 2-11 shows a rectangular arrangement of cells that does not have a preferred order. Not only can we not determine a good order for cell generation, but we can not determine a good order for routing the wires in the four wiring channels.

A floorplan that does have a preferred composition order is the Slicing floorplan. A slicing floorplan has the following definition.

A Slicing Cell is either
1) A Leaf cell,
2) Two Slicing cells with one to the right of the other, or
3) Two Slicing cells with one above the other.

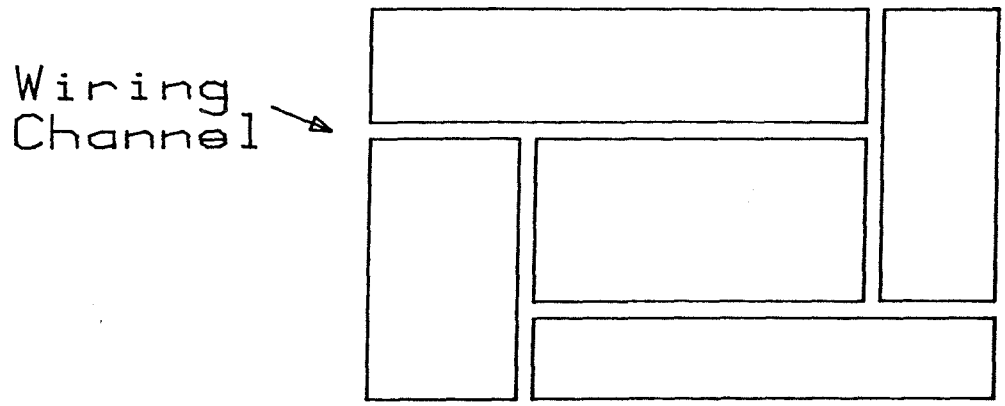


Fig. 2-11: Floorplan with no Preferred Order

Figure 2-12 shows the three possible types of slicing cells. Due to the recursive nature of this definition, we have the capability of designing a rather large collection of chips. Figure 2-13 illustrates the process of decomposing a chip by slicing.

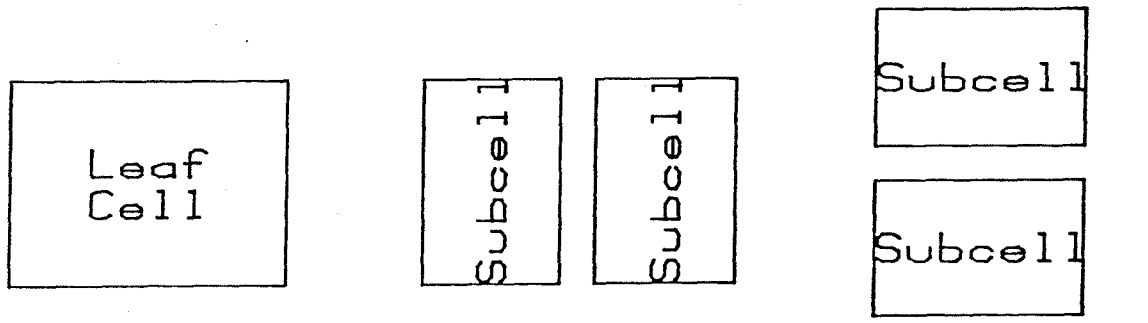


Fig. 2-12: The Three Slicing Cell Floorplans

For the systems described in this thesis, we will use the Slicing floorplan as the floorplan model. While other floorplans can use these same techniques, the mechanics of building the tools may be more difficult, and the examples may not be as clear as Slicing examples.

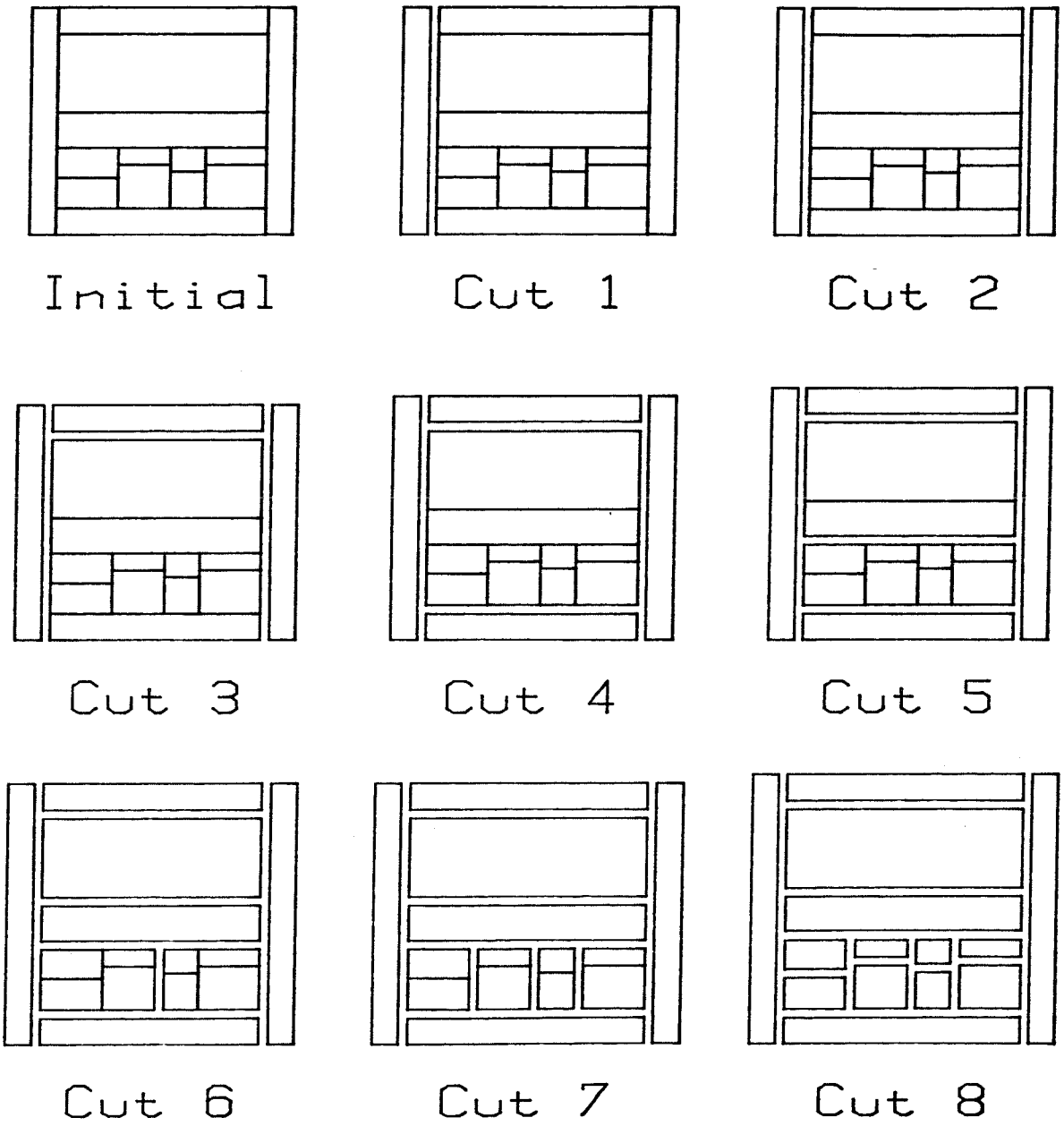


Fig. 2-13: Slicing a Chip

2.4: Global versus Local Optimization

In Small Scale Integration (SSI) and Medium Scale Integration (MSI) designs, much time was devoted to performing 'Local' optimizations on the circuits. This was because the entire circuit was usually considered 'Local'. For LSI, and certainly for VLSI, the situation has changed. No longer is the entire chip design considered a

'Local' design. Where time was previously spent performing local optimizations, time must now be spent performing global optimizations. As our designs increase in size, we must depend more upon global design optimization. Local optimizations can actually hurt the design from a global point of view.

For example, logic design in discrete or TTL design regimes involves 'logic minimization', which actually means transistor minimization. Much effort was spent reducing the number of transistors required to implement functions, because transistors were the expensive part of the design, the wires were free. In silicon design, the majority of the chip area is devoted to wiring. The actual area for the transistors is less than 40% of the total chip area for 'good' designs. In many instances, the transistors are free; they are placed under the wires.

Using wire-wrap boards, the designer has completely arbitrary interconnectability: any pin of any chip can be connected to any other pin of any other chip, regardless of where the chips are positioned on the board and independent of any other interconnections. In silicon, it is very inexpensive to interconnect shared edges of adjacent cells if the connections are well correlated (in approximately the same order in each of the cells). Almost any other circumstance, however, costs a great deal. Wires can not be arbitrarily drawn across the chip because wires can not arbitrarily cross other wires or cross through cells. Wires consume a great deal of the chip area.

A final contrast between TTL design and VLSI design is the difference in wire loading. In TTL design, the chips are each capable of easily driving fairly long wires (from one side of the board to the other). In silicon, however, the wires can add a large amount of loading to devices, which slows down the operation of the circuit. A small gate can not drive a wire from one corner of the chip to the other in a short period of time. Hence, in VLSI design, circuits which must communicate must be fairly close together, adjacent if possible.

Each of these points argue for global planning. The communication costs for VLSI are the dominating cost of the design. Global optimization of the communication paths can provide greater performance increase than local optimization of each circuit on the chip.

2.5: Conclusions

As we move towards VLSI, we can not continue to design chips as we have in the past. The complexity of design causes the design cost to rise exponentially. Our design tools must be designed to cope with the complexity of the design. We have seen some techniques which aid in the complexity management issues, and we have seen some planning tools which will aid in the global optimization of designs. The following chapters discuss tools which are built upon the techniques presented in this chapter.

Chapter 3: Imbedded Languages

When using the cellular design approach, quite often it is the case that a family of similar cells is developed. Each cell instance within this family shares most of the characteristics common to the family, but has its own personalization which distiguishes it from the others. For example, a group of cells may be designed where the cells perform the same function but each consumes a different amount of power. Another example might be a collection of similar cells where the aspect ratio of the cells varies amoung members.

Purely graphical systems dictate that each member of the group be completely designed, because graphic systems emphasize the differences between cells. For a small family of cells, this is not a problem, but for a large collection, perhaps containing many independant variables, it is not practical to design the entire family. In these cases, users would copy and edit the cell which most closely approximates the required cell.

If constructs for specifying conditional circuitry were added to a graphics language, a designer could specify a cell which represents a family of cells, using the conditional operators to distinguish between members of the family. To use these conditional constructs, methods would be added to allow parameter passing to the cells, so that these parameters could participate in the evaluation of the conditionals. It would also require the use of expression evaluators, so that the parameters could be operated upon as the conditionals were evaluated. Looping constructs would be very handy for generating arrays and vectors of cells.

By the time these features were added to a graphics system, the system would no longer be a high level graphics system but a low level programming language. Rather than add these complexities to a simple graphics system, we might add graphics primitives to existing programming languages. Using this new approach, cells would easily be designed which can be parametrized and which actually generate a whole family of cell instances depending upon the parameters passed into the cell.

Another advantage of designing classes of cells has to do with the binding of design decisions. In standard graphics designs, virtually all of the design parameters must

be bound before the cells are designed. Using the software programming approach, the exact parameter values are not needed, but rather a range of acceptable values is required. The cells can then be designed to produce correct layouts over these ranges. When the actual parameter values are known, these cell programs are called with the appropriate values and the layout is generated. The design can proceed before the details are completely known.

A third advantage of designing families of cells has to do with the granularity or size of cells. With graphics approaches, cells usually contain 10 to 100 primitive components, or transistors. This limitation is brought about by limitations on CRT terminals and on the ability of the human mind to design large circuits. These small cells are assembled to form the chips. Rarely are configurations of these small cells stored as a large cell in the library because of the fact that the large cells are exact physical elements which cannot be changed. Inefficiencies in a particular instantiation are usually not tolerated, so the large cell is redesigned in each context with the minor variations that each context requires. With the software approach, these large cells can be parametrized to vary the arrangements of smaller cells and remove the inefficiencies in the layout. Thus large cells can be designed and saved in libraries and still yield efficient layouts.

These software languages are referred to as Imbedded Languages. The construction for generating graphics primitives are imbedded in a previously existing programming language. There are two classes of imbedded languages: translation based languages and data structure based languages. The translator imbedded languages output the graphic primitives as they are encountered during the execution of the program. Data structure imbedded languages build up a data structure representing the entire chip as the program is executed. Once this data structure is built, the graphic primitives are output. The latter approach allows programs to modify the design after it is generated, while the former approach forbids such modification. Imbedded Languages exist in several languages at Caltech: ICLIC, written in ICL [4]; LAP, written in Simula [19]; Clap, written in C; others, written in Pascal, Fortran [31], and Basic. The language presented here is ICLIC, which is an example of a data structure imbedded language.

3.1: ICLIC

ICLIC is a series of functions and datatypes defined within ICL to allow the user to describe integrated circuits. ICLIC was written in ICL by Ron Ayres and Maureen Stone. Integrated circuit descriptions are ultimately geometrical regions, so the primitive constructs in ICLIC are representations of simple geometrical shapes. The most primitive shape is the BOX, which we will define to be all points on the plane whose x-coordinates are between the lower and upper x limits of the box (inclusive) and whose y-coordinates are between the lower and upper y limits of the box. The following ICL code defines the BOX datatype and a function TO which aids the user in generating a box:

```
TYPE     BOX= [LOW,HIGH:POINT];
DEFINE TO(A,B:POINT)=BOX:    [LOW: A MIN B    HIGH: A MAX B]    ENDDEFN
```

POINT is a pre-defined ICL datatype which has two real values labeled X and Y. The MIN and MAX functions are defined for POINTs to work coordinate-wise: the MIN of two points has an X value which is the minimum of the two point's X values and a Y value which is the minimum of the two point's Y values. A user may generate a box in one of the following two manners:

```
VAR B1,B2=BOX;
B1:= TO(3#4,10#12) ;
B2:= 3#12 \TO 10#4;
```

The two boxes are identical because the point values are sorted. A second primitive geometrical region is a polygon. A polygon is defined to be the set of all points in the plane which lie 'inside' the line segments which comprise the edges of the polygon. We can represent the polygon in the computer's memory as a list of points which are the verticies of the edge segments. The following code declares the type POLYGON:

```
TYPE     POLYGON= { POINT };
```

Here we have declared that a polygon is an arbitrary list of points. To generate a triangle, the following constructions can be used:

```
VAR P1,P2=POLYGON;
P1:= { 3#2 ; 10#4 ; 8#12 };
P2:= { 3#2 ; 10#.+2 ; .-2#.+8 };
```

The second example makes use of the relative-point feature in ICL. The final primitive geometric region used in ICLIC is a WIRE: Formally speaking a WIRE is the set of all points which lie within a fixed distance from any point on a given series of line segments. The collection of line segments is called the 'path' of the wire and the discrimination distance is called the 'radius' of the wire. This formal definition of a wire requires that circular arcs be present in the boundary of the wire. ICLIC approximates a formal WIRE by 'squaring off' all of the round edges. A WIRE can be defined in ICL by stating:

```
TYPE  WIRE= [WIDTH:REAL  PATH: { POINT } ];
```

An example of a wire might be

```
VAR W=WIRE;
W:=[WIDTH:2  PATH: {3#3;7#.;.#5;10#8;14#.} ];
```

Figure 3-1 illustrates each of the three primitive regions introduced to this point.

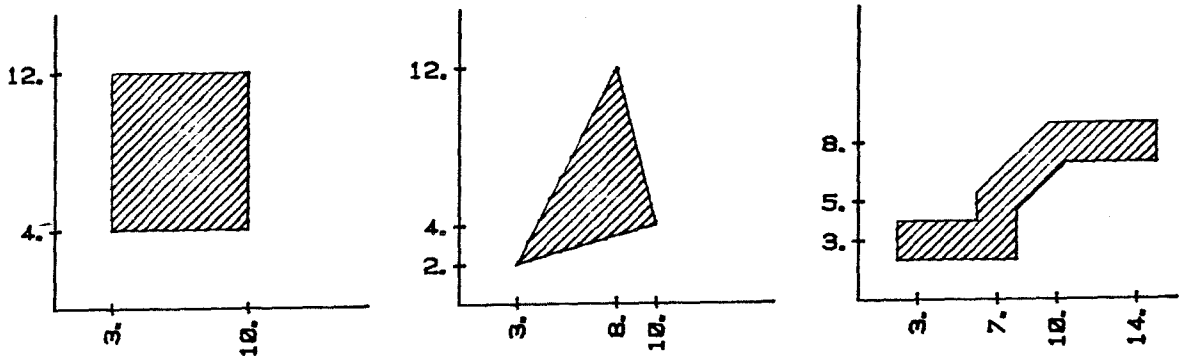


Fig. 3-1: ICLIC Primitives

We now have representations for the primitive features of our imbedded language. We would like to be able to talk about features in general, not just BOX-features, POLYGON-features, and WIRE-features. To do this, we need a new datatype which

can be either a BOX, POLYGON, or WIRE. A datatype of this form is called a 'variant' in ICL. We declare a datatype RG (for ReGion) which can be any one of the three primitives:

```
TYPE    RG= EITHER
              BOX=     BOX
              POLY=    POLYGON
              WIRE=    WIRE
               .
               .
               .
              ENDOR;
```

If we have a variable of type RG, we can assign to it a BOX, POLYGON, or WIRE. Unfortunately, we can only describe single features with this datatype. Usually IC masks contain more than just one geometric primitive. We can extend the definition of an RG to contain the possibility of many primitives by adding the following line to the definition of an RG:

```
               .
               .
              UNION=   { RG }
               .
               .
```

This now states that an RG may also be an arbitrary list of RGs. In addition to many features in an IC design, there are also many 'layers' to an IC specification. To discriminate between layers, the features have a color associated with the region. We can incorporate this possibility in our definition of RG by adding this line:

```
               .
               .
              COLOR=   [PAINT:RG  WITH:SCALAR(RED,BLUE,GREEN,BLACK,YELLOW)]
               .
               .
```

Finally, we may wish to reposition previously declared regions. In almost every instance, we are describing features relative to a local origin rather than in absolute chip coordinates. Once these sub-pieces are generated, we would like to reposition them into the absolute chip coordinates (or to a higher level local coordinate system). The primary repositioning operations we would like to perform are translation, rotation, and mirroring. These operations can all be represented as a transformation matrix which should be applied to all coordinates of the region to be displaced. By adding the matrix displacement case to the RG definition, we can

arbitrarily reposition, mirror, rotate, and scale subcells. The following case is added to the RG definition:

```
        .
        .
DISPLACE= [DISPLACE:RG  BY:MATRIX]
        .
        .
```

and the MATRIX datatype is declared with:

```
TYPE    MATRIX= [A, B, C,
                 D, E, F: REAL];
```

We have now completely described the RG datatype definition. With this datatype, we can represent the features of integrated circuit masks.

The datatype definitions presented above are an approximation of the actual ICLIC datatype definitions. Appendix 1 lists a more complete description of the datatype and functions defined for both generating and examining layouts. The primary difference between the definitions presented here and those in the appendix have to do with capturing the minimum bounding box (MBB) of the layouts. The MBB of a layout is a very useful quantity, and for efficiency, the layout datatype in ICLIC is MRG, which stands for Minimum bounding box with ReGion.

3.2: Parametrized Cells

To illustrate the design of parametrized cells, let us consider the task of designing a shift register cell. The shift register circuit we will implement is shown in figure 3-2. This circuit consists of a pair of inverters with a transmission gate connecting them and a transmission gate connecting the input to the first inverter. We can design the layout of the shift register as shown in figure 3-3. To design this layout, we have computed the expected power requirements and aspect ratio of the cell.

As we use our shift register cell in various places in several chips, we may find that the power requirements in some cases differ from the power requirements of our original cell, so we must design a new cell for these new uses. In other instances, we may find that we need to fit the cell into a different aspect ratio. Again, we

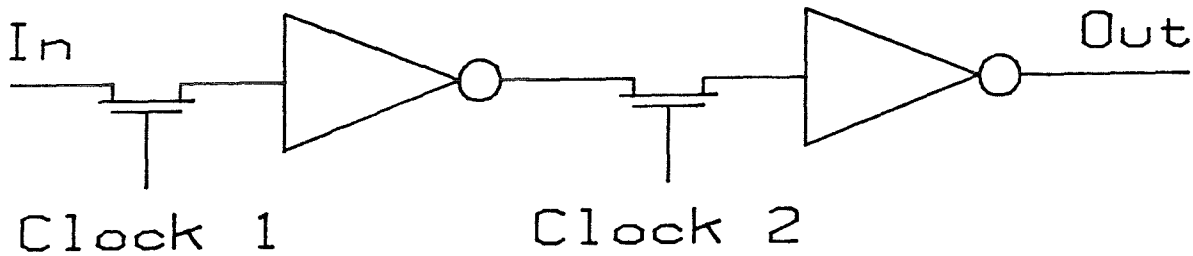


Fig. 3-2: Shift Register Circuit

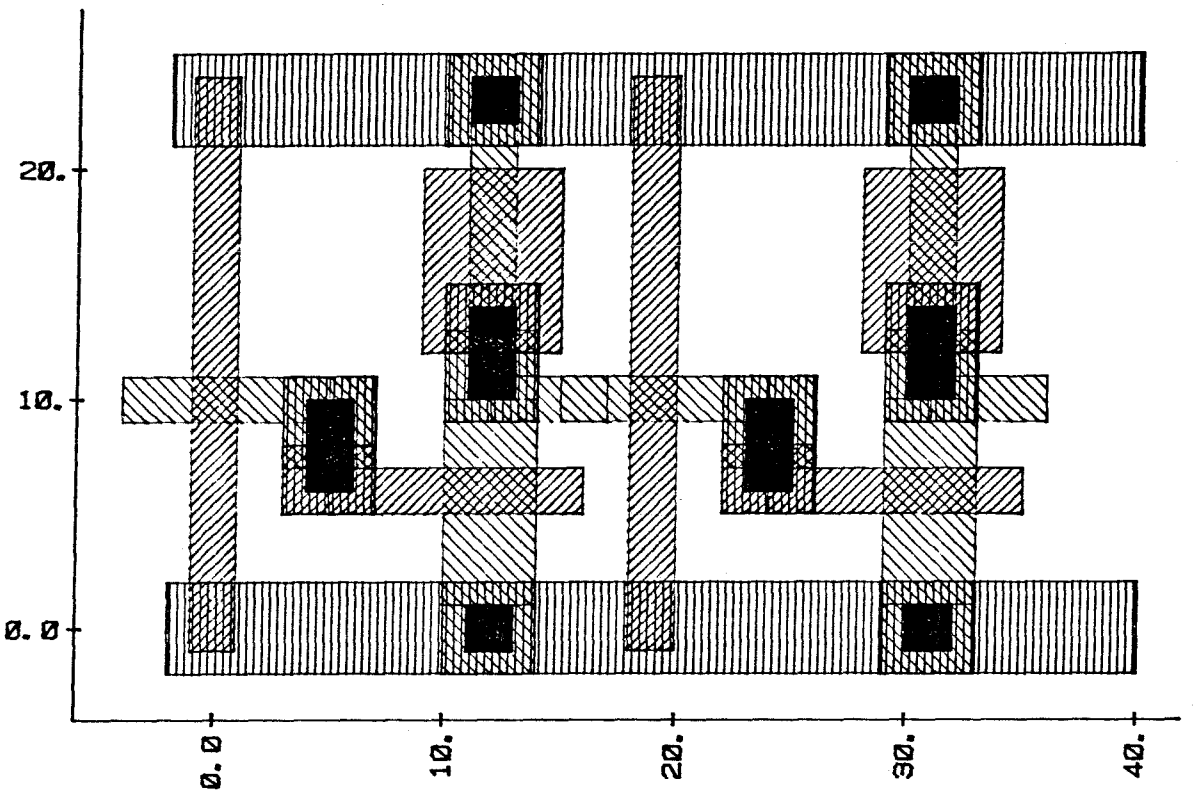


Fig. 3-3: Shift Register Layout

must redesign our cell to fit these new requirements.

Each time we must redesign the cells, we increase the chances of errors in the design. We also proliferate cell instances in our database, and we must expend the effort to document the new cell.

In our shift register example, it is very easy to mathematically describe our cell layout as a function the power requirements. In figure 3-4 we show the layout where some of the coordinates are not fixed, but rather are functions of the power

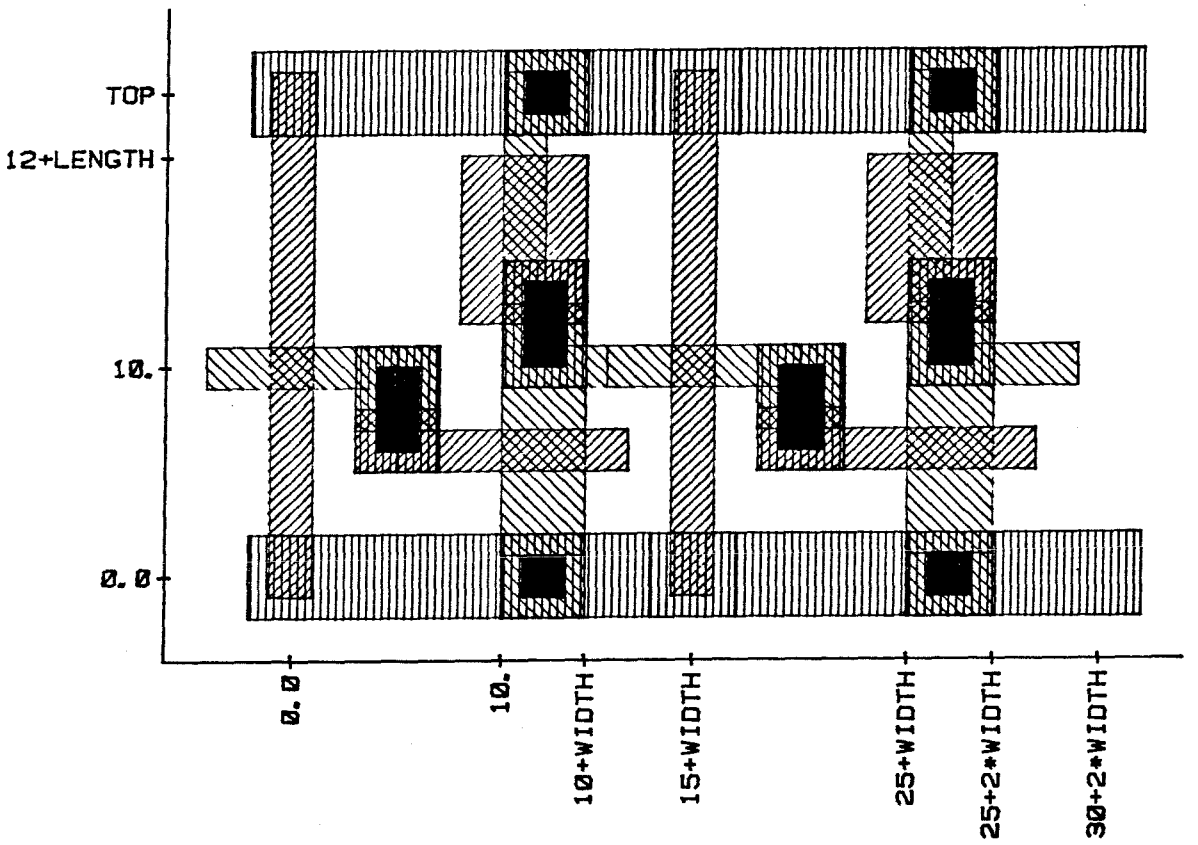


Fig. 3-4: Parametrized Layout

requirements of the cell. Given this description of the shift register layout, we can now generate a new cell every time we compute a new power requirement. In fact, we can write a little program which will generate this new cell for us.

```
DEFINE SHIFT_REGISTER_CELL(POWER:REAL)=MRG:
    BEGIN   VAR WIDTH,LENGTH,TOP=REAL;
    DO  LENGTH:= 2/POWER MAX 4;
        WIDTH:= 32/LENGTH MAX 2;
        TOP:=LENGTH+15 MAX 20;
    GIVE   ( WIRE(RED, (0#0;.#TOP));
             WIRE(GREEN, (-3.#10;4#.));
             WIRE(RED, (6#6;11+WIDTH#.));
             WIRE(GREEN, (11#0;.#TOP));
             WIRE(GREEN, (11#10;13+WIDTH#.));
             BOX(GREEN,10#3\TO 10+WIDTH#9);
             BOX(RED,8#12\TO 14#12+LENGTH);
             BOX(YELLOW,8#10\TO 14#14+LENGTH);
             WIRE(BLUE, (0#0;15+WIDTH#.));
             WIRE(BLUE, (0#TOP;15+WIDTH#.));
             GCB\AT (12#0;.#TOP);
             GRCBD\AT 5#9;
             GRCBU\AT 12#11 )\AT (0#0;15+WIDTH#0)
    END
ENDDEFN
```

Figure 3-5 shows the results of calling this program with power requirements of 1/2 and 1/8. Rather than call these two specific sets of geometrical primitives cells, why not call the program the cell? And we can call these layouts instances of the cell. Each instance of the layout may have a completely different set of geometrical primitives, yet they are all instances of the one cell.

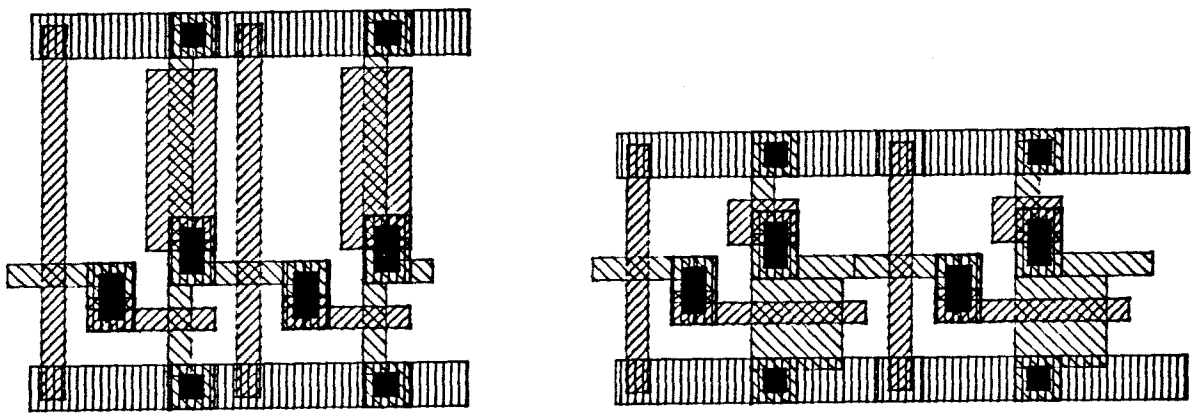


Fig. 3-5: Two Cell Instances

We can design this cell program before we know the actual power requirements of the cell. In our original approach, we had to compute the power requirements before we could begin the layout design. With this programmatical approach, we can estimate reasonable ranges we are willing to accept as power requirements, and design our cell before we know all of the implementation details.

Let's continue to parametrize our shift register cell. We could have designed our cell with many different aspect ratios. Depending upon how and where this shift register is used, the optimal aspect ratio for the cell changes. Figure 3-6 shows 6 different aspect ratios for the shift register cell. The first aspect ratio is approximately square, and takes the least area. In some cases, however, the horizontal space is more costly than the vertical space, so we might wish to use a narrower cell, even though the cell takes more vertical area, as in the second type of layout. The third and forth layouts were designed so that vertical space is at an absolute minimum, while the fifth and sixth layouts use an absolute minimum amount of horizontal space. Each of these layouts are parametrized with respect to the power requirements. We can now write a cell program which is parametrized in terms of both power requirements and aspect ratio. When we call the shift register cell, the program chooses the layout which most closely matches our desired aspect ratio, and generates the corresponding layout.

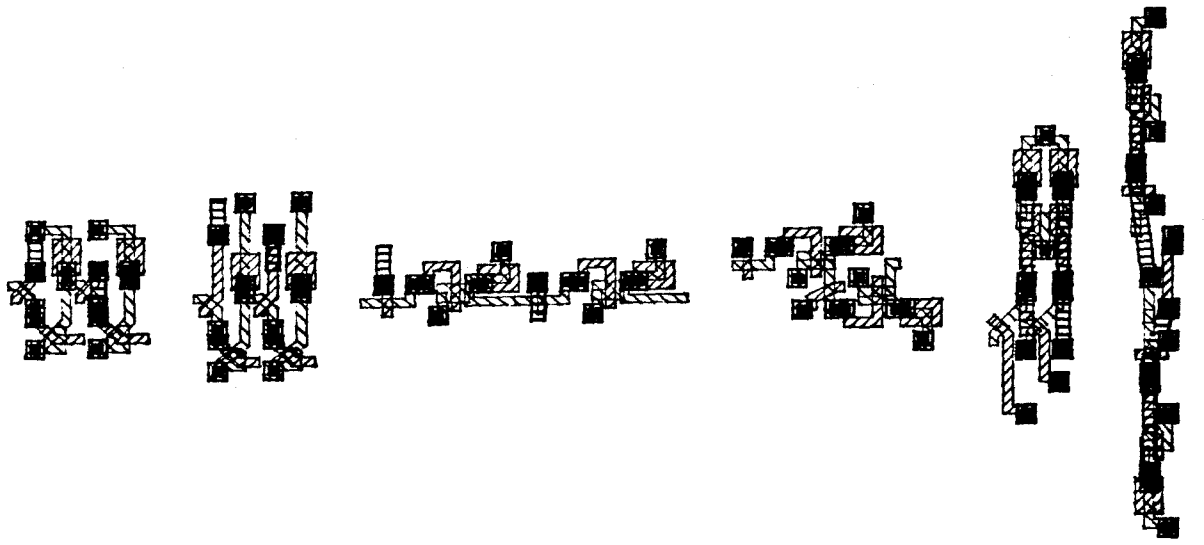


Fig. 3-6: Six Shift Register Layouts

Rarely are single shift register bits used. In most cases, a whole string of bits are required. In standard approaches, one does not think of a shift register row as a single primitive cell, rather a single bit is the primitive cell, and the user must interconnect each of the cells into a row for each shift register row needed. This is done because there is much variability in the requirements of the shift row (such as power, aspect ratio, number of bits). Because of this variability, fixed cells are usually not helpful. Since we are designing programmable cells, we can program

this additional variability into the cell.

For instance, the user might wish to state how many bits are to be in the shift register, so the program could generate the whole row. The user may also wish to have more than one shift register. Some area can be saved by placing two identical shift registers side-by-side. When long shift register chains are used, the chip area is long and thin. By folding the long shift registers, the chip area becomes more square, usually a desirable option. So let's design our shift register cell to take these parameters: number of bits wide, number of bits long, power per bit, and desired area. Our cell will take the power and compute cell sizes for the six different aspect ratios. It will then produce a single row and attempt to fold the row to match the desired area. Finally the cell will return a layout of the entire array, using the single bit layout and folding factors that will produce the best layout.

What if none of the possible implementations of the shift array fit within the desired area? Rather than having the program flag an error or abort the design, we may tell the program how to choose the next best area. For instance, we may like to state that the desired area is 500 by 800 lambda, but if nothing fits, the x size is free to grow while the y size must not get larger than 800 lambda. Or, we may say that the area should be 400 by 400, and if nothing fits, the instantiation with the smallest area should be used. To allow these possibilities, we will add one more parameter to our cell program which is a weight factor: if none of the instantiations fit, we will compute an excess cost for all prospective candidates by summing the x oversize times the x-coordinate of the weight and the y oversize times the y-coordinate of the weight. We select the candidate with the lowest excess cost.

The ICL code for our cell is listed in appendix 2. The organization of the code is as follows. We have routines for generating single bits of the shift register, named SHIFT1_CELL through SHIFT6_CELL. These routines are parametrized in terms of pullup transistor size, pulldown transistor size, and power line widths (PU=pullup length, PD=pulldown width, SP=width of single row power line, DP=width of double row power line, and HP=width of half-row power line). These return the layout for a single bit of the shift array. Next we have routines which generate rows of these single bits (SHIFT1_ROW through SHIFT6_ROW) plus a routine which turns these rows into a complete array (FINISH). These routines are also

parametrized in terms of total power, number of bits, and folding factor (TP=width of total power line, NR=number of bits per row, RB=number of rows in each shift register, NB=number of shift registers, and NL=number of bits in the last row of each register. TB=total number of bits in each shift register=NR*(RB-1)+NL). The functions SHIFT1_ARRAY through SHIFT6_ARRAY simply generate the entire array.

Given these shift array functions, we would like a routine which determines the area of each possible shift array. The SIZE function will return the area of a candidate and a routine which, when executed, will generate that candidate. We don't want to generate the actual layouts of every candidate to select the best layout because this would take a lot of space in the computer's memory plus it would take a long time. Instead, SIZE computes what the size would be, and generates a function reference which we may execute if the candidate is selected as best. The function SHIFT_CELL, which is the function a user calls, checks many candidates and selects the one best fitting the user's description. The best candidate is determined by the following algorithm:

> If there are candidates whose x and y values are less than the desired size, the one whose x and y values are closest (sum of squares) is chosen.
>
> If no candidates fit, a weight is determined for each candidate, and the candidate with the smallest weight is used. The weight is determined as follows:
>
> > If the x value is less than the desired x value, use 0 otherwise use the difference between the actual and desired x values.
> >
> > Multiply this number by the x weight and square the result.
> >
> > Similarly, compute the y weight.
> >
> > The total weight is the sum of the x and y weights.

The remaining functions in the listing (GRAPH and TABLE) produce a graph and tabular listing of the candidate sizes. These are useful if a designer wishes to see all of the candidate sizes for any particular size of shift array.

This parametrized cell is used as follows. The designer determines the number of bits in the array. For our example, we require 4 shift registers of 100 bits each. We would like these to be fairly low power, so our power requirements will be 1/8. Due to chip area constraints, we would like the array to be approximately 500

by 800 lambda. We can get a tabular listing of possible candidates by entering ICL and typing the command

```
TABLE(4,100,.125,23,1000#1600);
```

The first parameter is the number of shift registers (4), the second is the number of bits per register (100), the third is the power requirement (.125), the fourth states what the maximum number of folds in the shift register should be (23), and the last parameter states the maximum size we want listed in the table. Concerning this last parameter, the program will generate all possible candidates meeting the other parameters, but will only list those candidates whose x dimensions are less than the given x limit (1000) and whose y dimensions are less than the y limit (1600). ICL will print the following table:

```
CLASS:1 ROWS/BIT:3      SIZE:982.#405.
CLASS:1 ROWS/BIT:5      SIZE:590.#673.        <---<<<
CLASS:1 ROWS/BIT:7      SIZE:450.#941.        <---<<<
CLASS:1 ROWS/BIT:9      SIZE:366.#1209.
CLASS:1 ROWS/BIT:11     SIZE:310.#1477.
CLASS:2 ROWS/BIT:3      SIZE:845.#531.
CLASS:2 ROWS/BIT:5      SIZE:509.#883.        <---<<<
CLASS:2 ROWS/BIT:7      SIZE:389.#1235.
CLASS:2 ROWS/BIT:9      SIZE:317.#1587.
CLASS:3 ROWS/BIT:11     SIZE:872.#575.
CLASS:3 ROWS/BIT:13     SIZE:704.#679.
CLASS:3 ROWS/BIT:15     SIZE:620.#783.
CLASS:3 ROWS/BIT:17     SIZE:536.#887.        <---<<<
CLASS:3 ROWS/BIT:19     SIZE:536.#991.
CLASS:3 ROWS/BIT:21     SIZE:452.#1095.
CLASS:3 ROWS/BIT:23     SIZE:452.#1199.
CLASS:4 ROWS/BIT:5      SIZE:875.#523.
CLASS:4 ROWS/BIT:7      SIZE:665.#731.
CLASS:4 ROWS/BIT:9      SIZE:539.#939.
CLASS:4 ROWS/BIT:11     SIZE:455.#1147.
CLASS:4 ROWS/BIT:13     SIZE:371.#1355.
CLASS:4 ROWS/BIT:15     SIZE:329.#1563.
CLASS:5 ROWS/BIT:3      SIZE:566.#807.5       <---<<<
CLASS:5 ROWS/BIT:5      SIZE:342.#1343.5
CLASS:6 ROWS/BIT:1      SIZE:838.5#499.
CLASS:6 ROWS/BIT:3      SIZE:310.5#1491.
```

None of the candidates fit into the area we requested, but there are five entries in the table which are the approximate size we require. We could also make a plot showing these candidate sizes by using the GRAPH function, which takes the same parameters as the TABLE function:

```
PLOT(GRAPH(4,100,.125,23,1000#1600),HP_7221A);
```

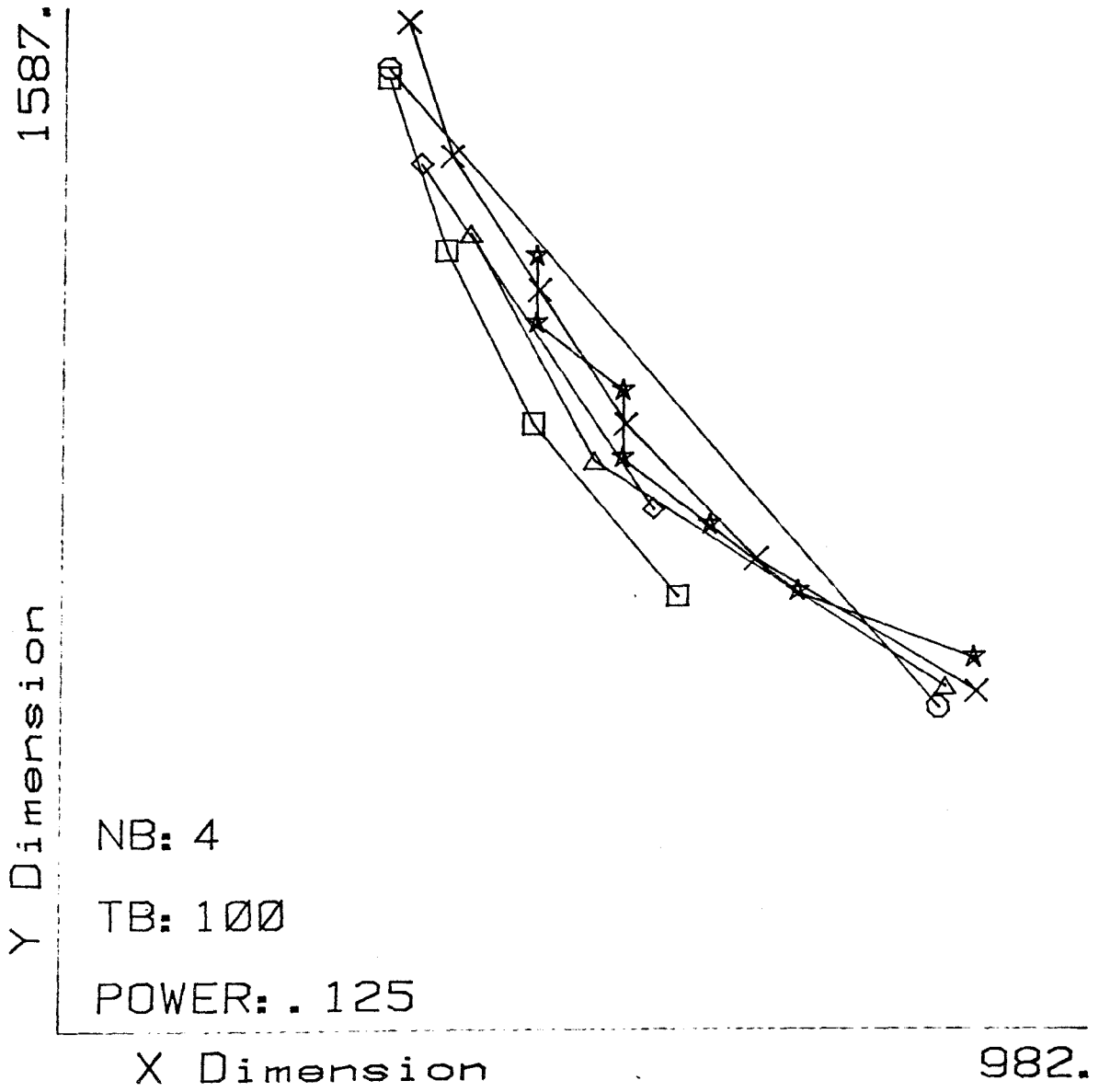


Fig. 3-7: Graph of Size Possibilities

This graph is shown in figure 3-7. To actually create the layout, we would call the SHIFT_CELL function. The following code generates five separate shift register arrays differing in the area costs. The desired area for all arrays is 500#800. The first array requires that the y dimension is fixed while the x dimension is free to vary. The second array fixes the x dimension and allows the y dimension to grow. The third array allows x and y to vary equally. The fourth array has the x dimension costing a bit more than the y dimension, but both are free to vary. The final array uses more power, but fits within the 500#800 space requirement. Figure 3-8 shows the metal2 layer for each of these arrays.

```
VAR ARRAY1,ARRAY2,ARRAY3,ARRAY4,ARRAY5=MRG;
ARRAY1:=SHIFT_CELL(4,100,.125,500#800,1#99999);
ARRAY2:=SHIFT_CELL(4,100,.125,500#800,99999#1);
ARRAY3:=SHIFT_CELL(4,100,.125,500#800,1#1);
ARRAY4:=SHIFT_CELL(4,100,.125,500#800,1.5#1);
ARRAY5:=SHIFT_CELL(4,100,.25,500#800,1#1);
```

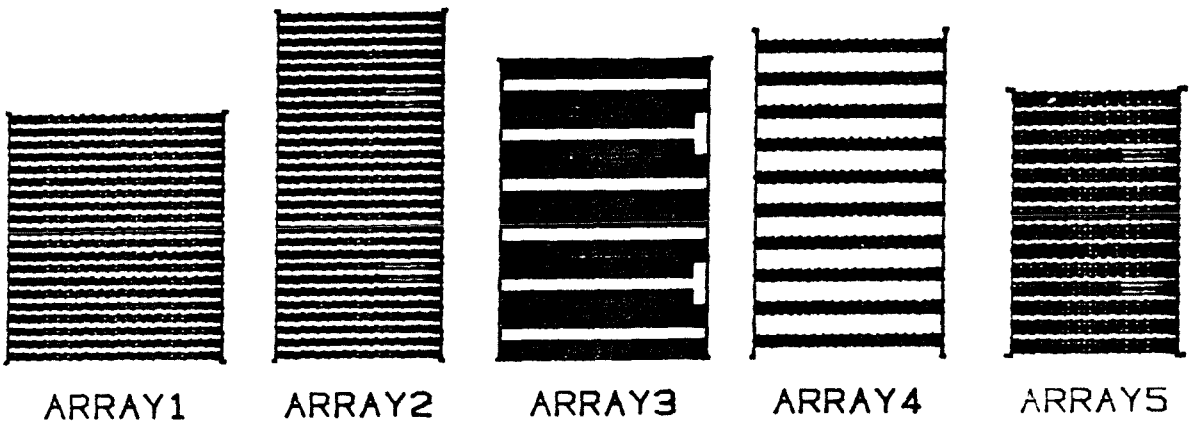


Fig. 3-8: Five Shift Array Candidates

3.3: Conclusions

We have seen the description of an imbedded language system and how this system can be used to construct integrated circuit layouts. We have also seen the benefits of using imbedded languages to design chips.

One of the advantages of imbedded language systems is that they allow the user to design a whole family of cell layouts at one time. Based upon parameters given to the cell program, the program will compute and generate the correct layout for the particular usage of the cell. This emphasizes the similarities between members of the cell family. This also reduces the number of cells in the cell libraries: The cell program is saved rather than the many cell instances.

The cell parameters typically refer to behavioral information, not geometrical information. Our shift register is parametrized in terms of number of bits and

power requirements, not inter-cell spacings and transistor sizes. This allows for partitioning of the design. The user of the cell thinks in terms of parameters interesting to him, and he does not have to know the details of the cell implementation.

Along these same lines, parametrized cells delay the binding of design decisions. The shift register program was implemented before the power or area requirements were known. Also, since this cell is now flexible, the entire chip layout can be designed before the power requirements are known. When the requirements change, a few simple parameter changes will completely correct the layout.

The task of making design decisions is also aided with parametrized cells and chips. When the cells are parametrized in the manner presented here, the user can alter the design parameters and actually see the effects these decisions make upon the design. The designer does not have to guess, the actual results can be seen.

Parametrized cells tend to encompass much larger functions than fixed cells. Parametrized cells are usually a complete function, whereas fixed cells tend to be rather small pieces of layouts which must be combined to construct a function. Since fixed cells can not reconfigure themselves depending upon how they are used, large fixed cells are not frequent since it is rare that a large function will be used identically in many places. Parametrized cells can reconfigure themselves, so similar uses of a function can efficiently use the same cell.

With imbedded languages, we are not designing chips as purely graphical data. We have the freedom to add additional information to our cells, information which can further aid the design process. In the next chapter, we explore some of these possibilities.

Chapter 4: Chip Assemblers

In the previous chapters, we have reviewed methods for generating leaf cells, which is only the first step to designing a chip. To complete the design of a chip, we need to generate the composition cells which interconnect the leaf cells. The task of interconnecting leaf cells is much harder than the generation of the leaf cells. The leaf cells are typically small, self-contained units which can be completely defined. Composition cells, on the other hand, deal with global information, and are fairly large, complex assemblies at the higher levels of the chip hierarchy. In this chapter we will explore some of the tools which can aid in the interconnection of the leaf cells [29].

4.1: Cell Composition

There are three phases of generating composition cells. The first phase deals with the specification of the interconnection between the cells: how should the cells be wired together? The second phase deals with the generation of the geometrical primitives required to interconnect the cells. The final phase deals with verification: was the interconnection specification correct, or did we just short VDD and GROUND?

Each interconnection methodology presents unique constraints upon these three phases of cell composition. In some interconnection strategies, the interconnection specification is implied by the cells themselves, freeing the user from the task of writing an interconnection list. Other techniques do not require wires to perform the interconnection, so the generation phase may be trivial.

Every interconnection methodology, however, should have a checking phase. Most of the errors in chip design have to do with erroneous interconnection of modules, virtually all of which would be caught by the checking phase of the interconnection. By TYPEing the connections to a cell, one can later verify that the connector was connected to the proper signal. For instance, one would not like to connect two outputs together. By adding this information to the layout representation, it can easily be verified that outputs do not connect to other outputs.

For the composition systems presented in this chapter, we will assume that the chip floorplan is a slicing type floorplan, as presented in Chapter 2.

4.2: Power Routing

Power signals are special signals in integrated circuits. They can not be routed as ordinary data signals, due to the finite resistance and current limits of the wires. Therefore, a strategy should be developed to deal specifically with power wires.

The first requirement that one might state about power lines is that they should always run in metal from very close to the transistor terminals to the edge of the chip. With two polarities of power lines in NMOS design, this means that some planning must be done before the cell design is begun. Without this planning stage, the power requirements may be impossible to satisfy.

We can analyse the structure of NMOS design to develop a general model of power routing [14]. In specific cases, special purpose power routing schemes are used, but in the general case, the following power routing scheme has been shown to produce close to optimal designs. We define a cell to have not only a rectangular outline, but also to have a VDD terminal in the North-East corner of the cell and a Ground terminal in the South-West corner of the cell. The cell must also contain power consumption information, so that the power lines can be made of the appropriate width.

We will place the following conventions upon the definition of the VDD and Ground points. To properly connect power to the cell, we need to touch the VDD point with a metal VDD box and to touch the Ground point with a metal Ground box. We are free to run Ground lines anywhere along the bottom edge of the cell, up to the Ground point, or we may run metal Ground lines anywhere along the left edge of the cell, up to the Ground point. Similar statements can be made about running VDD lines. Figure 4-1 illustrates these conventions. The first example has the power lines running horizontally while the second example routes the lines vertically.

We may define a datatype CELL which encapsulates the information needed for handling this style of power routing.

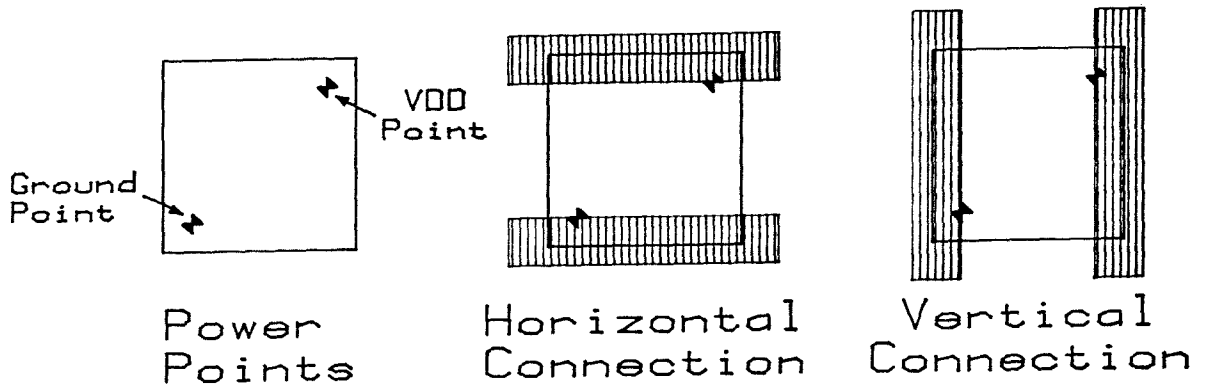


Fig. 4-1: Power Line Conventions

```
TYPE    CELL=
            [NAME:      QS
             LAYOUT:    MRG
             VDD,GND:   POINT
             POWER:     REAL];
```

The CELL has a name, layout, the two power points, and a power consumption variable. We will represent the power consumption by a REAL number which indicates the effective conductance (reciprocal of resistance) of the internal circuitry. A pre-defined procedure WIDTH converts this conductance into the minimum wire width needed to supply the required power.

We have stated that we will use the slicing floorplan for our chips. As shown in Chapter 2, this means that all possible chip floorplans can be implemented as a hierarchy of binary cell fusions. If we write routines which will properly interconnect two cells in any legal configuration, we will be able to route the power for any slicing chip whose cells use the two-point power convention. We may recall that there are precisely two legal configurations of two cells in the slicing floorplan: one cell may be to the right of the other, or one cell may be above the other. We will call these two orientations HORIZONTAL and VERTICAL, respectively.

Let us consider the horizontal case. Given two cells that have already been given appropriate relative positions, how do we connect the power lines? Figure 4-2 gives an example of how this might be done. In the figure, we route boxes from the power points to a larger power box which is a suitable distance from the two cells. The widths of the two vertical power boxes connected to the left cell are W1, which is equal to WIDTH(left.POWER). Similarly, the right cell's power box

widths are W2, which is WIDTH(right.POWER). The widths of the large power boxes are W3, which is WIDTH(left.POWER+right.POWER). Thus, the vertical boxes are wide enough to supply power to one of the cells, while the horizontal boxes are wide enough to supply power to both cells.

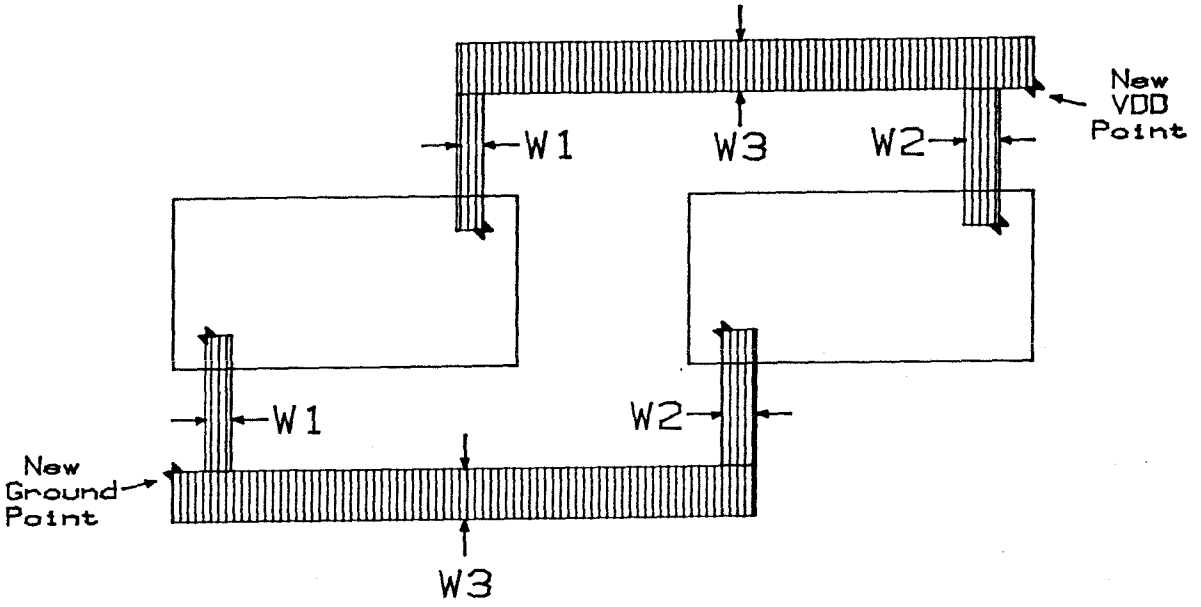


Fig. 4-2: Horizontal Power Connections

It may be noticed that the layout of the power boxes shown in figure 4-2 is fairly inefficient. We will now produce more efficient routings. Consider the vertical VDD box of either cell. We have its lower right corner touching the power point of the cell. If we had the lower left corner touch the power point, the power box may extend past the cell's bounding box if the power requirement is large, as shown in figure 4-3a. If we always lined the right edge of the power box with the cells bounding box, the box would never extend past the cell's bounding box, but the power box may not touch the power point, as shown in figure 4-3b. To efficiently align the power box, we need to examine the power box width and power point location. If the power box width is less than the distance from the power point to the cells bounding box, we will align the box to the power point (fig. 4-3c). If the power box width is greater than this distance, we align the power box to the cell's bounding box (fig. 4-3d).

Next, let us consider the position of the horizontal power box. In figure 4-2, it was placed a considerable distance from either cell, so as not to interfere with the

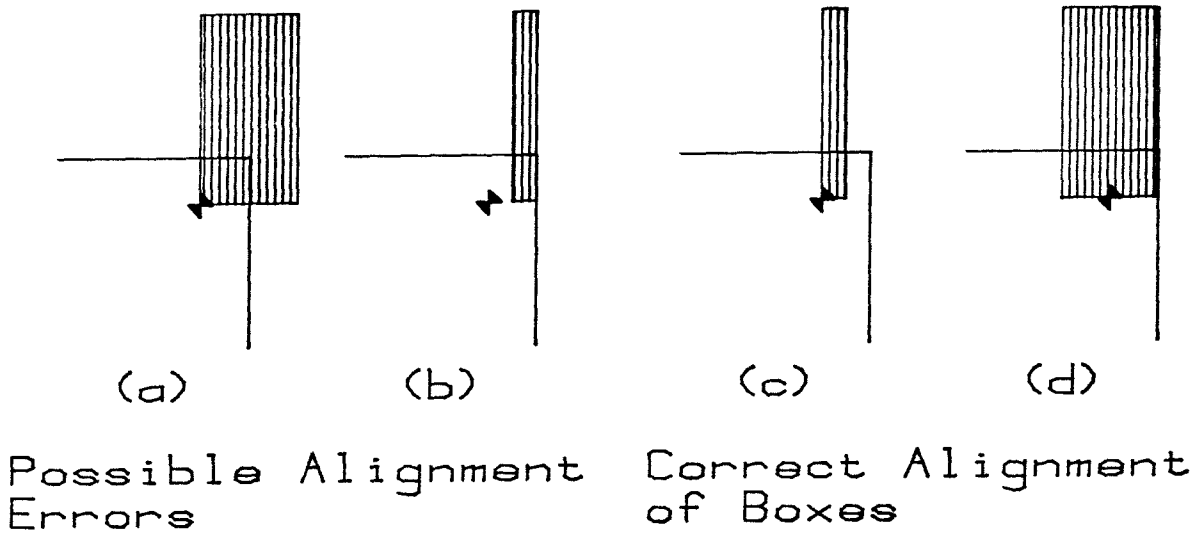


Fig. 4-3: Alignment of Vertical Boxes

geometry within the cells. On the other hand, our power routing convention states that we may run any VDD boxes we wish above the cell, as long as the box stays above the VDD point. Hence, what we might do is lower the VDD box until it just rests upon either of the two VDD points, which ever is higher. Figure 4-4 shows the only possible situations. If the left VDD point is above the right VDD point, the horizontal box rests upon the left's VDD point. Similarly, if the right's point is higher, the box rests upon the right's VDD point. If both points have the same height, the box rests on both. Notice in the first case that the left's vertical power box is not required, since the horizontal box completely overlaps the area where the vertical box would be. The second case does not require the right power box, and the third case requires neither.

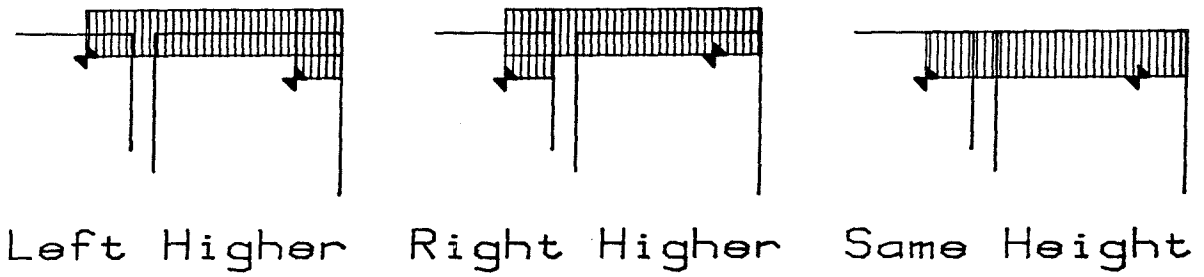


Fig. 4-4: Positioning Horizontal Box

To complete the routing of the VDD lines, we need to determine where the VDD point for the composition cell should be. The definition of the power point is that we may route any VDD boxes to the right or above the specified point. The x

component of the point can be determined solely from the right cell. The right cell's VDD point stated where we could run boxes over the right cell. This same x coordinate can be used for the composition cell. For the y coordinate, we need to examine both the left and the right cells, but again, they have given us acceptable values for running horizontal wires. We need only satisfy both cells' requirements. This is done by using the larger of the two cells' VDD point's Y values. Since the left cell's VDD point x is always less than the right cell's VDD point x, we can state that the new VDD point is simply the maximum of the left and right cell's VDD points.

The analysis of the VDD boxes can be used to analize the Ground boxes, with appropriate sign changes. We can now code the routine for horizontally fusing two cells.

```
DEFINE HORIZONTAL_POWER_BOXES(L,R:CELL  NAME:QS)=CELL:
    BEGIN   VAR W1,W2,W3,X1,X2,X3,X4,VDDY,GNDY=REAL;
    DO  W1:=WIDTH(L.POWER);
        W2:=WIDTH(R.POWER);
        W3:=WIDTH(L.POWER+R.POWER);
        X1:=MBB(L.LAYOUT).HIGH.X;
        X1:= IF X1-W1<L.VDD.X THEN X1-W1 ELSE L.VDD.X FI;
        X2:=MBB(R.LAYOUT).HIGH.X;
        X2:= IF X2-W2<R.VDD.X THEN X2-W2 ELSE R.VDD.X FI;
        X3:=MBB(L.LAYOUT).LOW.X;
        X3:= IF X3+W1>L.GND.X THEN X3 ELSE L.GND.X-W1 FI;
        X4:=MBB(R.LAYOUT).LOW.X;
        X4:= IF X4+W2<R.GND.X THEN X4 ELSE R.GND.X-W2 FI;
        VDDY:= L.VDD.Y MAX R.VDD.Y;
        GNDY:= L.GND.Y MIN R.GND.Y;
    GIVE    [NAME:NAME
             LAYOUT: [L.LAYOUT;
                      R.LAYOUT;
                      BOX(BLUE,X1#VDDY\TO X2+W2#VDDY+W3);
                      BOX(BLUE,X3#GNDY-W3\TO X4+W2#GNDY);
                      IF VDDY\IS_CLOSE_TO L.VDD.Y THEN
                         IF VDDY\IS_CLOSE_TO R.VDD.Y THEN NIL
                         ELSE BOX(BLUE,X2#R.VDD.Y\TO X2+W2#VDDY+W3) FI
                      ELSE BOX(BLUE,X1#L.VDD.Y\TO X1+W1#VDDY+W3) FI;
                      IF GNDY\IS_CLOSE_TO L.GND.Y THEN
                         IF GNDY\IS_CLOSE_TO R.GND.Y THEN NIL
                         ELSE BOX(BLUE,X4#GNDY-W3\TO X2+W2#R.GND.Y) FI
                      ELSE BOX(BLUE,X1#GNDY-W3\TO X1+W1#L.GND.Y) FI]
             VDD: L.VDD MAX R.VDD
             GND: L.VDD MIN R.VDD
             POWER: L.POWER+R.POWER]
    END
ENDDEFN
```

Figure 4-5 shows the resulting layout. The routine for vertical fusion is similar to the horizontal routine.

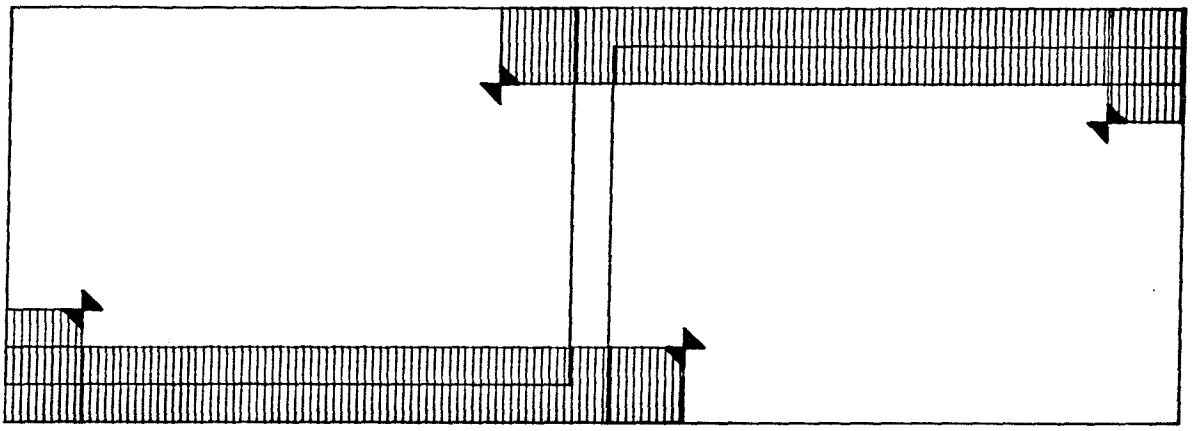


Fig. 4-5: Completed Power Connections

One final observation. We have located the VDD point and Ground point within the cell boundaries. Also, when we produced the composition cell, we kept these points well within the boundaries of the new cell. Why was this done? To conserve area. The higher levels in the chip heirarchy can share this power channel with the route done at this level. If another cell were added to the left of our composition cell, a larger power box would overlap the horizontal power box drawn for this composition cell. Overlapping the boxes does not cause problems, because the larger power box is wide enough to supply power for all three of the cells. Figure 4-6 contrasts the layout produced when the power points are inside the cell to the layout produced when the power points are at the corners of the cells.

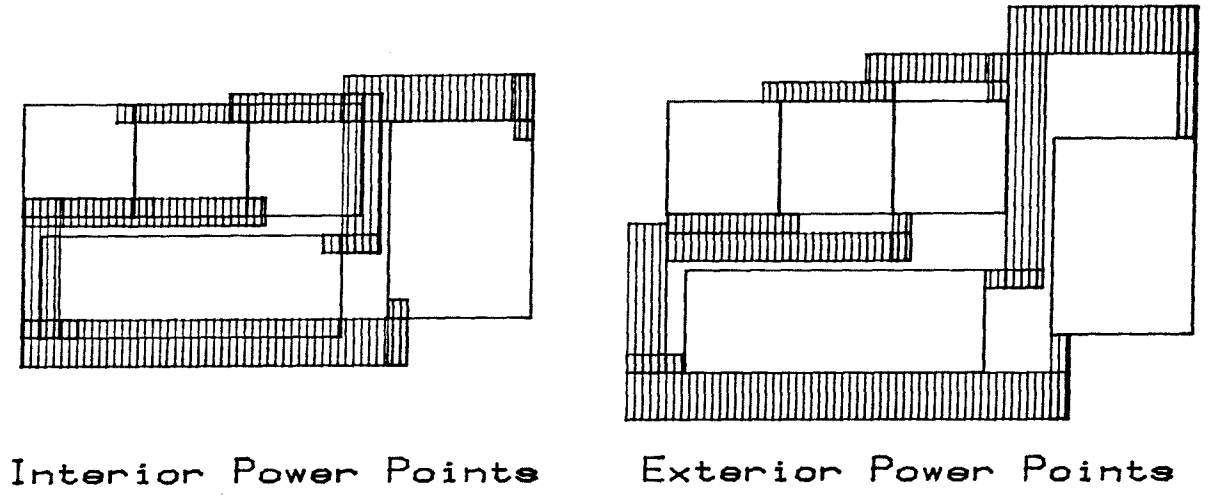


Fig. 4-6: Hierarchically Sharing Boxes

4.3: Composition Methods

We will now look at some of the data line interconnection philosophies, and notice what requirements are made upon the three phases of cell composition.

4.3.1: Cell Abutment

The simplest interconnection philosophy is that of cell abutment. In this style of composition, interconnection between cells is accomplished merely by abuting the two cells [24][27]. It is assumed that the interconnection points of the two cells are in precisely the correct position so that simple abutment properly connects each pair of ports. Figure 4-7 illustrates this concept. Here we wish to join cells A and B, with A 'to the left' of B. Given the bounding box information from the two cells, we can automatically position the two cells to get the interconnection. The following code will generate a fusion of the two cells.

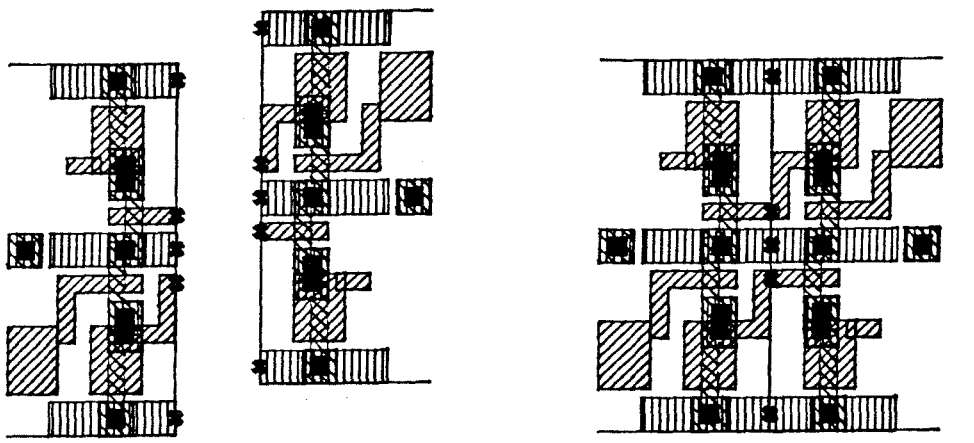


Fig. 4-7: Cell Abutment

```
DEFINE ABUTT_HORIZONTAL(A,B:CELL  NAME:NAME)=CELL:
        DO      B::=\AT A.LAYOUT\MBB\LR - B.LAYOUT\MBB\LL;
        GIVE    HORIZONTAL_POWER_BOXES(A,B,NAME)
ENDDEFN

DEFINE ABUTT_VERTICAL(A,B:CELL  NAME:NAME)=CELL:
        DO      B::=\AT A.LAYOUT\MBB\UL-B.LAYOUT\MBB\LL;
        GIVE    VERTICAL_POWER_BOXES(A,B,NAME)
```

```
ENDDEFN

DEFINE AT(C:CELL  P:POINT)=CELL:
        DO      C.LAYOUT::=\AT P;
                C.VDD::=+P;
                C.GND::=+P;
        GIVE    C
ENDDEFN

DEFINE LL(B:BOX)=POINT: B.LOW MIN B.HIGH    ENDDEFN

DEFINE UR(B:BOX)=POINT: B.LOW MAX B.HIGH    ENDDEFN

DEFINE LR(B:BOX)=POINT: UR(B).X # LL(B).Y   ENDDEFN

DEFINE UL(B:BOX)=POINT: LL(B).X # UR(B).Y   ENDDEFN
```

These abutment routines will handle the composition of two cells. Notice that the specification phase is trivial: we only specify which two cells to fuse, and in which order. Similarly, the generation phase is trivial: we need only position one cell relative to the other, then call our power box routines. On the other hand, we have done no verification of the design. We have no idea whether the implied connection locations of the two cells line up. This little piece of checking, if rigorously applied at all levels of the design, will catch most of the design errors.

To add the verification system to the existing cell system would require a large program that would analyse the layout portions of the two cells, extracting the circuit information. The program would then have to verify that the composition of the two circuits is still a valid circuit. This is a very akward way of determining the port configuration of a cell. This is like writing a software program which examines a core dump to see if all subroutine linkages are correct. A more logical approach would be to have the user specify the intended port configuration of the low-level cells. This information is trivial for the user to specify, since he has to generate this information for the cell documentation. Rather than keeping the port information in the cell documentation, we will keep the information with the cell in machine-readable form, and use it to verify the composition of the cells.

What sorts of information would we need in the ports of a cell? Obvious data are location and layer. To aid the user in examining a cell, we may want to add a name to each connector. These names could convey the intent of the signal. We would also like to know if a connector was an input, output, or bidirectional signal. With this information, we can verify that inputs connect to outputs, and that bidirectional signal connect to bidirectional signals. These three types of signals are

not inclusive, but they will suffice to illustrate the point. In addition to the direction of the signal, we would also like to know when the signal is valid. Even if we have connected an output to an input, if the output is only valid when the clock is high and the input only samples when the clock is low, we have a design error. We will add timing information to the connectors. Using a simplified two-phase clock model, we can have signals valid during PHI-1, PHI-2, or always valid. Finally, we would like to know if we have connected an incredibly large load on a frail driver. For the purposes of this discussion, we will model the load and drive capabilities of connectors by REAL numbers. When we connect two connectors, we wish that the sum of the drives exceeds the sum of the loads. The following datatypes hold the information presented here.

```
TYPE    CONNECTOR=
            [NAME:              NAME
             LOAD,DRIVE:        REAL
             COLOR:             COLOR
             AT:                POINT
             TYPE:              CONNECTOR_TYPE
             VALID:             VALID ];

        CONNECTOR_TYPE= SCALAR(IN,OUT,IO);

        VALID= SCALAR(PHI1,PHI2,ALWAYS);

        CONNECTORS= { CONNECTORS };
```

If we add a CONNECTORS component to our cell definition, our cell designers can append this connector information directly to the other information about the cell. Due to the implied conventions regarding connectors, we know that all connectors must lie on the perimeter of the cell, and that the connectors can not be on the metal layer (because the power boxes may run in metal).

To complete the connector addition to our data structures, there are a few routines which must be modified. When we move a cell with the AT routine, we must also move the connection points. Secondly, when we abut two cells, we must verify that the connectors line up and have the proper characteristics. Finally, we must extend connectors so that they lie on the perimeter of the new cell. When we add the power boxes, the boxes may extend the bounding box of the cell. If this happens, our connectors will no longer be on the perimeter of the cell. We check, therefore, and if a connector no longer lies on the perimeter, we will move the connector and draw a wire of the appropriate color from the old to new points.

This cell abutment technique is a very layout-efficient interconnection technique. Since the interconnection requires no area, the interconnection is as efficient as possible. On the other hand, this is not a very general technique. The only time when cells abut is when they were designed to abut, which makes for a very rigid system. If any of the cells change, several neighboring cells may also have to be changed. One would use abutment in special cases, when the set of cells is small and well defined.

4.3.2: Cell Stretching

A second composition methodology is very similar to the cell abutment approach. Suppose that we wish to simply abut two cells, but the connectors are not at the same positions. To avoid generating wires to perform the interconnection, we need to convert the original cells into cells which can simply abut, which means we need to arrange the connectors to be in the same positions. This is done by cell stretching. Consider figure 4-8. Here we have two cells whose connectors are in the same order, on the same mask layers, but in different positions. To align the 'A' connectors, we need to increase the distance between the bottom of the right cell and connector 'A'. We can not decrease the distance between the bottom of the left cell and connector 'A' because presumably the left cell was designed to have these distances minimized. Hence, we stretch out the right cell as shown in figure 4-8b. Next, we need to align the 'B' connectors. We stretch out the left cell, as shown in fig. 4-8c. This process continues until all of the connectors have the same positions, at which point we can call the abut routines to connect the cells.

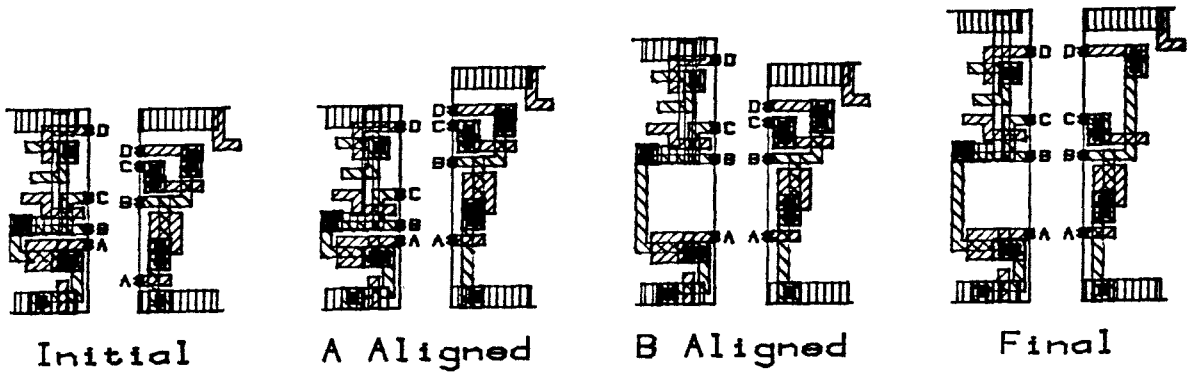


Fig. 4-8: Cell Stretching

This approach has the interconnection program reaching inside the subcells, modifying the layout, to perform the interconnection. External stretching is a very dangerous thing to do: by arbitrarily modifying a cell's layout, the electrical properties of the cell will change, and the cell may cease to function. Rather, one should design the cell to respond to requests to stretch. The system would ask the cell to move a connector, and the cell would be responsible for generating the new layout. In this manner, the cell can monitor changes in the performance of the circuitry, and correct for the cell stretching.

It may seem that this approach is wasteful, because cells are deliberately expanded to take more room, creating a larger chip. In actual fact, smaller chips can result from stretching. The space lost at the low level by stretching may be more than compensated for globally because the wiring cells are not needed. Similarly, stretching may increase the loads on some signal lines, so it would seem that performance would suffer. On the other hand, the routing required between cells degrades the performance of those wires. So stretching the cells may actually increase the performance of the system from a global standpoint, even though local performance has suffered. Finally, by stretching two cells to fit, the resulting layout might be much greater in the stretch direction that either of the two original cells, as the example in figure 4-8 shows. These arguments illustrate the dangers of arbitrarily stretching cells, but there are well-defined cases where stretching does pay off.

4.3.3: River Routing

In the cell-stretching interconnection scheme, we fused cells with connectors in the same order but in different positions. We stretched the cells so that the connectors were in the same positions. Alternatively, we can draw wires to perform the interconnection. Since the two sets of connectors are in the same order, the wires that we draw do not have to cross. A routing between cells where wires do not cross is called a 'River Route'. Figure 4-9 shows a river route between two cells. A very simple algorithm for generating a river route follows. Draw wires from each connector on the left cell over one unit. Then, as long as all connectors are not in the proper position to connect to the right cell, draw wires from the new connector positions up or down, coming as close to the final height as possible without getting too close to neighboring wires. This process of moving to the side

one unit, then approaching the desired height, continues until all wires are at the appropriate positions. Once this is done, the two cells can be fused using the standard abutment routine.

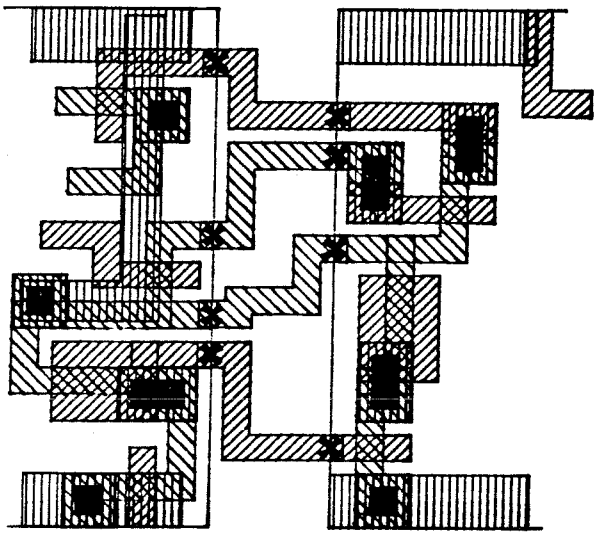


Fig. 4-9: River Routing

The river routing scheme is topologically identical to the stretching and abutment schemes. Because of this, the interconnection requirements are similar to the requirements of the other schemes. We do not need to specify the interconnection list, because this information is implied from the cells. We have mentioned one algorithm for generating the interconnection wires. Finally, the interconnection is verified using the simple abutment routine.

The river routing interconnection scheme is more generally useful than either stretching or abuting, since the connector positions are free to move without drastically affecting the cell size or performance. The connectors are still restricted to being in the same order and on a single mask layer. River routers are useful in chip assemblers, however, because there are cases where the connectors are in the proper order and on the proper layers, but not at the proper positions. For example, if the user connects buffers to each connector on a particular side of a cell, the buffer cell can be designed to have the appropriate number of connectors in the correct order so that the cells can be river-routed together.

There are several schemes for improving and generalizing the river route process. Appendix 3 discusses river routes in some detail.

4.3.4: One-sided General Interconnect

In each of the wiring methodologies presented above, the connectors of the two cells were required to be in the proper order on the proper layers. For general purpose cell composition, such is not the case. For the connectors to satisfy these requirements, both cells would have been designed with the interface specification known, so that the connectors can be put in the proper locations. This means that the wiring is done inside the cells! The user has to do the wiring by hand. There is also a one-to-one correspondence between connectors of the two cells, which is a serious limitation on the interconnectability of cells.

A more general interconnection scheme would permit arbitrary interconnections between the signals on adjacent edges of cells [5]. The user would specify the interconnections as net-lists, which are lists of connectors to be connected together. Using this style of interconnection, the user is required to specify the interconnection information, whereas the previously presented methods implied the interconnection information.

An example of a general interconnection is shown in figure 4-10. We no longer restrict the connectors' layers or positions. We do not require that there be the same number of connectors on the two cells. The only requirement is that the interconnections between two cells have the connectors on the edges between the cells.

An advantage of this interconnection technique is that the design of the chip can easily be partitioned. The two cells can be designed by independent design teams given only a functional specification of the interface between the cells. Also, if a cell is redesigned, the interconnection program is re-run with the original specification and the new composition cell is complete.

One of the disadvantages of this technique is that the user has to specify the interconnection between the two cells. This can be a fairly large specification if there are many connectors on the cells. Also, the possibility of errors requires

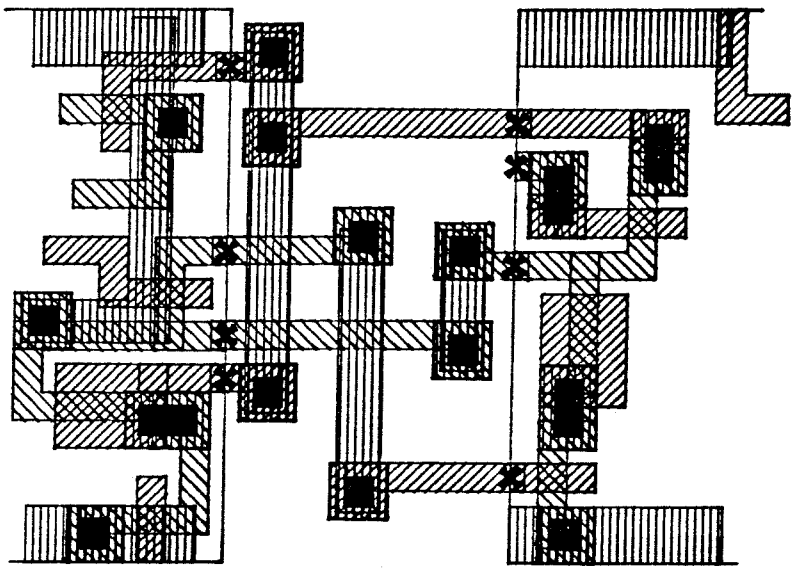


Fig. 4-10: General Interconnection

checking of the specification. Signal typing will catch most of the dumb mistakes, but many of the logical errors can only be caught by checking the specifications. Another disadvantage is that this style of interconnection consumes more chip area than the other approaches. Because of these disadvantages, one would like to use the stretching and river routing techniques where they logically fit, and reserve the general interconnection schemes for the remaining routes.

4.3.5: Four-sided General Interconnect

In the One-sided general interconnector, we require that all interconnections between adjacent cells use connectors on the shared edge of the cells. While this technique may be useful in many circumstances, there are times when the connectors do not lie between the cells. Figure 4-11 shows a route which connects to signals on the North and South edges of the cells, in addition the the shared edges of the cells. This style of interconnection is termed 'Four-sided interconnect', since the connectors may be on any of the four sides of a cell [5].

There exists a technique which converts the four-sided interconnect problem into a series of one-sided interconnections. This means that the four-sided interconnection can be as time and area efficient as the one-sided interconnect, but that the generality of the four-sided interconnect can be capitalized upon. In figure 4-12, we show three steps in the fusion of cells. In this figure, we perform all of

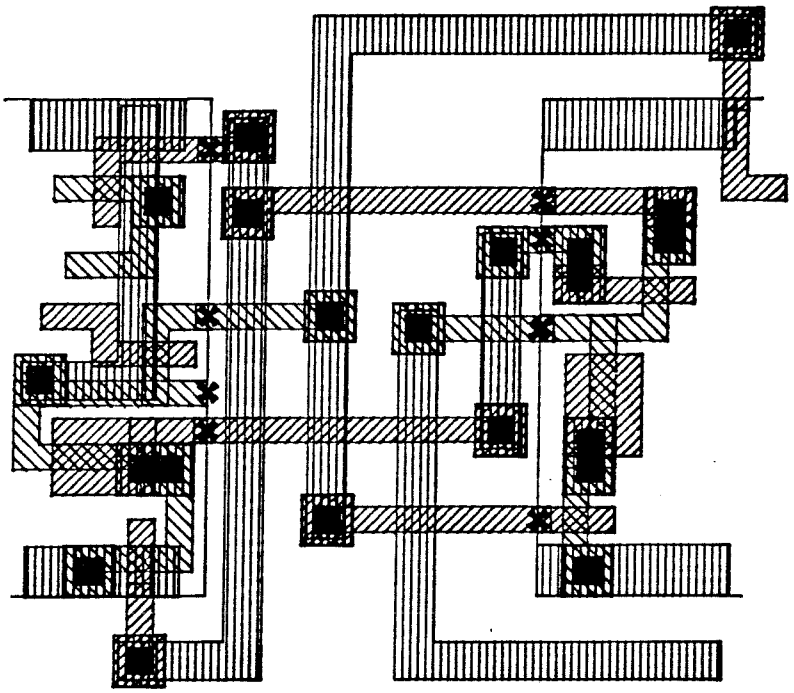


Fig. 4-11: Four Sided Interconnection

the interconnections at one level before moving to the next level. We perform an Immediate fusion of the two cells. When we do this, some of our interconnecting wires must route out of the channel between the two cells. For example, in the figure 4-12a, some of the wires route on the east sides of the two cells, which is the channel between cells in figure 4-12b. The first two cells have taken channel area from the next higher level. This higher level channel route cannot share the area used in this lower level route. If, instead of routing outside the channel, we only routed inside the channel, but kept a list of incompleted connections, we can share the channels for the various levels in the hierarchical fusion. In figure 4-13, we show the same interconnection, but with the Delayed technique. We have only routed in the channel, but kept the incomplete routes with the composition cell. When we go to fuse this cell to neighboring cells, we add these incomplete routes to the routes required by the new interconnection and route all of the wires in the new channel. The resulting layout is considerably smaller than the immediate interconnection layout.

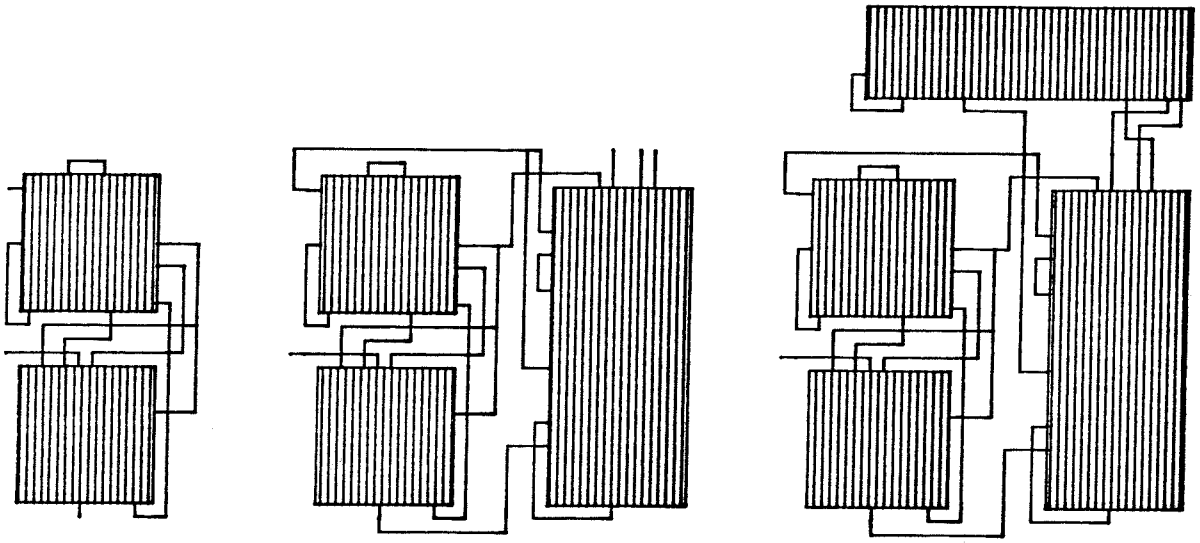


Fig. 4-12: Immediate Interconnect

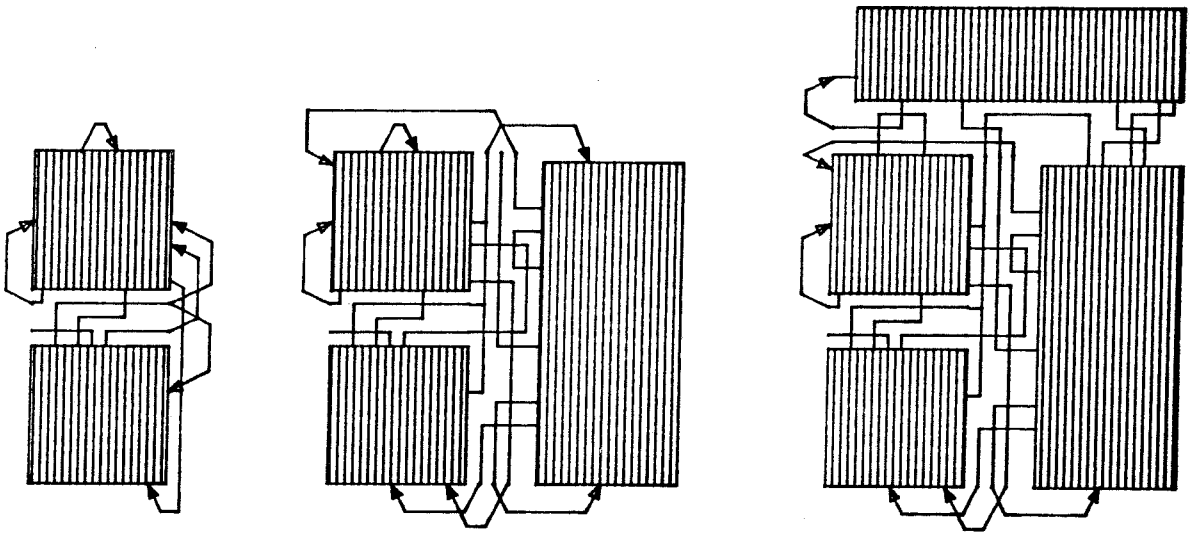


Fig. 4-13: Delayed Interconnect

4.4: Conclusions

In VLSI design, the design of the glue which interfaces cells is considerably harder than the design of the cells themselves. Much effort has gone into building systems to aid in the construction of the cells, but the interconnect problem has largely been ignored. In this chapter we have seen several techniques for fusing cells together. A Chip Assembler which contains these interconnectors, would greatly aid in the design of large chips.

We have also introduced checking into the design of chips. Rather than analysing the results of a chip design to verify the interconnection, we design layouts that are correct by construction. The analysis style of verification is becomes impractical as chip sizes and densities increase. We must move to the synthesis technique of correctness by construction if we wish to design correct layouts at a reasonable cost.

Chapter 5: A Simple Silicon Compiler

To illustrate the concepts involved in silicon compilation, this chapter will develop a simple yet complete compiler. This compiler may be called the Random Logic Compiler: it is designed to compile TTL-style circuits. Following a discussion of the floorplan for this particular compiler, we will see the code for the chip assembler and silicon compiler. After this, we will explore some of the possible extensions which allow higher-level user specification of the design.

A silicon compiler is a program which translates a high-level, behavioral chip specification into the 'machine language' of silicon design: a set of VLSI masks. The foundation of a silicon compiler is an Imbedded Language system. Within the imbedded language, the structure of the compiler's floorplan is designed. The floorplan is the logical and physical arrangement of circuitry that the compiler generates. Given this structure and the graphics language, procedures are written which generate the 'cells' or circuits to be used on the chips. These cells can take parameters and perform calculations as the layout is generated. These cells also generate logical information, such as the list of connection points, in addition to the actual physical information that describes the design. The user specification is used to provide the parameter values for the cell procedures. The compiler links these sublayouts together to complete the chip.

5.1 The Floorplan

The floorplan limits the capabilities of any compiler. The more limited or fixed the floorplan, the smaller the class of compilable chips; the more relaxed or generalized the floorplan, the broader the class. On the other hand, the more specific the compiler, the more specialized it can be for a particular design style, which has two-fold benefits: the resulting layouts are usually more optimized, and the specification for any particular chip are very concise.

For our example compiler, we want to generate arbitrary interconnections of NAND, NOR, and INVERT gates. These gates will be positioned horizontally in a single row, as illustrated in figure 5-1. The power lines will run along the top and bottom of the row, signal lines will run horizontally between the power lines, and the gates will be positioned vertically.

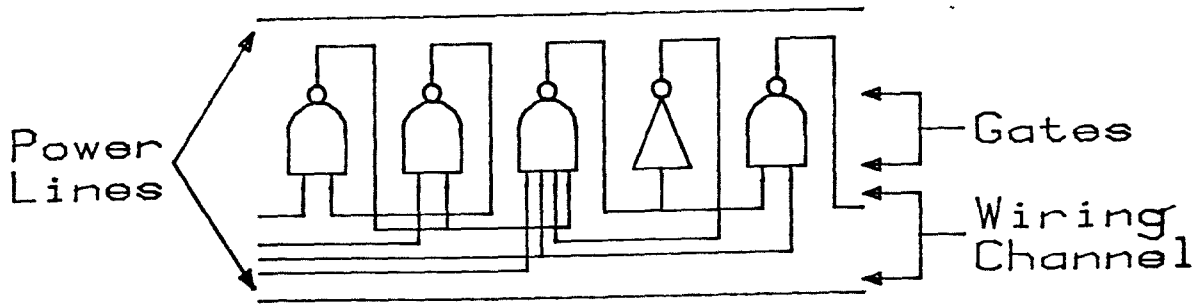


Fig. 5-1: RLC Floorplan

Since we are not restricting the number of gates, nor the interconnection possibilities, component locations cannot be fixed to exact physical locations. For instance, the location of the upper power line can not be fixed since the power line width is related to the power consumed by the circuit, which is a function of the number of gates in the circuit. Hence, unless we arbitrarily limit the number of gates, we can not state where the upper power line should be for all designs. These positions can, however, be parametrized in terms of global variables. For our compiler, the variable 'YVDD' will be set to the y-coordinate for the center of the VDD line. All of our cells will be designed to use 'YVDD' when referring to features associated with the VDD line, allowing us to position this line after we know how many gates are needed in the circuit. Similarly, 'YGND' will be the y-coordinate for the center of the ground line, and 'POWER' will be the width of the power lines.

In addition to the physical aspects of the floorplan as described above, we will need conventions for communication of information between the cells and the compiler. There is some information that the compiler needs which the cells compute, and there is some information that the cells need which the compiler computes. In our logic gate compiler, the procedures which generate each type of gate know where the inputs and outputs of the gates should connect relative to the cell's origin, while the compiler knows the origins for each cell. If the compiler were required to compute the connection locations, the compiler would be tied to specific cell implementations. One could not change a cell without having to change the compiler as well, and verification of the changes would be a formidable task. For the same reason, local cells should not have to generate information that belongs in the compiler.

In the logic gate compiler, there are two bilateral communication paths that are needed: the compiler gives each cell the x-coordinate of its origin, while the cells report their width to the compiler, so that the compiler can compute the next origin; the compiler assigns vertical position for each interconnection wire, but the cells must give the endpoints of the wires based on where the wire connects inside the cell. The first communication, involving cell origins, is done by direct parameter passing. The gate procedures are passed a REAL number which the procedures use for a horizontal origin. Each gate returns its width by setting a global variable CWIDTH. The second communication, for interconnection positions, is done through instances of a datatype called PHYSICAL_WIRE. PHYSICAL_WIREs receive y-values from the compiler. The gates can inspect this information in the PHYSICAL_WIREs to determine which channel the wire uses. The gates may pass x-values to the PHYSICAL_WIREs so that the wires will extend to the proper horizontal positions.

5.2 Chip Assembler

Having defined the conventions of the compiler, the cell generation routines may be written. The following code gives the implementation routines for the logic gate compiler:

```
TYPE    PHYSICAL_WIRE= [HEIGHT,LEFT,RIGHT:REAL  NAME:QS];
        PHYSICAL_WIRES= { PHYSICAL_WIRE };

VAR YVDD,YGND,PWIDTH,CWIDTH=REAL;

DEFINE CONNECT(WIRE:PHYSICAL_WIRE  X:REAL):
    @(WIRE).LEFT::= MIN X;
    @(WIRE).RIGHT::= MAX X;
ENDDEFN

DEFINE PULLUP(OUTPUT:PHYSICAL_WIRE  X:REAL)=MRG:
    DO  CONNECT(OUTPUT,X-2);
    GIVE    {BOX(RED,X-16#0\TO X-5#6);
             BOX(YELLOW,X-16#-2.\TO X-5#9);
             WIRE(GREEN,2,{X-13#YVDD;.#3;X-8#.;.#.-5;.+5#.;.#OUTPUT.HEIGHT});
             GCB\AT {X-12#YVDD;X-2#OUTPUT.HEIGHT};
             GRCBU\AT X-7#-1.}
ENDDEFN

DEFINE NAND(INPUTS:PHYSICAL_WIRES  OUTPUT:PHYSICAL_WIRE  X:REAL)=MRG:
    BEGIN   VAR IN=PHYSICAL_WIRE;NUMBER=INT;X2=REAL;
    DO  NUMBER:= +1 FOR IN $E INPUTS;;
```

```
        X2:=X-10-2*NUMBER;
        DO CONNECT(IN,X2); FOR IN $E INPUTS;
        CWIDTH:=X2-5;
    GIVE {GCB\AT X-8#YGND;
          BOX(GREEN,X2+3#YGND-2\TO X-7#-1.);
          COLLECT {RCB\AT X2#IN.HEIGHT;
                   WIRE(RED,2,{X2#IN.HEIGHT;X-6#.})} FOR IN $E INPUTS;;
          PULLUP(OUTPUT,X)}
    END
ENDDEFN

DEFINE NOR(INPUTS:PHYSICAL_WIRES  OUTPUT:PHYSICAL_WIRE  X:REAL)=MRG:
    BEGIN   VAR IN=PHYSICAL_WIRE;
    DO  DO CONNECT(IN,X-16); FOR IN $E INPUTS;
        CWIDTH:=X-24;
    GIVE {GCB\AT X-19#YGND;
          WIRE(GREEN,2,{X-20#YGND;.#-6.});
          WIRE(GREEN,2,{X-8#YGND+PWIDTH/2+9;.#-2.});
          COLLECT {RCB\AT X-16#IN.HEIGHT;
                   WIRE(RED,2,{X-15#IN.HEIGHT+1;X-11#.;.#.+5});
                   WIRE(GREEN,2,{X-20#IN.HEIGHT+4;X-8#.})}
          FOR IN $E INPUTS;;
          PULLUP(OUTPUT,X)}
    END
ENDDEFN

DEFINE INVERT(INPUTS:PHYSICAL_WIRES  OUTPUT:PHYSICAL_WIRE  X:REAL)=MRG:
    BEGIN   VAR IN=PHYSICAL_WIRE;
    DO  IN:=INPUTS[1];
        CONNECT(IN,X-12);
        CWIDTH:=X-17;
    GIVE {GCB\AT X-8#YGND;
          BOX(GREEN,X-9#YGND-2\TO X-7#-1.);
          RCB\AT X-12#IN.HEIGHT;
          WIRE(RED,2,{X-12#IN.HEIGHT;X-6#.});
          PULLUP(OUTPUT,X)}
    END
ENDDEFN
```

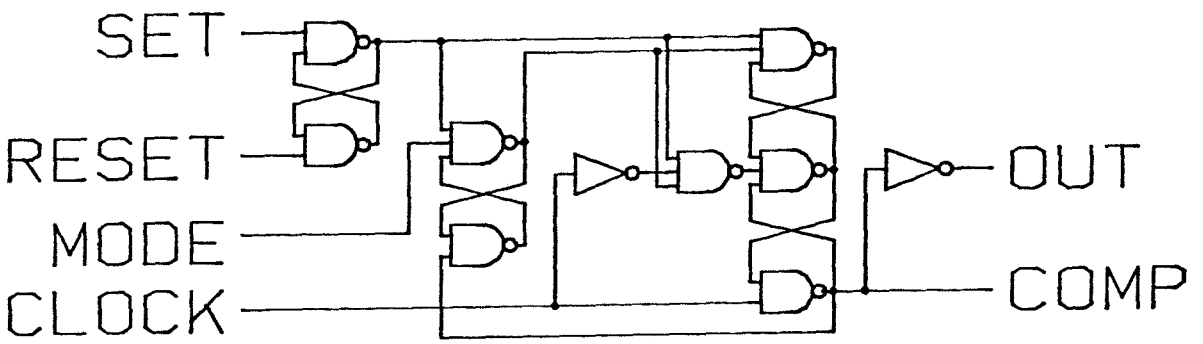


Fig. 5-2: Pulse Synchronizer Circuit

At this point, we have routines for implementing NAND, NOR, and INVERT gates. We can assemble chips by generating the required PHYSICAL_WIREs, initializing parameters in each wire, calling the appropriate gate functions, collecting the

resulting cells, and drawing the interconnection wires. The following example illustrates how one could use our chip assembler for designing a 'pulse synchronizer'. Figure 5-2 gives the logic diagram of the circuit. This code will produce the layout shown in figure 5-3:

```
POWER:=4;
YVDD:=9;
YGND:=-69.;
CWIDTH:=0;

VAR WIRES=PHYSICAL_WIRES;  WIRE=PHYSICAL_WIRE;
WIRES:={[HEIGHT:-8.  LEFT:-999999.  RIGHT:-999999.];
        [HEIGHT:-17. LEFT:-999999.  RIGHT:-999999.];
        [HEIGHT:-26. LEFT:-999999.  RIGHT:-999999.];
        [HEIGHT:-35. LEFT:-999999.  RIGHT:-999999.];
        [HEIGHT:-44. LEFT: 999999.  RIGHT:-999999.];
        [HEIGHT:-53. LEFT: 999999.  RIGHT:-999999.];
        [HEIGHT:-17. LEFT: 999999.  RIGHT:-999999.];
        [HEIGHT:-44. LEFT: 999999.  RIGHT:-999999.];
        [HEIGHT:-17. LEFT: 999999.  RIGHT:-999999.];
        [HEIGHT:-62. LEFT: 999999.  RIGHT:-999999.];
        [HEIGHT:-35. LEFT: 999999.  RIGHT: 999999.];
        [HEIGHT:-17. LEFT: 999999.  RIGHT:-999999.];
        [HEIGHT:-8.  LEFT: 999999.  RIGHT:-999999.];
        [HEIGHT:-8.  LEFT: 999999.  RIGHT: 999999.]};

VAR RESULT=MRG;
RESULT:=NAND({WIRES[11]},WIRES[14],CWIDTH);
RESULT::=\UNION NAND({WIRES[3];WIRES[9]},WIRES[11],CWIDTH);
RESULT::=\UNION NAND({WIRES[8];WIRES[10];WIRES[11]},WIRES[9],CWIDTH);
RESULT::=\UNION NAND({WIRES[6];WIRES[9];WIRES[13]},WIRES[10],CWIDTH);
RESULT::=\UNION NAND({WIRES[6];WIRES[7];WIRES[13]},WIRES[8],CWIDTH);
RESULT::=\UNION NAND({WIRES[3]},WIRES[7],CWIDTH);
RESULT::=\UNION NAND({WIRES[11];WIRES[13]},WIRES[12],CWIDTH);
RESULT::=\UNION NAND({WIRES[4];WIRES[6];WIRES[12]},WIRES[13],CWIDTH);
RESULT::=\UNION NAND({WIRES[2];WIRES[5]},WIRES[6],CWIDTH);
RESULT::=\UNION NAND({WIRES[1];WIRES[6]},WIRES[5],CWIDTH);
RESULT::=\UNION {COLLECT WIRE(BLUE,3,{CWIDTH MAX WIRE.LEFT # WIRE.HEIGHT;
                                      0 MIN WIRE.RIGHT#.})
                 FOR WIRE $E WIRES;};
RESULT::=\UNION {BOX(BLUE,CWIDTH+3#YVDD-3\TO 4#YVDD+(POWER-3 MAX 2));
                 BOX(BLUE,CWIDTH-1#YGND+2-POWER\TO 0#YGND+2)};

PLOT(RESULT,'Q'\AIF);
```

This example shows how our chip assembler has raised the level of user specification away from the low-level wires and boxes, yet there are still many implementation details left for the user to specify. Too, this specification is not in a form conceptually clear for the user. The designer will make many specification errors, and these errors will be very difficult to locate, because of the obscure nature of the specification language.

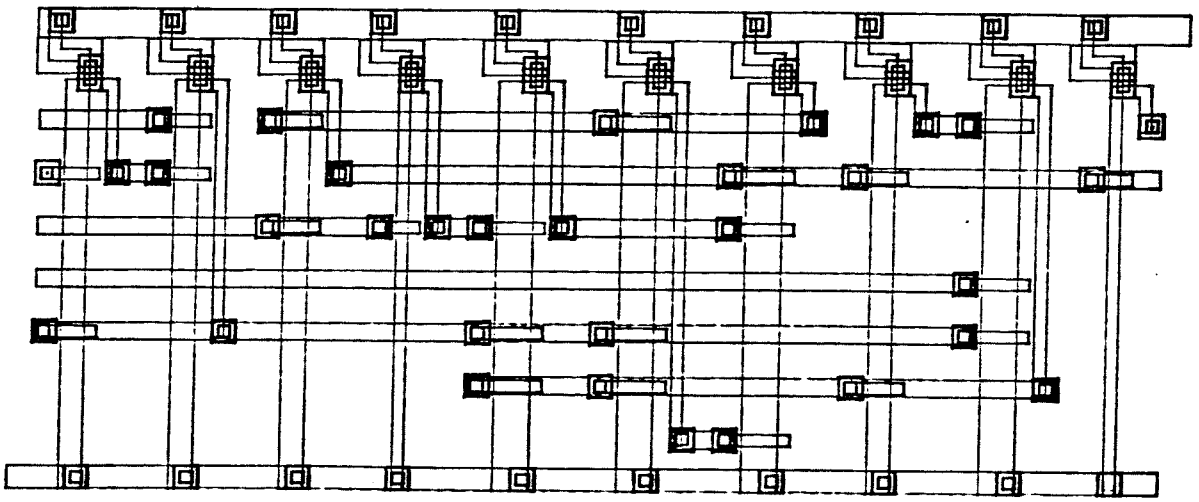


Fig. 5-3: Layout of Pulse Synchronizer

5.3 The Compiler

It is rather clumsy to generate chips in the assembler form given above. The user must constantly be concerned with implementation details, and design errors are common. If the implementation details could be hidden from the user so that the user could design with a higher level description, the design task would be easier and many errors would be eliminated. We will generate new data structures that allow us to describe the chip in a more functional manner, without the physical details, and write a program which will handle the physical concerns, given one of these new data structures. The following section of code lists both the data structures and the compiler:

```
TYPE    SIGNAL_WIRE= [FROM:GATE
                      TO:GATES
                      NAME:QS
                      PHYSICAL:PHYSICAL_WIRE
                      VLEFT,VRIGHT,VHEIGHT:INT];

        SIGNAL_WIRES= { SIGNAL_WIRE };

        GATE= [INPUTS:SIGNAL_WIRES
               OUTPUT:SIGNAL_WIRE
               TYPE:GATE_TYPE
               INDEX:INT];

        GATES= { GATE };
```

```
        GATE_TYPE= SCALAR(NAND,NOR,INVERT);

        CHIP= [GATES:GATES
               INPUTS,OUTPUTS,SIGNALS:SIGNAL_WIRES
               SIGNAL_COUNT:INT
               NAME,DESCRIPTION:QS];


DEFINE PHYSICAL(SW:SIGNAL_WIRE)=PHYSICAL_WIRE:  SW.PHYSICAL ENDDEFN

DEFINE PHYSICAL(SWS:SIGNAL_WIRES)=PHYSICAL_WIRES:
    BEGIN   VAR S=SIGNAL_WIRE;
    {COLLECT S\PHYSICAL FOR S $E SWS;}
    END
ENDDEFN


DEFINE PACK(C:CHIP):
    BEGIN   VAR SWS=SIGNAL_WIRES;H=INT;G=GATE;S=SIGNAL_WIRE;
            DEFINE SORT(SWS:SIGNAL_WIRES)=SIGNAL_WIRES:
                BEGIN   VAR OUT=SIGNAL_WIRES;W=SIGNAL_WIRE;I,J,K=INT;
                DO  OUT:=NIL;
                    WHILE DEFINED(SWS); DO
                        I:=-1;
                        FOR W $E SWS;&& FOR J FROM 1 BY 1; DO
                            IF W.VLEFT>I THEN
                                I:=W.VLEFT;
                                K:=J; FI
                        END
                        OUT:= SWS[K] <$;
                        SWS[K-]:=SWS[K+1-];
                    END
                GIVE OUT
                END
            ENDDEFN
            DEFINE DRAW_WIRE(LEFT:INT):
                BEGIN   VAR W=SIGNAL_WIRE;I=INT;
                IF THERE_IS W.VLEFT>LEFT FOR W $E SWS;&& FOR I FROM 1 BY 1;
                THEN    SWS[I-]:=SWS[I+1-];
                        @(W).VHEIGHT:=H;
                        DRAW_WIRE(W.VRIGHT); FI
                END
            ENDDEFN
    FOR G $E C.GATES;&& FOR H FROM 1 BY 1;DO @(G).INDEX:=H; END
    FOR S $E C.SIGNALS; DO
        @(S).VLEFT:= IF DEFINED(S.TO)
                     THEN S.FROM.INDEX MIN MIN G.INDEX FOR G $E S.TO;
                     ELSE S.FROM.INDEX FI;
        @(S).VRIGHT:= S.FROM.INDEX MAX MAX G.INDEX FOR G $E S.TO;;
    END
    FOR S $E C.INPUTS;DO @(S).VLEFT:=0; END
    FOR S $E C.OUTPUTS;DO @(S).VRIGHT:=999999; END
    SWS:=C.SIGNALS\SORT;
    WHILE DEFINED(SWS);&& FOR H FROM 1 BY 1; DO DRAW_WIRE(-1); END
    END
ENDDEFN

DEFINE SETUP_DIMENSIONS(C:CHIP):
    BEGIN   VAR G=GATE;S=SIGNAL_WIRE;H=REAL;
    POWER:= WIDTH(+.25 FOR G $E C.GATES;) MAX 4;
    YGND:= -9.*(MAX S.VHEIGHT FOR S $E C.SIGNALS;)-4-POWER/2;
```

```
    YVDD:=6+POWER/2 MAX 9;
    END
ENDDEFN

DEFINE INITIALIZE_WIRES(C:CHIP):
    BEGIN   VAR S=SIGNAL_WIRE;
    FOR S $E C.SIGNALS; DO
        @(S).PHYSICAL:=[LEFT:999999
                        RIGHT:-999999.
                        HEIGHT:1-9*S.VHEIGHT
                        NAME:S.NAME];
    END
    FOR S $E C.INPUTS; DO
        @(S).PHYSICAL.LEFT:=-999999.;
    END
    FOR S $E C.OUTPUTS; DO
        @(S).PHYSICAL.RIGHT:=999999;
    END
    END
ENDDEFN

DEFINE DRAW_CELLS(C:CHIP)=MRG:
    BEGIN   VAR X=REAL;G=GATE;M=MRG;
    {COLLECT DO  M:= CASE G.TYPE OF
                     NOR: NOR(G.INPUTS\PHYSICAL,G.OUTPUT\PHYSICAL,CWIDTH)
                     NAND:NAND(G.INPUTS\PHYSICAL,G.OUTPUT\PHYSICAL,CWIDTH)
                     INVERT:INVERT(G.INPUTS\PHYSICAL,G.OUTPUT\PHYSICAL,CWIDTH)
                     ENDCASE;
             GIVE M
     FOR G $E REVERSE(C.GATES);}
    END
ENDDEFN

DEFINE DRAW_WIRES(C:CHIP)=MRG:
    BEGIN   VAR S=SIGNAL_WIRE;LEFT,RIGHT=REAL;
    DO  LEFT:=CWIDTH+5;
        RIGHT:=-2.;
    GIVE    {COLLECT WIRE(BLUE,3,(S.PHYSICAL.LEFT#S.PHYSICAL.HEIGHT;
                                  S.PHYSICAL.RIGHT#.))
             FOR S $E C.SIGNALS;
             EACH_DO @(S.PHYSICAL).LEFT::= MAX LEFT;
                     @(S.PHYSICAL).RIGHT::= MIN RIGHT;;;
             BOX(BLUE,CWIDTH+3#YVDD-POWER/2\TO 4#YVDD+POWER/2);
             BOX(BLUE,CWIDTH-1#YGND-POWER/2\TO 0#YGND+POWER/2)}
    END
ENDDEFN

DEFINE LOAD(S:SIGNAL_WIRE)=REAL:
    BEGIN   VAR G=GATE;T=SIGNAL_WIRE;
    (+ CASE G.TYPE OF
         NOR:    1
         INVERT: 1
         NAND:   +1 FOR T $E G.INPUTS;
       ENDCASE FOR G $E S.TO;)*Q_LOAD +
    LOAD(BLUE,WIDTH(BLUE),S.PHYSICAL.RIGHT-S.PHYSICAL.LEFT)
    END
ENDDEFN

DEFINE COMPILE(C:CHIP)=MRG:
    BEGIN   VAR M=MRG;G=GATE;S=SIGNAL_WIRE;
    DO  CWIDTH:=0;
```

```
        PACK(C);
        SETUP_DIMENSIONS(C);
        INITIALIZE_WIRES(C);
        M:=DRAW_CELLS(C);
        M:={M;DRAW_WIRES(C)};
    GIVE M
    END
ENDDEFN
```

There are two basic datatypes defined here: SIGNAL_WIRE and GATE. These are abstract representations for PHYSICAL_WIREs and instances of the gates. There is an additional datatype, CHIP, which holds references to all of the gates and wires which comprise the chip. The COMPILE function consumes a CHIP and produces an MRG, which is the ICLIC representation for layout. COMPILE calls five procedures. The first assigns horizontal channels to each of the interconnection wires. The second procedure computes the values for the global positioning variables. The third procedure initializes the PHYSICAL_WIREs. The fourth procedure calls each of the gate cells. The final procedure draws the actual interconnection wires.

We now have a program which will take an abstract structure representing the behavioral definition of a chip and generate the layout. To facilitate the construction of these abstract chip specifications, support routines may be designed. The following code provides routines for modifying this data structure, followed by routines for generating this data structure.

```
VAR CHIP=CHIP;

DEFINE EQ(A,B:GATE)=BOOL:    MACRO-10('LSPEQ$')

DEFINE EQ(A,B:SIGNAL_WIRE)=BOOL:     MACRO-10('LSPEQ$')

DEFINE LINK_INPUT(G:GATE  S:SIGNAL_WIRE):
    @(S).TO::= G <$;
    @(G).INPUTS::= S <$;
ENDDEFN

DEFINE LINK_OUTPUT(G:GATE  S:SIGNAL_WIRE):
    @(G).OUTPUT:=S;
    @(S).FROM:=G;
ENDDEFN

DEFINE UNLINK_INPUT(G:GATE  S:SIGNAL_WIRE):
    BEGIN   VAR Q=GATE;R=SIGNAL_WIRE;
    @(S).TO:={COLLECT Q FOR Q $E S.TO;WITH -(Q\EQ G);};
    @(G).INPUTS:={COLLECT R FOR R $E G.INPUTS;WITH -(R\EQ S);};
    END
ENDDEFN

DEFINE UNLINK_OUTPUT(G:GATE  S:SIGNAL_WIRE):
    @(S).FROM:=NIL;
```

```
    @(G).OUTPUT:=NIL;
ENDDEFN

DEFINE ELIMINATE(G:GATE):
    BEGIN   VAR Q=GATE;
    CHIP.GATES:={COLLECT Q FOR Q $E CHIP.GATES;WITH -(Q\EQ G);};
    END
ENDDEFN

DEFINE ELIMINATE(S:SIGNAL_WIRE):
    BEGIN   VAR R=SIGNAL_WIRE;
    CHIP.SIGNALS:={COLLECT R FOR R $E CHIP.SIGNALS;WITH -(R\EQ S);};
    CHIP.INPUTS:={COLLECT R FOR R $E CHIP.INPUTS;WITH -(R\EQ S);};
    CHIP.OUTPUTS:={COLLECT R FOR R $E CHIP.OUTPUTS;WITH -(R\EQ S);};
    END
ENDDEFN

DEFINE FUSE(A,B:SIGNAL_WIRE):
    BEGIN   VAR G=GATE;C=CHAR;S=SIGNAL_WIRE;
    IF DEFINED(B.FROM)! THERE_IS S\EQ B FOR S $E CHIP.INPUTS; THEN
        IF DEFINED(A.FROM) ! THERE_IS S\EQ A FOR S $E CHIP.INPUTS; THEN HELP;
        ELSE    @(A).INPUT:=B.INPUT;
                G:=B.FROM;
                IF DEFINED(G) THEN
                    UNLINK_OUTPUT(G,B);
                    LINK_OUTPUT(G,A); FI FI FI
    IF THERE_IS S\EQ B FOR S $E CHIP.OUTPUTS; THEN CHIP.OUTPUTS::= A <$; FI
    FOR G $E B.TO; DO
        UNLINK_INPUT(G,B);
        LINK_INPUT(G,A);
    END
    ELIMINATE(B);
    END
ENDDEFN

LET QS BECOME SIGNAL_WIRE BY
    BEGIN   VAR S=SIGNAL_WIRE;
    IF THERE_IS S.NAME\EQ QS FOR S $E CHIP.SIGNALS; THEN S
    ELSE    DO  S:=[NAME:QS];
                CHIP.SIGNALS::= S <$;
            GIVE S  FI
    END;

DEFINE NEW_SIGNAL=SIGNAL_WIRE:  SC((CHIP.SIGNAL_COUNT::=+1;))   ENDDEFN

DEFINE SET(S:SIGNAL_WIRE  G:GATE):  LINK_OUTPUT(G,S);   ENDDEFN

LET GATE BECOME SIGNAL_WIRE BY
    BEGIN   VAR S=SIGNAL_WIRE;
    DO  S:=NEW_SIGNAL;
        SET(S,GATE);
    GIVE S
    END;

DEFINE INPUT(QS:QS):    CHIP.INPUTS::= QS <$;   ENDDEFN

DEFINE OUTPUT(QS:QS):   CHIP.OUTPUTS::= QS <$;  ENDDEFN
```

```
DEFINE NEW_CHIP:    CHIP:=NIL;  ENDDEFN

DEFINE FINISH:
    CHIP.GATES:=REVERSE(CHIP.GATES);
ENDDEFN

DEFINE NEW_GATE(SWS:SIGNAL_WIRES  TYPE:GATE_TYPE)=GATE:
    BEGIN   VAR GATE=GATE;SW=SIGNAL_WIRE;
    DO  GATE:=[INPUTS:SWS  TYPE:TYPE];
        CHIP.GATES::= GATE <$;
        DO @(SW).TO::= GATE <$; FOR SW $E SWS;
    GIVE GATE
    END
ENDDEFN

DEFINE NAND(SWS:SIGNAL_WIRES)=GATE: NEW_GATE(SWS,NAND)       ENDDEFN

DEFINE NOR(SWS:SIGNAL_WIRES)=GATE:  NEW_GATE(SWS,NOR)        ENDDEFN

DEFINE INVERT(SW:SIGNAL_WIRE)=GATE: NEW_GATE({SW},INVERT)    ENDDEFN

DEFINE AND(SWS:SIGNAL_WIRES)=GATE:  SWS\NAND\INVERT          ENDDEFN

DEFINE OR(SWS:SIGNAL_WIRES)=GATE:   SWS\NOR\INVERT           ENDDEFN

DEFINE NAND(A,B:SIGNAL_WIRE)=GATE:  NAND({A;B})              ENDDEFN

DEFINE NOR(A,B:SIGNAL_WIRE)=GATE:   NOR({A;B})               ENDDEFN

DEFINE AND(A,B:SIGNAL_WIRE)=GATE:   AND({A;B})               ENDDEFN

DEFINE OR(A,B:SIGNAL_WIRE)=GATE:    OR({A;B})                ENDDEFN
```

To specify the function of a chip, we call these new procedures. To start the description of a chip, we call NEW_CHIP, which initializes the system. Next, we enter the logical equations by calling the SET function. We then state which signals are inputs or outputs of the chip by calling the INPUT or OUTPUT procedures. Finally, we call the FINISH routine, which completes the linking of various portions of the description. Signal wires are identified by enclosing their names in single quotes. Logical equations are specified by calling the NAND, NOR, AND, OR, and INVERT functions. To specify the 'pulse synchronizer' from above, the following code could be used:

```
NEW_CHIP;

SET('ENABLE',NAND('SET',NAND('ENABLE','RESET')));

SET('COMP',NAND('CLOCK','X'));

SET('X',NAND({NAND({INVERT('CLOCK');'ENABLE';'Y'});
        NAND({'ENABLE';'Y';'X'});
        'COMP'}));
```

```
SET('Y',NAND({'ENABLE';'MODE';NAND('COMP','Y')}));

SET('OUT',INVERT('COMP'));

INPUT('SET');INPUT('RESET');INPUT('CLOCK');INPUT('MODE');

OUTPUT('OUT');OUTPUT('COMP');

FINISH;
```

Notice how concise this description is compared to the description required for the chip assembler. In addition, this description is more natural for the designer, which assures fewer specification errors. In the compiler, we referred to signal wires by name, whereas in the assembler we used indexes into a global list. The compiler allows us to work with more of our own semantics, and to include more of this semantics in the chip description.

5.4 Compiler Extensions

There is a major difference between the assembler and compiler specifications of a chip. With the assembler, we write a program which contains the specification of the chip; with the compiler, we generate a data structure which contains this information. The data structure representation limits our design capabilities since the data structure is not as general as a programming language, but there is an advantage to data structure representations: we can write programs to modify, generate, or examine our chip specification.

In the RLC, we may wish to perform logic minimization upon a set of equations to reduce the number of gates required to implement those equations. Programs of this class are called Optimizers, which are discussed in section 5.4.1. In addition, the user may wish to specify the equations using mathematical notation, letting the program translate this formal mathematical notation into the appropriate data structures. Section 5.4.2 shows examples of these Generators and Parsers. Our data structure contains more information than strictly a layout. The user may wish to examine this information. In RLC, the user may wish to simulate the circuit. Such programs are called Examiners, which are discussed in section 5.4.3.

These extensions have been added to the compiler presented above. Appendix 3 contains a users guide to the complete compiler, along with all source listings of the

compiler.

5.4.1 Optimizers

Through the several levels of chip design (architecture, block, logic, gate, etc.), much thought is devoted to optimizing the design. Many of the optimizations are algorithmic in nature: a formula or program can be stated which will apply the optimization to the design. Since our compiler's input is a data structure, we can design programs which will operate on the input data in attempts to produce more optimal chips.

One optimization we might consider is the removal of unnecessary inverters. When using predefined cells, the user may need to invert a signal before connecting to an input of the cell, only to have the signal re-inverted by a gate within the cell. One, perhaps both, of the inverters are superfluous and can be removed. We can design an optimization program which scans for series inverters and removes the unnecessary inverters. Figure 5-4 illustrates this process. In the first example, both polarities of the signal are required, in which case the second inverter is the only unnecessary inverter. The second example shows a case where the signal is inverted twice, but the intermediate signal is never used, in which case both inverters can be removed. The following routines are used to perform this optimization.

```
DEFINE GET_INVERT(S:SIGNAL_WIRE)=SIGNAL_WIRE:
    BEGIN   VAR T=SIGNAL_WIRE;G=GATE;
    IF S.FROM.TYPE=INVERT THEN
        GIVING S.FROM.INPUTS[1]
        DO  IF -(DEFINED(S.TO)! THERE_IS T\EQ S FOR T $E CHIP.OUTPUTS;) THEN
                G:=S.FROM;
                UNLINK_OUTPUT(G,S);
                UNLINK_INPUT(G,G.INPUTS[1]);
                ELIMINATE(G);
                ELIMINATE(S); FI
        END
    EF THERE_IS G.TYPE=INVERT FOR G $E S.TO; THEN G.OUTPUT
    ELSE INVERT(S) FI
    END
ENDDEFN

DEFINE REMOVE_INVERTERS:
    BEGIN   VAR G=GATE;S,T=SIGNAL_WIRE;
    FOR G $E CHIP.GATES;WITH G.TYPE=INVERT;WITH DEFINED(G.OUTPUT); DO
        S:=G.OUTPUT;
        T:=G.INPUTS[1];
```

```
        UNLINK_OUTPUT(G,S);
        UNLINK_INPUT(G,T);
        ELIMINATE(G);
        FUSE(T\GET_INVERT,S);
    END
    END
ENDDEFN
```

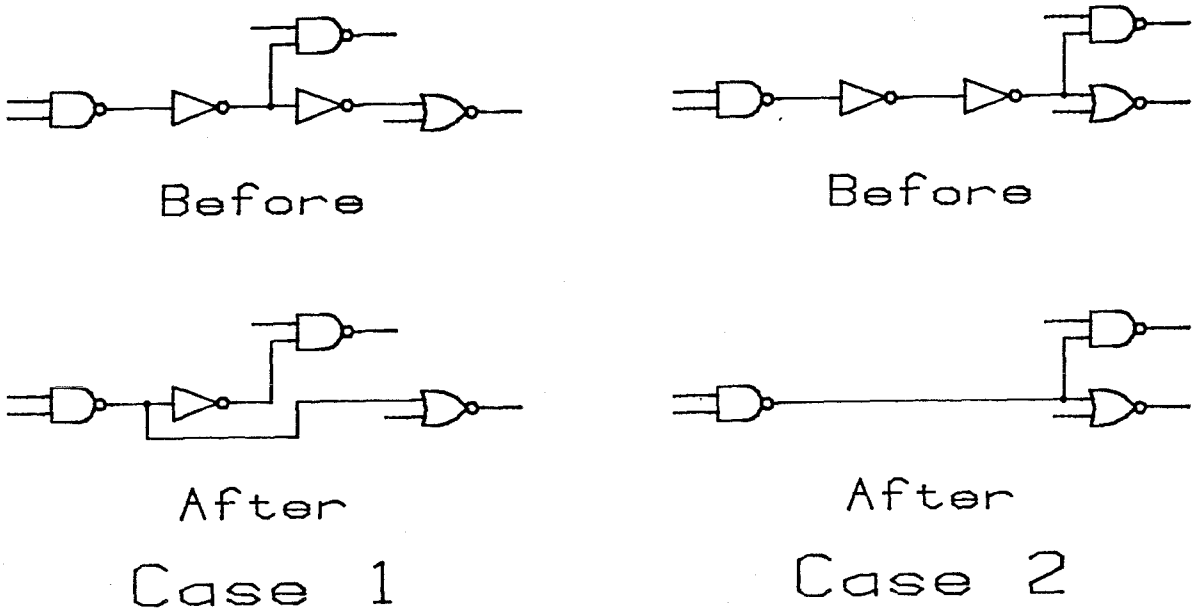


Fig. 5-4: Examples of Redundant Inverters

The GET_INVERT function is used to efficiently invert a signal. Figure 5-5 depicts the various conditions tested by GET_INVERT. In the first case, the inversion of a signal (marked by the '*') is required. The signal does not come from an INVERTER, and no INVERTERs connect to this signal. In this case, an INVERTER is added to the circuit and its output (marked by the '**') is returned. In the second case, the original signal does not come from an INVERTER, but an INVERTER does connect to this signal, in which case the output of the INVERTER is used. In the third case, the signal comes from an INVERTER and is used other places, in which case the input of the INVERTER is used. In the final case, the signal comes from an INVERTER, and the signal is not used in other gates, in which case the INVERTER can be eliminated and its input signal returned.

Given the GET_INVERT function, the REMOVE_INVERTERS function is straightforward: remove all INVERTERs from the chip and instead fuse the outputs

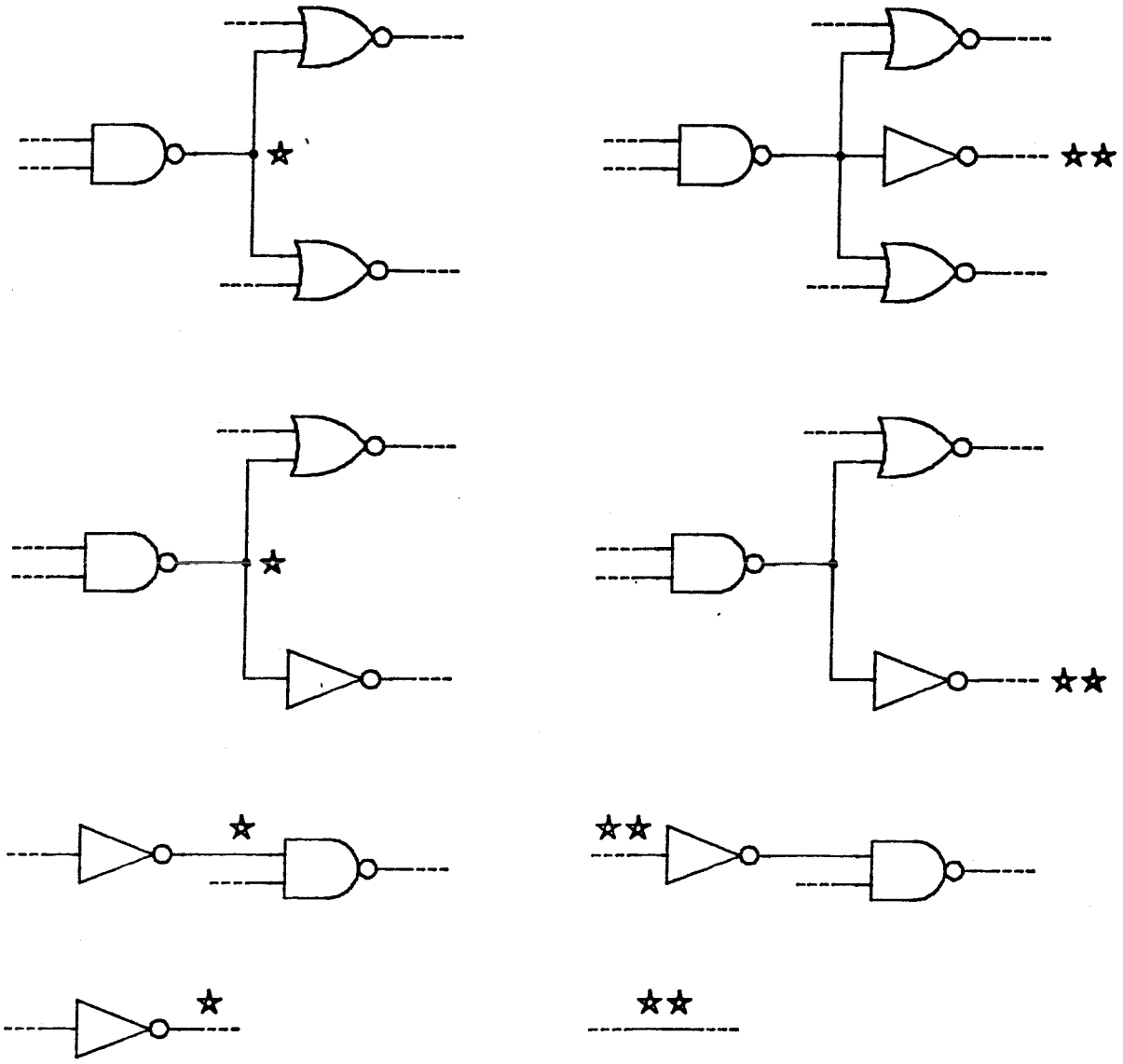


Fig. 5-5: Operation of GET INVERT

to the 'GET INVERT' of the input.

Other optimizers in the RLC remove redundant gates (for instance two NAND gates whose inputs are identical), attempt to replace NAND gates with NOR gates if the gate count would be reduced, and vice versa, and to merge NAND gates whenever possible. These optimizers presented so far look only at the logical specification of the chip and attempt to produce a more optimal logical specification by reducing the number of gates. Other optimizers look at wire lengths and gate loads to perform eletrical optimizations on the design. These optimizers to not change the functional

specification of the chip, merely the realization of that specification. This frees the designer from many of the design constraints while composing the functional specification of the chip.

5.4.2 Generators and Parsers

The input to the RLC is a data structure containing the functional specification of the chip. We have presented routines which allow the user to directly generate these data structures. On the other hand, we can write programs which generate these data structures for us. One such program might be a parser which accepts mathematical equations and produces proper RLC input for implementing those equations. With such a parser, our pulse synchronizer could be specified as follows.

```
DEFINE PULSE_SYNCHRONIZER(INPUTS:SET,RESET,CLOCK,MODE
                          OUTPUTS:OUT,COMP
                          LOCALS:ENABLE,X,Y):
    ENABLE= SET & (ENABLE & RESET)
    COMP= CLOCK & X
    X= (-CLOCK & ENABLE & Y) & (ENABLE & Y & X) & COMP
    Y= ENABLE & MODE & (COMP & Y)
    OUT= -COMP
ENDDEFN
```

The parser which accepts this mathematical notation is listed with the RLC compiler in appendix 3.

We might also write programs that generate the data structures for us. These programs specialize in the construction of certain classes of circuits. For instance, we might like a program that produces divide-by-n circuits. We would call the program, passing the divisor n, along with an input and output signal, and the program would generate the circuitry for the counter. The following code is in fact the program for producing divide-by-n logic.

```
DEFINE DFLOP(DATA,CLOCK,OUT,BAR:SIGNAL_WIRE):
    BEGIN   VAR X1,X2,X3,X4=SIGNAL_WIRE;
    X1:=NEW_SIGNAL;
    X2:=NEW_SIGNAL;
    X3:=NEW_SIGNAL;
    X4:=NEW_SIGNAL;
    SET(X1,NAND(DATA,X2));
    SET(X2,NAND((X4;X1;CLOCK)));
    SET(X3,NAND(X1,X4));
    SET(X4,NAND(X3,CLOCK));
```

```
    SET(OUT,NAND(X4,BAR));
    SET(BAR,NAND(X2,OUT));
    END
ENDDEFN

DEFINE COUNTER(N:INT  IN,OUT:SIGNAL_WIRE):
    BEGIN   VAR FW=FW;TOGGLE,NEXT,Q,QBAR,D=SIGNAL_WIRE;OUTPUT=SIGNAL_WIRES;
    OUTPUT:=NIL;
    FW:=N-1\FW;
    IF N<2 THEN HELP; FI
    WHILE FW<>L(0); DO
        Q:=NEW_SIGNAL;
        QBAR:=NEW_SIGNAL;
        D:=NEW_SIGNAL;
        IF DEFINED(OUTPUT) THEN
            NEXT:=NEW_SIGNAL;
            SET(NEXT,NOR(QBAR,INVERT(TOGGLE)));
            SET(D,NOR((OUT;NEXT;NOR(TOGGLE,Q))));
        ELSE
            NEXT:=Q;
            SET(D,NOR(OUT,Q)); FI
        DFLOP(D,IN,Q,QBAR);
        TOGGLE:=NEXT;
        OUTPUT::= IF FW BIT 0 THEN QBAR ELSE Q FI <$;
        FW:=FW SHIFTR 1;
    END
    SET(OUT,NOR(OUTPUT));
    END
ENDDEFN
```

The following input generates three dividers, with ratios of 5, 3, and 25.

```
NEW_CHIP;

COUNTER(5,'IN','FIVE');

COUNTER(3,'IN','THREE');

COUNTER(25,'IN','TWENTY-FIVE');

INPUT('IN');OUTPUT('FIVE');OUTPUT('THREE');OUTPUT('TWENTY-FIVE');

FINISH;
```

A plot of the layout is shown in figure 5-6. The layout has been transformed to fit the page better.

This technique of building procedures within the compiler to aid in the generation of the compiler input is very powerful. The user can build his own environment within the compiler. With a handful of routines similar to this, the user can quickly and easily design new chips or experiment with multiple implementations of a single chip.

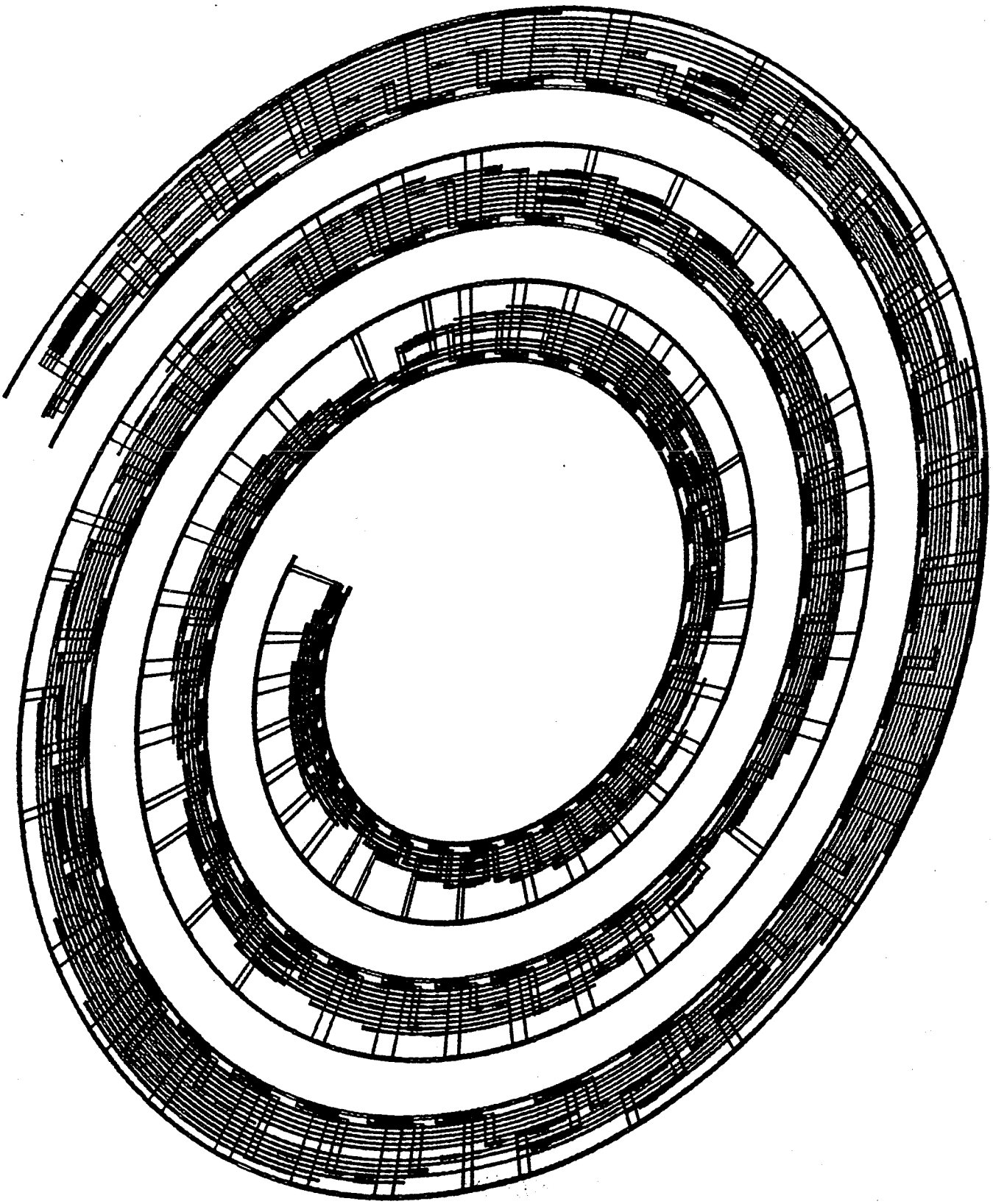
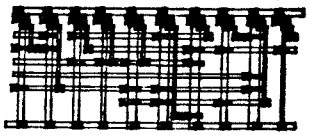


Fig. 5-6: Three Frequency Dividers (Transformed)

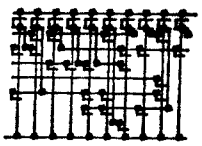
5.4.3 Examiners

Our chip specification is an abstract representation of the chip, containing only functional information. As such, it is not particularly tied to any technology or set of design rules. There are a very few routines which actually convert the data structure to physical layouts. The majority of the RLC code is independent of the physical implementation. Therefore, by modifying the few physical routines, we can generate output for a new technology.

This concept can easily be included in the RLC through the use of ICL's suspendable functions. A datatype TECHNOLOGY is defined which includes all of the technology dependent information. The user may generate several technology variables, which allow him to generate masks for any of these technologies. Figure 5-7 shows eight different implementations of the pulse synchronizer. Some of the 'technologies' are merely pictures, and not meant to be actual mask layouts.

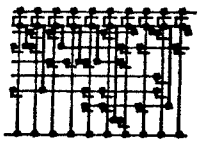


NMOS



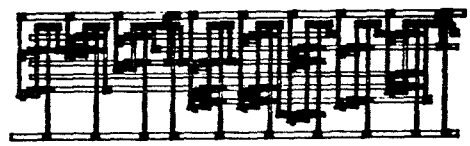
NMOS Sticks

Metal2 NMOS



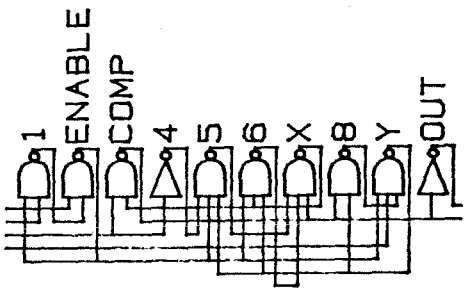
Metal2 Sticks

Fig. 5-7: Multiple Representations

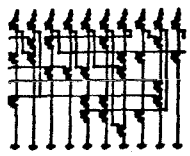


CMOS

Metal2 CMOS



Logical



Transistor

Fig. 5-7: Multiple Representations (cont.)

With this capability, the user may design a chip before the technology is available. When the technology is available, the masks can be generated. Also, if designs are archived by saving the data structure rather than the mask sets, the designs can be updated to new technologies quickly.

The user may also wish to simulate his circuit. Again, since we have an abstract representation of the circuit, it is a simple matter to simulate the chip. In RLC, we generate a new data structure from the chip specification data structure. This new data structure contains the information required to simulate the chip. The following input constructs the simulation data structure for the pulse synchronizer and plots the result of the simulation, as shown in figure 5-8.

```
MAKE_SIMULATOR;

CLOCK([PHASE:500  HIGH:1000  LOW:1000  VALUE:FALSE  INPUT:'CLOCK']);

WAVEFORM([VALUE:TRUE  DELTAS:{200;7000;8000;21000;22000}  INPUT:'RESET']);

WAVEFORM([VALUE:FALSE  DELTAS:{4000;5000;16000;17000;24000;25000}
          INPUT:'SET']);
```

```
WAVEFORM([VALUE:FALSE  DELTAS:[12000;26000]  INPUT:'MODE']);

RUN(30000);

Simulation terminated at time=30000.

PLOT([('CLOCK';'MODE';'SET';'RESET';'OUT';'COMP';'X';'Y';'ENABLE'),
     'Q'\AIF,.005);
```

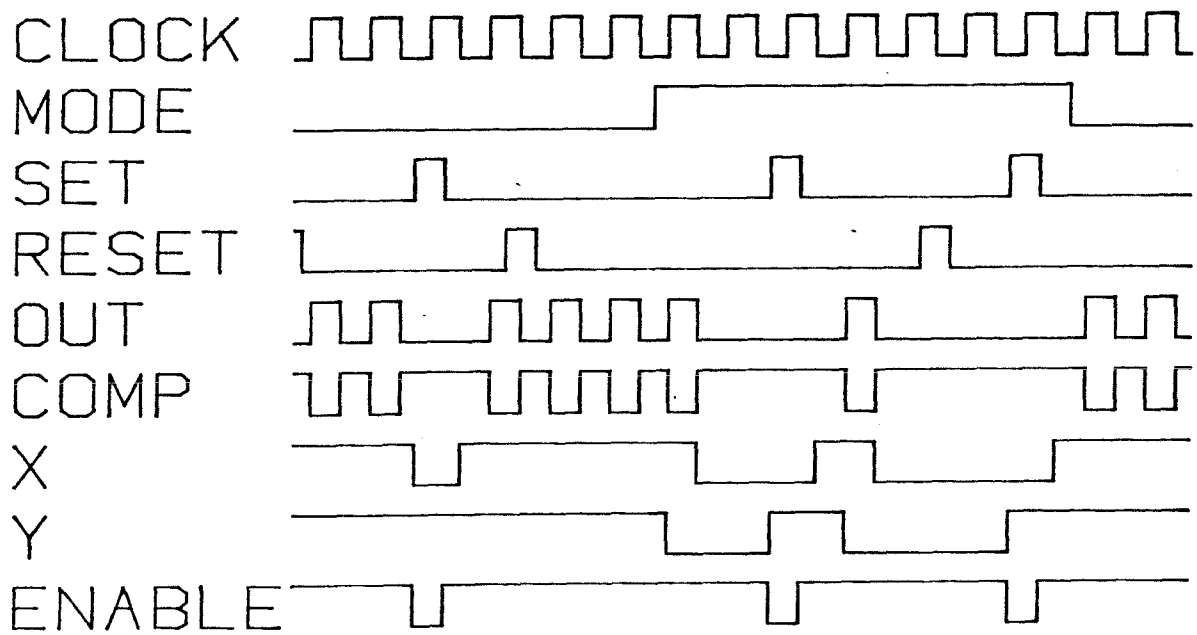


Fig. 5-8: Simulation Plot

A very important advantage of having the simulation driven from precisely the same chip description data structure is that we are guaranteed that the simulator is simulating the circuit that the layout generators produce. If the simulator required a different specification than the layout producers, the user would manually have to verify that the specifications matched (plus he would have twice as much typing to do).

5.5: Conclusions:

In this chapter we have seen the basics of a silicon compiler. The Random Logic Compiler is a very simple compiler, yet it illustrates the techniques and advantages of using silicon compilers.

Virtually the only disadvantage of using a silicon compiler is the restriction of the floorplan. The only chips that may be designed are those that fit the floorplan, and forcing a chip into a given floorplan may lead to inefficiencies. On the other hand, the floorplan aids the user in specifying his chip, and helps in the verification of the design. To ease the floorplan restrictions, several compilers will be designed, each one finely-tuned for generating one class of chip or portion of a chip.

One of the major advantages of using a silicon compiler is that the user can work in his own language. We have seen with the parsers that the user writes logic equations. Logic equations are natural to the user, and the functional specification is typically given in terms of logic equations. When the user completes the functional specification of the chip, the chip can be generated immediately.

With this rapid specification-to-layout cycle, the user can explore many of the design tradeoffs that would otherwise be impossible. When a decision must be made, the user can try several alternatives and quickly see the accurate cost of each possibility. This can dramatically shorten the functional design cycle, and the resulting chip can be significantly more optimal than a similar chip whose functional specification was virtually frozen before the physical layout was begun.

The user can extend the language. Every working group develops its own language for intercommunication. Similarly, software designers develop subroutine libraries for commonly used routines. In the same manner, users may extend the language of the silicon compiler, adding constructs and procedures which allow a more efficient communication of the chip specifications.

Compilers give us technology independence. Just as FORTRAN is available on many machines, and programs written in FORTRAN are portable between installations, silicon compilers allow designs to be portable across technologies. When the technology changes, the code generation routines are rewritten, but the user need never see the change. The old design specifications are still valid, and can quickly generate masks in the new technology.

The silicon compiler gives us three guarantees: there will be no design rule violations in the generated artwork, the circuit will correctly perform the specified function, and multiple representations of the circuit indeed represent the same

circuit. These capabilities and guarantees give the silicon compiler fantastic advantages over the traditional design techniques.

Part Two

Bristle Blocks

Chapter 6: Introduction to Bristle Blocks

As the cost of VLSI integrated circuit design increases, the desirability of automated circuit design programs grows. Previous automated circuit design systems have evolved from the TTL gate technology, and focus attention upon the logic equation specification of the design [5][9][10][11][23][26]. None of these tools have confronted the problem of generating efficient designs in the VLSI technology. In VLSI design, the communication network is the expensive portion of the design, whereas in TTL design the communication network is essentially free and the components are expensive. TTL design optimization focuses upon the reduction of the number of components at the expense of increased interconnections. Hence, TTL-based design systems yield undesirable results when applied to the design of VLSI circuits.

The Bristle Blocks system addresses the central issues of VLSI design. By adhering to a wiring strategy which optimizes communication, designs are generated which compare favorably with hand designs in terms of area and performance. This wiring strategy provides the framework for both the layout and the user's specification.

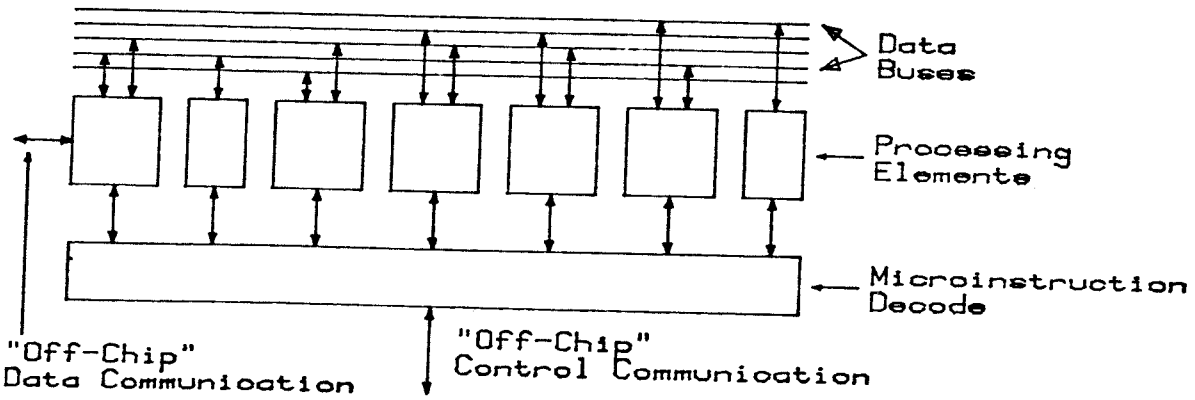


Fig. 6-1: Generalized Datapath Block Diagram

The wiring structure implemented in Bristle Blocks is that of a datapath, which supports Register Transfer (RT) operations. Figure 6-1 is the block diagram of a datapath. A datapath may consist of several data processing elements, such as Arithmetic/Logic Units (ALUs) and shifters, and storage nodes (registers or latches), interconnected by data buses. The datapath elements are controlled by a microcontrol word decoder. The microcontrol word is an arbitrarily long series of

binary logic values which describe the current operation of the datapath. Portions of the microcontrol word may be driven by datapath elements, while the remainder of the logic value sources are external to the datapath. Given a list of data processing elements and a behavioral description of the register transfer operations to be performed, Bristle Blocks will compile a datapath and control logic layout which implements those operations.

For any preliminary specification of a chip, there may be many structures which can be used to implement the specifications. The datapath structure is one which can be used to implement a variety of functions. In chapter 9 we see examples of pipelined chips, signal processing chips, general purpose computing chips, and special application chips implemented in Bristle Blocks.

Although general purpose in nature, restrictions are imposed upon the designs by the physical floorplan and the logical and temporal schema of Bristle Blocks. One restriction is that all of the data processing elements be of the same width. This means that all registers and ALUs, for instance, contain the same number of bits. Another major restriction is that complex instruction sequencing is implemented in a very inefficient manner.

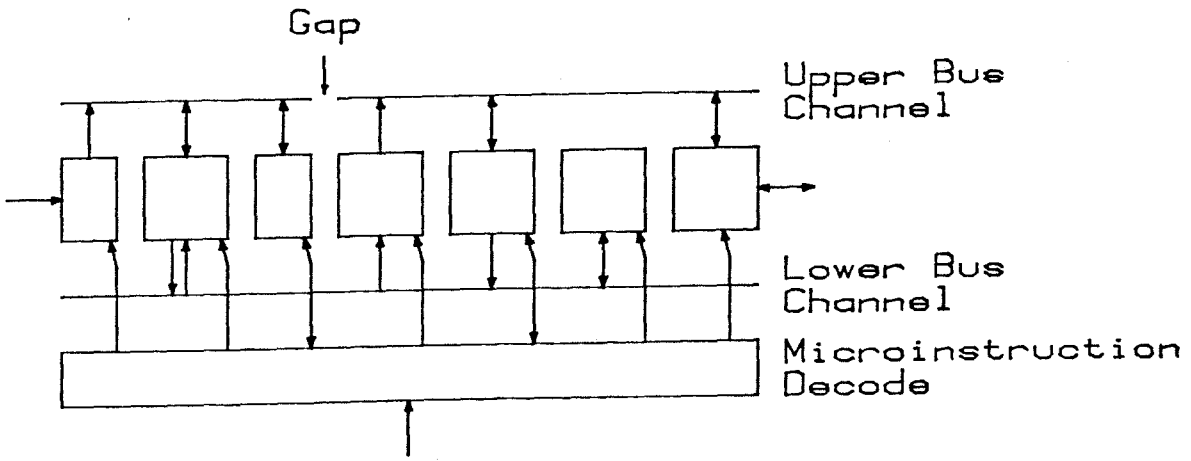


Fig. 6-2: Bristle Blocks Logical Floorplan

The logical block diagram of Bristle Blocks is shown in figure 6-2. There is a single row of data processing elements with a limit of two data buses running past any element. There can be more than two data buses on a chip by placing a gap in one of the two busing channels. The two busing channels are refered to as the 'Upper Bus

Channel' and the 'Lower Bus Channel', and the buses in those channels are referred to as the 'Upper Bus' and the 'Lower Bus'. These two buses are designed into each of the data processing elements, which does limit the number of buses in the system. However, by designing these buses into the cells rather than externally routing the bus wires, considerable chip area is saved.

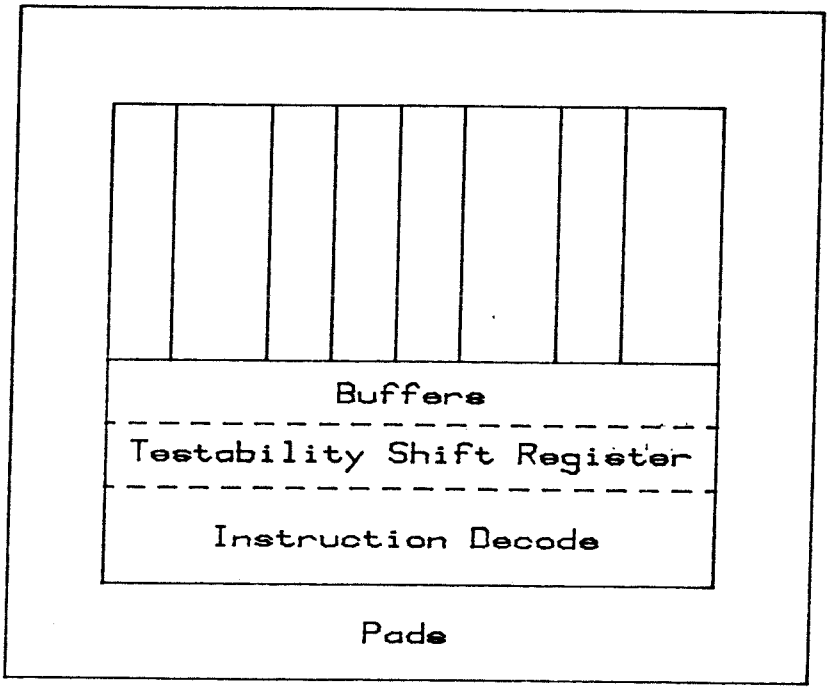


Fig. 6-3: Bristle Blocks Physical Floorplan

The physical floorplan of Bristle Blocks is very similar to the logical block diagram. The physical floorplan is shown in figure 6-3. The datapath elements are horizontally abuted in the order they are encountered in the user's specification. The buffers, testability shift register, and the instruction decoder are placed below the datapath core. Finally, pads are placed around the perimeter of the chip.

Bristle Blocks uses the two-phase clocking scheme presented in Mead and Conway [20]. Each of the data buses transfers data from the source register to the destination register(s) when the PHI 1 clock is high. To improve the performance of the chip, these buses are precharged during PHI 2, so that the source registers need only pull appropriate bus lines low. If the registers are asked to refresh their internal values, refreshing will occur during PHI 2. The processing elements have

the opposite timing conventions. The carry chains and other internal nodes are precharged during PHI 1, and the computations occur during PHI 2. Output registers are loaded during PHI 2. With this timing scheme, data can be transfered into an ALU's input registers and the ALU can load its output register in one PHI 1 - PHI 2 clock cycle.

The control line buffers isolate the instruction decoding from the datapath core control lines. Each buffer samples an instruction decoder output during one clock phase, and drives its control line on the opposite clock phase. This delay allows the instruction decoder and datapath core to operate in parallel, and eliminates race conditions in the instruction decoder. The bus transfer controls, which are active during PHI 1, are driven by microcode conditions existing during the previous PHI 2. Similarly, all ALU operations are decoded during PHI 1 and are then performed the during the next PHI 2.

When both system clocks are low, the control line buffers dynamically latch the values which will drive each control line. If the two testability clocks are strobed, each buffer will transfer its value to its righthand neighbor. The leftmost buffer gets its new value from the testability input pad, and the rightmost buffer transfers its value to the testability output pad. By repeatedly strobing the two testability clocks, the user can examine the state of each control line buffer's latch, and can set each latch to new values. The instruction decoder can be tested by examining the state of the testability vector, and the datapath core can be tested by setting the testability vector to specific values and observing the results.

The remaining chapters describe Bristle Blocks in greater detail. Chapter 7 documents the input specifications accepted by the parser. Chapter 8 describes how Bristle Blocks generates a layout from a specification. Chapter 9 presents several examples of chips compiled by Bristle Blocks. Finally, Chapter 10 presents the history of Bristle Blocks, and proposes a new Bristle Blocks system.

Chapter 7: The Bristle Blocks Input Language

The Bristle Blocks Input Language is a formal language which allows for the specification of datapath chips. There are four pieces of information needed by Bristle Blocks to compile a chip: the name of the chip, the width of the datapath, the data processing elements needed in the datapath, and the structure of the microcontrol word. The name is used to identify the datapath, since many datapaths may reside in the system at any one time. The datapath width is required, since Bristle Blocks can generate datapaths of arbitrary width. In fact, many times the difference between a 16-bit chip specification and a 32-bit chip specification is only this single number. The microcontrol word is described to facilitate the specification of element operations. The data processing elements are listed in the order they are to appear in the final layout. As these elements are listed, parameter values are given which define how each element is to behave.

The input parser for Bristle Blocks converts all lower case letters to upper case, so the input may be typed in either style. All examples presented here will use strictly upper case to improve the readability of the text. The parser recognizes the following tokens:

<ID> Identifiers, which are a single letter followed by an arbitrarily long sequence of letters, digits, or underscores. Examples: A Hi_There x49 R2D2

<MASK> Masks, which are composed of X, I, and 0 characters. These are used to indicate which bits in the datapath are to be operated upon. The number of characters in the mask must be equal to the datapath width. Examples for 8-bit wide datapaths: XII00IXX oooooiii XioxiOIx

<INT> Integers, which are composed of an arbitrarily long, non-empty set of digits. Examples:
1 32424134 0080

<BLANK> Blank characters. All spaces, tabs, carriage return, and line-feed tokens are ignored by the parser.

<OTHER> Any other character. Any character which can not be interpreted as a token by the above definitions becomes a token of this type. Examples: (+ .

The following rules state the syntax for a Bristle Blocks input file.

```
<CHIP>          :::=    <NAME> <BODY> END

<NAME>          :::=    NAME <ID> <INT> ;

<BODY>          ::=     <DECLARATION>
<BODY>          ::=     <BODY> <DECLARATION>
```

These rules state that a <CHIP>, which is the grammar accepted by Bristle Blocks, is composed of a <NAME>, followed by a <BODY>, followed by the token 'END'. A <NAME> is the token 'NAME', followed by an <ID>, followed by an <INT>, followed by the token ';'. A <BODY> is either a single <DECLARATION>, or it is a <BODY> followed by a single <DECLARATION>. This recursive definition for <BODY> states that a <BODY> can be any arbitrarily long, non-empty set of <DECLARATION>s. An example of a <CHIP> might be

```
NAME SAMPLE 8;
...
END
```

where we have represented the <BODY> by '...'. The <ID> in the <NAME> is the name of the chip, while the <INT> is the width of the datapath. We can see here that the name of the chip is SAMPLE and that the datapath is 8 bits wide.

The <DECLARATION>s are specifications of datapath elements, and the description of the microcontrol word. The following sections define the syntax and semantics of <DECLARATION>s.

7.1: Field Declarations

To specify the functioning of a datapath element, the user must be able to state microcode conditions associated with each operation of the element. For example, if the element is to increment an internal value, the user must state when the incrementation is to occur. This is done by describing the states of the microcode inputs which should cause this operation to occur. This microcode condition specification is called an EQUATION. The user therefore gives the EQUATIONs associated with the elements' functions when specifying the datapath.

To facilitate the specification of these EQUATIONs, the microcode inputs, or control word, can be broken into FIELDs, so that the EQUATIONs become pairs of FIELDs with associated values. When the microcode inputs corresponding to each FIELD have the associated value, the EQUATION becomes TRUE, and the element performs the desired operation.

These FIELDs are described in declarations, using the following syntax:

```
<DECLARATION>    :::=    FIELD <FIELD_DECL>

<FIELD_DECL>     ::=     <FIELD_SPEC> , <FIELD_DECL>
<FIELD_DECL>     ::=     <FIELD_SPEC> ;
```

Informally, these rules state that fields are declared by the keyword 'FIELD' followed by an arbitrary, non-empty set of field specifications, each separated by commas, followed by a semi-colon. Field declarations may occur anywhere in the datapath specification, but the fields must be declared before they are used.

One form of a field specification is the field name followed by numbers indicating which bits of the microcontrol word compose the field. For instance, a field specification might be

```
REG_SELECT<1,3,21>
```

This specification has declared a new field, named REG_SELECT, which is bits 1, 3, and 21 of the microcontrol word. In most instances, fields contain contiguous bits, so a shorthand can be used: if two of the integers in the list of bits are separated by a colon instead of a comma, all of the integers between and including these two integers are included in the list. Therefore, the following two specifications are identical:

```
ALU_OP<1,2,3,4,5>
ALU_OP<1:5>
```

Bits can not be repeated in a single field. Therefore, this specification is in error:

```
SHIFT_CONST<1,2,1,3,2>
```

On the other hand, using the short hand notation, if the second integer equals the first, no error occurs:

```
A_SOURCE<3,7,9:9>
```

Fields may have bits in common. For instance, the following three fields all share bits 3, 4, and 5 of the microcontrol word, but notice that the third field uses the bits in reverse order:

```
FIELD_1<1:5>,FIELD_2<3:7>,FIELD_3<8,5:3>
```

To aid the use of macros in the field specifications, simple arithmetic operations upon the integers in the bit specifications is needed. Therefore, each of the integers in the bit specifications can be replaced by a simple equation involving addition and subtraction.

```
FIELD_X<1,4-2:7+3-5>
```

At times, one would like to describe a field not as a collection of specific microcontrol word bits, but rather as a subfield of a previously declared field. This can be specified as follows:

```
FIELD FIELD_A<3,5,2,8,4,7>,FIELD_B=FIELD_A<4:2>;
```

Here, FIELD_A is declared to be six randomly ordered bits in the microcontrol word. FIELD_B is bits 4 through 2 of FIELD_A, which corresponds to bits <8,2,5> of the microcontrol word. Additionally, one might like to specify a field which is a concatenation of existing fields. This is done as follows

```
FIELD  A<2:4>,B<8:6>,C= A & B<2>;
```

Here, A is bits 2, 3, and 4, while B is bits 8, 7, and 6. Field C contains all the bits of A and the second bit of B, so C contains bits 2, 3, 4, and 7. One final word about field specifications: each field name must be an identifier, which is a letter followed by an arbitrary string of letters, digits, and underscores. These rules concerning field specifications can be summed up in the following syntax rules:

```
<BITSPEC>        ::=     < <BITSPEC1>

<BITSPEC1>       ::=     <INTSPEC> : <INTSPEC> <BITSPEC2>
<BITSPEC1>       ::=     <INTSPEC> <BITSPEC2>

<BITSPEC2>       ::=     , <BITSPEC1>
<BITSPEC2>       ::=     >

<FIELD1>         ::=     <ID>
<FIELD1>         ::=     <ID> <BITSPEC>

<FIELD_LIST>     ::=     = <FIELD1>
<FIELD_LIST>     ::=     <FIELD_LIST> & <FIELD1>

<FIELD_SPEC>     ::=     <ID> <BITSPEC>
<FIELD_SPEC>     ::=     <ID> <FIELD_LIST>

<INTSPEC>        ::=     <INT>
<INTSPEC>        ::=     <INTSPEC> + <INT>
<INTSPEC>        ::=     <INTSPEC> - <INT>
```

7.2: Microcode Equations

To specify the operations for many of the datapath elements, the user declares EQUATIONs, which associate values with fields. When the microcontrol words associated with the fields have the specified value, the EQUATION is TRUE, and the datapath element performs its operation.

The syntax for EQUATIONs can be summarized by the following rules.

```
<EQUATION>      ::=     ALWAYS
<EQUATION>      ::=     <EQUATION1>
<EQUATION>      ::=     GND
<EQUATION>      ::=     NEVER
<EQUATION>      ::=     PAD
<EQUATION>      ::=     VDD

<EQUATION1>     ::=     <EQUATION1> OR <EQUATION2>
<EQUATION1>     ::=     <EQUATION2>

<EQUATION2>     ::=     <EQUATION2> AND <EQUATION3>
<EQUATION2>     ::=     <EQUATION3>

<EQUATION3>     ::=     ( <EQUATION1> )
<EQUATION3>     ::=     <ID> = <BITS>
<EQUATION3>     ::=     IF <EQUATION1> THEN <EQUATION1>
                                       ELSE <EQUATION1> FI
<EQUATION3>     ::=     NOT ( <EQUATION1> )

<BITS>          ::=     <BIT>
<BITS>          ::=     <BITS> <BIT>

<BIT>           ::=     I
<BIT>           ::=     O
<BIT>           ::=     X
```

In the simplest case, the EQUATION would state that a single field have one specific value. Given the field declaration

```
FIELD   SELECT<1:3>,ENABLE<4:5>,OP<6:8>;
```

an EQUATION might be

```
SELECT=IXO
```

This states that the first bit of SELECT should be high and the third bit should be low. The state of the second bit of SELECT does not matter. Notice that the high and low specifications are the letters I and O, not the digits 1 and 0. The SELECT field is three bits long, therefore the value to be associated with that field must be three bits long.

A more general equation might state that several fields have fixed values. Given the field declaration from above, the following example shows use of the AND function.

```
SELECT=IXO AND ENABLE=XI
```

Here we require the second bit of the ENABLE field to be high in addition to the value required in the SELECT field. The AND function is practically free in terms of

chip area, so the use of AND is welcomed and encouraged.

To allow more than one value to be associated with a field, an OR function is required. If we had written the equation as

```
SELECT=IXO OR ENABLE=XI
```

then the equation would be TRUE when either SELECT=IXO or ENABLE=XI or both. The OR function does cost some area in the instruction decoder, so some care should be exercised in its use. The OR functions will apply after all of the AND functions: we say that OR has a lower precedence than AND. Therefore,

```
SELECT=IXX AND ENABLE=XI OR SELECT=XXO AND ENABLE=OX
```

will group as

```
(SELECT=IXX AND ENABLE=XI) OR (SELECT=XXO AND ENABLE=OX)
```

rather than as

```
SELECT=IXX AND (ENABLE=XI OR SELECT=XXO) AND ENABLE=OX
```

To get the second grouping, the parentheses must be used.

To invert the polarity of an equation, the NOT function is used. The following equation is TRUE unless SELECT=IXX and ENABLE=XI.

```
NOT( SELECT=IXX AND ENABLE=XI )
```

The parenthesis are required. Notice that the following two specifications are not equivalent.

```
NOT( SELECT=100 )
SELECT=011
```

The first equation will go TRUE if SELECT<1> is low OR if SELECT<2> is high OR if SELECT<3> is high, whereas the second equation will go TRUE only when SELECT<1> is low AND SELECT<2> is high AND SELECT<3> is high.

Other equations can use IF...THEN...ELSE...FI constructs. One might say

```
IF SELECT=IXO THEN ENABLE=XI AND OP=OIO ELSE OP=IXX FI
```

This equation is TRUE if SELECT=IXO and ENABLE=XI and OP=OIO or if SELECT<>IXO and OP=IXX. Each of the IF, THEN, and ELSE clauses may be any of the equations specified up to this point, including other IF...THEN...ELSE...FIs. One caution, however: The IF...THEN...ELSE...FI equation can take a relatively large area in the instruction decoder. One should not include equations of this form with reckless abandon.

Each of the equation constructs presented so far deal with variable equations, equations that depend on microcode inputs. Other equations may have fixed values, such as always being low. Fixed equations may have one of five values: ALWAYS, NEVER, VDD, GND, and PAD. In the ALWAYS case, the equation will always be TRUE; in the NEVER case, the equation will always be FALSE. In the VDD and GND cases, the control line is tied directly to the appropriate power line. In the PAD case, a pad will be added to the chip, and this control line will be the sole signal which depends upon that pad's value.

7.3: Parameters

The datapath elements are parametrized cells. They consume parameters specifying the configuration required for the particular instance of the cell and produce the corresponding layout. There are several kinds of parameters used in the Bristle Blocks cells. The first form of parameter is an EQUATION, where the equation specifies when a certain operation should occur. Another type of parameter is a REGISTER_SPECIFICATION, which describes a register, for example, the input register for an incrementer. A third parameter is an integer. For Bristle Blocks, integers are restricted to positive, usually non-zero values. A fourth kind of parameter is a FIELD, which might indicate a shift constant, for instance. Another parameter type is an OUTPUT, which is used to drive a signal from a datapath element to either an output pad or into the instruction decoder. A sixth parameter type is a MASK, which is used to specify which bits in the datapath are being operated upon. A DECODE parameter is used to decode a field into one of many instructions. Finally, SOURCE and DESTINATION parameters are used to connect bits

from the datapath to bits in the instruction decoder. Each of these parameter types will be discussed in more detail, with examples.

There is a uniform syntax for specifying each of the elements in the datapath. The first token is an identifier specifying the class of the element, and the second token is always an identifier which is the name of that element. For example,

```
REGISTER PC ..... ;
ALU ALU ...;
```

Here we have a REGISTER named PC and an ALU named ALU. Following the name is a list of keywords and parameter values. The keywords are a function of the element class. REGISTERs have one set keywords, while the ALU has a different set. Some of the parameters are required, others are optional. The cell documentation lists the parameter keywords, types, and requirement status for each of the element classes. The following rules define the syntax for calling a datapath element:

```
<DECLARATION>    ::=    <ID> <ID> ;
<DECLARATION>    ::=    <ID> <ID> <PARAMS>

<PARAM>          ::=    <ID> : <DECODE>
<PARAM>          ::=    <ID> : <DESTS>
<PARAM>          ::=    <ID> : <EQUATION>
<PARAM>          ::=    <ID> : <ID>
<PARAM>          ::=    <ID> : <INT>
<PARAM>          ::=    <ID> : <MASK>
<PARAM>          ::=    <ID> : <OUT>
<PARAM>          ::=    <ID> : <REG_SPEC>
<PARAM>          ::=    <ID> : <SOURCES>
<PARAM>          ::=    <ID> : <VAR_EQUATION>

<PARAMS>         ::=    <PARAM> , <PARAMS>
<PARAMS>         ::=    <PARAM> ;
```

7.3.1: Equations

One of the Bristle Blocks elements is a bus precharge unit. This cell will precharge the upper data bus when its PRECHARGE parameter is high. The PRECHARGE parameter is an EQUATION, but the parameter is optional. If the user does not specify the parameter value, the cell will use a default value which always precharges the bus. The documentation of the cell reflects these characteristics:

```
Element: PRECHARGE_UPPER
Required Parameters: NONE
Optional Parameters:
    Keyword: PRECHARGE     Type: EQUATION     Default: ALWAYS
```

The type of the element is PRECHARGE_UPPER. There are no required parameters and one optional parameter, which is of type EQUATION. The default value for the parameter is ALWAYS. One might use this element as follows.

```
FIELD PCHG<1>;

PRECHARGE_UPPER    CELL_TO_PRECHARGE_UPPER_BUS    PRECHARGE:PCHG=1;
```

The element is of type PRECHARGE_UPPER. The name of this particular upper-bus-precharger is CELL_TO_PRECHARGE_UPPER_BUS. The one and only parameter for this cell has the keyword PRECHARGE. The user has specified that the bus is to precharge whenever the PCHG field is high. The following code uses the default value for the PRECHARGE parameter:

```
PRECHARGE_UPPER    CELL_TO_PRECHARGE_UPPER_BUS;
```

7.3.2: Register Specifications

A second common parameter type is REGISTER_SPECIFICATION, or REG_SPEC. A REG_SPEC describes a register that can be used as an input or output register of a datapath element. For example, an ADDER has two input registers and an output register. The user specifies how the register should interface to the data buses. Equations may be given to control the reading or writing of the two buses. Additionally, the register can be made to refresh its internal value, or load with a predetermined (fixed) constant. The syntax of a REG_SPEC is

```
<REG_SPEC>       ::=     <REG_SPEC1> ]

<REG_SPEC1>      ::=     <REG_SPEC1> , <ID> : <REG_VAL>
<REG_SPEC1>      ::=     [ <ID> : <REG_VAL>

<REG_VAL>        ::=     <EQUATION>
<REG_VAL>        ::=     <MASK>
```

The keywords for a REG_SPEC are READ_UPPER, READ_LOWER, WRITE_UPPER, WRITE_LOWER, REFRESH, SUGGEST, and VALUE. These are all EQUATIONs except VALUE, which is a MASK. When SUGGEST is TRUE, the VALUE is loaded into the register (Xs in the mask indicate bits of the register that are not modified by the suggest operation). For example,

```
NAME EXAMPLE 8;

FIELD   REG_OP<1:3>;

.... [READ_UPPER: REG_OP=100,
      WRITE_UPPER:REG_OP=101,
      READ_LOWER: REG_OP=110,
      WRITE_LOWER:REG_OP=111,
      REFRESH:    ALWAYS,
      SUGGEST:    REG_OP=011,
      VALUE:      X11X00XX]....
```

When the REG_OP field is 100, this register will take the value from the upper bus and store it in its internal node. When REG_OP=101, the register drives the upper bus with the data contained in its internal node. Similar functions occur with the lower bus. The register refreshes its internal value every cycle. When REG_OP=011, the second and third bits of the register are set high, while the fifth and sixth bits are set low. The remaining bits are not modified. All of the parameters in the REG_SPEC are optional. Also, none of the read or suggest equations should be TRUE when either of the write equations are TRUE, because the data buses could be loaded with garbage. Unfortunately, the compiler can not verify that these equations are exclusive, due to the fact that the various register equations may be driven by independent sets of control bits. The correctness of these equations must be insured in the software.

To illustrate the use of a REG_SPEC, consider an INCREMENTER. The documentation for an INCREMENTER is

```
Element: INCREMENTER
Required Parameters:
    Keyword: INPUT_REGISTER     Type: REGISTER
    Keyword: LOAD               Type: EQUATION
Optional Parameters:
    Keyword: OUTPUT_REGISTER    Type: REGISTER
    Keyword: PRECHARGE          Type: EQUATION    Default: ALWAYS
    Keyword: CARRY_OUT          Type: OUTPUT
```

The INCREMENTER takes the data from its input register, adds one to this value, and stores it in the output register when the LOAD equation is true. If the OUTPUT_REGISTER parameter is not specified, the INCREMENTER will store the value into its input register. The following code shows two incrementers, INC1 and INC2. INC1 has only a single register; INC2 has separate input and output registers.

```
NAME INCREMENTER_EXAMPLE 8;

FIELD RESET<1>,OP<2:3>;

INCREMENTER INC1
    INPUT_REGISTER: [ WRITE_UPPER: OP=OI,
                      SUGGEST: RESET=I,
                      VALUE: OOOOOOOO ],
    LOAD: ALWAYS;

INCREMENTER INC2
    INPUT_REGISTER: [ READ_UPPER: OP=XI,
                      REFRESH: ALWAYS,
                      SUGGEST: OP=IO,
                      VALUE: OOOOXXXX ],
    OUTPUT_REGISTER: [ WRITE_UPPER: OP=II ],
    LOAD: ALWAYS;

PRECHARGE_BOTH  PCHG;

END
```

When RESET is high, the first incrementer clears its value. The value in this first incrementer is incremented every cycle. When the OP field equals OI, INC1 writes its value onto the upper bus.

The second incrementer always increments its input value and stores it in its output register. When the OP field is OO, the input register for INC2 does not load with a new value, so effectively no operation is done by INC2. When the OP field is OI, the input register is reading from the bus, while INC1 is writing to the bus, so this operation is a transfer from INC1 to INC2. When OP is IO, the input register suggests, but only the four most significant bits are altered: they are cleared. When OP is II, the input register is also reading from the upper bus, but the output register is writing to the bus, so this operation transfers data from the output of INC2 back to the input.

7.3.3: Integers

The third parameter type is that of Integer. Integers in Bristle Blocks must be positive, and usually must be non-zero, although they may have leading zeros. An element which takes an integer as a parameter is the STACK element. The documentation for a STACK is

```
Element: STACK
Required Parameters:
    Keyword: DEPTH      Type: INTEGER
    Keyword: TOP        Type: REGISTER
    Keyword: POP        Type: EQUATION
    Keyword: PUSH       Type: EQUATION
Optional Parameters:
    Keyword: MIDDLE     Type: REGISTER    Default: [REFRESH:ALWAYS]
    Keyword: BOTTOM     Type: REGISTER    Default: [REFRESH:ALWAYS]
    Keyword: REFRESH    Type: EQUATION    Default: ALWAYS
```

The STACK is implemented as a TOP register followed by DEPTH-1 MIDDLE registers, followed by a BOTTOM register. Between adjacent pairs of registers lie circuitry for transfering data between the registers. When the PUSH equation is TRUE, data in the TOP register transfers into the first MIDDLE register as data from the first MIDDLE register is transfered into the second MIDDLE register, etc. The POP control performs the inverse operation. The following STACK has depth 6:

```
NAME STACK_TEST_1 8;

FIELD OP<1:2>;

STACK SAMPLE_STACK
    DEPTH: 6,
    PUSH:  OP=IO,
    POP:   OP=II,
    TOP:   [ READUPPER: OP=XO,
             WRITEUPPER:OP=OI,
             REFRESH: ALWAYS];

PRECHARGE_BOTH  PCHG;

END
```

When OP=OO, the TOP register reads data from the upper bus, overwriting what used to be on the top of the stack. When OP=OI, the data on the top of the stack writes to the upper bus, but the stack does not POP. When OP=IO, the stack does a PUSH, and the register loads from the bus. When OP=II, the stack POPs, but the TOP register does not write to the upper bus. The stack can not perform a POP operation on the same cycle that the register is writing to a bus because the bus will be written with garbage. It is ok to read from a bus while the stack is doing a PUSH, however. Also, the stack should not do both a PUSH and a POP at the same time, unless the depth of the stack is 1. For longer stacks, registers in the middle of the stack would be loaded from their two neighbor registers at the same time, so garbage would appear in these registers. For a stack of depth 1, however, there are only two registers (the TOP and the BOTTOM registers), so a simultaneous PUSH and POP will do a swap of the two register values, as illustrated in the following

example.

```
NAME STACK_TEST_2 8;

FIELD OP<1:4>;

STACK SWAPPER
    TOP:     [READ_UPPER: OP=1OXX,
              WRITE_UPPER: OP=11XX,
              REFRESH:ALWAYS,
              SUGGEST: OP=1111,
              VALUE:00000000],
    BOTTOM:  [READ_UPPER: OP=XX10,
              WRITE_UPPER: OP=XX11 AND NOT(OP=11XX),
              REFRESH:ALWAYS,
              SUGGEST: OP=1111,
              VALUE:00000000],
    PUSH: OP=XX01,
    POP:  OP=01XX,
    DEPTH:1;

PRECHARGE_BOTH  PCHG;

END
```

The following table lists the operations performed by this stack.

OP	Operation	OP	Operation
0000:	No Change	1000:	Load TOP from bus
0001:	Copy from TOP to BOTTOM	1001:	Push into TOP
0010:	Load BOTTOM from bus	1010:	Load both TOP and BOTTOM
0011:	Read BOTTOM to bus	1011:	BOTTOM goes to TOP and bus
0100:	Copy from BOTTOM to TOP	1100:	Read TOP to bus
0101:	Swap TOP and BOTTOM	1101:	TOP goes to BOTTOM and bus
0110:	Push into BOTTOM	1110:	TOP goes to BOTTOM and bus
0111:	BOTTOM goes to TOP and bus	1111:	Clear TOP and BOTTOM

7.3.4: Fields

Parameters of type FIELD are used to specify shift constants or bit selects. For example, a 16-bit datapath may have a shifter capable of shifting data left from 0 to 15 places in one cycle. A 4-bit field can specify the size of the shift in this case. For a 32-bit datapath, however, the shifter can shift between 0 and 31 places in one cycle, which requires a 5-bit field to specify the shift constant. The SIMPLE_SHIFTER element is one example of an element which requires a field to supply the shift constant. Documentation for the SIMPLE_SHIFTER is

```
Element: SIMPLE_SHIFTER
Required Parameters:
    Keyword: MOST_SIGNIFICANT_WORD      Type: REGISTER
    Keyword: LEAST_SIGNIFICANT_WORD     Type: REGISTER
    Keyword: OUTPUT_REGISTER            Type: REGISTER
    Keyword: SHIFT_CONSTANT             Type: FIELD
    Keyword: LOAD                       Type: EQUATION
Optional Parameters: NONE
```

One might use this shifter as follows.

```
NAME SHIFT_TEST 16;

FIELD REG_SELECT<1:3>,SHIFT_CONST<4:7>;

SIMPLE_SHIFTER SHIFTER
    LOAD: ALWAYS,
    SHIFT_CONSTANT: SHIFT_CONST,
    OUTPUT_REGISTER: [WRITE_UPPER: REG_SELECT=1XX],
    MOST_SIGNIFICANT_WORD: [READ_UPPER:REG_SELECT=XX1, REFRESH:ALWAYS],
    LEAST_SIGNIFICANT_WORD: [READ_UPPER:REG_SELECT=XX1, REFRESH:ALWAYS];

PRECHARGE_BOTH  PCHG;

END
```

7.3.5: Outputs

Signals like the carry output of an adder come from the datapath, and may go either to pads or into the instruction decoder. If the signal goes to a pad, Bristle Blocks will add an output pad to the chip and connect the pad to the control wire. If the signal goes to the instruction decoder, it is treated like any other microcontrol word bit, and so can modify the operation of the datapath. The syntax for specifying the operation of an output is

```
<OUT>          ::=     <ID>
<OUT>          ::=     <ID> BIT <INT>
<OUT>          ::=     PAD
<OUT>          ::=     UNUSED
```

In the first case, the output specification is a field name. The control line from the datapath element will drive the first bit of the field. In the second case, a field name and an index are given. The index indicates which bit of the field will be driven by the datapath element. The specification of 'PAD' states that the control line should connect to an output pad. The 'UNUSED' option indicates that the control line should not connect to anything. This is equivalent to not specifying the parameter. In the register example, the incrementer element was seen to have a parameter with keyword CARRY_OUT. This parameter is of type output.

Augmenting the register example to include connection of the carry output signal to a pad, we get the following code:

```
NAME OUTPUT_EXAMPLE 8;

FIELD RESET<1>,OP<2:3>;

INCREMENTER INC1
    INPUT_REGISTER: [ WRITE_UPPER: OP=OI,
                      SUGGEST: RESET=I,
                      VALUE: OOOOOOOO ],
    LOAD: ALWAYS,
    CARRY_OUT: PAD;

INCREMENTER INC2
    INPUT_REGISTER: [ READ_UPPER: OP=XI,
                      REFRESH: ALWAYS,
                      SUGGEST: OP=IO,
                      VALUE: OOOOXXXX ],
    OUTPUT_REGISTER: [ WRITE_UPPER: OP=II ],
    LOAD: ALWAYS
    CARRY_OUT: PAD;

PRECHARGE_BOTH  PCHG;

END
```

Each of the incrementers' carry outputs will go to pads.

7.3.6: Masks

MASKs are used to indicate which bits of the datapath are to be affected by a particular operation. Recall in the register example that one of the incrementers' input register had a suggest value of OOOOXXXX. This indicates that the four most significant bits should be set low, which the four least significant bits were to be left unchanged. Notice that the length of the MASK is required to be the same as the width of the datapath, since each character in the MASK represents one bit in the datapath. The first bit in the MASK is associated with the most significant bit in the datapath, while the last bit in the MASK is associated with the least significant bit.

7.3.7: Variable Timing Equations

For almost every control line in Bristle Blocks, we can state precisely which clock phase should enable the line. Registers always write to a bus during PHI_1; ALUs always operate during PHI_2. However, for the input and output ports, we can not say what the timing requirements are, for these are dictated by off-chip concerns. Hence, the control lines driving the ports must have the flexibility of changing the timing information. <VAR_EQUATION>s are <EQUATION>s with the capability of having this modified timing.

```
<VAR_EQUATION>  ::=     <EQUATION>
<VAR_EQUATION>  ::=     <EQUATION> [ <VAR_TIMING> ]

<VAR_TIMING>    ::=     CLOCKED_PHI_1
<VAR_TIMING>    ::=     CLOCKED_PHI_2
<VAR_TIMING>    ::=     NOT_CLOCKED
```

We see that a VAR_EQUATION may be a standard equation, in which case the timing takes the default clock phase, or an equation followed by one of the three timing specifications. The VAR_EQUATION may take on PHI_1 or PHI_2 as the enabling clock, or may asynchronously drive the control line directly. To see an example of these variable equations, consider an output port. The documentation for the element is given, followed by an example showing its use.

```
Element: OUTPUT_PORT
Required Parameters:
    Keyword: REGISTER     Type: REGISTER
Optional Parameters:
    Keyword: DRIVE        Type: EQUATION    Variable Timing
```

```
NAME OUTPUT_TEST 8;

OUTPUT_PORT PORT_1
   REGISTER: [REFRESH:ALWAYS];

OUTPUT_PORT PORT_2
   REGISTER: [REFRESH:ALWAYS],
   DRIVE: PAD;

OUTPUT_PORT PORT_3
   REGISTER: [REFRESH:ALWAYS],
   DRIVE: PAD[CLOCKED_PHI_1];

PRECHARGE_BOTH  PCHG;

END
```

The first port always drives the pads. The second port drives the pads during PHI_2 only when its input (coming from an input pad) was high during the previous PHI_1. The third port drives the pads during PHI_1 only when its input (coming from a different input pad) was high during the previous PHI_2.

7.3.8: Decode Operations

The Arithmetic-Logic Unit (ALU) is an example of a cell which can perform a wide variety of operations, but which has relatively few control lines. The particular operation performed by the ALU depends upon the state of several control lines. It is very difficult to specify the operation of the ALU in terms of its control line. One naturally thinks of the specification of the ALU operation in terms of operations like ADD and SUBTRACT. A DECODE parameter specifies how a field should be decoded to perform the appropriate operations. For example, the following is a partial listing of the ALU's documentation.

```
Element: ALU
Required Parameters:
    Keyword: INPUT_A              Type: REGISTER
    Keyword: INPUT_B              Type: REGISTER
    Keyword: OUTPUT_1             Type: REGISTER
    Keyword: DECODE               Type: DECODE
        Operations:  DONT_CARE
                     ADD
                     ADD_W_CARRY
                     SUBTRACT
                     SUB_W_BORROW
                     INCREMENT_A
                     INCREMENT_B
                     DECREMENT_A
                     DECREMENT_B
                     XOR
                     AND
                     SETA
                     OR
                     NAND
                     NOR
                     ...
Optional Parameters:
    Keyword: OUTPUT_2             Type: REGISTER
    Keyword: PRECHARGE            Type: EQUATION    Default: ALWAYS
    Keyword: CARRY_OUT            Type: OUTPUT
    Keyword: CARRY_INTO_MSB       Type: OUTPUT
    Keyword: MSB                  Type: OUTPUT
    Keyword: ZERO                 Type: OUTPUT
    Keyword: WRITE_OUTPUT_1       Type: EQUATION    Default: ALWAYS
    Keyword: WRITE_OUTPUT_2       Type: EQUATION    Default: GND
```

We can specify the operation of the ALU as follows.

```
FIELD ALU_OP<1:2>;

ALU .....
    DECODE: ALU_OP
        0=> ADD
        1=> SUBTRACT
        2=> AND
        3=> OR
    ....;
```

When the ALU_OP field has the value 00, the ALU will perform an addition operation, while an ALU_OP of 01 will cause a subtraction. ELSE can be used as the last case in the decode, which can save effort in a large, sparse decode.

```
....DECODE: ALU_OP
        0=> ADD
        2=> AND
        ELSE=> OR  .....
```

Another shorthand available allows several field values to be associated with one operation, using the BITSPEC construction.

```
....DECODE: ALU_OP
        0=> ADD
        2=> AND
        <1,3>=> OR  .....
```

The formal syntax for DECODE parameters is

```
<DECODE>        ::=     <DECODE> <DECODE1>
<DECODE>        ::=     <ID> <DECODE1>

<DECODE1>       ::=     <BITSPEC> => <ID>
<DECODE1>       ::=     ELSE => <ID>
<DECODE1>       ::=     <INT> => <ID>
```

This states that a DECODE is an <ID>, which is the field to decode, followed by a list of associations. Each association ties a field value or values to an operation. If there are some field values which are not associated with operations, a DON'T CARE is assumed. If the decoded field ever contains one of these values, the operation performed is unspecified and unguaranteed.

7.3.9: Sources

In Bristle Blocks datapaths, we have data lines running horizontally and control lines running vertically. There are times, however, when one would like to turn data lines into control lines. For example, flags from a register leave the register as data, but should enter the instruction decoder as control lines. The lines have to 'turn the corner'. Another example would be an instruction register. The instruction register is loaded with data, the operation to be performed, and it must communicate this data to the decoder. Bristle Blocks needs to know which bits of the register should connect to which inputs of the instruction decoder or to which pads. A parameter of type SOURCE conveys this information.

In the simplest case, a SOURCE parameter is a list of bit index and instruction bit pairs. For example,

```
{ 1 => FLAG ; 2 => ENABLE }
```

indicates that bit 1 (the most significant bit) of the register in question connects to the FLAG field, which must be a field containing only one bit. Similarly, the second bit of the register connects to the ENABLE field, again a single-bit field. To connect

to multiple-bit fields, the BITSPEC shorthand is used:

```
{ <1:4> => OPCODE }
```

Here, the OP_CODE field must be a four-bit field, which is driven from the four most significant bits in the register.

One element that uses SOURCES is a DATA_TO_CONTROL element. This element will function as an instruction register. Data in the register can drive bits in the instruction decoder. The documentation for this element is

```
Element: DATA_TO_CONTROL
Required Parameters:
    Keyword: REGISTER     Type: REGISTER
    Keyword: MAP          Type: SOURCES
Optional Parameters: NONE
```

One might use this element as follows.

```
NAME IR_TEST 8;

FIELD FROM<1:4>, TO<5:8>;

INPUT_PORT INSTRUCTION_PORT
    REGISTER: [WRITE_UPPER: FROM=0000],
    LOAD: ALWAYS;

DATA_TO_CONTROL INSTRUCTION_REGISTER
    REGISTER: [READ_UPPER: TO=0000,
               SUGGEST: NOT(TO=0000),
               VALUE: 00000000],
    MAP: { <1:4> => FROM ; <5:8> => TO };

INCREMENTER PC
    INPUT_REGISTER: [READ_UPPER: TO=0001,
                     REFRESH:ALWAYS,
                     WRITE_LOWER: FROM=0000];
    LOAD: FROM=0000;

OUTPUT_PORT ADDRESS
    REGISTER: [READ_LOWER: FROM=0000, REFRESH:ALWAYS];

PRECHARGE_BOTH  PCHG;

END
```

This example is portion of the Fetch/Execute section of a simple microprocessor. The Instruction Register drives the TO and FROM fields of the microcontrol word. Notice that if the TO field is not 0000, the instruction suggests to 00000000 for the next cycle. The 00000000 operation causes data in the Instruction Port to be loaded

into the instruction register, and the PC value increments. Thus, after every instruction which does not write into the instruction register, the instruction register automatically loads with the FETCH instruction.

Sources can also specify that certain bits in a register should connect to pads. If many bits from a register are connecting to pads, OUTPUT PORTS should be used, but if only a few bits connect to the decoder and a few connect to pads, a DATA_TO_CONTROL register can be used. Pads are indicated by the token 'PAD' in place of a field specification. The syntax for SOURCES is

```
<SINGLE_SOURCE> ::=     <BITSPEC> => <ID>
<SINGLE_SOURCE> ::=     <BITSPEC> => PAD
<SINGLE_SOURCE> ::=     <INTSPEC> => <ID>
<SINGLE_SOURCE> ::=     <INTSPEC> => PAD

<SOURCE>        ::=     <SINGLE_SOURCE> ; <SOURCE>
<SOURCE>        ::=     <SINGLE_SOURCE> }

<SOURCES>       ::=     { <SOURCE>
```

7.3.10: Destinations

The SOURCE parameters indicate how to turn data lines into control lines. The inverse operation is also useful: turning control lines into data lines, which allows equations from the instruction decoder to load into registers, to be used in the datapath during later cycles. The format for specifying a DESTINATION parameter is very similar to the SOURCE parameter format.

```
<DEST>          ::=     <EQUATION1> => <INT> ; <DEST>
<DEST>          ::=     <EQUATION1> => <INT> }

<DESTS>         ::=     { <DEST>
```

Informally, a DESTINATION parameter is a list of EQUATIONs with associated bit indicies. The following example illustrates calls of this type. The documentation for a CONTROL_TO_DATA element is given, along with a datapath using this element.

```
Element: CONTROL_TO_DATA
Required Parameters:
    Keyword: REGISTER     Type: REGISTER
    Keyword: MAP          Type: DESTS
    Keyword: LATCH        Type: EQUATION
Optional Parameters: NONE


NAME DESTINATION_EXAMPLE 4;

FIELD INPUT<1:2>;

CONTROL_TO_DATA DECODE
    REGISTER:    [WRITE_UPPER: ALWAYS],
    MAP: {INPUT=00 => 1; INPUT=01 => 2; INPUT=10 => 3; INPUT=11 => 4},
    LATCH: ALWAYS;

PRECHARGE_BOTH  PCHG;

END
```

The LATCH equation states that the register should be loaded from the DESTINATION parameter values every clock cycle. The DESTINATION parameter in this example 'decodes' the value of the input field: When the field has value 0, the most significant bit of the register will be the only bit with a high value; when the field has value 1, the next most significant bit will be the high bit, etc. Any bits of the register not specified in the DESTINATION parameter will be unaffected by the LATCH signal.

7.4: Comments and Macros

In addition to the language constructs presented above, the Bristle Blocks parser has two meta-commands: comments and macros. These constructs are not part of the formal language definition, but are processed by the parser before the formal language is parsed.

Comments consist of all characters between double-quote characters. The parser removes the double-quotes and all characters between them, before tokenizing the input stream. This allows comments to be inserted anywhere, even in the middle of an indentifier or number.

Macros are simple text-replacement facilities which reduce the amount of typing required to specify a design. They also aid in the reduction of errors. A macro has a

name, a set of parameters, and a body. When the macro is instantiated, the body is inserted into the character stream. Any parameter values to the macro are inserted in the text where the parameters occur in the macro body. An example of macro usage follows.

```
MACRO TEST1(ABC,DEF)
% THIS IS A TEST:  ?ABC? IS PARAMETER 1,
                   ?DEF? IS PARAMETER 2.
%

/*TEST1(HI THERE,XYZZY)*/
```

We have defined the macro TEST1. This macro takes two parameters, which are identified as ABC and DEF. The body of the macro consists of all characters between the percent signs. Within the macro body, tokens within question marks refer to instantiations of parameter values. For instance, the value of the first macro parameter is inserted in the macro body where the characters ?ABC? occur. Following the macro definition, we have a macro instantiation. The characters /* signify the start of a macro call, while */ indicates the end of the call. Between these indicators, we have the macro name and parameter values. Here we are stating that the macro TEST1 is to be called, with the first parameter set to the characters 'HI THERE' and the second parameter set to the characters 'XYZZY'. The above macro definition and instantiation is identical to the following text.

```
THIS IS A TEST:  HI THERE IS PARAMETER 1,
                 XYZZY IS PARAMETER 2.
```

Macro parameters may be given default values. The following example gives default values for the first and third parameters of the macro.

```
MACRO TEST2(P1/123,P2,P3/HI MOM) % IF ?P2?=?P1? THEN WRITE('?P3?'); FI %

/*TEST2(453,231,WHAT?)*/

/*TEST2(,X,)*/

/*TEST2(,X)*/

/*TEST2()*/
```

These four macro instantiations will expand into the following text.

```
 IF 231=453 THEN WRITE('WHAT?'); FI
IF X=123 THEN WRITE('HI MOM'); FI
IF X=123 THEN WRITE('HI MOM'); FI
IF =123 THEN WRITE('HI MOM'); FI
```

In the first example, we specified values for all three parameters, which were inserted into the text. In the second example, we let the first and third parameters take their default values. This was done by not specifying a value for the parameters. In the third parameter, we terminated the parameter list after the second parameter, so the third parameter again took on its default value. In the final example, the parameter list was empty, so every parameter took on its default value. Since the second parameter did not have a default value, an empty set of characters was used.

Parameter values consist of all characters upto but not including the first comma or close parenthesis. There are times, however, when one would like to pass these characters in as parameter values. To allow this, parameter values or default values may be enclosed in percent signs. For example,

```
/*TEST2(,X,%ILLEGAL CONDITION, PLEASE TRY AGAIN%)*/
```

produces the following text.

```
 IF X=123 THEN WRITE('ILLEGAL CONDITION, PLEASE TRY AGAIN'); FI
```

Macros are instantiated before the parser tokenizes the input, in the same manner as comments are removed. This allows identifiers to be 'split' across macro instantiations: part of an identifier or number is generated outside of a macro instantiation, while the remainder is generated by the macro. Macro instantiations may nest, and macro definitions may instantiate other macros. Macros must be defined before they are instantiated.

A macro definition is treated like a declaration to the parser. A formal statement of macro definition syntax is presented here. Rules which use :*= instead of ::= do not allow arbitrary insertion of blanks.

```
<DECLARATION>     :::=     MACRO <ID> <MACRO_HEADER> <MACRO_BODY>

<MACRO_HEADER>    ::=      <MACRO_HEADER1> )

<MACRO_HEADER1>   ::=      (
<MACRO_HEADER1>   ::=      ( <PARAM_DECL>
<MACRO_HEADER1>   ::=      <MACRO_HEADER1> , <PARAM_DECL>

<PARAM_DECL>      ::=      <ID>
<PARAM_DECL>      ::=      <ID>  <PARAM_DEFAULT>

<PARAM_DEFAULT>   :*=      / all-characters-until-comma-or-)
<PARAM_DEFAULT>   :*=      / % all-characters-until-% %

<MACRO_BODY>      :*=      <MACRO_BODY1> %

<MACRO_BODY1>     :*=      %
<MACRO_BODY1>     :*=      <MACRO_BODY1> <MACRO_BODY_ELEMENT>

<MACRO_BODY_ELEMENT>      :*=      all-characters-until-%-or-?
<MACRO_BODY_ELEMENT>      ::=      ? <ID> ?
```

Chapter 8: How Bristle Blocks Works

In this chapter, we will discuss the operations performed by Bristle Blocks. We will use the following chip specification as our example. This chip may be thought of as a datapath for a simple control processor. We have eight internal registers and an ALU, along with an input port and an output port. Figure 8-1 gives a block diagram representation of the circuit. The input specification to Bristle Blocks is listed here.

```
NAME SAMPLE 8;

FIELD REG_SELECT<1:3>,ENABLES<4:6>,ALU_OP<7:9>;

INPUT_PORT INPUT LOAD:ALWAYS,REGISTER:[WRITE_LOWER:ALWAYS];

MACRO ADDRESS(ADDR)
%REGISTER R?ADDR? OPTIONS:[READ_UPPER: REG_SELECT=?ADDR? AND ENABLES=XX1,
                           WRITE_UPPER: REG_SELECT=?ADDR? AND ENABLES=XX0,
                           REFRESH:ALWAYS];%

/*ADDRESS(000)*/
/*ADDRESS(001)*/
/*ADDRESS(010)*/
/*ADDRESS(011)*/
/*ADDRESS(100)*/
/*ADDRESS(101)*/
/*ADDRESS(110)*/
/*ADDRESS(111)*/

PRECHARGE_BOTH PCHG;

ALU ALU
    INPUT_A:[READ_LOWER:ALWAYS],
    INPUT_B:[READ_UPPER: ENABLES=1XX, REFRESH:ALWAYS],
    OUTPUT_1:[WRITE_UPPER: ENABLES=XX1],
    DECODE: ALU_OP
        0=> OR
        1=> INCREMENT_A
        2=> AND
        3=> SUBTRACT
        4=> XOR
        5=> ADD
        6=> ZERO
        7=> DECREMENT_A;

OUTPUT_PORT OUTPUT REGISTER:[READ_UPPER: ENABLES=X1X, REFRESH:ALWAYS],
    DRIVE:ALWAYS;

END
```

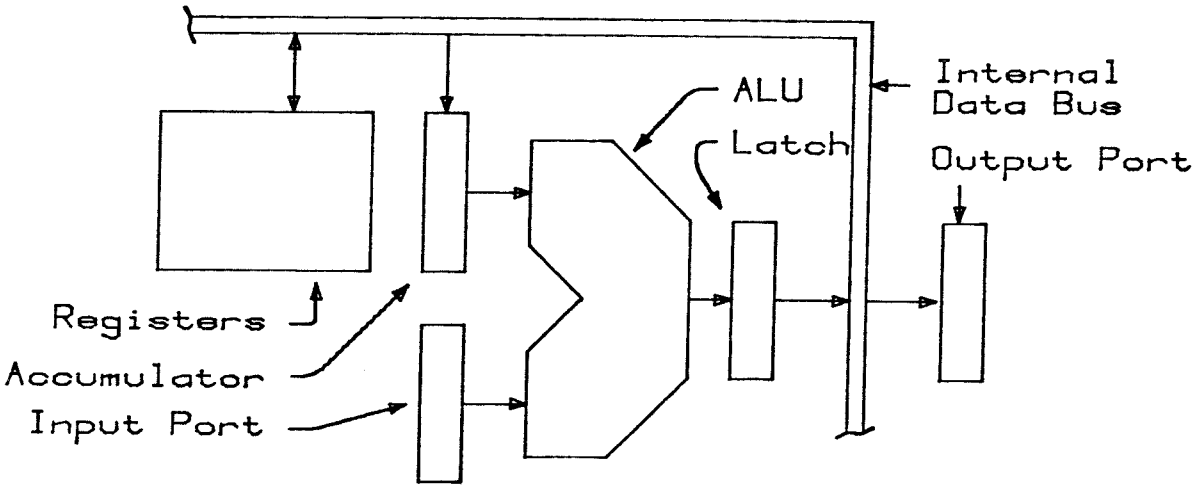


Fig. 8-1: Sample Chip Block Diagram

8.1: Parse Input

The first step taken by Bristle Blocks is to parse the user input, determining the elements and element configurations needed for the chip. The parser's output will be a series of function calls which, when evaluated, will generate the chip's layout.

In our example, we see that the name of the chip is SAMPLE and that the datapath width is 8. This name will be kept with the chip, to identify the current chip from other chips that may reside in the system. This name is also used to compute file names. For instance, the CIF file name will be SAMPLE.CIF, and the log file, which lists the testability vector and pad order, will be SAMPLE.CPL. The file names adhere to the DEC-10 conventions, which limit file names to six alphanumeric characters, the first of which should be a letter. In our example, the name SAMPLE is an acceptable file name. In other examples, the chip's name may not be acceptable, so Bristle Blocks computes a file name which bears a strong resemblance to the given name.

The datapath width is used for determining how many bits to place in each register and each processing element. In addition, for elements like the barrel shifter, the number of control lines for the element is a function of the datapath width.

The next line of text in the sample file contains the micro-control word specification. The user states that the micro-control word will be nine bits long,

and that this word can be thought of as three non-overlapping fields. The REG_SELECT field will be used to address one of the registers in the datapath, the ENABLES field will be used to control the transfer of data across the internal data bus, and the ALU_OP field will control the operation of the ALU.

Following the micro-control word specification, the input port is declared. This is an element of type INPUT_PORT, which the user chose to call 'INPUT'. This element is to 'ALWAYS' load its register from its pads, and its register is to 'ALWAYS' drive the lower data bus. The timing conventions presented in Chapter 6 state that the port unit actually loads data from the pads to its register every PHI_2, and that its register drives the lower bus every PHI_1. We will use the lower bus to transfer data from the input port to the ALU. The parser will generate a call to an internal function called PORT_IN. One parameter to this function gives the equations to drive a register, another parameter is the equation to control loading from the pads. The register has only one equation, which controls writing the the lower bus, and is set to PHI_1. The load parameter is set to PHI_2, since the port always loads from the pad, independent of the micro-control word.

Next, the user wants to specify the register array. This register array is composed of eight registers which function almost identically. To save typing, and to reduce the possibility of specification errors, the user uses a macro. The macro takes one parameter, which is the address of the register, and generates the specification for that register. The MACRO name is ADDRESS, and the single parameter's name is ADDR. The macro call /*ADDRESS(abc)*/ will generate the text

```
REGISTER Rabc OPTIONS: [READ_UPPER: REG_SELECT=abc AND ENABLES=XXI,
                        WRITE_UPPER: REG_SELECT=abc AND ENABLES=XXO,
                        REFRESH:ALWAYS];
```

Following this macro definition, the user calls the macro eight times, passing the eight register addresses. When this macro is expanded, the parser will see eight register specifications, so will generate eight calls to the internal REGISTER function. These registers each have three equations: reading the upper bus, writing the upper bus, and refreshing. The bus read/write operations occur during PHI_1, while the refresh occurs during PHI_2.

After the registers are specified, the user adds the bus precharge element. The buses in Bristle Blocks are dynamic. They are precharged during PHI_2, and transfer data during PHI_1. To write on the bus, a datapath element pulls the bits low to write a zero, or leaves the bus alone to write a one.

Following this, the user specifies an ALU, whose name is ALU. Following the three ALU register specifications, the user gives the operations performed by the ALU. The ALU has 13 control lines which are used to determine the operation done by the ALU. To perform an ADD operation, these 13 lines must be set in particular states, while a SUBTRACT operation requires different states on these wires. Rather than having the user specify these states, the parser allows the user to specify the operations. Here, the user has specified that the ALU should perform an ADD when the ALU_OP field is IOI, and a SUBTRACT when the field is OII. The other operations are seen in the input text. The parser must convert this operation-wise specification into a control-line-wise specification before calling the internal ALU function. This conversion will be discussed in section 8.2.

Following the ALU, the user specifies the output port, and then the END. When the END is reached, the parser will have collected 12 function calls to internal datapath element procedures, along with the description of the micro-control word and datapath width. Before these function calls can be made, the parser must generate the instruction decoder functions.

8.2: Generate Instruction Decoder Functions

The instruction decoder used in Bristle Blocks is nothing more than a series of NOR gates, as shown in figure 8-2. Each NOR gate drives one of the control lines, based upon the states of its input lines. Given a structure like this, only very-uninteresting decodes can be performed. The NOR gates can be thought of as actually AND gates, if all the microcode inputs were negated. Thus, we could only perform AND functions in the instruction decoder. To allow the inclusion of OR functions in the decoder, we allow some of the NOR gates to drive new decoder inputs, rather than driving control lines. Figure 8-3 shows some of these NOR gates. We can now perform OR functions in the decoder, although the OR functions cost more both in area and in time than the AND functions. In fact, we use this technique to generate the compliments of microcode inputs. The user may state

that an equation is dependent upon a microcode input being low rather than high, in which case a single-input NOR gate is used to generate the compliment of the actual input signal.

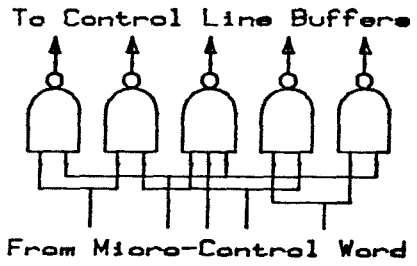


Fig. 8-2: NOR Gate Decoder

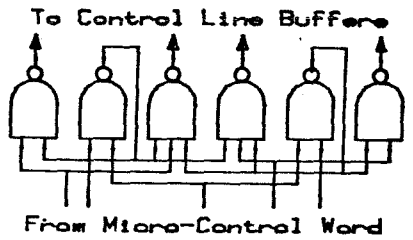


Fig. 8-3: Decoder with Minterm Gates

Each of the microcode equations passed to the internal element functions in Bristle Blocks must be the NOR form of equations. Hence, the parser must convert all non-NOR functions to NOR functions by declaring these new 'microcode inputs' and specifying the NOR gates which will drive these inputs. We convert an AND function to a NOR function simply by complimenting all of the input signals. Therefore,

```
a AND b AND -c
```

becomes

```
NOR(-a,-b,c)
```

An OR function is converted to a NOR function by inverting the output, which is done with another NOR gate. Therefore,

```
a OR b OR -c
```

becomes

```
NOR(NOR(a,b,-c))
```

An IF...THEN...ELSE...FI function is converted to NOR functions by realizing that

```
IF a THEN b ELSE c FI
```

can be stated as

```
AND(a,b) OR AND(-a,c)
```

or as

```
NOR(NOR(NOR(-a,-b),NOR(a,-c)))
```

Similarly, the decode functions, as in the ALU unit, are converted into NOR functions. In our example, we wish to perform the following decode.

```
ALUOP       OUTPUTs
0 0 0        I I I 0 0 0 0 I 0 I I 0 I
0 0 I        I I 0 0 0 0 I I 0 I I 0 0
0 I 0        I 0 0 0 0 I I I 0 I I 0 I
0 I I        I 0 0 I 0 0 I 0 0 I I 0 0
I 0 0        0 I I 0 I 0 0 I 0 I I 0 I
I 0 I        0 I I 0 0 0 0 I 0 I I 0 I
I I 0        0 0 0 0 I I I I 0 I I 0 I
I I I        0 0 I I 0 0 0 0 0 I I 0 I
```

The parser converts this to the following code:

```
FIELD NEW_FUNCTION<n>;
NEW_FUNCTION:= ALU_OP=0XI;
OUTPUT_1:= ALU_OP=0XX;
OUTPUT_2:= ALU_OP=X0X;
OUTPUT_3:= NEW_FUNCTION=0 AND OUTPUT_6=0;
OUTPUT_4:= ALU_OP=XII;
OUTPUT_5:= ALU_OP=IX0;
OUTPUT_6:= ALU_OP=XI0;
OUTPUT_7:= OUTPUT_3=0;
OUTPUT_8:= OUTPUT_4=0;
OUTPUT_9:= NEVER;
OUTPUT_10:= ALWAYS;
OUTPUT_11:= ALWAYS;
OUTPUT_12:= NEVER;
OUTPUT_13:= NEW_FUNCTION=0;
```

Once these conversions are completed, all of the equations are in the NOR form, which can easily be implemented in the instruction decoder. We have effectively widened the microcontrol word, and we also have a list of equations which drive these extra microcontrol word inputs.

8.3: Build Datapath Core

At this point, we know the width of the datapath, the equations for each of the control lines and virtual microcontrol word inputs, plus we have the 12 datapath element functions. We are set to generate the layout for the core of the datapath. The datapath core consists of the actual registers and ALUs, without the control line buffers or instruction decoder.

Before we actually generate the layouts, we need to determine the physical sizes of the datapath bits. In Bristle Blocks, we chose to perform global optimization by having all datapath bits the same physical height over local optimization with the required routing between cells. Figure 8-4 illustrates the two possible alternatives. In one case, we would leave the individual cells with their minimum sizes, and

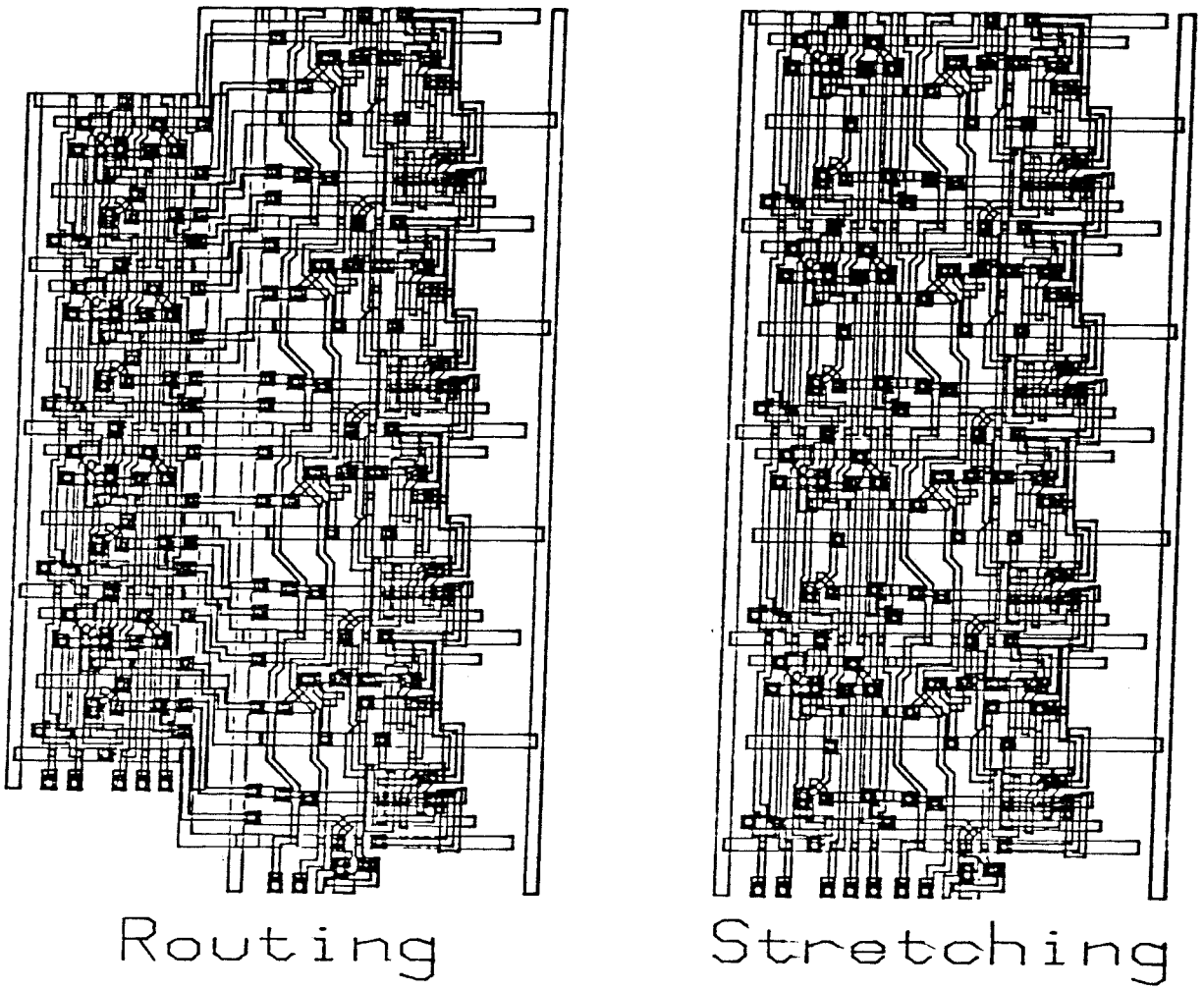


Fig. 8-4: Comparison of Stretching and Routing

route between the cells. The electrical properties of the individual cells would not change, but we would degrade the bus signals, and we would take horizontal chip area for the routing. In the second case, we would stretch the cells so that the interfaces match, and the cells could plug together with no stretching. Horizontal area is saved at the cost of some vertical area. Additionally, the control line signals would degrade, and the electrical properties of the cells could change. After an analysis of the situation, it was determined that the best approach would be to stretch the cells. But rather than externally stretching the cells, which would play havoc with the electronics, we design the cells to accept stretching parameters, so that the cell generates the stretched layout. In this way, the cell can monitor the stretching and alter its geometries to preserve the electronics. The cell may also select one circuit topology from several potential topologies, depending upon the physical size of the datapath.

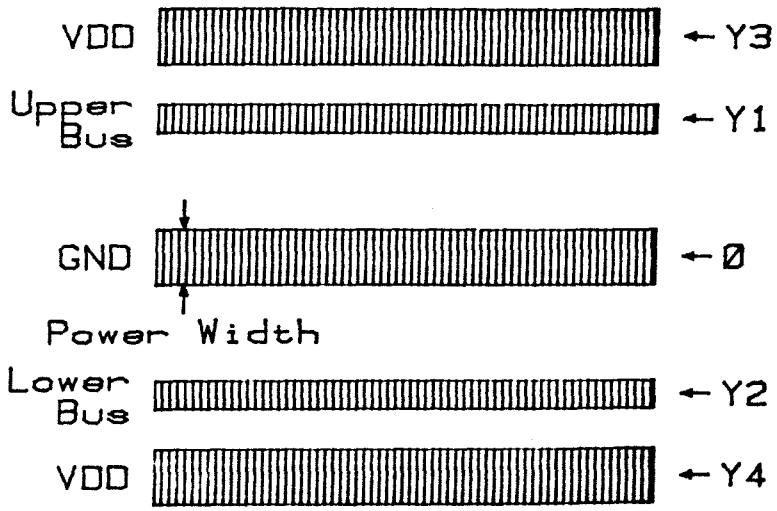


Fig. 8-5: Single Bit Floorplan

The first step, therefore, is to determine the stretch point values. Figure 8-5 shows the floorplan of a single bit of the datapath. We have chosen to position the Ground line which runs through the cell along the x-axis. Above this line we have the upper bus, at a y-coordinate of 'Y1', and a VDD line, at a y-coordinate of 'Y3'. Similarly, we have the lower bus and VDD line below the Ground line, at y-coordinates of 'Y2' and 'Y4'. Finally, the width of the power lines is POWER_WIDTH. Each datapath element function can be thought of as an object, as in object-oriented programming. To determine the stretch point values, we just ask each function what its minimum requirement is for each spacing. We take the largest of each of these values as the spacing between stretch points. In addition, we ask each element for its power consumption. By summing the power consumptions, we can determine the necessary width of the power lines. In figure 8-6 we show the register cell stretching itself to match the requirements of the system, while figure 8-7 shows the stack cell. The stack cell uses an alternate layout when the stretching is great enough.

After we have computed the stretch point values, we can call the individual element functions, requesting the layouts. These functions will examine the stretch points, along with the parameters passed from the user's specification, to determine which layout to use. For example, we have used several types of registers in the sample chip, yet there is only one register function. The following register configurations have been requested.

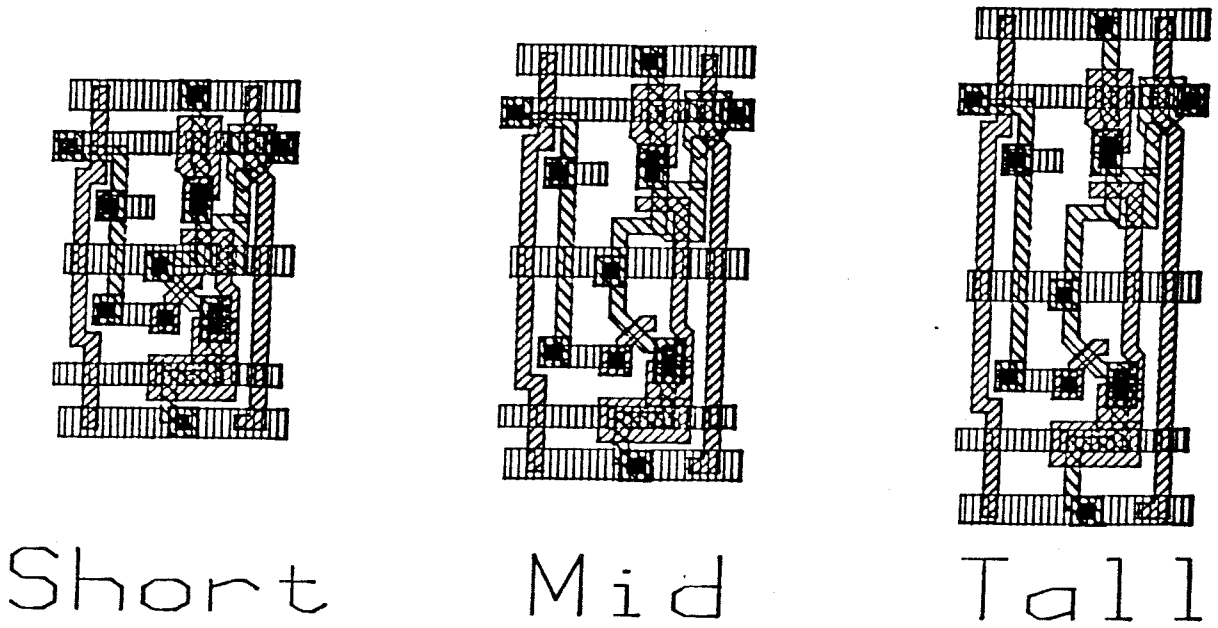


Fig. 8-6: Stretching Register Cell

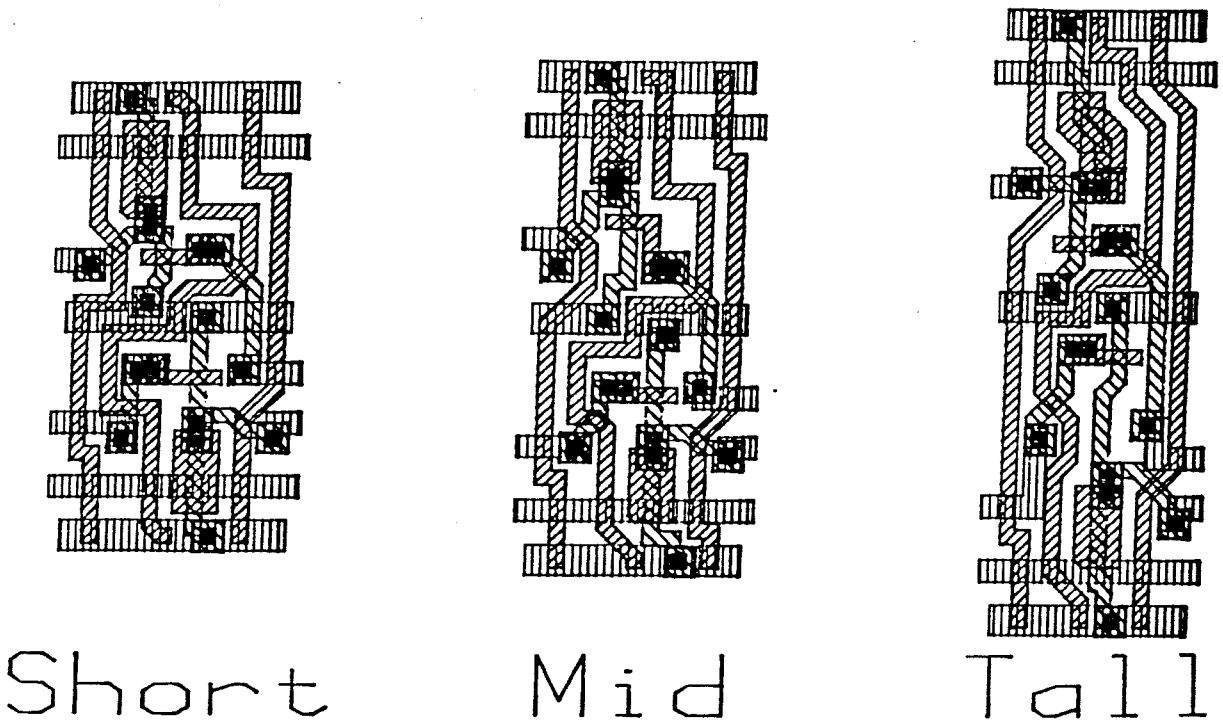


Fig. 8-7: Stretching Stack Cell

1) Write Lower
2) Read and Write Upper, and Refresh
3) Read Lower
4) Read Upper and Refresh
5) Write Upper

A register cell layout which performs all five of the basic operations required here is shown in figure 8-8. If we were to use this layout in all of the locations that require a register, we would be wasting a lot of space, since each of these functions require area. On the other hand, we do not wish to design 31 different registers, one for each of the possible configurations. What we do is design one register as a program which computes the appropriate layout from the functions required. Figure 8-9 shows the five resulting layouts needed by our sample chip.

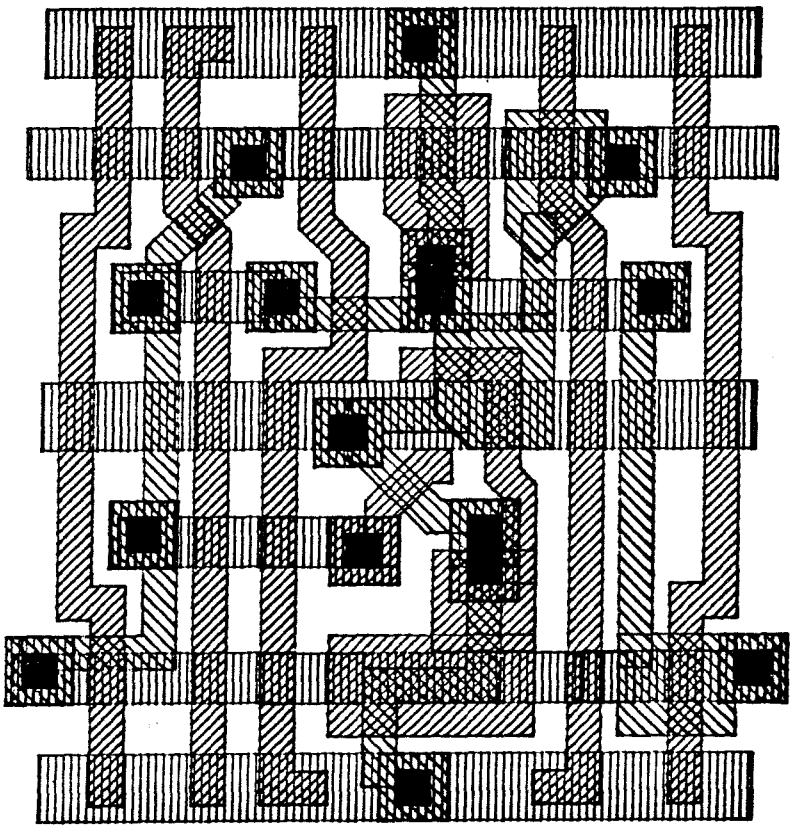


Fig. 8-8: Complete Register Cell

As the cell is computing the layout, it is very easy to add information about connection points: where the control lines interface to the datapath core. If this information were not captured with the layout, some program would have to determine these positions later, which means that this program would have to have intimate knowledge of each datapath cell, and would have to duplicate much of the

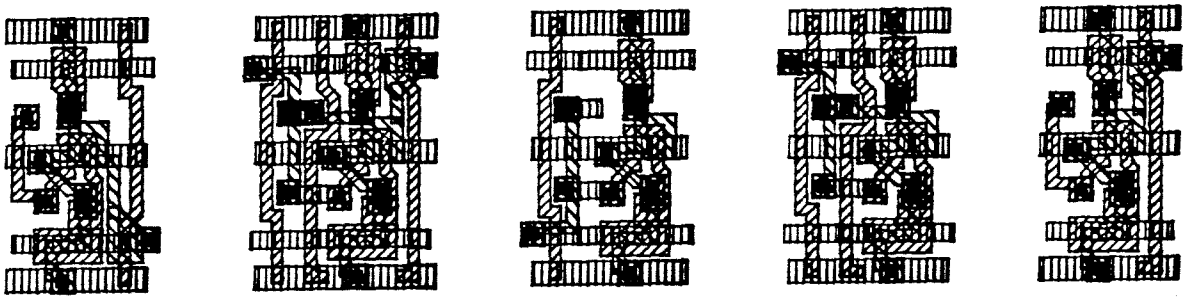


Fig. 8-9: Sample Chip Register Instances

computation performed by the cells. Instead, we choose to add information to the layout datastructure to indicate where the control lines connect. In fact, we need to know more than just position. What are these connection points to connect to? We have told the datapath elements which microcode equations drive each line, and the datapath element knows what style of buffer is required for each line. By adding this information to the connection point, the buffer program can generate the buffers by looking at the datapath core layout, and the instruction decoder program can generate its layout by looking at the result of the buffer program.

Once we have generated the layouts for each datapath element, we may abut the elements and finish the datapath core. To simplify the abutment procedure, we have defined the following conventions regarding the left and right edge characteristics of datapath cells: All geometry within a cell must have positive x-coordinates. All geometric primitives must be at least half the minimum design rule spacing from x=0. For instance, a diffusion feature must be at least 1.5 lambda from the edge of the cell. We can state the width of the cell as being the minimum x-coordinate that is at least half of the minimum design rule spacing from all geometric features of the cell. Therefore, if a diffusion edge has the largest x coordinate, the cell's width is 1.5 lambda beyond that coordinate. If we place the first datapath element at the origin, and displace all other elements by the widths of all elements to their left, we will have no design rule violations between cells. Notice that the two data buses and the power buses do not enter into the width calculation, for these lines must connect between cells. The layouts produced in this manner are large. Most of the elements communicate with the buses with diffusion connections. We will therefore allow a cell to place a diffusion-to-metal feedthrough on either edge of a cell, to connect to either bus. If a neighbor cell also connects to the bus, they will both share the same feedthrough.

Once we have completed the abutment process, we have finished the datapath core. Figure 8-10 shows the layout of the core for our sample chip. In addition to the layout, we have connection points along the lower edge of the layout for connecting to the buffers, and we have connection points on the left and right edges for connecting to pads.

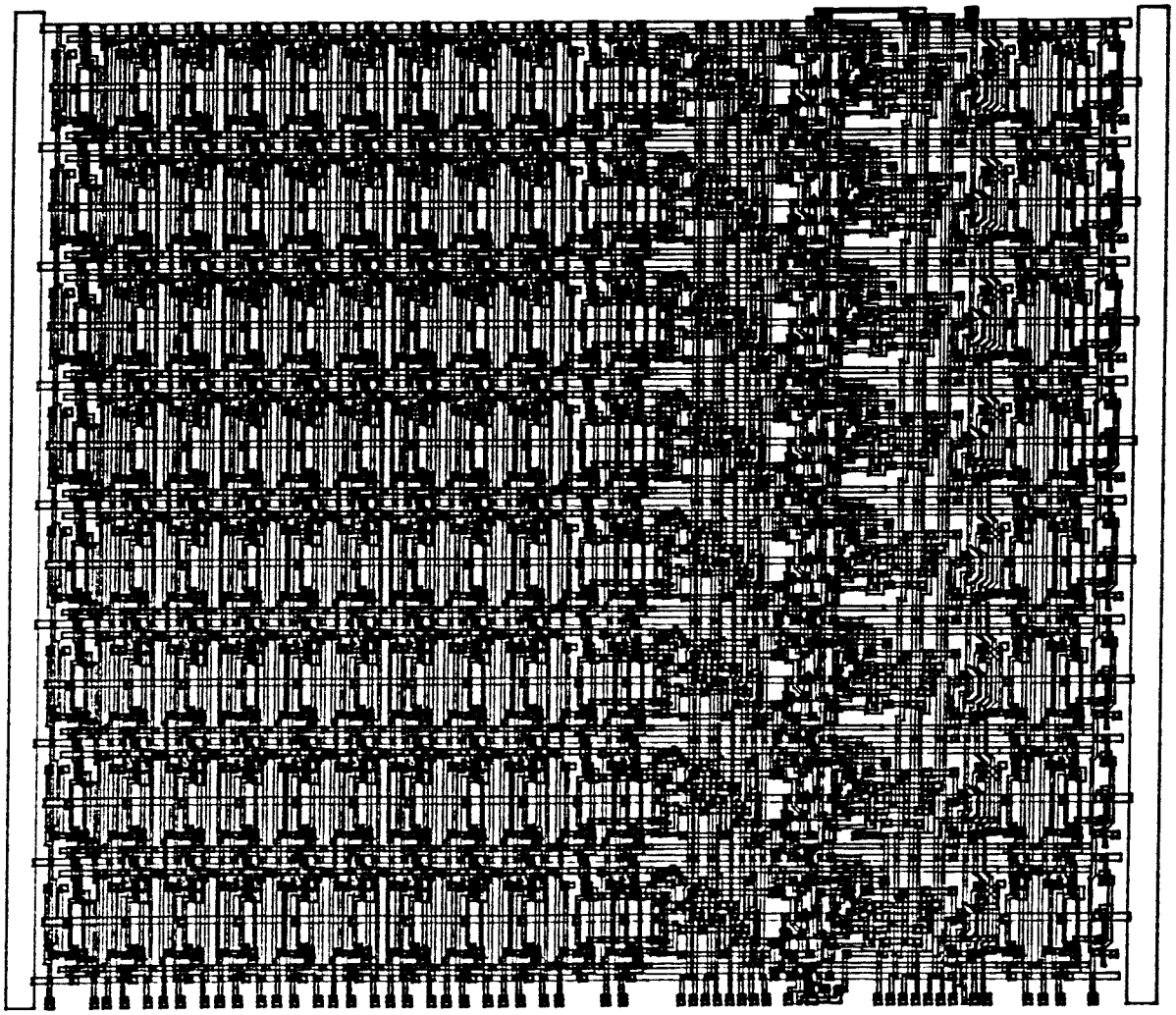


Fig. 8-10: Sample Chip Core Layout

8.4: Add Buffers and PLSRs

Given a datapath core, we need to add buffers to each of the control lines. These buffers latch values from the instruction decoder during one clock phase and drive the control lines on the other clock phase. These buffers satisfy the electrical

constraints of driving large loads from the weak signals of the decoder. They also satisfy the timing constraints by allowing the instruction decoder and datapath to work in parallel rather than in series. This parallelism allows the chips to run faster, and removes the possibility of race conditions.

To facilitate the testing of chips, we would like to independently test the instruction decoder and the datapath. If we had to test the two units together, it could take a fantastically long time to verify that the chip functions correctly. By splitting the testing task into testing the two pieces in isolation, one can hope of completely testing the chip. We do this by adding Parallel-Load Shift Registers (PLSRs) between the decoder and the datapath. As it turns out, the circuitry required by the buffers and the PLSRs have a lot in common. If we therefore design the PLSRs into the buffers, we can save a lot of area. The buffer routine adds the output driver of the buffers, while the PLSR routine adds the remainder of the buffers and the PLSRs.

The datapath core tells us which buffers it needs on which control lines, since this information is present in the connection points. We can arrange our buffer program to generate the buffer layouts in the same order as the connection points, so that we may river route between the buffers and the core. To generate the buffers, we need to compute the positions of the individual buffers. If we take the positions of the connection points as a first approximation, we will generate buffers which are as close as possible to the wires they drive. Given this first approximation, we move any buffers which are too close to neighboring buffers. We continue to shift buffer positions until none of the buffers overlap, and then we river route to the core. Figure 8-11 shows the buffer programs output for the sample chip.

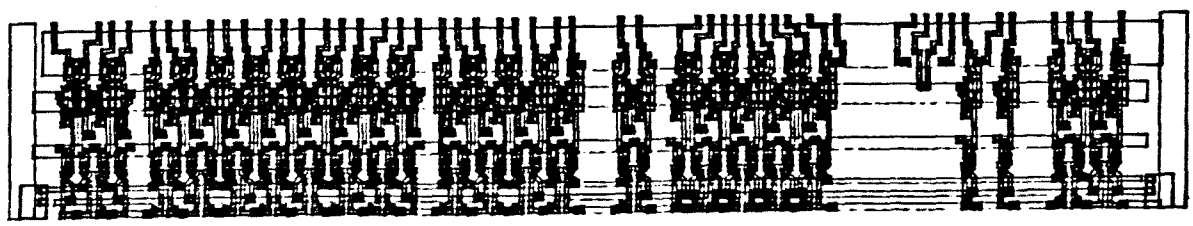


Fig. 8-11: Sample Chip Buffers

Some of the buffers drive control lines independent of the microcontrol word. Whenever the user specifies ALWAYS, NEVER, VDD, or GND for a control line function, that control line does not connect to the instruction decoder. Because some control lines may not connect to the instruction decoder, and because the buffers may have to shift positions to avoid overlaps, we put connection points on the buffer layouts. The PLSR program does not have to compute the positions of the buffer inputs, this information is given in the connection points. In fact, the type of the PLSR which needs to connect to each control line can be deduced from information in these connection points. The PLSR program operates in the same manner as the buffer program, positioning and shifting the individual circuits to avoid overlaps. Figure 8-12 shows the PLSR circuit and river route.

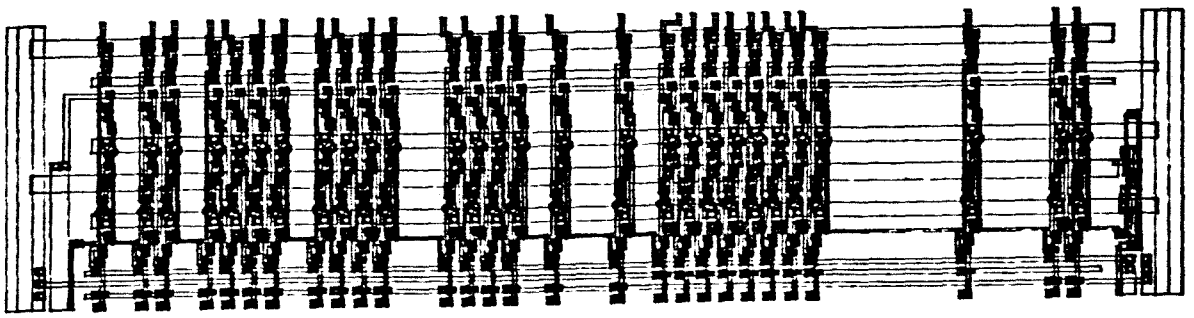


Fig. 8-12: Sample Chip PLSRs

8.5: Add Instruction Decoder

After the buffers and PLSRs are added to the core, we are ready to add the instruction decoder. The PLSR cells have connection points which state position and microcode equations. In addition, we have the microcode equations for the virtual control word inputs from the OR functions and the DECODE functions. We generate the instruction decoder layout in three steps. The first step initialized the decoder. The second step adds the virtual input NOR gates and connections to pads. The third step packs the wires to conserve chip area.

To initialize the decoder, we add the NOR gates which drive the PLSRs. These NOR gates are inserted in the column closest to the PLSR which is to be driven. Next, we add the NOR gates to generate the virtual equations. These NOR gates are driving the inputs of other NOR gates. We may potentially have a NOR gate driving a wire

which extends the whole width of the chip and drives many NOR gates. If this were to be allowed, the instruction decoder would run very slow. To avoid these long delays, we will limit the loading which we will add to a NOR gate. As we are scanning across the decoder, if we notice the load getting too great, we will terminate the line and regenerate the signal where it is needed. To do the scanning, we need to sort the virtual inputs before adding the NOR gates. We sort the list so that equations in the list only depend on equations occurring later in the list and not on any equations earlier in the list. We then take equations off the list in order, adding the NOR gates to the decoder as we go. When we have finished adding the virtual equations, the equations remaining in the list are in fact the actual microcontrol word inputs, so we connect each of these to wires which will connect to pads.

When we have completely generated the instruction decoder, we pack the wires to save space. The packed instruction decoder for the sample chip is shown in figure 8-13.

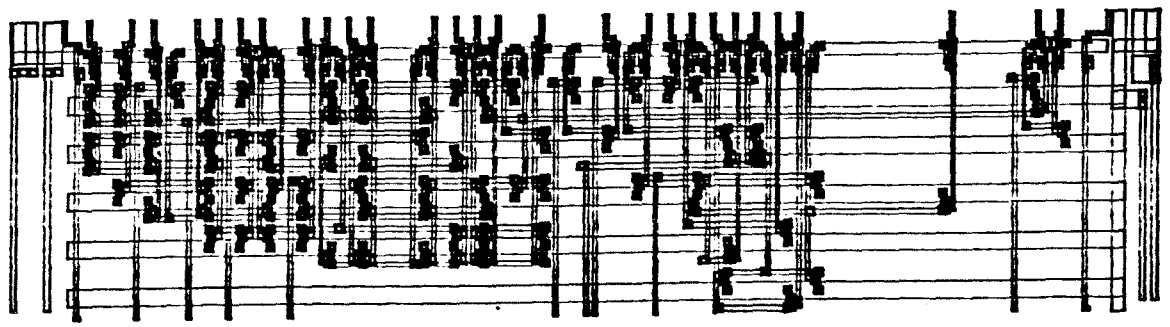


Fig. 8-13: Sample Chip Decoder

8.6: Add Pads

When the instruction decoder has inputs which come from pads, it adds wires to the edge of the cell. To the ends of these wires, it attaches connection points which will tell the pad router of the existence of the wire and of the type of pad required. Similarly, the datapath elements have previously generated connection points calling for pads. Based upon this information, along with power consumption information, the pad router can add the pads to the chip. If this datapath is to be a complete chip, the pad router can be called, which completes the chip. If this

datapath is to be a portion of a chip, the datapath can be used as is, and the connection points are available to aid in the interfacing to the datapath.

The pad router gathers all connection points which connect to pads. It then determines how many pads are needed, and can tell what types of pads are required. It places the pads and 'Roto-Routes' them as described in the River Router appendix. This Roto-Route shifts the pads around the chip in an attempt to minimize the wire lengths. The box river router is then called to route wires to the pads, and the chip is complete. Figure 8-14 shows the pad layout for the chip.

8.7: Conclusions

This chapter has described how Bristle Blocks builds a chip from the user specification. It can be seen that much of the task is geared toward this particular style of chip. This focus upon the floorplan does restrict the capabilities of the compiler to a very specific class of chip. On the other hand, this also allows Bristle Blocks to compile very optimal chips, and it also relieves the user of a lot of specification, since much of the specification can be implied from the structure of Bristle Blocks chips.

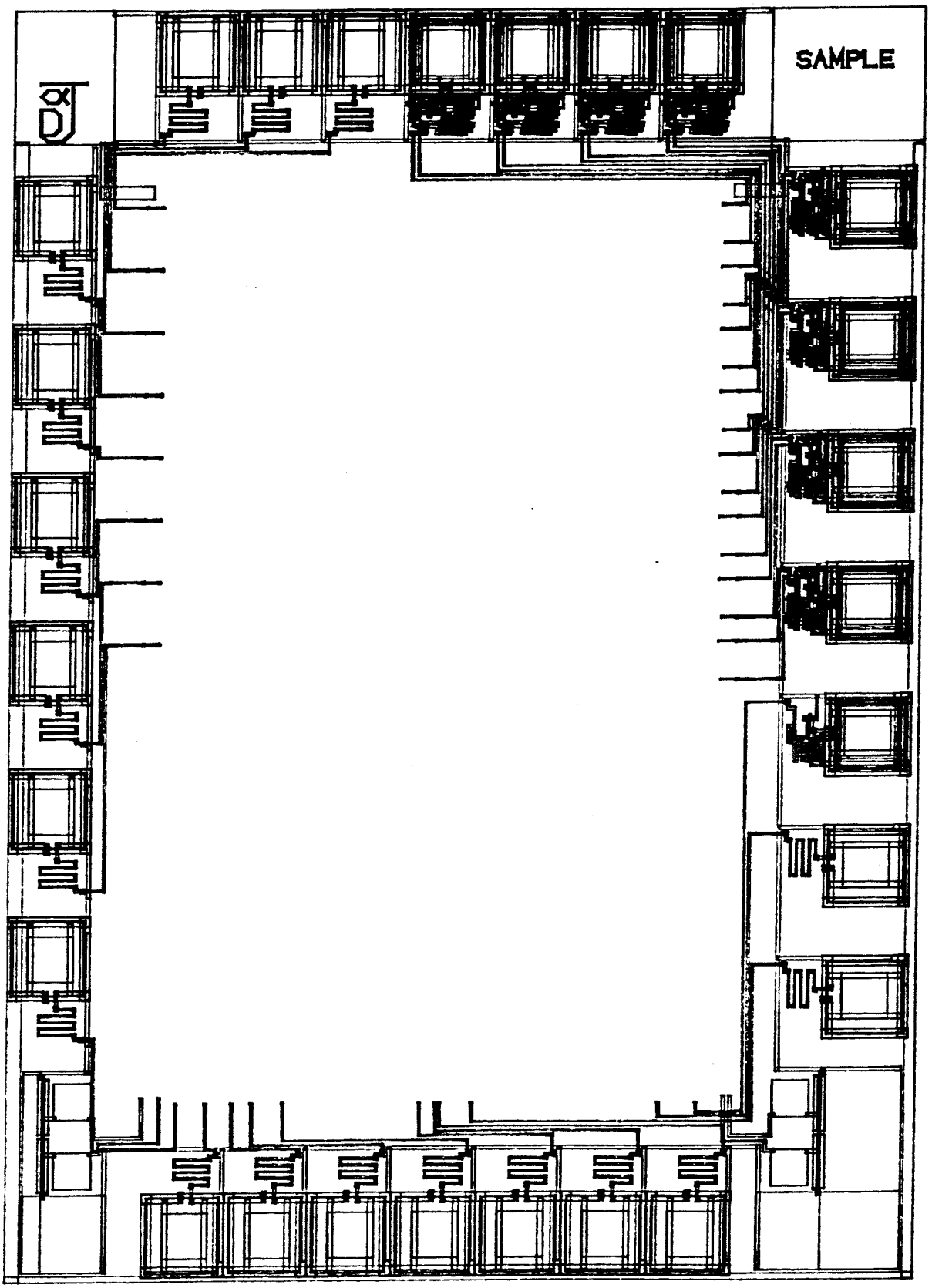


Fig. 8-14: Sample Chip Pads

Chapter 9: Bristle Blocks Examples

In this chapter, we will show several chips designed by Bristle Blocks. These examples will not only reinforce the language aspects of Bristle Blocks, but also illustrate how the design methodology is impacted by a silicon compiler, even one with a limited floorplan.

9.1: Lamp Dimmer

The lamp dimmer chip is a variation of a chip designed by Ron Ayres. Ron wanted a chip which he could use to control the brightness of a lamp. A diagram of Ron's setup is shown in figure 9-1. Several of these lamp control chips would be connected to a small processor via a serial bus. The processor could send commands to the lamp chips over this bus. The commands would be to select a particular device by its address or to set a device's lamp brightness to a given value. The lamp chips would drive Triacs, which controlled each lamp's power supply.

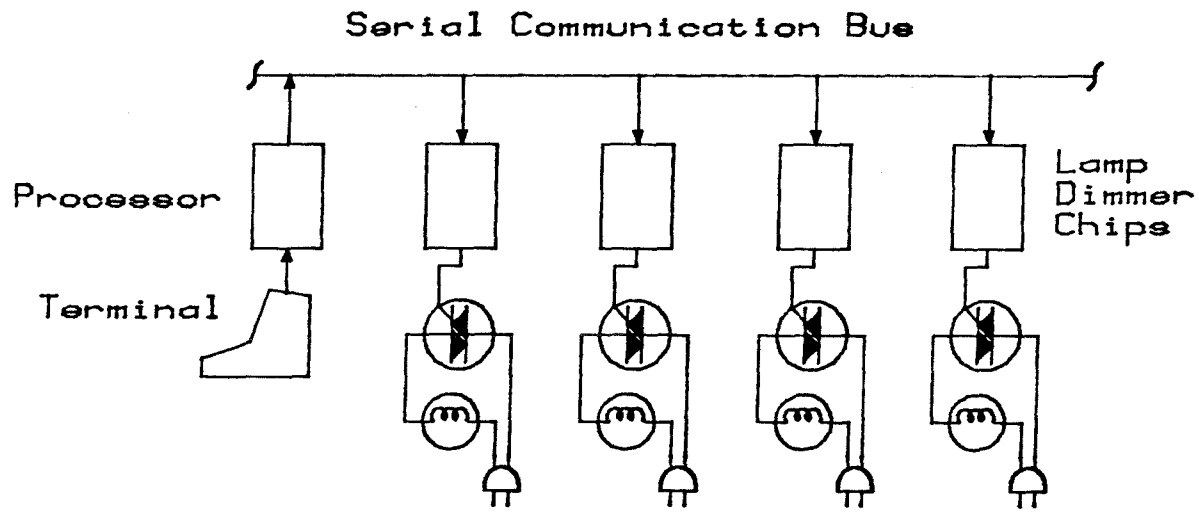


Fig. 9-1: Lamp Dimmer System

A block diagram of the lamp dimmer is shown in figure 9-2. We have an 8-bit shift register which reads the serial data from the command bus and drives the 6-bit data bus and 2-bit instruction bus. The data bus can load into the address register for modifing the device's bus address. The data bus is also compared with the value in the address register during the select operation to determine if the

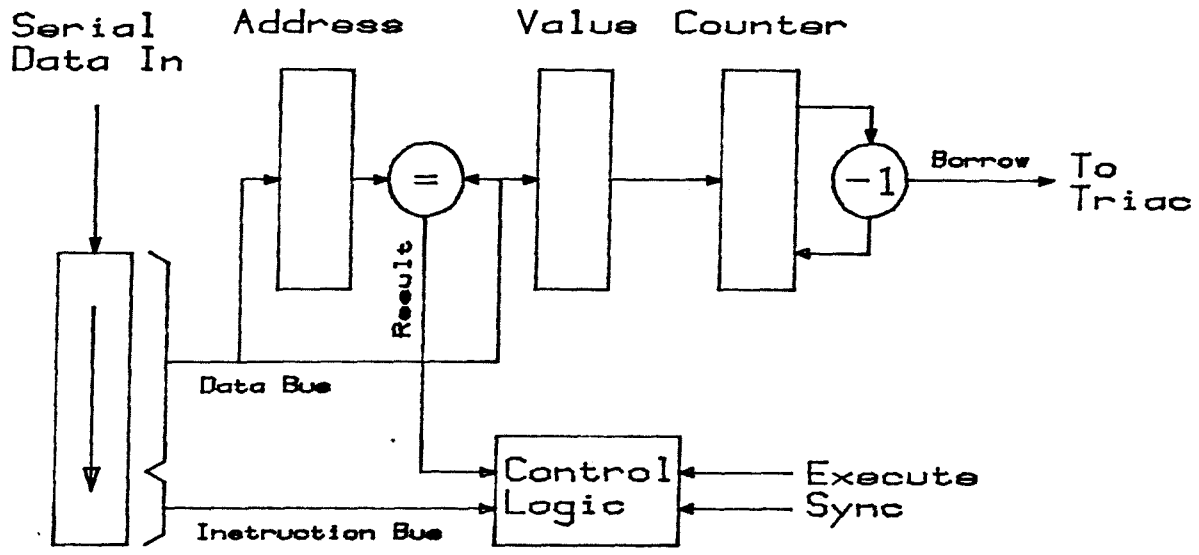


Fig. 9-2: Lamp Dimmer Chip Block Diagram

microprocessor is selecting this device. Finally, the data bus can load into the value register, which holds the current lamp brightness value.

To drive the Triac, we need to convert the data in the value register to a time. For a bright lamp, we want to pulse the Triac soon after the zero crossings of the AC line current. Conversely, for a very dim lamp, we should trigger the Triac just before the zero crossings. We will convert the data value to a time by comparing the value register's contents with the contents of a counter. The counter will be reset at the zero crossings of the AC current, and will be clocked so that the counter reaches full count just before the next zero crossing.

The 2-bit instruction bus drives the control logic section of the chip. The EXECUTE pin is used to indicate when the instruction bus and data bus contain valid data. When EXECUTE is high, the 2-bits are decoded as follows. An instruction of 00 initializes every device to its initial address. This initial address is read from a 6-bit input port, which is hard wired on each chip to a unique number. When the instruction is 01, the processor is selecting a new device. Each chip compares the data bus value to its address value and, if they match, the chip becomes enabled. When the instruction is 10, all selected devices will load their address registers from the data bus, allowing the processor to change the address of any device. Finally, when the instruction is 11, all selected devices will load their value register from the data bus.

From this description, we can start mapping the chip specification into Bristle Blocks. Since most of the data widths are 6 bits, we will set the chip width to 6. We can also state what the microcontrol word looks like. We need a bit to state whether this chip is the selected chip. We call this the ACTIVE bit. Next, we need two bits which contain the current instruction from the shift register, which we call the OP bits. The SYNC bit clears the counter. This signal goes high at each zero-crossing of the AC line. EXECUTE is an input which states when the instruction and data values are valid. We also need a data input, which we call INPUT. Our specification to Bristle Blocks now looks like this.

```
NAME RONS_CHIP  6;
PRECHARGE_BOTH PCHG;
FIELD ACTIVE<1>,OP<2:3>,SYNC<4>,EXECUTE<5>,INPUT<6>;
```

We define macros for each of the four basic instructions executed by the lamp dimmer chip. We have also defined a macro NOT_INITIALIZE which is true for any instruction but INITIALIZE.

```
MACRO INITIALIZE() % OP=00 AND EXECUTE=1 %
MACRO SELECT() % OP=01 AND EXECUTE=1 %
MACRO LOAD_ADDR() % OP=10 AND EXECUTE=1 AND ACTIVE=1 %
MACRO SET_VALUE() % OP=11 AND EXECUTE=1 AND ACTIVE=1 %
MACRO NOT_INITIALIZE() % NOT(OP=00) AND EXECUTE=1 %
```

We can now list the datapath elements we require for this chip. These elements are the command shift register, the initial address port, the address register and comparison unit, the value register and comparison unit, and the counter.

The command shift register must be 8-bits long, but our datapath is only 6-bits wide. However, we can think of the 8-bit register as a 6-bit register followed by a 2-bit register. The 6-bit register will contain the data portion of the command when the ENABLE bit is TRUE, at which time the 2-bit register is holding the operation portion of the command. The 6-bit register is simply a LEFT_RIGHT_SHIFTER, while the 2-bit register is a SHIFTING_IR, since we need to access the register's value in the instruction decoder. These two elements are specified by

```
FIELD MSB<7>;

LEFT_RIGHT_SHIFT    DATA
    INPUT_REGISTER: [WRITE_LOWER: /*NOT_INITIALIZE*/],
    SHIFT_LEFT:     NEVER,
    SHIFT_RIGHT:    ALWAYS,
    INPUT:          INPUT=I,
    MSB:            MSB;

SHIFTING_IR  OP_CODE
    MAP:            (<2:1>=>OP),
    INPUT:          MSB=I,
    SHIFT_LEFT:     ALWAYS,
    SHIFT_RIGHT:    NEVER;
```

The data in the 6-bit register should write to the lower bus for every operation except INITIALIZE. This shifter always shifts right, never left. The input comes from the INPUT pin, and the output drives a new microcode bit called MSB. This bit supplies the input data for the 2-bit shifter. The 2-bit shifter always shifts left, never right, and we feed the input of the shifter from the MSB bit, which is the output of the 6-bit shifter. The first and second bits in this shifter drive the OP field of the microcontrol word.

The next element we would like to design is the initial address port. This element is an input port which should transfer its data to the address register during an INITIALIZE instruction. The data input shift register does not write the lower bus during INITIALIZE, so we can have the input port drive the lower bus. The specification for the input port is simply

```
INPUT_PORT  FIXED_ADDRESS
    REGISTER:   [WRITE_LOWER: /*INITIALIZE*/],
    LOAD:       ALWAYS;
```

This element always loads its internal register from the pads, and drives the lower bus during the INITIALIZE instruction.

The address register and comparison unit must contain a latch to save the device address, a comparator to compare the device address to the select address, and a mechanism for saving the result of the comparison. To maintain the comparison result, we can either have a single bit latch for holding the value, or we may have a register to hold the select address and continuously perform the comparison. In Bristle Blocks, all registers have the same width as the datapath, so a 1-bit register takes as much area as a 6-bit register. Therefore, we choose to have a register for the select address and we will continuously compare the address and select

registers. To compare two registers' values, we use a subtractor with a 'value checker' on its output. This unit will compute the difference between its two input values, then compare this difference to a fixed constant. With a fixed constant of 0, this element's RESULT will be TRUE when the two input values are equal.

```
SUBTRACTER_WITH_VALUE_CHECK  ADDRESS_CHECKER
    VALUE:        000000,
    RESULT:       ACTIVE,
    INPUT_A:      [READ_LOWER: /*INITIALIZE*/ OR /*LOAD_ADDR*/,
                   REFRESH:ALWAYS],
    INPUT_B:      [READ_LOWER: /*SELECT*/,
                   REFRESH:ALWAYS],
    LOAD:         ALWAYS;
```

The INPUT_A register is the device address register. This register reads data from the lower bus during the INITIALIZE instruction and the LOAD_ADDRESS instruction if the chip is currently the selected device. The INPUT_B register contains the select address. This register reads the lower bus during the SELECT operation.

Next, we will specify the value register. This register should load from the lower bus during the SET_VALUE instruction. The contents of this register should be available for comparison with the counter's value. We can use the upper bus for this transfer. Since there are no other transfers on the upper bus, we can simply drive the upper bus from the register every clock cycle.

```
REGISTER     VALUE
    OPTIONS:      [READ_LOWER: /*SET_VALUE*/,
                   REFRESH:ALWAYS,
                   WRITE_UPPER:ALWAYS];
```

Finally, we need to specify the counter and comparison unit. In the chip specification, we stated that we wish to compare the data in the value register to the value in a counter. This counter is reset at the zero-crossing of the AC current, and simply increments each clock cycle. The clock cycle for the chip is adjusted so that the counter overflows at the next zero-crossing of the AC current. Rather than having an incrementer and a comparison unit, we can have a decrementer which is initialized to the value in the VALUE register at the zero crossing, and simply decrements each clock cycle. When this decrementer's value passes zero, the triac is strobed.

```
DECREMENTER     OUTPUT
    INPUT_REGISTER:  [READ_UPPER:SYNC=1 AND NOT(/*SET_VALUE*/),
                      READ_LOWER:SYNC=1 AND /*SET_VALUE*/],
    LOAD:            ALWAYS,
    CARRY_OUT:       PAD;
```

We use the upper bus to transfer data from the VALUE register to the decrementer at the zero crossings. Problems arise when the zero crossing occurs during the same clock cycle as a SET_VALUE instruction, because the VALUE register would be loading from one bus while driving the other. The register documentation in chapter 7 states that simultaneous read and write operations cause garbage data to be driven onto the written bus. Therefore, the decrementer reads its data from the lower bus if SYNC is high and a SET_VALUE instruction is being executed.

We have now described each of the elements for the lamp dimmer chip. We need only decide the order the elements should be placed in the datapath, since the order of implementation will be the order in which we specify the elements. The order does not matter a great deal, although the port cells are more efficient at either of the two ends of the datapath. We will place the fixed address input port on the left end of the chip. The complete specification for the lamp dimmer chip is listed here.

```
NAME RONS_CHIP  6;

PRECHARGE_BOTH PCHG;

FIELD ACTIVE<1>,OP<2:3>,SYNC<4>,EXECUTE<5>,INPUT<6>,MSB<7>;

MACRO INITIALIZE() % OP=00 AND EXECUTE=1 %

MACRO SELECT() % OP=01 AND EXECUTE=1 %

MACRO LOAD_ADDR() % OP=10 AND EXECUTE=1 AND ACTIVE=1 %

MACRO SET_VALUE() % OP=11 AND EXECUTE=1 AND ACTIVE=1 %

MACRO NOT_INITIALIZE() % NOT(OP=00) AND EXECUTE=1 %

INPUT_PORT  FIXED_ADDRESS
    REGISTER:    [WRITE_LOWER: /*INITIALIZE*/],
    LOAD:        ALWAYS;

LEFT_RIGHT_SHIFT     DATA
    INPUT_REGISTER:  [WRITE_LOWER: /*NOT_INITIALIZE*/],
    SHIFT_LEFT:      NEVER,
    SHIFT_RIGHT:     ALWAYS,
    INPUT:           INPUT=1,
    MSB:             MSB;

SHIFTING_IR  OP_CODE
    MAP:             {<2:1>=>OP},
    INPUT:           MSB=1,
```

```
    SHIFT_LEFT:      ALWAYS,
    SHIFT_RIGHT:     NEVER;

SUBTRACTER_WITH_VALUE_CHECK  ADDRESS_CHECKER
    VALUE:       000000,
    RESULT:      ACTIVE,
    INPUT_A:     [READ_LOWER: /*INITIALIZE*/ OR /*LOAD_ADDR*/,
                  REFRESH:ALWAYS],
    INPUT_B:     [READ_LOWER: /*SELECT*/,
                  REFRESH:ALWAYS],
    LOAD:        ALWAYS;

REGISTER    VALUE
    OPTIONS:     [READ_LOWER: /*SET_VALUE*/,
                  REFRESH:ALWAYS,
                  WRITE_UPPER:ALWAYS];

DECREMENTER    OUTPUT
    INPUT_REGISTER: [READ_UPPER:SYNC=1 AND NOT(/*SET_VALUE*/),
                     READ_LOWER:SYNC=1 AND /*SET_VALUE*/],
    LOAD:        ALWAYS,
    CARRY_OUT:   PAD;

END
```

Bristle Blocks compiled the layout for this chip in 1.8 minutes. The chip dimensions were 78.9 mil by 102.4 mil, and the chip consumed 26 ma. Figure 9-3 shows the bounding boxes for the various sections of the chip.

9.2: Random Tune Generator

The Player chip was designed to play pseudo-random melodies. The system block diagram is shown in figure 9-4. External to the player chip is an EPROM memory chip which contains the melody algorithm. Using the algorithm in the ROM, the player chip computes a square wave signal. This square wave is multiplied by the note amplitude to generate an 8-bit output value. The output value is converted to an analog voltage by a Digital-to-Analog Converter (DAC).

The melody algorithm is contained in an object-oriented data structure contained in the melody ROM. The ROM is organized as as 256 note 'objects'. Each object specifies a note, containing a duration, amplitude, and frequency, along with potential future notes. A note object is graphically illustrated in figure 9-5. When the player chip is playing a note, it generates a square wave with the specified duration, amplitude, and frequency. When the given note has finished, the player chip will follow one of the four next-note pointers to find the next note. This

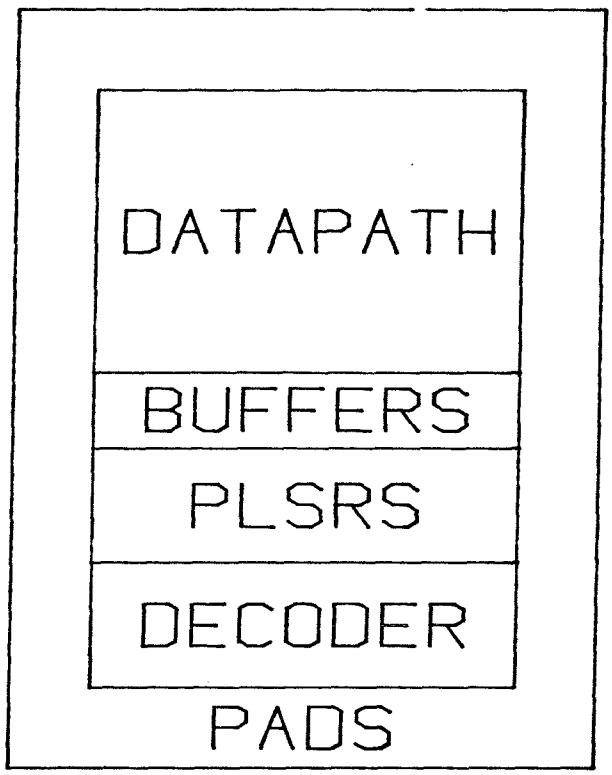


Fig. 9-3: Lamp Dimmer Chip Bounding Box Plot

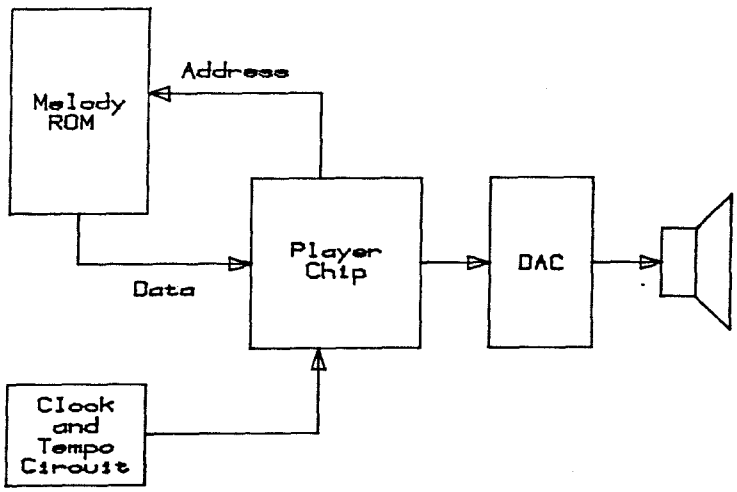


Fig. 9-4: Random Tune System Block Diagram

```
CONTROL_TO_DATA_AND_BACK  IR
    TO_CONTROL:       {<1:3>=> OP;<4:6>=> PAD},
    LOAD:             OP=0XX OR TIME=I,
    REGISTER:         [SUGGEST:TIME=0 AND OP=IXX],
    TO_DATA:          {IF OP=0II THEN RND=XXI ELSE OP=0X0 FI => 3;
                       IF OP=0II THEN RND=XIX ELSE OP=00I OR OP=0I0 FI => 2;
                       OP=0II=> 1;
                       IF OP=0II THEN RND=XXI ELSE OP=0X0 FI => 6;
                       IF OP=0II THEN RND=XIX ELSE OP=00I OR OP=0I0 FI => 5;
                       OP=0II=> 4};
```

The TO_CONTROL parameter specifies that bits 1-3 drive the OP field, while bits 4-6 drive pads, as stated above. The register is loaded from the instruction decoder when the OP field equals OXX or when TIME=I. When the OP field's MSB is low, the chip is reading in the note parameters, so the sequencer increments the OP field value. When the final parameter is read, the OP field is loaded with I00, I0I, II0, or III, depending upon the next note to be played. The sequencer then waits until the TIME field goes high, indicating that the note has finished playing.

The pseudo-random number generator uses a shift register with feedback logic. The feedback logic computes the shifter input value as a function of the current shift register data. With an appropriate feedback function, the random number stream repeats every 255 cycles, which is the maximal cycle length attainable using this form of generator. The RESET2 input, which comes from a pad, will clear the shift register. This input allows the user to alter the random number sequence. Without providing this reset, the system may only produce one fixed melody if the random number shift register always initializes with the same value on power up. The random number generator is specified by the following code.

```
SHIFTING_IR  RANDOM
    MAP:              {<1,7,8>=>RND},
    SHIFT_RIGHT:      OP=000,
    SHIFT_LEFT:       NEVER,
    INPUT:            RND=I00 OR RND=0I0 OR RND=00I OR RND=III,
    REGISTER:         [SUGGEST: RESET2=I, VALUE:00000000, REFRESH:OP=IXX];
```

The ROM interface is fairly straightforward. An output port supplies the upper 8 address bits for the ROM. These bits select which note object is the active note. This register is loaded with a new value when the chip begins to play a new note. The register is cleared when RESET1 is high, which allows the user to reinitialize the melody. An input port reads the data from the ROM. This port always drives the data unto the lower bus. The Bristle Blocks specification of these two ports is shown here.

```
OUTPUT_PORT  SEGMENT
    REGISTER:    [READ_LOWER: OP=1XX AND TIME=1,
                  SUGGEST: RESET1=1,
                  VALUE:00000000];

INPUT_PORT  DATA
    LOAD:        ALWAYS,
    REGISTER:    [WRITE_LOWER:ALWAYS];
```

The frequency divider is implemented as a 16-bit down-counter. This counter is initialized to the frequency value read from the ROM. The counter then decrements once each clock cycle. When the counter's data reaches zero, the frequency divider is reinitialized to the frequency value, and the square wave output changes sign. The 16-bit counter is implemented as a pair of 8-bit decrementers. Both decrementers decrement their values each clock cycle. If the least-significant word's value does not cause a carry, the most-significant value is reset to its pre-decremented value. In effect, the most-significant word is not decremented unless the least-significant word caused a carry. When both decrementers have a carry output, both counters are set to the frequency value and the square wave changes sign. The frequency divider is specified as follows.

```
SWAPPING_DECREMENTER  FREQUENCY_LOW
  ACTIVE:      [SUGGEST:NEVER],
  BACKUP:      [READ_LOWER:OP=OII, REFRESH:ALWAYS],
  RESTORE:     FREQ=II,
  LOAD:        ALWAYS,
  CARRY_OUT:   FREQ BIT 1;

REGISTER FREQUENCY_HIGH  OPTIONS:[WRITE_UPPER:ALWAYS,
                   READ_LOWER:OP=OIO,
                   REFRESH:ALWAYS];

SWAPPING_DECREMENTER  FREQUENCY_HIGH_DEC
  ACTIVE:      [READ_UPPER:FREQ=II OR OP=OII],
  BACKUP:      [READ_UPPER:FREQ=II OR OP=OII, REFRESH:ALWAYS],
  LOAD:        ALWAYS,
  CARRY_OUT:   FREQ BIT 2,
  RESTORE:     FREQ=XO,
  SAVE:        FREQ=XI;
```

Next, we need a timer. The timer is preset to the note duration. The timer's value is decremented when the TEMPO input is high. When the timer's value becomes zero, TIME becomes high, and the next note is played.

```
DECREMENTER TIMER
  INPUT_REGISTER:[READ_LOWER:OP=OOO, REFRESH:TEMPO=O],
```

```
LOAD:       TEMPO=I,
CARRY_OUT:  TIME;
```

To generate the output value, we need to multiply the square wave by the note amplitude. As it turns out, square waves have only two values: +1 and -1. When the square wave is high, the output value is just the note amplitude, and when the square wave is low, the output value is the inverse of the note amplitude. Our output section has a swapping output port. The two registers are loaded with the amplitude and the inverse amplitude when the note parameters are read. Each time the frequency divider produces a carry output, the data in these two registers swap places. The output pads are driven with the data contained in one of these register. The 'multiplying' output unit is implemented by the following datapath elements.

```
SUBTRACTER  NEGATE
    INPUT_A:            [SUGGEST:ALWAYS, VALUE:00000000],
    INPUT_B:            [READ_LOWER: OP=001],
    OUTPUT_REGISTER:    [WRITE_UPPER: OP=010],
    LOAD:               ALWAYS;

PRECHARGE_BOTH  PRECHARGE;

SWAPPING_OUTPUT_PORT  OUTPUT
    ACTIVE:             [READ_LOWER: OP=001, SUGGEST:ALWAYS],
    BACKUP:             [READ_UPPER: OP=010, SUGGEST:ALWAYS],
    RESTORE:            OP=1XX AND FREQ=11,
    SAVE:               OP=1XX AND FREQ=11;
```

The complete chip specification is listed next. Bristle Blocks compiled the chip in 3.67 CPU minutes, and the final chip size is 140 by 154 mil. The chip consumes 59 ma. of power at 5 volts.

```
NAME PLAYER 8;

FIELD OP<1:3>,RND<4:6>,TIME<7>,FREQ<8:9>,RESET1<10>,TEMPO<11>,RESET2<12>;

OUTPUT_PORT  SEGMENT
    REGISTER:    [READ_LOWER: OP=1XX AND TIME=1,
                  SUGGEST: RESET1=1,
                  VALUE:00000000];

INPUT_PORT  DATA
    LOAD:        ALWAYS,
    REGISTER:    [WRITE_LOWER:ALWAYS];

SHIFTING_IR  RANDOM
    MAP:                {<1,7,8>=>RND},
    SHIFT_RIGHT:        OP=000,
    SHIFT_LEFT:         NEVER,
    INPUT:              RND=100 OR RND=010 OR RND=001 OR RND=111,
    REGISTER:           [SUGGEST: RESET2=1, VALUE:00000000, REFRESH:OP=1XX];
```

```
CONTROL_TO_DATA_AND_BACK  IR
     TO_CONTROL:       {<1:3>=> OP;<4:6>=> PAD},
     LATCH:            OP=0XX OR TIME=1,
     REGISTER:         [SUGGEST:TIME=0 AND OP=1XX],
     TO_DATA:          {IF OP=011 THEN RND=XX1 ELSE OP=0X0 FI => 3;
                        IF OP=011 THEN RND=X1X ELSE OP=001 OR OP=010 FI => 2;
                        OP=011=> 1;
                        IF OP=011 THEN RND=XX1 ELSE OP=0X0 FI => 6;
                        IF OP=011 THEN RND=X1X ELSE OP=001 OR OP=010 FI => 5;
                        OP=011=> 4};

SWAPPING_DECREMENTER  FREQUENCY_LOW
     ACTIVE:           [SUGGEST:NEVER],
     BACKUP:           [READ_LOWER:OP=011, REFRESH:ALWAYS],
     RESTORE:          FREQ=11,
     LOAD:             ALWAYS,
     CARRY_OUT:        FREQ BIT 1;

REGISTER FREQUENCY_HIGH  OPTIONS: [WRITE_UPPER:ALWAYS,
                                   READ_LOWER:OP=010,
                                   REFRESH:ALWAYS];

SWAPPING_DECREMENTER  FREQUENCY_HIGH_DEC
     ACTIVE:           [READ_UPPER:FREQ=11 OR OP=011],
     BACKUP:           [READ_UPPER:FREQ=11 OR OP=011, REFRESH:ALWAYS],
     LOAD:             ALWAYS,
     CARRY_OUT:        FREQ BIT 2,
     RESTORE:          FREQ=X0,
     SAVE:             FREQ=X1;

PRECHARGE_AND_BREAK_UPPER  CUT;

DECREMENTER TIMER
     INPUT_REGISTER: [READ_LOWER:OP=000, REFRESH:TEMP0=0],
     LOAD:           TEMP0=1,
     CARRY_OUT:      TIME;

SUBTRACTER  NEGATE
     INPUT_A:          [SUGGEST:ALWAYS, VALUE:00000000],
     INPUT_B:          [READ_LOWER: OP=001],
     OUTPUT_REGISTER:  [WRITE_UPPER: OP=010],
     LOAD:             ALWAYS;

PRECHARGE_BOTH  PRECHARGE;

SWAPPING_OUTPUT_PORT  OUTPUT
     ACTIVE:           [READ_LOWER: OP=001, SUGGEST:ALWAYS],
     BACKUP:           [READ_UPPER: OP=010, SUGGEST:ALWAYS],
     RESTORE:          OP=1XX AND FREQ=11,
     SAVE:             OP=1XX AND FREQ=11;

END
```

9.3: Frequency Scaler Chip

Jeff Sondeen, employed by Hewlett-Packard, Colorado Springs, was on temporary assignment to Caltech when he designed the frequency scaler (FRESCA) chip. The chip specification presented here is a slightly modification of Jeff's design. Jeff

wanted a chip which scales the frequency of an input waveform. The chip would accept a binary waveform, and generate a new binary waveform with the frequency scaled, but with the duty factor of the output wave as close as possible to the input wave's duty factor.

The chip counts the number of clock cycles that occur while the input waveform is high, and the number of clock cycles occurring while the input signal is low. The sum of these two numbers is the period of the input signal. These two numbers are multiplied by one user-supplied constant, and divided by another constant, to generate two output period numbers. The output generator sets the output high for the number of clock cycles indicated by the scaled high period value, then sets the output low for the number of clock cycles indicated by the scaled low period value.

Rather than perform a multiply and divide on the chip, Jeff used incremental techniques to achieve the same results. Rather than incrementing a value during the high period and multiplying this by one of the scaling factors, we accumulate the scaling factor over the high period. We do the divide and decrement by repeated subtractions. The simplified block diagram of the FRESCA chip is shown in figure 9-6. The input section computes the high and low periods, scaled by one of the two scale parameters. The storage section stores these two values. The output unit computes the output signal, using the period values from the storage section and the other scale parameter. Finally, the state section computes when various signals change.

Some additional complexity has been added to the simplified block diagram to correct for round off errors during the counting processes. The SAVE_D_BAR and TO_OUTPUT elements are the elements added to improve the counting accuracy. Bristle Blocks compiled the FRESCA chip in 3.0 minutes. The chip size was 124 by 177 mil, and the chip consumed 68 ma. at 5 volts. The Bristle Blocks specification for the chip is shown here.

```
NAME FRESCA 16;

MACRO CONST1() % 000000000000XXXX %
MACRO CONST2() % 111111111111XXXX %

FIELD   IN<1>,
        LOAD<2:3>,
        OLD_IN<4>,
```

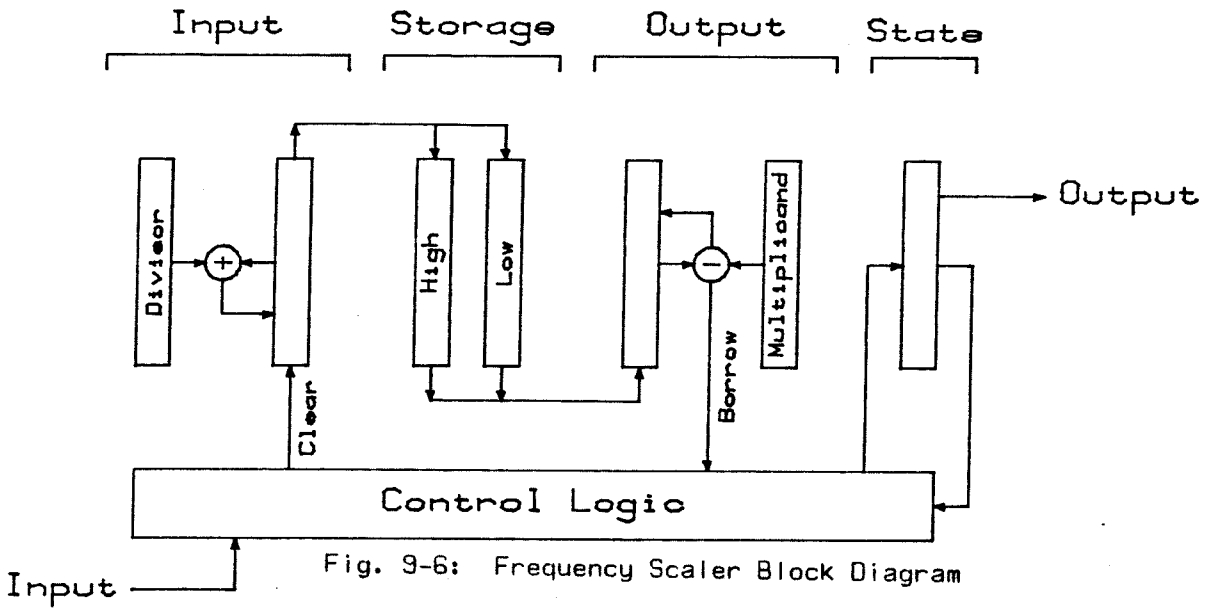


Fig. 9-6: Frequency Scaler Block Diagram

```
        DELTA_IN<5>,
        DELTA_OUT<6>,
        OUT<7>,
        DATA<8:11>;

    "Input Section:"

CONTROL_TO_DATA   SAVE_M
    REGISTER:    [REFRESH:ALWAYS,
                  SUGGEST:LOAD=0X,
                  VALUE: /*CONST1*/,
                  WRITE_LOWER: DELTA_IN=0],
    MAP:         { DATA=XXXI => 16 ;
                   DATA=XXIX => 15 ;
                   DATA=XIXX => 14 ;
                   DATA=IXXX => 13 },
    LATCH:       LOAD=0X;

CONTROL_TO_DATA   SAVE_D_BAR
    REGISTER:    [REFRESH:ALWAYS,
                  SUGGEST:LOAD=X0,
                  VALUE: /*CONST2*/,
                  WRITE_LOWER: DELTA_IN=1],
    MAP:         { DATA=XXX0 => 16 ;
                   DATA=XX0X => 15 ;
                   DATA=X0XX => 14 ;
                   DATA=0XXX => 13 },
    LATCH:       LOAD=X0;

ADDER   INPUT
    INPUT_A:     [READ_UPPER: DELTA_IN=0,
                  READ_LOWER: DELTA_IN=1],
    INPUT_B:     [READ_LOWER: DELTA_IN=0],
    OUTPUT_REGISTER:    [WRITE_UPPER:ALWAYS],
    LOAD:               ALWAYS,
    CARRY_IN_BAR:       DELTA_IN=0;
```

```
PRECHARGE_AND_BREAK_LOWER    GAP1;

    "Storage Section"

REGISTER  HIGH
    OPTIONS:              [READ_UPPER: DELTA_IN=I AND IN=0 AND
                                       NOT(DELTA_OUT=0 AND OUT=0),
                           REFRESH: ALWAYS,
                           WRITE_LOWER: DELTA_OUT=0 AND OUT=0];

REGISTER  LOW
    OPTIONS:              [READ_UPPER: DELTA_IN=I AND IN=I AND
                                       NOT(DELTA_OUT=0 AND OUT=I),
                           REFRESH: ALWAYS,
                           WRITE_LOWER: DELTA_OUT=0 AND OUT=I];

PRECHARGE_AND_BREAK_UPPER    GAP2;

    "State Section"

CONTROL_TO_DATA_AND_BACK    STATE
    REGISTER:             [REFRESH:ALWAYS],
    LATCH:                ALWAYS,
    TO_CONTROL:           { 1=> PAD ;
                            2=> OUT ;
                            3=> OLD_IN ;
                            4=> DELTA_IN },
    TO_DATA:              { OUT=I => 1 ;
                            IF DELTA_OUT=0 THEN OUT=0 ELSE OUT=I FI => 2 ;
                            IN=I => 3 ;
                            IF IN=I THEN OLD_IN=0 ELSE OLD_IN=I FI => 4 };

ADDER    TO_OUTPUT
    INPUT_B:              [READ_LOWER: DELTA_OUT=0, REFRESH:ALWAYS],
    INPUT_A:              [READ_UPPER: DELTA_OUT=I,
                           WRITE_UPPER:DELTA_OUT= 0],
    LOAD:                 ALWAYS;

PRECHARGE_AND_BREAK_LOWER    GAP3;

    "Output Section:"

SUBTRACTER    OUTPUT
    INPUT_A:              [READ_UPPER: ALWAYS],
    INPUT_B:              [READ_LOWER: LOAD=XI],
    OUTPUT_REGISTER:      [WRITE_UPPER: DELTA_OUT=I],
    LOAD:                 ALWAYS,
    CARRY_OUT_BAR:        DELTA_OUT;

CONTROL_TO_DATA    SAVE_D
    REGISTER:     [REFRESH:ALWAYS,
                   SUGGEST:LOAD=XO,
                   VALUE: /*CONST1*/,
                   WRITE_LOWER: LOAD=XI],
    MAP:          { DATA=XXXI => 16 ;
                    DATA=XXIX => 15 ;
                    DATA=XIXX => 14 ;
                    DATA=IXXX => 13 },
    LATCH:        LOAD=XO;
```

```
PRECHARGE_BOTH    END;
END
```

9.4: SDLC Chip

John Wawrzynek, a member of Caltech's Silicon Structures Project (SSP), was interested in building a synchronous, serial communication chip, similar to IBM's Synchronous Data Link Control chip, or the synchronous portion of INTEL's 8251A USART chip. He found that each of these chips had undesirable 'features' because the chip designers wanted a 'universal' chip. John realized that with a silicon compiler, chips can be optimized to their application, rather than being 'general purpose' in nature.

The SDLC chip is designed to be used with an 8-bit microprocessor. The chip contains both a transmit and receive buffer, along with a status/command register. The microprocessor interface consists of an 8-bit data port, a read (RD) line, a write (WR) line, and a control/data (C_DBAR line). The system interface consists of a reset (RESET) line, transmit clock signal (TXC), and receive clock signal (RXC), along with the standard power and clock signals. The network interface consists of the transmit data (TX) line and the receive data line (RX).

Upon RESET, or when the microprocessor sets bit 3 in the status/command register, the receiver enters the HUNT mode. In HUNT mode, the receiver circuitry attempts to match each 8-bit window in the incoming bit stream, scanning for the SYNC character, which is fixed as IOOOOOOI. When the sync character is received, the SDLC chip terminates HUNT mode and begins assembling characters.

Upon RESET, the SDLC chip will transmit SYNC characters until data is written into the transmitter buffer. Additionally, whenever a character has finished being transmitted, and the transmitter buffer is not full, the SYNC character will be transmitted.

The Bristle Blocks code for the SDLC chip is listed here. Bristle Blocks compiled the chip in 2.4 minutes, and the resulting chip size was 95 by 148 mils. The chip consumed 36 ma. of power.

```
NAME SDLC 8;

MACRO SYNC() % [0000001] %

FIELD RESET<1>,RD<2>,WR<3>,C_DBAR<4>,RXC<9>,TXC<10>,TDONE<5>,RDONE<11>,
      TXBUF_FULL<6>,RXBUF_FULL<7>,HUNT_MODE<8>,IS_SYNC<12>,RX<13>;

IO_PORT  DATA
    OUTPUT_REGISTER:  [READ_UPPER: RD=1,
                       WRITE_UPPER: WR=1,
                       REFRESH: ALWAYS],
    LOAD:             WR=1,
    DRIVE:            RD=1;

CONTROL_TO_DATA_AND_BACK  STAT_CMD
    REGISTER:        [READ_UPPER: WR=1 AND C_DBAR=1,
                      WRITE_UPPER: RD=1 AND C_DBAR=1,
                      SUGGEST: RESET=1,
                      VALUE: 00000000,
                      REFRESH: ALWAYS],
    TO_CONTROL:      { 1=> RXBUF_FULL; 2=> TXBUF_FULL; 3=> HUNT_MODE },
    TO_DATA:         { RDONE=1 OR RXBUF_FULL=1 AND RD=0
                         OR RXBUF_FULL=1 AND C_DBAR=1 => 1;
                       WR=1 AND C_DBAR=0 OR TDONE=0 AND TXBUF_FULL=1 => 2;
                       IS_SYNC=0 AND HUNT_MODE=1 => 3 },
    LATCH:           ALWAYS;

REGISTER  TXBUF
    OPTIONS:    [READ_UPPER: WR=1 AND C_DBAR=1,
                 WRITE_LOWER: TDONE=1,
                 REFRESH: ALWAYS];

SHIFTING_IR  T
    REGISTER:    [READ_LOWER: TDONE=1 AND TXBUF_FULL=1,
                  SUGGEST: TXBUF_FULL=0 OR RESET=1,
                  VALUE: /*SYNC*/,
                  REFRESH: ALWAYS],
    SHIFT_RIGHT: TXC=1,
    SHIFT_LEFT:  NEVER,
    MAP:         { 8=> PAD };

PRECHARGE_AND_BREAK_LOWER  LOWER_CHARGE;

PRECHARGE_BOTH  BOTH_CHARGE;

REGISTER  RXBUF
    OPTIONS:    [WRITE_UPPER: RD=1 AND C_DBAR=0,
                 READ_LOWER: RDONE=1,
                 REFRESH: ALWAYS];

SHIFTER_WITH_VALUE_CHECK  R
    REGISTER:       [WRITE_LOWER: RDONE=1,
                     REFRESH: ALWAYS],
    SHIFT_RIGHT:    RXC=1,
    SHIFT_LEFT:     NEVER,
    VALUE:          /*SYNC*/,
    RESULT:         IS_SYNC,
    INPUT:          RX=1;

LEFT_RIGHT_SHIFT  TCOUNT
    INPUT_REGISTER:  [SUGGEST: RESET=1 OR HUNT_MODE=1,
```

```
                        VALUE: 0000000I,
                        REFRESH: ALWAYS],
    SHIFT_LEFT:        TXC=I,
    SHIFT_RIGHT:       NEVER,
    MSB:               TDONE;

LEFT_RIGHT_SHIFT  RCOUNT
    INPUT_REGISTER:    [SUGGEST: RESET=I,
                        VALUE: 0000000I,
                        REFRESH: ALWAYS],
    SHIFT_LEFT:        RXC=I,
    SHIFT_RIGHT:       NEVER,
    MSB:               RDONE;

END
```

In another application, the same basic function was required, but due to processor overhead time, FIFOs were required on the transmit and receive buffers. In the following listing, 8-word deep FIFOs have been added to the two buffers. The compile time for this new chip was 6.67 CPU minutes, the chip size was 222 by 199 mils, and the power requirements were 103 ma.

```
NAME SDLC2 8;

MACRO SYNC() % 10000001 %

FIELD RESET<1>,RD<2>,WR<3>,C_DBAR<4>,RXC<9>,TXC<10>,TDONE<5>,RDONE<11>,
      TXBUF_FULL<6>,RXBUF_FULL<7>,HUNT_MODE<8>,IS_SYNC<12>,RX<13>,
      TXRA<14:21>,TXWA<22:29>,RXRA<14:21>,RXWA<22:29>;


IO_PORT  DATA
    OUTPUT_REGISTER:   [READ_UPPER: RD=I,
                        WRITE_UPPER: WR=I,
                        REFRESH: ALWAYS],
    LOAD:              WR=I,
    DRIVE:             RD=I;

CONTROL_TO_DATA_AND_BACK  STAT_CMD
    REGISTER:          [READ_UPPER: WR=I AND C_DBAR=I,
                        WRITE_UPPER: RD=I AND C_DBAR=I,
                        SUGGEST: RESET=I,
                        VALUE: 00000000,
                        REFRESH: ALWAYS],
    TO_CONTROL:        { 1=> RXBUF_FULL; 2=> TXBUF_FULL; 3=> HUNT_MODE },
    TO_DATA:           { RDONE=I AND
                             (RXRA=IXXXXXXX AND RXWA=XXXXXXXI OR
                              RXRA=XIXXXXXX AND RXWA=IXXXXXXX OR
                              RXRA=XXIXXXXX AND RXWA=XIXXXXXX OR
                              RXRA=XXXIXXXX AND RXWA=XXIXXXXX OR
                              RXRA=XXXXIXXX AND RXWA=XXXIXXXX OR
                              RXRA=XXXXXIXX AND RXWA=XXXXIXXX OR
                              RXRA=XXXXXXIX AND RXWA=XXXXXIXX OR
                              RXRA=XXXXXXXI AND RXWA=XXXXXXIX) => 1;
                         WR=I AND C_DBAR=0 AND
                             (TXRA=IXXXXXXX AND TXWA=XXXXXXXI OR
                              TXRA=XIXXXXXX AND TXWA=IXXXXXXX OR
```

```
                        TXRA=XXIXXXXX AND TXWA=XIXXXXXX OR
                        TXRA=XXXIXXXX AND TXWA=XXIXXXXX OR
                        TXRA=XXXXIXXX AND TXWA=XXXIXXXX OR
                        TXRA=XXXXXIXX AND TXWA=XXXXIXXX OR
                        TXRA=XXXXXXIX AND TXWA=XXXXXIXX OR
                        TXRA=XXXXXXXI AND TXWA=XXXXXXIX)
                    OR TXBUF_FULL=I => 2;
                IS_SYNC=0 AND HUNT_MODE=I => 3;
                 NOT(RXRA=IXXXXXXX AND RXWA=IXXXXXXX OR
                     RXRA=XIXXXXXX AND RXWA=XIXXXXXX OR
                     RXRA=XXIXXXXX AND RXWA=XXIXXXXX OR
                     RXRA=XXXIXXXX AND RXWA=XXXIXXXX OR
                     RXRA=XXXXIXXX AND RXWA=XXXXIXXX OR
                     RXRA=XXXXXIXX AND RXWA=XXXXXIXX OR
                     RXRA=XXXXXXIX AND RXWA=XXXXXXIX OR
                     RXRA=XXXXXXXI AND RXWA=XXXXXXXI) =>4;
                 NOT(TXRA=IXXXXXXX AND TXWA=IXXXXXXX OR
                     TXRA=XIXXXXXX AND TXWA=XIXXXXXX OR
                     TXRA=XXIXXXXX AND TXWA=XXIXXXXX OR
                     TXRA=XXXIXXXX AND TXWA=XXXIXXXX OR
                     TXRA=XXXXIXXX AND TXWA=XXXXIXXX OR
                     TXRA=XXXXXIXX AND TXWA=XXXXXIXX OR
                     TXRA=XXXXXXIX AND TXWA=XXXXXXIX OR
                     TXRA=XXXXXXXI AND TXWA=XXXXXXXI) =>5},
    LATCH:          ALWAYS;

MACRO TXBUFREG(NAME,ADR)
% REGISTER  TXBUF_?NAME?
    OPTIONS: [READ_UPPER: WR=I AND C_DBAR=0 AND TXRA=?ADR?,
              WRITE_LOWER: TDONE=I AND TXWA=?ADR?,
              REFRESH: ALWAYS];  %

/*TXBUFREG(1,00000001)*/
/*TXBUFREG(2,00000010)*/
/*TXBUFREG(3,00000100)*/
/*TXBUFREG(4,00001000)*/
/*TXBUFREG(5,00010000)*/
/*TXBUFREG(6,00100000)*/
/*TXBUFREG(7,01000000)*/
/*TXBUFREG(8,10000000)*/

SHIFTING_IR  TXREAD_POINTER
    SHIFT_LEFT: NEVER,
    SHIFT_RIGHT: WR=I AND C_DBAR=0,
    MAP: {<1:8> => TXRA},
    REGISTER: [SUGGEST: RESET=I,
               VALUE:10000000],
    INPUT: TXRA=00000001;

SHIFTING_IR  TXWRITE_POINTER
    SHIFT_LEFT: NEVER,
    SHIFT_RIGHT: TDONE=I,
    MAP: {<1:8> => TXWA},
    REGISTER: [SUGGEST: RESET=I,
               VALUE:10000000],
    INPUT: TXWA=00000001;

SHIFTING_IR  T
    REGISTER:     [READ_LOWER: TDONE=I AND TXBUF_FULL=I,
                   SUGGEST: TXBUF_FULL=0 OR RESET=I,
                   VALUE: /*SYNC*/,
```

```
                    REFRESH: ALWAYS],
    SHIFT_RIGHT: TXC=1,
    SHIFT_LEFT:  NEVER,
    MAP:         { 8=> PAD };

PRECHARGE_AND_BREAK_LOWER  LOWER_CHARGE;

PRECHARGE_BOTH  BOTH_CHARGE;

MACRO RXBUFREG(NAME,ADR)
% REGISTER  RXBUF_?NAME?
    OPTIONS: [WRITE_UPPER: RD=1 AND C_DBAR=0 AND RXRA=?ADR?,
              READ_LOWER: RDONE=1 AND RXWA=?ADR?,
              REFRESH: ALWAYS];  %

/*RXBUFREG(1,00000001)*/
/*RXBUFREG(2,00000010)*/
/*RXBUFREG(3,00000100)*/
/*RXBUFREG(4,00001000)*/
/*RXBUFREG(5,00010000)*/
/*RXBUFREG(6,00100000)*/
/*RXBUFREG(7,01000000)*/
/*RXBUFREG(8,10000000)*/

SHIFTING_IR  RXREAD_POINTER
    SHIFT_LEFT: NEVER,
    SHIFT_RIGHT: RD=1 AND C_DBAR=0,
    MAP: {<1:8> => RXRA},
    REGISTER: [SUGGEST: RESET=1,
               VALUE:10000000],
    INPUT: RXRA=00000001;

SHIFTING_IR  RXWRITE_POINTER
    SHIFT_LEFT: NEVER,
    SHIFT_RIGHT: RDONE=1,
    MAP: {<1:8> => RXWA},
    REGISTER: [SUGGEST: RESET=1,
               VALUE:10000000],
    INPUT: RXWA=00000001;

SHIFTER_WITH_VALUE_CHECK  R
    REGISTER:       [WRITE_LOWER: RDONE=1,
                     REFRESH: ALWAYS],
    SHIFT_RIGHT:    RXC=1,
    SHIFT_LEFT:     NEVER,
    VALUE:          /*SYNC*/,
    RESULT:         IS_SYNC,
    INPUT:          RX=1;

LEFT_RIGHT_SHIFT  TCOUNT
    INPUT_REGISTER:  [SUGGEST: RESET=1 OR HUNT_MODE=1,
                      VALUE: 00000001,
                      REFRESH: ALWAYS],
    SHIFT_LEFT:      TXC=1,
    SHIFT_RIGHT:     NEVER,
    MSB:             TDONE;

LEFT_RIGHT_SHIFT  RCOUNT
    INPUT_REGISTER:  [SUGGEST: RESET=1,
                      VALUE: 00000001,
                      REFRESH: ALWAYS],
```

```
    SHIFT_LEFT:     RXC=1,
    SHIFT_RIGHT:    NEVER,
    MSB:            RDONE;

END
```

9.5: A Microprogrammed Microprocessor

In this next example, we will see how a silicon compiler allows the user to explore alternate system architectures. We will design a microprogrammed microprocessor system, similar to the OM2 [15][16] system designed at Caltech. The basic architectural plan of the OM system is shown in figure 9-7. We have a datapath chip, which contains the scratchpad registers and ALU for the system, a microcode controller, which generates microcode addresses, and a microcode memory, which contains the instruction code for the machine. Surrounding these three modules are application dependent peripheral circuits. The basic system communicates with the peripheral circuitry across two 16-bit data buses, called the Left bus and the Right bus.

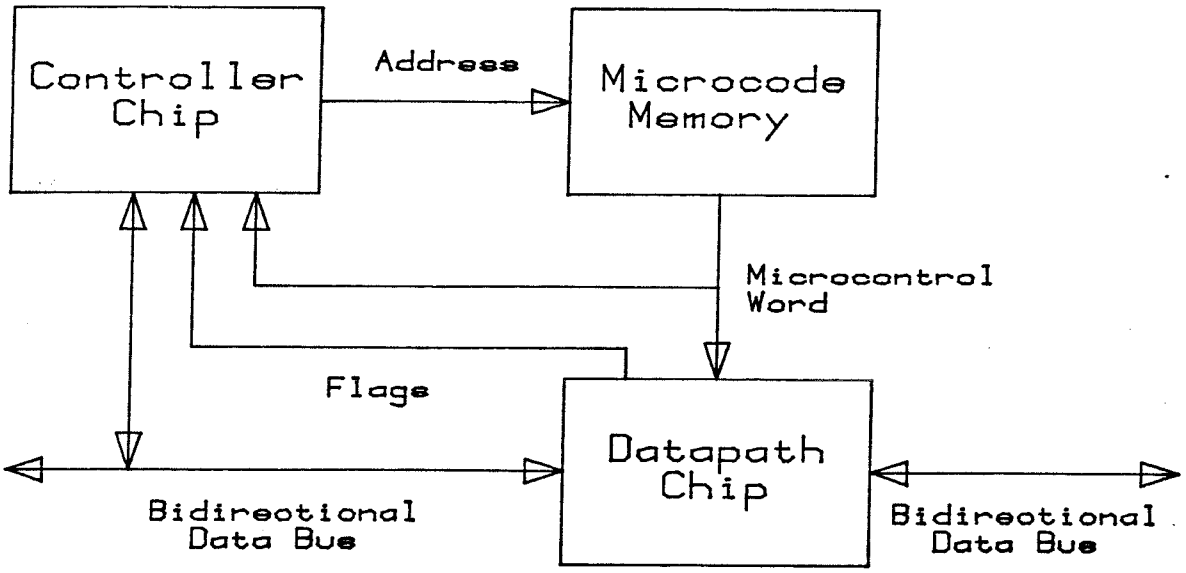


Fig. 9-7: OM System Block Diagram

We will begin by designing a controller chip. The controller provides microcode addresses. We need a register to hold the current microcode Program Counter (mPC). The usual operation of the controller will be to sequence through a series of microcode words, so the mPC will need an incrementer. If we used an adder instead of an incrementer, we can perform relative microcode branches. Under normal

operation, one input to the adder can be set to the value 1, so that the adder performs the increment operation. To branch, we merely load this adder input with the offset. To do a jump, we can force new data into the mPC register. By including a small stack on the chip, we can have subroutines in our microcode.

Based upon these desires, we can design a Register Transfer (RT) level diagram of the datapath, as shown in figure 9-8. We have drawn each of the registers and transfer paths. The transfer paths have been labeled to aid in the description of the chip operation. The upper bus is used to transfer the new mPC to the PORT unit, which drives the address lines of the microcode memory (note: the least-significant address is connected to the PHI-2 clock line, so that two words are read from the microcode memory every clock cycle). Since we want a new mPC value each clock cycle, the A control line should be high every clock cycle, and one of C, D, or E should also be high. The mPC latch, which is one of the adder input registers, should also be loaded every clock cycle, so the B control line is always high, too. For normal operation, we want to increment the mPC value each clock cycle, so the OFFSET register should normally contain a value of 1, and the NEW mPC register, which is the adder output, should normally drive the upper bus. Therefore, the L control, which loads the OFFSET register with 1, and the C control lines should normally be high. To perform a branch, we want to load the OFFSET register with the data in the IO PORT. This transfer is done by enabling the F and J control lines. To do a jump, we wish to directly transfer the IO PORT data into the mPC, so we enable the E control line instead of the C control line.

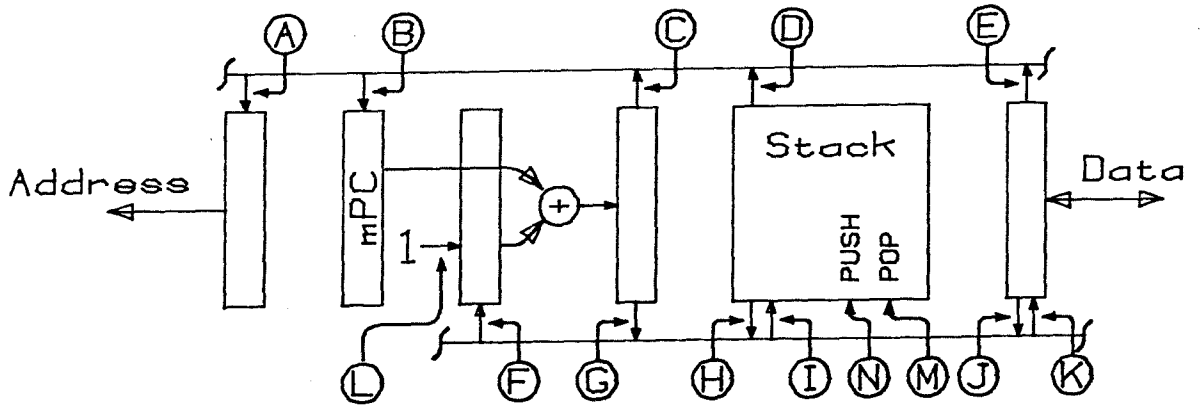


Fig. 9-8: Controller Register Transfer Diagram

The STACK unit allows calls and returns in the microcode. To perform a CALL operation, we need to push the NEW_mPC value onto the stack and load the mPC from the IO_PORT. This operation requires setting the G, I, N, and E control lines high. To perform a RETURN operation, we simply pop the top value off the STACK and into the mPC. Setting D and M high will perform this transfer.

We have described five operations performed by the controller chip, which means that a 3-bit microcode field is required to specify the operation. We can have up to eight operations specified by the 3-bit field, so we can add three more instructions to the controller's repertoire without impacting the microcode cost. If we can perform these new operations with the existing controller hardware, these new instructions are virtually free. One operation we may wish to have is a SAVE operation, which will push new data unto the STACK. This operation allows us to store a jump address in the controller chip several clock cycles before the jump is to occur. When the time comes to jump, the RETURN instruction will transfer the jump address to the mPC. We may like to use the two remaining instructions as loop control operations. One of the operations would be used at the start of the loop, the other at the end. The form of loop we will implement is a DO loop. The DO instruction will push the NEW_mPC value on the stack, and the ENDDO instruction will move the top-of-stack value into the mPC.

To allow conditional operations, there will be a condition input to the chip. If the condition is TRUE (i.e. the pin is high), the instructions will be executed as stated above. If the condition is FALSE, the normal operation, which increments the current mPC value, will be executed. If the ENDDO instruction is executed when the condition is FALSE, we will say that an UNDO instruction is executed, which causes the controller to 'fall out' of the loop. We will increment the mPC value and discard the top value on the STACK.

The following table summarizes the driving functions for each of the control lines.

Operation	Condition	Active Control Lines	Optional Active Controls
NOP	TRUE	A,B,L,C	G,H,J
	FALSE	A,B,L,C	G,H,J
JUMP	TRUE	A,B,E	L,F,G,H,J
	FALSE	A,B,L,C	G,H,J

CALL	TRUE	A,B,E,L,G,I,N	
	FALSE	A,B,L,C	G,H,J
RETURN	TRUE	A,B,D,M	L,F,G,H,J
	FALSE	A,B,L,C	G,H,J
BRANCH	TRUE	A,B,C,F,J	
	FALSE	A,B,L,C	G,H,J
SAVE	TRUE	A,B,C,L,I,N,J	
	FALSE	A,B,L,C	G,H,J
DO	TRUE	A,B,C,L,G,I,N	
	FALSE	A,B,L,C	G,H,J
ENDDO (UNDO)	TRUE	A,B,D	G,H,J,L,F
	FALSE	A,B,L,C,M,H,K	

The translation of this chip description into the Bristle Blocks specification language is straightforward. The Bristle Blocks input is listed here. Bristle Blocks compiled the chip in 4.06 CPU minutes, the chip area was 171 by 195 mil, and the power requirements were 88.8 ma.

```
NAME CONTROLLER 16;

FIELD OP<1:3>,CONDITION<4>,LOAD<5>,DRIVE<6>;

MACRO NOP()     % OP=000 OR  CONDITION=0 %
MACRO JUMP()    % OP=001 AND CONDITION=1 %
MACRO CALL()    % OP=010 AND CONDITION=1 %
MACRO RETURN()  % OP=011 AND CONDITION=1 %
MACRO BRANCH()  % OP=100 AND CONDITION=1 %
MACRO SAVE()    % OP=101 AND CONDITION=1 %
MACRO DO()      % OP=110 AND CONDITION=1 %
MACRO ENDDO()   % OP=111 AND CONDITION=1 %
MACRO UNDO()    % OP=111 AND CONDITION=0 %

OUTPUT_PORT PC
    REGISTER: [READ_UPPER:ALWAYS];

ADDER NEW_PC
    INPUT_A: [READ_UPPER:ALWAYS],
    INPUT_B: [READ_LOWER: /*BRANCH*/, SUGGEST: NOT(/*BRANCH*/),
              VALUE:0000000000000001],
    LOAD:ALWAYS,
    OUTPUT_REGISTER:
        [WRITE_UPPER: /*NOP*/ OR /*BRANCH*/ OR /*SAVE*/ OR /*DO*/,
         WRITE_LOWER: NOT(/*UNDO*/ OR /*BRANCH*/ OR /*SAVE*/)];

PRECHARGE_BOTH PCHG;

STACK STACK
    DEPTH:16,
    TOP: [WRITE_UPPER: /*RETURN*/ OR /*ENDDO*/,
          WRITE_LOWER:/*UNDO*/,
          READ_LOWER: /*SAVE*/ OR /*CALL*/ OR /*DO*/,
          REFRESH: NOT( /*RETURN*/ OR /*UNDO*/ )],
    POP: /*RETURN*/ OR /*UNDO*/,
```

```
    PUSH: /*CALL*/ OR /*SAVE*/ OR /*DO*/;

IO_PORT DATA
    OUTPUT_REGISTER: [WRITE_UPPER: /*CALL*/ OR /*JUMP*/,
                      WRITE_LOWER:/*BRANCH*/ OR /*SAVE*/,
                      READ_LOWER: /*UNDO*/,
                      REFRESH:ALWAYS],
    LOAD: LOAD=I,
    DRIVE: DRIVE=I;

END
```

We can experiment to see how the stack size affects the area and power requirements of the chip. After compiling controllers with stack depths of 8 and 12, and interpolating and extrapolating the results, the power requirements were found to be approximately 28.8 + 3.75*depth ma. and the width of the chip was found to be approximately 83 + 5.5*depth mils.

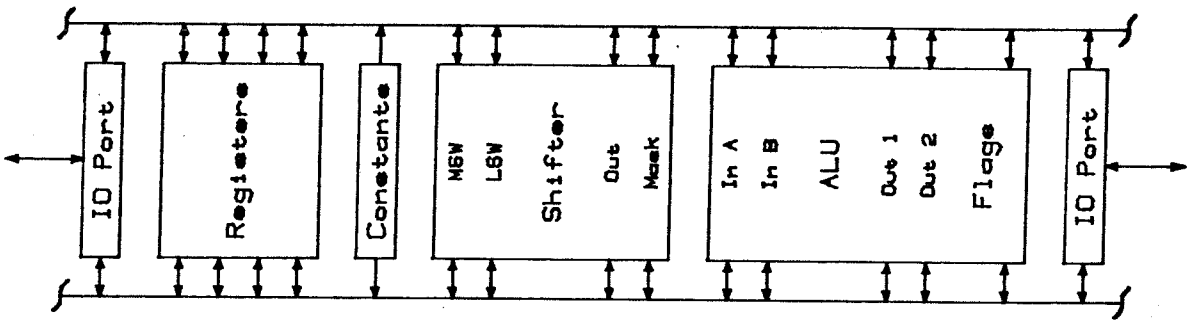
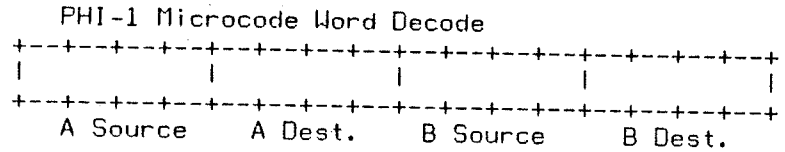
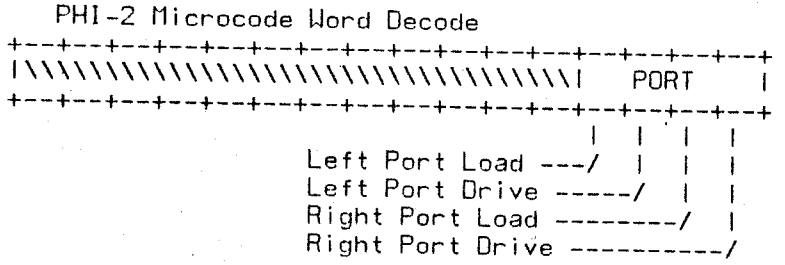


Fig. 9-9: Datachip Block Diagram

Next, we can design the datachip for the microprogrammed processor. We need two bi-directional data ports, some general purpose registers, a fixed constant source, a shifter, and an Arithmetic/Logic Unit (ALU). A block diagram of the proposed chip is shown in figure 9-9. Each of the registers in the chip communicate with two data buses. We can assign a unique bus address for each of the registers. We can decode the microcode to allow two transfers per clock cycle. There are 16 data sources for each bus, and 15 data sinks (due to the constant value). Hence, we can decode a 16-bit microcode word as four 4-bit address field. One address specifies the upper bus (A bus) source, another specifies the destination. We decode the two lower bus (B bus) addresses in the same manner.



The left and right data ports are implemented as single-register IO_PORT. The left port's register is assigned bus address 2, while the right port's register is assigned bus address 3. The load and drive controls for these ports come directly from a microcode field of the PHI-2 microcode word. When the least-significant bit of the PORT field is high, the right port will drive from its internal register to the pads. The next bit controls when the right port reads from the pads into the internal register. The two high-order bits of the PORT field control the left port.



It is useful to have a source of constant data in the datapath. Besides giving us a known value, a constant 'register' does not read data from the bus. Hence, we have an unassigned bus destination address. If we do not wish to perform a transfer on one of the two data buses, we can 'transfer' into this non-existing register. We must choose what our two constant values will be. To aid in the generation of masks and shift operations, the upper bus constant will be 0 and the lower bus constant will be -1.

We will use a barrel shifter for the shift element. The MASKED_SHIFTER has registers for the input most-significant word and least-signficant word, along with an output register and mask register. With the masked writing capabilities, we can do field extractions and field insertions. We have a 4-bit shift constant field in the microcode, along with a two bit field specifing how to load under mask. If the two mask bits are low, the shifter does not write into its output register. If both bits are high, the shifter directly loads its output register. If the lower bit of the mask op field is high, the shifter writes into the output register bits whose corresponding mask register bits are low. If the upper bit of the mask op field is high, the shifter writes into the output register bits whose corresponding mask register bits are high.

```
   PHI-2 Microcode Word Decode
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|   SHIFT   |\\\\\\\\\\\| MASK|\\\\\\|   PORT    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

                          0  0   No Write
                          0  1   Write where mask is low
                          1  0   Write where mask is high
                          1  1   Write every bit
```

For the ALU, we use the Bristle Blocks ALU_WITH_FLAGS element. This element has two input registers, either one or two output registers, and a flag register. We will use both output registers. The CARRY, MSB, and ZERO flags from the flag register will drive pads, so that external circuitry can sense the state of the flags. To allow external conditions to modify the ALU operations, we will have a condition input which drives the ALU operation decode. The ALU portions of the PHI-2 microcode word are listed here.

```
   PHI-2 Microcode Word Decode
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|   SHIFT   |    ALU    | MASK| LOAD|   PORT    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
            |  |  |  |        |  |
            |  |  |  |        |  \--- Load ALU output register 2
            |  |  |  |        \------ Load ALU output register 1
            0  0  0  0  Divide Step*
            0  0  0  1  Increment A
            0  0  1  0  Subtract with Borrow*
            0  0  1  1  Subtract
            0  1  0  0  Add with Carry*
            0  1  0  1  Add
            0  1  1  0  Decrement A
            0  1  1  1  Negate A
            1  0  0  0  Multiply Step*
            1  0  0  1  Select A/B*
            1  0  1  0  OR
            1  0  1  1  AND
            1  1  0  0  A
            1  1  0  1  XOR
            1  1  1  0  TEST
            1  1  1  1  Compliment A

   * indicates that the operation performed is a function of the
     condition input.
```

Finally, we add the general purpose registers. We have four free bus addresses, so we will add four registers to the chip. The Bristle Blocks specification for this chip is listed here. A chip enable input has been added to the chip specification. When chip enable is low, none of the registers' contents will be modified.

```
NAME DATAPATH 16;

FIELD A_SOURCE<1:4>,A_DEST<5:8>,B_SOURCE<9:12>,B_DEST<13:16>,
      ENABLE<17>,SHIFT_CONST<1:4>,ALU<5:8>,MASK<9:10>,LOAD<11:12>,
      PORT<13:16>,CONDITION<18>,ALU_OP= ALU & CONDITION;

MACRO ADDR(ADDR)
%    [READ_UPPER: A_DEST=?ADDR? AND ENABLE=I,
      READ_LOWER: B_DEST=?ADDR? AND ENABLE=I,
      WRITE_UPPER: A_SOURCE=?ADDR?,
      WRITE_LOWER: B_SOURCE=?ADDR?,
      REFRESH: ALWAYS ]  %


IO_PORT LEFT_PORT
    OUTPUT_REGISTER: /*ADDR(0010)*/,
    LOAD:            PORT=1XXX AND ENABLE=I,
    DRIVE:           PORT=X1XX AND ENABLE=I;

REGISTER R12 OPTIONS:/*ADDR(1100)*/;
REGISTER R13 OPTIONS:/*ADDR(1101)*/;
REGISTER R14 OPTIONS:/*ADDR(1110)*/;
REGISTER R15 OPTIONS:/*ADDR(1111)*/;

ROM_PAIR C0
    LEFT_ENABLE: A_SOURCE=0000, LEFT_UPPER:0000000000000000,
    RIGHT_ENABLE:B_SOURCE=0000, RIGHT_LOWER:1111111111111111;

PRECHARGE_BOTH PCHG;

MASKED_SHIFTER SHIFTER
    MOST_SIGNIFICANT_WORD:  /*ADDR(1000)*/,
    LEAST_SIGNIFICANT_WORD: /*ADDR(1001)*/,
    OUTPUT_REGISTER:        /*ADDR(1010)*/,
    MASK_REGISTER:          /*ADDR(1011)*/,
    SHIFT_CONSTANT:         SHIFT_CONST,
    LOAD_IF_0:              MASK=X1 AND ENABLE=I,
    LOAD_IF_1:              MASK=1X AND ENABLE=I;

ALU_WITH_FLAGS ALU
    INPUT_A:         /*ADDR(0100)*/,
    INPUT_B:         /*ADDR(0101)*/,
    OUTPUT_1:        /*ADDR(0110)*/,
    OUTPUT_2:        /*ADDR(0111)*/,
    FLAGS:           /*ADDR(0001)*/,
    LOAD_FLAGS:      LOAD=1X AND ENABLE=I,
    WRITE_OUTPUT_1: LOAD=1X AND ENABLE=I,
    WRITE_OUTPUT_2: LOAD=X1 AND ENABLE=I,
    TO_CONTROL:      {<1,2,9>=>PAD},
    DECODE: ALU_OP
        <0> => SUBTRACT
        <1> => ADD
        <2,3> => INCREMENT_A
        <4> => SUBTRACT
        <5> => SUB_W_BORROW
        <6,7> => SUBTRACT
        <8> => ADD
        <9> => ADD_W_CARRY
        <10,11> => ADD
        <12,13> => DECREMENT_A
        <14,15> => NEGATE_A
```

```
        <16> => SETA
        <17> => ADD
        <18> => SETA
        <19> => SETB
        <20,21> => OR
        <22,23> => AND
        <24,25> => SETA
        <26,27> => XOR
        <28,29> => TEST
        <30,31> => SETCA;

IO_PORT RIGHT_PORT
    OUTPUT_REGISTER: /*ADDR(0011)*/,
    LOAD:            PORT=XXIX AND ENABLE=I,
    DRIVE:           PORT=XXXI AND ENABLE=I;

END.
```

This chip was compiled in 6.2 minutes, resulting in a chip whose area was 203 by 276 mil, and whose power consumption was 96 ma.

The microcode writers were unsatisfied with the limited number of general purpose registers. There were only four registers in the original chip specification that were not used by the data processing elements, although one of the ALU output registers can be used if the user never loaded the register from the ALU. The system designers, on the other hand, wished to keep the microcode width at 16-bits, which presented an addressing problem. How can we address more registers in the datapath. Four schemes were pursued which lead to an increased register count in the data chip.

The first scheme involved rearranging the PHI-1 microcode word. Instead of having 4-bit addressing for both the A and B buses, we tried having 5-bit addresses for the A bus and 3-bit addresses for the B bus. We would limit the number of registers which could communicate across the lower bus and at the same time increase the number of registers which can use the A bus. With this technique, we were able to add 16 more registers to the chip. The chip area increased to 229 by 272 mil, and the power consumption rose to 126 ma. The specification for this new chip is listed here.

```
NAME DATAPATH2 16;

FIELD A_SOURCE<1:5>,A_DEST<6:10>,B_SOURCE<11:13>,B_DEST<14:16>,
      ENABLE<17>,SHIFT_CONST<1:4>,ALU<5:8>,MASKS<9:10>,LOAD<11:12>,
      PORT<13:16>,CONDITION<18>,ALU_OP= ALU & CONDITION;

MACRO ADDR_BOTH(ADDR)
%    [READ_UPPER: A_DEST=00?ADDR? AND ENABLE=I,
```

```
     READ_LOWER: B_DEST=?ADDR? AND ENABLE=I,
     WRITE_UPPER: A_SOURCE=00?ADDR?,
     WRITE_LOWER: B_SOURCE=?ADDR?,
     REFRESH: ALWAYS ]  %

MACRO ADDR_A(ADDR)
%    [READ_UPPER: A_DEST=?ADDR? AND ENABLE=I,
     WRITE_UPPER: A_SOURCE=?ADDR?,
     REFRESH: ALWAYS ]  %


IO_PORT LEFT_PORT
    OUTPUT_REGISTER: /*ADDR_BOTH(III)*/,
    LOAD:            PORT=IXXX AND ENABLE=I,
    DRIVE:           PORT=XIXX AND ENABLE=I;

REGISTER R1 OPTIONS:/*ADDR_BOTH(001)*/;
REGISTER R2 OPTIONS:/*ADDR_BOTH(010)*/;
REGISTER R3 OPTIONS:/*ADDR_BOTH(011)*/;
REGISTER R15 OPTIONS:/*ADDR_A(01111)*/;
REGISTER R16 OPTIONS:/*ADDR_A(10000)*/;
REGISTER R17 OPTIONS:/*ADDR_A(10001)*/;
REGISTER R18 OPTIONS:/*ADDR_A(10010)*/;
REGISTER R19 OPTIONS:/*ADDR_A(10011)*/;
REGISTER R20 OPTIONS:/*ADDR_A(10100)*/;
REGISTER R21 OPTIONS:/*ADDR_A(10101)*/;
REGISTER R22 OPTIONS:/*ADDR_A(10110)*/;
REGISTER R23 OPTIONS:/*ADDR_A(10111)*/;
REGISTER R24 OPTIONS:/*ADDR_A(11000)*/;
REGISTER R25 OPTIONS:/*ADDR_A(11001)*/;
REGISTER R26 OPTIONS:/*ADDR_A(11010)*/;
REGISTER R27 OPTIONS:/*ADDR_A(11011)*/;
REGISTER R28 OPTIONS:/*ADDR_A(11100)*/;
REGISTER R29 OPTIONS:/*ADDR_A(11101)*/;
REGISTER R30 OPTIONS:/*ADDR_A(11110)*/;
REGISTER R31 OPTIONS:/*ADDR_A(11111)*/;

ROM_PAIR C0
    LEFT_ENABLE: A_SOURCE=00000, LEFT_UPPER:0000000000000000,
    RIGHT_ENABLE:B_SOURCE=000, RIGHT_LOWER:IIIIIIIIIIIIIIII;

PRECHARGE_BOTH PCHG;

MASKED_SHIFTER SHIFTER
    MOST_SIGNIFICANT_WORD:  /*ADDR_A(01000)*/,
    LEAST_SIGNIFICANT_WORD: /*ADDR_A(01001)*/,
    OUTPUT_REGISTER:        /*ADDR_A(01010)*/,
    MASK_REGISTER:          /*ADDR_A(01011)*/,
    SHIFT_CONSTANT:         SHIFT_CONST,
    LOAD_IF_0:              MASKS=XI AND ENABLE=I,
    LOAD_IF_1:              MASKS=IX AND ENABLE=I;

ALU_WITH_FLAGS ALU
    INPUT_A:         /*ADDR_BOTH(100)*/,
    INPUT_B:         /*ADDR_A(01100)*/,
    OUTPUT_1:        /*ADDR_BOTH(101)*/,
    OUTPUT_2:        /*ADDR_A(01101)*/,
    FLAGS:           /*ADDR_A(01110)*/,
    LOAD_FLAGS:      LOAD=IX AND ENABLE=I,
    WRITE_OUTPUT_1: LOAD=IX AND ENABLE=I,
    WRITE_OUTPUT_2: LOAD=XI AND ENABLE=I,
```

```
    TO_CONTROL:       {<1,2,9>=>PAD},
    DECODE: ALU_OP
        <0> => SUBTRACT
        <1> => ADD
        <2,3> => INCREMENT_A
        <4> => SUBTRACT
        <5> => SUB_W_BORROW
        <6,7> => SUBTRACT
        <8> => ADD
        <9> => ADD_W_CARRY
        <10,11> => ADD
        <12,13> => DECREMENT_A
        <14,15> => NEGATE_A
        <16> => SETA
        <17> => ADD
        <18> => SETA
        <19> => SETB
        <20,21> => OR
        <22,23> => AND
        <24,25> => SETA
        <26,27> => XOR
        <28,29> => TEST
        <30,31> => SETCA;

IO_PORT RIGHT_PORT
    OUTPUT_REGISTER: /*ADDR_BOTH(III)*/,
    LOAD:            PORT=XXIX AND ENABLE=I,
    DRIVE:           PORT=XXXI AND ENABLE=I;

END
```

Another proposed method for increasing the number of datapath registers was to add backup registers, similar to the alternate register set in the Zilog Z80 chip. We would have backup registers for each of the four general purpose registers, and when a swap instruction was executed, the register pairs would swap data values. For this method to work, we need a bit to indicate when to swap. We can free up one PHI-2 bit if we only have one ALU output register. The load bit for that register can then be used as the SWAP bit. The area for this new chip is 220 by 280 mil, and the power consumption is 114 ma.

```
NAME DATAPATH3 16;

FIELD A_SOURCE<1:4>,A_DEST<5:8>,B_SOURCE<9:12>,B_DEST<13:16>,
      ENABLE<17>,SHIFT_CONST<1:4>,ALU<5:8>,MASKS<9:10>,LOAD<11>,SWAP<12>,
      PORT<13:16>,CONDITION<18>,ALU_OP= ALU & CONDITION;

MACRO ADDR(ADDR)
%    [READ_UPPER: A_DEST=?ADDR? AND ENABLE=I,
     READ_LOWER: B_DEST=?ADDR? AND ENABLE=I,
     WRITE_UPPER: A_SOURCE=?ADDR?,
     WRITE_LOWER: B_SOURCE=?ADDR?,
     REFRESH: ALWAYS ]  %

MACRO SWAP(ADDR)
%SWAPPING_REGISTERS R?ADDR?
```

```
    LEFT: [REFRESH: LOADS=XXXO,
           READ_UPPER: A_DEST=?ADDR? AND ENABLE=I,
           READ_LOWER: B_DEST=?ADDR? AND ENABLE=I,
           WRITE_UPPER: A_SOURCE=?ADDR?,
           WRITE_LOWER: B_SOURCE=?ADDR?],
    RIGHT:[REFRESH: LOADS=XXXO],
    RIGHT_TO_LEFT: SWAP=I,
    LEFT_TO_RIGHT: SWAP=I;  %


IO_PORT LEFT_PORT
    OUTPUT_REGISTER: /*ADDR(OOIO)*/,
    LOAD:            PORT=IXXX AND ENABLE=I,
    DRIVE:           PORT=XIXX AND ENABLE=I;

/*SWAP(IIOO)*/
/*SWAP(IIOI)*/
/*SWAP(IIIO)*/
/*SWAP(IIII)*/

ROM_PAIR C0
    LEFT_ENABLE: A_SOURCE=0000, LEFT_UPPER:0000000000000000,
    RIGHT_ENABLE:B_SOURCE=0000, RIGHT_LOWER:IIIIIIIIIIIIIIII;

PRECHARGE_BOTH PCHG;

MASKED_SHIFTER SHIFTER
    MOST_SIGNIFICANTWORD:  /*ADDR(IOOO)*/,
    LEAST_SIGNIFICANT_WORD: /*ADDR(IOOI)*/,
    OUTPUT_REGISTER:        /*ADDR(IOIO)*/,
    MASK_REGISTER:          /*ADDR(IOII)*/,
    SHIFT_CONSTANT:         SHIFT_CONST,
    LOAD_IF_0:              MASKS=XI AND ENABLE=I,
    LOAD_IF_1:              MASKS=IX AND ENABLE=I;

ALU_WITH_FLAGS ALU
    INPUT_A:         /*ADDR(OIOO)*/,
    INPUT_B:         /*ADDR(OIOI)*/,
    OUTPUT_1:        /*ADDR(OIIO)*/,
    FLAGS:           /*ADDR(OOOI)*/,
    LOAD_FLAGS:      LOAD=I AND ENABLE=I,
    WRITE_OUTPUT_1: LOAD=I AND ENABLE=I,
    TO_CONTROL:      {<1,2,9>=>PAD},
    DECODE: ALU_OP
        <0> => SUBTRACT
        <1> => ADD
        <2,3> => INCREMENT_A
        <4> => SUBTRACT
        <5> => SUB_W_BORROW
        <6,7> => SUBTRACT
        <8> => ADD
        <9> => ADD_W_CARRY
        <10,11> => ADD
        <12,13> => DECREMENT_A
        <14,15> => NEGATE_A
        <16> => SETA
        <17> => ADD
        <18> => SETA
        <19> => SETB
        <20,21> => OR
        <22,23> => AND
```

```
        <24,25> => SETA
        <26,27> => XOR
        <28,29> => TEST
        <30,31> => SETCA;

REGISTER R13 OPTIONS:/*ADDR(0111)*/;

IO_PORT RIGHT_PORT
    OUTPUT_REGISTER: /*ADDR(0011)*/,
    LOAD:            PORT=XXIX AND ENABLE=I,
    DRIVE:           PORT=XXXI AND ENABLE=I;

END
```

A simpler proposal was to share the shifter and ALU input registers, thereby freeing up two bus addresses. Since is is difficult to physically share the registers, we can share the registers in a logical sense: ALU input register A and shifter MSW register will have the same bus destination address, but only the ALU register will write the bus. Whenever a transfer is made to the ALU input register, the shifter register will also load. Whenever a transfer is made from the ALU input register, only the ALU register will write the bus. This chip has an area of 209 by 272 mil, and a power consumption of 100 ma.

```
NAME DATAPATH4 16;

FIELD A_SOURCE<1:4>,A_DEST<5:8>,B_SOURCE<9:12>,B_DEST<13:16>,
      ENABLE<17>,SHIFT_CONST<1:4>,ALU<5:8>,MASKS<9:10>,LOAD<11:12>,
      PORT<13:16>,CONDITION<18>,ALU_OP= ALU & CONDITION;

MACRO ADDR(ADDR)
%   [READ_UPPER: A_DEST=?ADDR? AND ENABLE=I,
     READ_LOWER: B_DEST=?ADDR? AND ENABLE=I,
     WRITE_UPPER: A_SOURCE=?ADDR?,
     WRITE_LOWER: B_SOURCE=?ADDR?,
     REFRESH: ALWAYS ]  %

MACRO HALF(ADDR)
%   [READ_UPPER: A_DEST=?ADDR? AND ENABLE=I,
     READ_LOWER: B_DEST=?ADDR? AND ENABLE=I,
     REFRESH: ALWAYS ]  %


IO_PORT LEFT_PORT
    OUTPUT_REGISTER: /*ADDR(0010)*/,
    LOAD:            PORT=IXXX AND ENABLE=I,
    DRIVE:           PORT=XIXX AND ENABLE=I;

REGISTER R8  OPTIONS:/*ADDR(1000)*/;
REGISTER R9  OPTIONS:/*ADDR(1001)*/;
REGISTER R12 OPTIONS:/*ADDR(1100)*/;
REGISTER R13 OPTIONS:/*ADDR(1101)*/;
REGISTER R14 OPTIONS:/*ADDR(1110)*/;
REGISTER R15 OPTIONS:/*ADDR(1111)*/;
```

```
ROM_PAIR C0
    LEFT_ENABLE: A_SOURCE=0000, LEFT_UPPER:0000000000000000,
    RIGHT_ENABLE:B_SOURCE=0000, RIGHT_LOWER:IIIIIIIIIIIIIIII;

PRECHARGE_BOTH PCHG;

MASKED_SHIFTER SHIFTER
    MOST_SIGNIFICANT_WORD:  /*HALF(0100)*/,
    LEAST_SIGNIFICANT_WORD: /*HALF(0101)*/,
    OUTPUT_REGISTER:        /*ADDR(1010)*/,
    MASK_REGISTER:          /*ADDR(1011)*/,
    SHIFT_CONSTANT:         SHIFT_CONST,
    LOAD_IF_0:              MASKS=XI AND ENABLE=I,
    LOAD_IF_1:              MASKS=IX AND ENABLE=I;

ALU_WITH_FLAGS ALU
    INPUT_A:          /*ADDR(0100)*/,
    INPUT_B:          /*ADDR(0101)*/,
    OUTPUT_1:         /*ADDR(0110)*/,
    OUTPUT_2:         /*ADDR(0111)*/,
    FLAGS:            /*ADDR(0001)*/,
    LOAD_FLAGS:       LOAD=IX AND ENABLE=I,
    WRITE_OUTPUT_1: LOAD=IX AND ENABLE=I,
    WRITE_OUTPUT_2: LOAD=XI AND ENABLE=I,
    TO_CONTROL:       {<1,2,9>=>PAD},
    DECODE: ALU_OP
        <0> => SUBTRACT
        <1> => ADD
        <2,3> => INCREMENT_A
        <4> => SUBTRACT
        <5> => SUB_W_BORROW
        <6,7> => SUBTRACT
        <8> => ADD
        <9> => ADD_W_CARRY
        <10,11> => ADD
        <12,13> => DECREMENT_A
        <14,15> => NEGATE_A
        <16> => SETA
        <17> => ADD
        <18> => SETA
        <19> => SETB
        <20,21> => OR
        <22,23> => AND
        <24,25> => SETA
        <26,27> => XOR
        <28,29> => TEST
        <30,31> => SETCA;

IO_PORT RIGHT_PORT
    OUTPUT_REGISTER: /*ADDR(0011)*/,
    LOAD:            PORT=XXIX AND ENABLE=I,
    DRIVE:           PORT=XXXI AND ENABLE=I;

END
```

The final proposal was to add a stack to the chip. We would again have to remove one of the ALU output registers to free up a control bit for the POP line. This stack pushes data whenever the top register is written to, and pops data whenever the

POP signal is high. The top of stack register can be read independent of whether the stack POPs or not. For an 8-deep stack, the chip area is 231 by 279 mil and the power consumption is 129 ma.

```
NAME DATAPATH5 16;

FIELD A_SOURCE<1:4>,A_DEST<5:8>,B_SOURCE<9:12>,B_DEST<13:16>,
      ENABLE<17>,SHIFT_CONST<1:4>,ALU<5:8>,MASKS<9:10>,LOAD<11>,POP<12>,
      PORT<13:16>,CONDITION<18>,ALU_OP= ALU & CONDITION;

MACRO ADDR(ADDR)
%    [READ_UPPER: A_DEST=?ADDR? AND ENABLE=I,
      READ_LOWER: B_DEST=?ADDR? AND ENABLE=I,
      WRITE_UPPER: A_SOURCE=?ADDR?,
      WRITE_LOWER: B_SOURCE=?ADDR?,
      REFRESH: ALWAYS ]  %


IO_PORT LEFT_PORT
    OUTPUT_REGISTER: /*ADDR(0010)*/,
    LOAD:            PORT=1XXX AND ENABLE=I,
    DRIVE:           PORT=X1XX AND ENABLE=I;

REGISTER R12 OPTIONS: /*ADDR(1100)*/;
REGISTER R13 OPTIONS: /*ADDR(1101)*/;
REGISTER R14 OPTIONS: /*ADDR(1110)*/;
REGISTER R15 OPTIONS: /*ADDR(1111)*/;

ROM_PAIR C0
    LEFT_ENABLE: A_SOURCE=0000, LEFT_UPPER:0000000000000000,
    RIGHT_ENABLE:B_SOURCE=0000, RIGHT_LOWER:1111111111111111;

PRECHARGE_BOTH PCHG;

MASKED_SHIFTER SHIFTER
    MOST_SIGNIFICANTWORD:  /*ADDR(1000)*/,
    LEAST_SIGNIFICANT_WORD: /*ADDR(1001)*/,
    OUTPUT_REGISTER:        /*ADDR(1010)*/,
    MASK_REGISTER:          /*ADDR(1011)*/,
    SHIFT_CONSTANT:         SHIFT_CONST,
    LOAD_IF_0:              MASKS=X1 AND ENABLE=I,
    LOAD_IF_1:              MASKS=1X AND ENABLE=I;

ALU_WITH_FLAGS ALU
    INPUT_A:         /*ADDR(0100)*/,
    INPUT_B:         /*ADDR(0101)*/,
    OUTPUT_1:        /*ADDR(0110)*/,
    FLAGS:           /*ADDR(0001)*/,
    LOAD_FLAGS:      LOAD=I AND ENABLE=I,
    WRITE_OUTPUT_1: LOAD=I AND ENABLE=I,
    TO_CONTROL:      {<1,2,9>=>PAD},
    DECODE: ALU_OP
        <0> => SUBTRACT
        <1> => ADD
        <2,3> => INCREMENT_A
        <4> => SUBTRACT
        <5> => SUB_W_BORROW
```

```
        <6,7> => SUBTRACT
        <8> => ADD
        <9> => ADD_W_CARRY
        <10,11> => ADD
        <12,13> => DECREMENT_A
        <14,15> => NEGATE_A
        <16> => SETA
        <17> => ADD
        <18> => SETA
        <19> => SETB
        <20,21> => OR
        <22,23> => AND
        <24,25> => SETA
        <26,27> => XOR
        <28,29> => TEST
        <30,31> => SETCA;

STACK  STACK
    DEPTH: 8,
    TOP: /*ADDR(0111)*/,
    POP: POP=1,
    PUSH: A_DEST=0111 OR B_DEST=0111;

IO_PORT RIGHT_PORT
    OUTPUT_REGISTER: /*ADDR(0011)*/,
    LOAD:            PORT=XX1X AND ENABLE=1,
    DRIVE:           PORT=XXX1 AND ENABLE=1;

END
```

The following table summarizes the results of the datachip modification experiments.

Name	Number of Free Registers	Size x	Size y	Power
DATAPATH	4	203	276	96
DATAPATH2	20	229	272	126
DATAPATH3	5 with 4 backups	220	280	114
DATAPATH4	6	209	272	100
DATAPATH5	4 with 8-deep stack	231	279	129

Figure 9-10 shows the bounding boxes for each of these chips. Given this comparison data, the microcode designers and the fabrication engineers can haggle over the design specs.

Later that afternoon, the members of the market staff came by, expressing a desire for combining the controller and datachip onto a single chip. Additionally, the width of the microcode was to be narrowed from 24-bits to 16-bits. One of the two bi-directional buses could also be eliminated. Using a handy text editor, the controller specification was merged with one of the datapath specifications. Bristle Blocks compiled the new chip in 7 minutes. The chip size was 244 by 246 mil, and

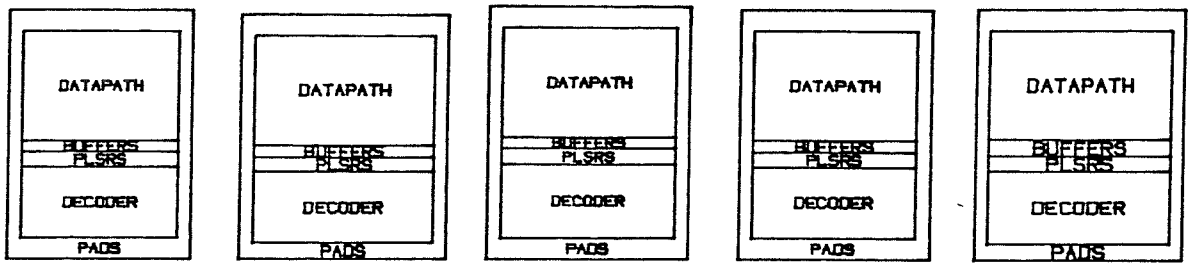


Fig. 9-10: Bounding Box Comparisions

the power consumption was 128 ma. The source code for the combined chip is listed here.

```
NAME COMBINED 16;

FIELD ADDRESS<1:3>,A_SOURCE<4:7>,A_DEST<8:11>,B_SOURCE<12:14>,
      B_DEST<15:16>,SHIFT_CONST<1:4>,SHIFT_LD<5:6>,ALU<7:10>,
      PORT<13:16>,CONDITION<17>,ALU_OP=ALU & CONDITION;

MACRO NOP()     % ADDRESS=000 OR  CONDITION=0 %
MACRO JUMP()    % ADDRESS=001 AND CONDITION=1 %
MACRO CALL()    % ADDRESS=010 AND CONDITION=1 %
MACRO RETURN()  % ADDRESS=011 AND CONDITION=1 %
MACRO BRANCH()  % ADDRESS=100 AND CONDITION=1 %
MACRO SAVE()    % ADDRESS=101 AND CONDITION=1 %
MACRO DO()      % ADDRESS=110 AND CONDITION=1 %
MACRO ENDDO()   % ADDRESS=111 AND CONDITION=1 %
MACRO UNDO()    % ADDRESS=111 AND CONDITION=0 %

MACRO REG_A(ADR)
    % READ_UPPER: A_DEST=?ADR?,
      WRITE_UPPER: A_SOURCE=?ADR?,
      REFRESH:ALWAYS %

MACRO REG_B_OUT(AADR,BADR)
    % [/*REG_A(?AADR?)*/, WRITE_LOWER:B_SOURCE=?BADR?] %

MACRO REG_B_IN(AADR,BADR)
    % [/*REG_A(?AADR?)*/, READ_LOWER:B_DEST=?BADR?] %

MACRO REG_A_ONLY(ADR)
    % [/*REG_A(?ADR?)*/] %

MACRO PORT_OUT() % PORT=000X %

MACRO PORT_IN() % NOT(/*PORT_OUT*/) %


OUTPUT_PORT ADDRESS
    REGISTER: [READ_UPPER: /*NOP*/ OR /*DO*/ OR /*BRANCH*/ OR /*SAVE*/,
               READ_LOWER: /*JUMP*/ OR /*CALL*/ OR /*RETURN*/ OR /*ENDDO*/];
```

```
ADDER NEW_PC
    INPUT_A: [READ_UPPER: /*NOP*/ OR /*DO*/ OR /*BRANCH*/ OR /*SAVE*/,
              READ_LOWER: /*JUMP*/ OR /*CALL*/ OR /*RETURN*/ OR /*ENDDO*/],
    INPUT_B: [READ_LOWER: /*BRANCH*/,
              SUGGEST: NOT(/*BRANCH*/),
              VALUE: 0000000000000001],
    LOAD: ALWAYS,
    OUTPUT_REGISTER: [WRITE_UPPER: ALWAYS];

STACK PC_STACK
    DEPTH:8,
    TOP: [READ_UPPER: /*CALL*/ OR /*DO*/,
          READ_LOWER: /*SAVE*/,
          WRITE_LOWER: /*RETURN*/ OR /*ENDDO*/],
    PUSH: /*CALL*/ OR /*DO*/ OR /*SAVE*/,
    POP: /*RETURN*/ OR /*UNDO*/;

PRECHARGE_AND_BREAK_UPPER PCHG1;

REGISTER LINK OPTIONS:
    [WRITE_LOWER: /*JUMP*/ OR /*CALL*/ OR /*BRANCH*/ OR /*SAVE*/,
     WRITE_UPPER: A_SOURCE=0001, READ_UPPER: A_DEST=0001];

PRECHARGE_AND_BREAK_LOWER  PCHG2;

PRECHARGE_BOTH  PCHG3;

ALU_WITH_FLAGS  ALU
    INPUT_A: /*REG_B_IN(1000,01)*/,
    INPUT_B: /*REG_A_ONLY(1001)*/,
    OUTPUT_1: /*REG_B_OUT(1010,100)*/,
    FLAGS: /*REG_A_ONLY(1011)*/,
    LOAD_FLAGS: NOT(ALU=1111),
    WRITE_OUTPUT_1: NOT(ALU=1111),
    TO_CONTROL:      {<1,2,9>=>PAD1,
    DECODE: ALU_OP
        <0> => SUBTRACT
        <1> => ADD
        <2,3> => INCREMENT_A
        <4> => SUBTRACT
        <5> => SUB_W_BORROW
        <6,7> => SUBTRACT
        <8> => ADD
        <9> => ADD_W_CARRY
        <10,11> => ADD
        <12,13> => DECREMENT_A
        <14,15> => NEGATE_A
        <16> => SETA
        <17> => ADD
        <18> => SETA
        <19> => SETB
        <20,21> => OR
        <22,23> => AND
        <24,25> => SETA
        <26,27> => XOR
        <28,29> => TEST
        <30,31> => DONT_CARE;

MASKED_SHIFTER  SHIFTER
    MOST_SIGNIFICANT_WORD:  /*REG_A_ONLY(0100)*/,
    LEAST_SIGNIFICANT_WORD: /*REG_B_IN(0101,10)*/,
```

```
    OUTPUT_REGISTER:        /*REG_B_OUT(0110,101)*/,
    MASK_REGISTER:          /*REG_A_ONLY(0111)*/,
    SHIFT_CONSTANT:         SHIFT_CONST,
    LOAD_IF_0:              SHIFT_LD=X1,
    LOAD_IF_1:              SHIFT_LD=1X;

ROM_PAIR  C0
    LEFT_ENABLE: A_SOURCE=0000,
    RIGHT_ENABLE:B_SOURCE=110,
    LEFT_UPPER:  0000000000000000,
    RIGHT_LOWER: 1111111111111111;

REGISTER  R2_REG
    OPTIONS: [WRITE_UPPER: A_SOURCE=0010,
              READ_UPPER: A_DEST=0010,
              REFRESH:ALWAYS,
              READ_LOWER: B_DEST=00 AND NOT(B_SOURCE=000),
              WRITE_LOWER: B_SOURCE=000];

REGISTER  R13  OPTIONS: /*REG_B_OUT(1101,001)*/;
REGISTER  R14  OPTIONS: /*REG_B_OUT(1110,010)*/;
REGISTER  R15  OPTIONS: /*REG_B_OUT(1111,011)*/;

IO_PORT RIGHT_PORT
    OUTPUT_REGISTER: [WRITE_UPPER: A_SOURCE=0011,
                      READ_UPPER:  A_DEST=0011,
                      READ_LOWER:  B_DEST=11,
                      WRITE_LOWER: B_SOURCE=111],
    LOAD:            /*PORT_IN*/,
    DRIVE:           /*PORT_OUT*/;

END
```

The new system still requires external logic associated with the microcode and external circuitry for the condition select operations. This external circuitry does provide system flexibility, but it also adds to system complexity. A final proposed system includes on-chip circuitry for providing strobe signals and condition select operations.

```
NAME COMPLETE 16;

FIELD ADDRESS<1:3>,A_SOURCE<4:7>,A_DEST<8:11>,B_SOURCE<12:14>,
      B_DEST<15:16>,SHIFT_CONST<1:4>,SHIFT_LD<5:6>,ALU<7:10>,
      PORT<13:16>,CONDITION<17>,ALU_OP=ALU & CONDITION,STROBES<18:24>,
      RESET<25>,FLAGS<26:28>,EXTERNAL<29:31>;

MACRO NOP()     % ADDRESS=000 OR  CONDITION=0 %
MACRO JUMP()    % ADDRESS=001 AND CONDITION=1 %
MACRO CALL()    % ADDRESS=010 AND CONDITION=1 %
MACRO RETURN()  % ADDRESS=011 AND CONDITION=1 %
MACRO BRANCH()  % ADDRESS=100 AND CONDITION=1 %
MACRO SAVE()    % ADDRESS=101 AND CONDITION=1 %
MACRO DO()      % ADDRESS=110 AND CONDITION=1 %
MACRO ENDDO()   % ADDRESS=111 AND CONDITION=1 %
MACRO UNDO()    % ADDRESS=111 AND CONDITION=0 %
```

```
MACRO REG_A(ADR)
    % READ_UPPER: A_DEST=?ADR?,
      WRITE_UPPER: A_SOURCE=?ADR?,
      REFRESH:ALWAYS %

MACRO REG_B_OUT(AADR,BADR)
    % [/*REG_A(?AADR?)*/, WRITE_LOWER:B_SOURCE=?BADR?] %

MACRO REG_B_IN(AADR,BADR)
    % [/*REG_A(?AADR?)*/, READ_LOWER:B_DEST=?BADR?] %

MACRO REG_A_ONLY(ADR)
    % [/*REG_A(?ADR?)*/] %


OUTPUT_PORT ADDRESS
    REGISTER: [READ_UPPER: /*NOP*/ OR /*DO*/ OR /*BRANCH*/ OR /*SAVE*/,
               READ_LOWER: /*JUMP*/ OR /*CALL*/ OR /*RETURN*/ OR /*ENDDO*/];

ADDER NEW_PC
    INPUT_A: [READ_UPPER: /*NOP*/ OR /*DO*/ OR /*BRANCH*/ OR /*SAVE*/,
              READ_LOWER: /*JUMP*/ OR /*CALL*/ OR /*RETURN*/ OR /*ENDDO*/,
              SUGGEST: RESET=1,
              VALUE:0000000000000000],
    INPUT_B: [READ_LOWER: /*BRANCH*/,
              SUGGEST: NOT(/*BRANCH*/),
              VALUE: 0000000000000001],
    LOAD: ALWAYS,
    OUTPUT_REGISTER: [WRITE_UPPER: ALWAYS];

STACK PC_STACK
    DEPTH:8,
    TOP: [READ_UPPER: /*CALL*/ OR /*DO*/,
          READ_LOWER: /*SAVE*/,
          WRITE_LOWER: /*RETURN*/ OR /*ENDDO*/],
    PUSH: /*CALL*/ OR /*DO*/ OR /*SAVE*/,
    POP: /*RETURN*/ OR /*UNDO*/;

PRECHARGE_AND_BREAK_UPPER PCHG1;

REGISTER LINK OPTIONS:
    [WRITE_LOWER: /*JUMP*/ OR /*CALL*/ OR /*BRANCH*/ OR /*SAVE*/,
     WRITE_UPPER: A_SOURCE=0001, READ_UPPER: A_DEST=0001];

PRECHARGE_AND_BREAK_LOWER  PCHG2;

ALU_WITH_FLAGS ALU
    INPUT_A: /*REG_B_IN(1000,01)*/,
    INPUT_B: /*REG_A_ONLY(1001)*/,
    OUTPUT_1: /*REG_B_OUT(1010,100)*/,
    FLAGS: /*REG_A_ONLY(1011)*/,
    LOAD_FLAGS: NOT(ALU=1111),
    WRITE_OUTPUT_1: NOT(ALU=1111),
    TO_CONTROL:      {<1,2,9>=>FLAGS},
    DECODE: ALU_OP
        <0> => SUBTRACT
        <1> => ADD
        <2,3> => INCREMENT_A
        <4> => SUBTRACT
        <5> => SUB_W_BORROW
        <6,7> => SUBTRACT
```

```
        <8> => ADD
        <9> => ADD_W_CARRY
        <10,11> => ADD
        <12,13> => DECREMENT_A
        <14,15> => NEGATE_A
        <16> => SETA
        <17> => ADD
        <18> => SETA
        <19> => SETB
        <20,21> => OR
        <22,23> => AND
        <24,25> => SETA
        <26,27> => XOR
        <28,29> => TEST
        <30,31> => DONT_CARE;

MASKED_SHIFTER  SHIFTER
    MOST_SIGNIFICANT_WORD:   /*REG_A_ONLY(0100)*/,
    LEAST_SIGNIFICANT_WORD: /*REG_B_IN(0101,10)*/,
    OUTPUT_REGISTER:         /*REG_B_OUT(0110,101)*/,
    MASK_REGISTER:           /*REG_A_ONLY(0111)*/,
    SHIFT_CONSTANT:          SHIFT_CONST,
    LOAD_IF_0:               SHIFT_LD=XI,
    LOAD_IF_1:               SHIFT_LD=IX;

ROM_PAIR  C0
    LEFT_ENABLE: A_SOURCE=0000,
    RIGHT_ENABLE:B_SOURCE=110,
    LEFT_UPPER:  0000000000000000,
    RIGHT_LOWER: IIIIIIIIIIIIIIII;

REGISTER  R0_REG
    OPTIONS: [WRITE_UPPER: A_SOURCE=0010,
              READ_UPPER: A_DEST=0010,
              REFRESH:ALWAYS,
              READ_LOWER: B_DEST=00 AND NOT(B_SOURCE=000),
              WRITE_LOWER: B_SOURCE=000];

REGISTER  B7
    OPTIONS: [WRITE_LOWER: B_SOURCE=111,
              READ_LOWER: B_DEST=11,
              REFRESH:ALWAYS];

REGISTER  R13  OPTIONS: /*REG_B_OUT(1101,001)*/;
REGISTER  R14  OPTIONS: /*REG_B_OUT(1110,010)*/;
REGISTER  R15  OPTIONS: /*REG_B_OUT(1111,011)*/;

PRECHARGE_AND_BREAK_LOWER  PCHG3;

REGISTER  LINK2
    OPTIONS:     [READ_UPPER: A_DEST=0011,
                  WRITE_UPPER: A_SOURCE=0011,
                  REFRESH: ALWAYS,
                  READ_LOWER: PORT=01XX OR PORT=0X1X OR PORT=0XX1,
                  WRITE_LOWER: PORT=11XX];

PRECHARGE_BOTH  PCHG4;

CONTROL_TO_DATA_AND_BACK  STROBES
    REGISTER:    [SUGGEST: RESET=1, VALUE:0000000000000000],
    LATCH:       ALWAYS,
```

```
    TO_CONTROL: (<1:7>=> STROBES; <8>=> CONDITION; <9:14>=> PAD),
    TO_DATA:    (STROBES=1XXXXXX AND NOT(PORT=0111) OR PORT=1111=>1;
                 STROBES=1XXXXXX AND NOT(PORT=0111) OR PORT=1111=>9;
                 STROBES=X1XXXXX AND NOT(PORT=0111) OR PORT=1110=>2;
                 STROBES=X1XXXXX AND NOT(PORT=0111) OR PORT=1110=>10;
                 STROBES=XX1XXXX AND NOT(PORT=0111) OR PORT=1101=>3;
                 STROBES=XX1XXXX AND NOT(PORT=0111) OR PORT=1101=>11;
                 STROBES=XXX1XXX AND NOT(PORT=1000) OR PORT=1100=>4;
                 STROBES=XXX1XXX AND NOT(PORT=1000) OR PORT=1100=>12;
                 STROBES=XXXX1XX AND NOT(PORT=1000) OR PORT=1011=>5;
                 STROBES=XXXX1XX AND NOT(PORT=1000) OR PORT=1011=>13;
                 STROBES=XXXXX1X AND NOT(PORT=1000) OR PORT=1010=>6;
                 STROBES=XXXXX1X AND NOT(PORT=1000) OR PORT=1010=>14;
                 STROBES=XXXXXX1 AND NOT(PORT=1000) OR PORT=1001=>7;
                 STROBES=XXXXXX1 AND NOT(PORT=1000) OR PORT=1001=>15;
                 CONDITION=00 OR
                 CONDITION=01 AND FLAGS=1XX OR
                 CONDITION=10 AND EXTERNAL=1XX OR
                 CONDITION=11 AND
                     IF SHIFT_CONST=1XXX THEN
                         SHIFT_CONST=1001 AND FLAGS=0XX OR
                         SHIFT_CONST=1010 AND FLAGS=XX0 OR
                         SHIFT_CONST=1011 AND FLAGS=X0X OR
                         SHIFT_CONST=1100 AND EXTERNAL=0XX OR
                         SHIFT_CONST=1101 AND EXTERNAL=X0X OR
                         SHIFT_CONST=1110 AND EXTERNAL=XX0 OR
                         SHIFT_CONST=1111 AND FLAGS=0X0
                     ELSE
                         SHIFT_CONST=0000
                         SHIFT_CONST=0001 AND FLAGS=1XX OR
                         SHIFT_CONST=0010 AND FLAGS=XX1 OR
                         SHIFT_CONST=0011 AND FLAGS=X1X OR
                         SHIFT_CONST=0100 AND EXTERNAL=1XX OR
                         SHIFT_CONST=0101 AND EXTERNAL=X1X OR
                         SHIFT_CONST=0110 AND EXTERNAL=XX1 OR
                         SHIFT_CONST=0111 AND NOT(FLAGS=0X0) FI => 8);

LOWER_ROM  C1  ENABLE: PORT=0001, VALUE:1111111111110000;
LOWER_ROM  C2  ENABLE: PORT=0010, VALUE:1111000011110000;
LOWER_ROM  C3  ENABLE: PORT=0011, VALUE:1111111100000000;
LOWER_ROM  C4  ENABLE: PORT=0100, VALUE:1110001110000000;
LOWER_ROM  C5  ENABLE: PORT=0101, VALUE:1111110000000000;
LOWER_ROM  C6  ENABLE: PORT=0110, VALUE:1111111111010000;

IO_PORT  PORT
    OUTPUT_REGISTER:     [WRITE_LOWER: PORT=0111,
                          READ_LOWER: PORT=11XX,
                          REFRESH: STROBES=000XXXX],
    LOAD: NOT(STROBES=000XXXX),
    DRIVE: NOT(STROBES=XXX0000);

END
```

Chapter 10: A History of Bristle Blocks

This chapter provides a brief overview of the Bristle Blocks project. The major results of a number of experiments are stated, and the motivation behind various design decisions are given. Finally, a description is given of what the next version of Bristle Blocks may be like.

10.1: The Past

Bristle Blocks was born out of the OM project [15][16][17]. The OM2 datapath chip was designed in nine months, three of which were spent designing the low level cells, and the remaining six of which were spent interconnecting all of the pieces. The chip was designed using a special purpose programming language, PAL [2]. A picture of the finished mask set is shown in chapter 2, figure 2-10.

There were many lessons learned from the OM project. The more dramatic (and painful) lessons dealt with the limited expressability of the language, the complexity of the global interconnect versus the simplicity of leaf cell design, and the limited expressibility of a purely graphical design system.

The PAL artwork language is a special purpose drafting language. The purpose of the language is to describe simple line drawings or printed circuit board layouts. There are relatively few standard programming language constructs. It is virtually impossible to design a parametrized cell in such a language, and there is little hope for designing automatic routing programs with such a system. Due to the limited power of PAL, yet the power of textual cell descriptions, imbedded languages were developed. The first imbedded language developed at Caltech was ICLIC, written by Ron Ayres and Maureen Stone in the ICL language [4]. Soon thereafter, Bart Locanthi programmed LAP in Simula [19].

The complexity issue of global interconnect had two manifestations in the OM project. The first was that the layout of the final portion of the chip took much longer than the design of the majority of the chip area, even though much time was spent planning the global structure of the chip. The leaf cells were small layouts, which could easily be plotted on small sheets of paper. The entire function of each

cell could be grasped as the cell was being designed. The control structure, on the other hand, was a very large cell, so that it was difficult to make detailed plots of the entire cell. The cell was hard to design because of the many timing and logical function details which had to be included in the cell. The second manifestation of the global interconnect complexity appeared when the chip was tested. It was in the global interconnections that all of the design errors were encountered. There were two timing errors, one logical error, and one design rule error in the interconnections. The first timing error set the chip speed at 2.5 MHz, one quarter of the intended operating speed. The second error caused the flag circuitry to become inoperative. The logical error was not fatal: the polarity of one of the control input pins was negated. The design rule error was the major design error. Six of the highest level wires ever so slightly missed their proper connection positions on the instruction decoder. They were off less than .2% of their total length. For 5000 micron long wires, however, this small error, which is invisible on all cell plots, caused six of the control input bits to be shorted to ground. Each of these errors was not caused because the global interconnection task for any particular signal was difficult, but because there were so many signals to be interconnected that the specific details were forgotten.

The third lesson learned from OM was that cells are more than just layout. There is documentation information about the cells that is just as important as the layout information. The design system which was used to create OM only allowed for the specification of geometric information, although I was able to add a block diagram description of the OM2 datapath chip to the system. As a designer, it was very frustrating not being able to add a little more information to the cells' descriptions. Even if additional information could be added to the cell, there was no way to access that information later. With the new design tools that have been developed, there has been a gradual increase in the flexibility of the cell data representation, so that additional designer intent can be encapsulated with the design.

When the OM2 datapath chip design errors were found, there was a strong motivation to develop better design tools: to cast away nine months of effort because of a few tiny implementation details is not an easy thing to do. The process was begun of designing programs to aid in the design of integrated circuits.

The first routine implemented was a simple, monochromatic river router. There were several places in the datapath chip where a river router could be used to interconnect cells. Although there were no design errors in the datapath's hand designed river routes, the generation of the 500 interconnection wires between two of the cells was not a pleasant task.

The second routine to be implemented was an instruction decode generator. In the datapath chip, the instruction decoder was implemented as a collection of 42 incredibly tiny cells. These cells measured 7 lambdas by 14 lambdas, and were used to tile large portions of the chip. The instruction decoders required close to 20,000 function calls, each of which required an absolute chip position parameter. This tedious and error prone task was accomplished without design error. However, the design was fixed for one particular chip instance, and if there were any change in the chip specification, this entire decoder would have to be re-implemented. An instruction decoder generator was written to automatically produce calls to cells very similar to the cells used in the datapath chip. Data structures were defined in ICL to desribe the instruction decoder operations, which became the input parameters to the generator. When this programming task was completed, a chip designer could rapidly generate an instruction decoder from a functional description of the decoder operation, plus positional information for the outputs of the decoder.

The next step in automating the design of chips was to add the timing information to the decoder routing, so that the buffers and decoder could automatically be added to the datapath. It was at this same time that Ron Ayres presented some fascinating news of his Programmed Logic Array (PLA) compiler, RELAY [5]. He pointed out some very obvious ideas which helped crystallize the Bristle Blocks framework. A short description of RELAY will be presented here.

Ron Ayres is a software computer scientist. He had a mathematical description of a chip he wanted implemented, yet he did not know how to design integrated circuits. He built a programming system that let him describe his formal, mathematical, chip descriptions. The system accepted a hierarchy of synchronous logic equations, and would allow the designer to alter the hierarchy of the logic while preserving the function of the description. The designer could simulate the operation of the chip at any time to verify the correctness of the specification. Ron

then met with a student in the LSI design course, and they composed a simple model of a PLA and of an interconnect algorithm. Ron added these models to his system, which allowed him to quickly see what a set of logic equations would look like when implemented in PLAs. He could observe the physical impact of editing the logic hierarchy. Finally, Ron borrowed a PLA generator and wrote an actual interconnect procedure. With these two routines, Ron was able to generate complete chip layouts from logic equation specifications.

To illustrate the form of RELAY input, the following cell examples will be given. These examples are not meant to teach the reader how to design chips with RELAY, but rather provide the user with the flavor of the design methodology.

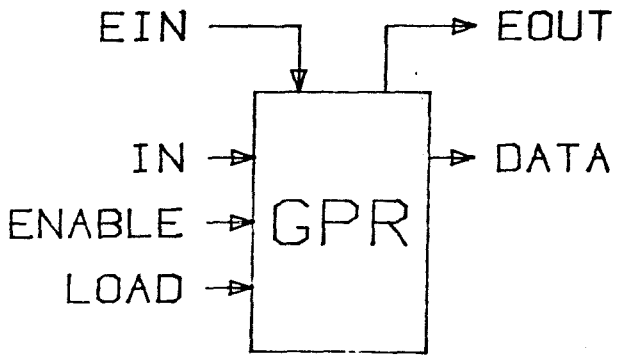


Fig. 10-1: General Purpose Register Block Diagram

The first cell is a General Purpose Register (GPR). A block diagram of this register is shown in figure 10-1. The register will load data from the IN pin when LOAD and ENABLE are TRUE. When ENABLE is TRUE, EOUT will be set to the value contained within the register, and when ENABLE is FALSE, EOUT will be set to the value of EIN. The RELAY specification for the GPR register is listed here.

```
VAR GPR=LL;
BEGIN    VAR DATA,IN,LOAD,ENABLE,EIN,EOUT=BIT;
     DATA:=NEW_BIT;
     IN:=NEW_BIT;
     LOAD:=NEW_BIT;
     ENABLE:=NEW_BIT;
     EIN:=NEW_BIT;
     EOUT:=NEW_BIT;
     GPR:=
          [EXTERNALS: [IN_PINS:    (EIN\NAMED 'EIN';
                                    ENABLE\NAMED 'ENABLE';
                                    LOAD\NAMED 'LOAD';
                                    IN\NAMED 'IN')
```

```
                OUT_PINS:   (EOUT\NAMED 'EOUT')]
        RELATIONS: (EOUT\EQU [IF:ENABLE THEN:DATA ELSE:EIN];
                    DATA\NEXT [IF:LOAD\AND ENABLE THEN:IN ELSE:DATA])];
END
```

We have declared GPR to be of type LL, which stands for Logic Level, the RELAY cell. Internal to a GPR, we have the following signals: DATA, IN, LOAD, ENABLE, EIN, and EOUT. We have declared the port characteristics of the GPR cell, and given the logic equations relating the signals within the GPR.

We can now define a cell which uses two of these GPR cells. This GPR_PAIR cell has a SELECT input which is used to select which GPR cell is being addressed.

```
VAR GPR_PAIR=LL;
BEGIN   VAR L,R=NAMED_LOGIC_LEVEL;IN,LOAD,SELECT,ENABLE=BIT;
    L:=GPR\NEW;
    R:=GPR\NEW;
    IN:=NEW_BIT;
    LOAD:=NEW_BIT;
    SELECT:=NEW_BIT;
    ENABLE:=NEW_BIT;
    GPR_PAIR:=
        [EXTERNALS: [IN_PINS:    (R\S 'EIN'\NAMED 'EIN';
                                  ENABLE\NAMED 'ENABLE';
                                  LOAD\NAMED 'LOAD';
                                  IN\NAMED 'IN';
                                  SELECT\NAMED 'SELECT')
                    OUT_PINS:   (L\S 'EOUT'\NAMED 'EOUT')]
         RELATIONS: (L\S 'EIN'\EQU R\S 'EOUT';
                     L\S 'IN'\EQU IN;
                     R\S 'IN'\EQU IN;
                     L\S 'LOAD'\EQU LOAD;
                     R\S 'LOAD'\EQU LOAD
                     L\S 'ENABLE'\EQU SELECT\AND ENABLE;
                     R\S 'ENABLE'\EQU NOT(SELECT)\AND ENABLE)];
END
```

In the same manner, we can define a few new register cells. The GPRO cell is similar to the GPR cell, except that the data contained within the cell is also available as a port. The GPRI cell is used as an interface cell, a shared register between two processors, for instance. When one processor writes into the cell, the second processor notices the effect in its corresponding interface cell.

```
VAR GPRO=LL;
BEGIN   VAR DATA,IN,LOAD,ENABLE,EIN,EOUT=BIT;
    DATA:=NEW_BIT;
    IN:=NEW_BIT;
    LOAD:=NEW_BIT;
    ENABLE:=NEW_BIT;
    EIN:=NEW_BIT;
    EOUT:=NEW_BIT;
    GPRO:=
```

```
        [EXTERNALS: [IN_PINS:    {EIN\NAMED 'EIN';
                                  ENABLE\NAMED 'ENABLE';
                                  LOAD\NAMED 'LOAD';
                                  IN\NAMED 'IN'}
                     OUT_PINS:   {EOUT\NAMED 'EOUT';
                                  DATA\NAMED 'DATA'}]
         RELATIONS: {EOUT\EQU [IF:ENABLE THEN:DATA ELSE:EIN];
                     DATA\NEXT [IF:LOAD\AND ENABLE THEN:IN ELSE:DATA]}];
END
```

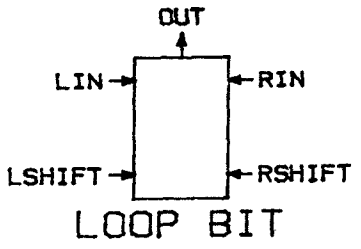
```
VAR GPRI=LL;
BEGIN   VAR DATA,IN,LOAD,ENABLE,EIN,EOUT,DIN=BIT;
    DATA:=NEW_BIT;
    IN:=NEW_BIT;
    LOAD:=NEW_BIT;
    ENABLE:=NEW_BIT;
    EIN:=NEW_BIT;
    EOUT:=NEW_BIT;
    DIN:=NEW_BIT;
    GPRI:=
        [EXTERNALS: [IN_PINS:    {EIN\NAMED 'EIN';
                                  ENABLE\NAMED 'ENABLE';
                                  LOAD\NAMED 'LOAD';
                                  IN\NAMED 'IN';
                                  DIN\NAMED 'DATA_IN'}
                     OUT_PINS:   {EOUT\NAMED 'EOUT';
                                  DATA\NAMED 'DATA_OUT'}]
         RELATIONS: {EOUT\EQU [IF:ENABLE THEN:DIN ELSE:EIN];
                     DATA\NEXT [IF:LOAD\AND ENABLE THEN:IN ELSE:DATA]}];
END
```

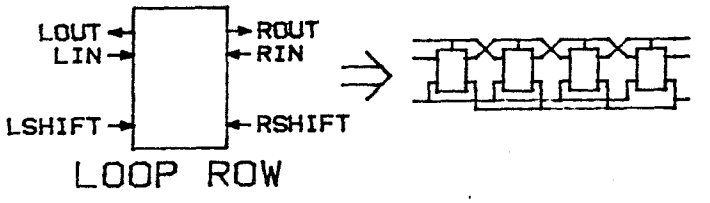
As a final example, a shift register loop is described. Externally, the shifter appears like a GPR cell, except that shift input signals are included in the interface of the cell. The top cell communicates with a series of short shift registers, each of which is composed of a series of bits. Hence, the shifter is a hierarchy of shift bits, as shown in figure 10-2.

```
VAR LOOP_BIT=LL;

BEGIN   VAR LIN,RIN,LSHIFT,RSHIFT,OUT=BIT;
    LIN:=NEW_BIT;
    RIN:=NEW_BIT;
    LSHIFT:=NEW_BIT;
    RSHIFT:=NEW_BIT;
    OUT:=NEW_BIT;
    LOOP_BIT:=
        [EXTERNALS: [IN_PINS:    {LIN\NAMED 'LIN';
                                  RIN\NAMED 'RIN';
                                  LSHIFT\NAMED 'LSHIFT';
                                  RSHIFT\NAMED 'RSHIFT'}
                     OUT_PINS:   {OUT\NAMED 'OUT'}]
         RELATIONS: {OUT\NEXT [IF:LSHIFT THEN:RIN
                               ELSE: [IF:RSHIFT THEN:LIN
                               ELSE: OUT]]}];
END
```

```
"A LOOP ROW IS A STRING OF N LOOP BITS, ALL PROPERLY CONNECTED."

DEFINE LOOP_ROW(N:INT)=LL:
BEGIN   VAR LOOP_BITS=NAMED_LOGIC_LEVELS;L,R,B1,BN=NAMED_LOGIC_LEVEL;
DO      LOOP_BITS:={COLLECT GPR\NEW REPEAT N;};
        B1:=LOOP_BITS[1];
        BN:=LOOP_BITS[N];
GIVE    [EXTERNALS: [IN_PINS:   {B1\S 'LIN';
                                 BN\S 'RIN';
                                 B1\S 'LSHIFT';
                                 B1\S 'RSHIFT'}
                    OUT_PINS:  {B1\S 'OUT'\NAMED 'LOUT';
                                BN\S 'OUT'\NAMED 'ROUT'}]
         RELATIONS: {FOR {L;R} $C LOOP_BITS; COLLECT
                        {L\S 'RIN'\EQU R\S 'OUT';
                         R\S 'LIN'\EQU L\S 'OUT';
                         R\S 'LSHIFT'\EQU B1\S 'LSHIFT';
                         R\S 'RSHIFT'\EQU B1\S 'RSHIFT'}}
         GUTS:  LOOP_BITS]
END
ENDDEFN


"A LOOP LOOKS MUCH LIKE A GPRO EXTERNALLY, BUT IT CONTAINS AN
 M*N+1 BIT SHIFT REGISTER.  EXTERNALLY, IT DOES HAVE THE RSHIFT AND
 LSHIFT SIGNALS."

DEFINE LOOP(M,N:INT)=LL:
BEGIN   VAR LOOPS=NAMED_LOGIC_LEVELS;L,R,B1,BN=NAMED_LOGIC_LEVEL;
            DATA,IN,LOAD,ENABLE,EIN,EOUT,LSHIFT,RSHIFT=BIT;
DO      DATA:=NEW_BIT;
        IN:=NEW_BIT;
        LOAD:=NEW_BIT;
        ENABLE:=NEW_BIT;
        EIN:=NEW_BIT;
        EOUT:=NEW_BIT;
        LSHIFT:=NEW_BIT;
        RSHIFT:=NEW_BIT;
        B1:=LOOP_ROW(M);
        LOOPS:={COLLECT B1\NEW REPEAT N;};
        B1:=LOOPS[1];
        BN:=LOOPS[N];
GIVE    [EXTERNALS: [IN_PINS:   {EIN\NAMED 'EIN';
                                 ENABLE\NAMED 'ENABLE';
                                 LOAD\NAMED 'LOAD';
                                 IN\NAMED 'IN';
                                 LSHIFT\NAMED 'LSHIFT';
                                 RSHIFT\NAMED 'RSHIFT'}
                    OUT_PINS:  {EOUT\NAMED 'EOUT';
                                DATA\NAMED 'DATA'}]
         RELATIONS: {EOUT\EQU [IF:ENABLE THEN:DATA ELSE:EIN];
                     DATA\NEXT [IF:LOAD\AND ENABLE THEN:IN
                         ELSE: [IF:LSHIFT THEN:BN\S 'ROUT'
                         ELSE: [IF:RSHIFT THEN:B1\S 'LOUT'
                         ELSE: DATA]]];
                     B1\S 'LIN'\EQU DATA;
                     BN\S 'RIN'\EQU DATA;
                     FOR {L;R} $C LOOPS; COLLECT
                        {L\S 'RIN'\EQU R\S 'LOUT';
                         R\S 'LIN'\EQU L\S 'ROUT';
```

```
                    R\S 'LSHIFT'\EQU LSHIFT;
                    R\S 'RSHIFT'\EQU RSHIFT};
                B1\S 'LSHIFT'\EQU LSHIFT;
                B1\S 'RSHIFT'\EQU RSHIFT}
        GUTS:  LOOPS]
END
ENDDEFN
```





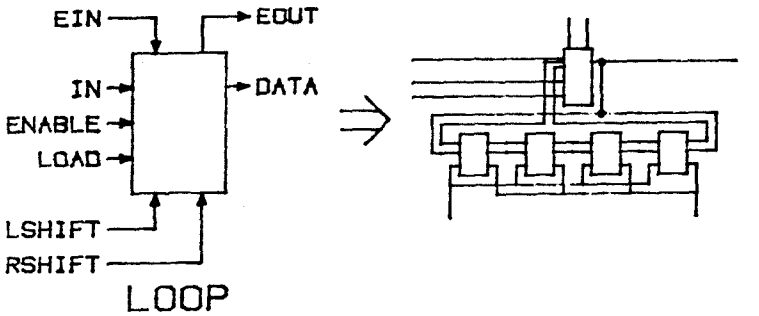


Fig. 10-2: Shifter Loop Block Diagram

These examples illustrate the design of leaf cells and composition cells. Each cell (LL) contains an interface specification (EXTERNALS), an interconnection specification (RELATIONS), and a subcell specification (GUTS). Leaf cells do not have any GUTS, only EXTERNALS and RELATIONS. Composition cells have values in all three areas.

The first version of Bristle Blocks was completed in December, 1978. Version one produced small datapath chips, in a variety of representations. The compiler produced the NMOS artwork, along with a stick diagram, transistor diagram, logic diagram, and block diagram of the chip. In all later versions of Bristle Blocks, the

capability of multiple representations (even multiple technologies) has been an integral part of the system, although the datapath cells were designed to produce only layouts, due to the press of time.

In the two and a half years since the first running of Bristle Blocks, there have been several areas of improvement upon the basic system. Work has been done on the Virtual Memory (VM) system, which greatly increased the compilable chip size. Many of the algorithms, like the river router and instruction decode generator, have been improved and tested. User interfaces have been added to allow non-specialists to use the system. Finally, the variety of datapath elements has increased, improving the efficiency and flexibility of Bristle Blocks.

To provide efficient generation of artwork, Bristle Blocks cells were designed to be programs, rather than data structures. If the cells were data structures, the user would be limited to designed cells expressable in the data structure. Since the user is allowed to write programs for the cells, the user is only limited by the expressability of the language Bristle Blocks is written in (ICL). ICL allows much greater expressability than a simple data structure would allow, so that the user can design very flexible cells.

Unfortunately, the PDP-10 computer has a very small address space, with only 18 bits for addresses. In current versions of ICL, programs are not swappable to the disk, although data structures can be swapped to the disk. Since data structures are swappable, we can have a very large effective address space by saving the information contained in the data structures in a disk file. The system can read this information as it is required, and when the data is no longer needed, the data can be written back into the file. With swapping, we can effectively have a much larger address space if our cells were data structures.

To make use of swapping, yet still retain the power of cells as programs, a compromise was made. Most cells have a lot of relatively fixed, or constant, layout. The fixed portion of the cell can be stored in a data structure, and thereby can be swapped to a disk file. The variable portions of the cell can be kept as a program. The cells compute the variable portions of the layout and swap in the fixed layout sections. Partitioning the cells in this manner does add to the complexity of the compiler and of the cells, but users of the system never see the additional

complexity.

To free up as much code space as possible, we need to have as much of the cells as possible represented in the swappable data structure. To this end, the data structures used in Bristle Blocks allow the representation of simple variations in the layout and connection points. In many cases, the actual code required by a cell simply checks the user's parameters and swaps in the cell implementation from the data file, which is called the Virtual Memory (VM) file.

The following data structure definitions describe the structures used in the current version of Bristle Blocks.

The first primitive user-defined datatype in Bristle Blocks is called the STRETCH_POINT. The name refers to the common use of the datatype, although a better name would probably be VARIABLE. The data structures refer to these STRETCH_POINTs using the ID number as a name. To stretch a layout, the appropriate STRETCH_POINT's value is modified, and the layout is effectively changed.

```
TYPE    STRETCH_POINT= [
            NAME:                SC
            ID,INITIAL,FINAL:    INT
            FRESH:               BOOL
            XFRM:                COORDINATE ];

        STRETCH_POINTS= { STRETCH_POINT };

VAR STRETCH_POINTS_VALID=BOOL;
```

The NAME component of STRETCH_POINTs holds the user's names for the STRETCH_POINTs. The system looks through the global STRETCH_POINT list to convert a name to a STRETCH_POINT. The ID is the internal identification assigned by Bristle Blocks to the STRETCH_POINTs. The remaining components are used to compute the value of a STRETCH_POINT. The XFRM component may contain an algorithm for computing a STRETCH_POINTs value: a STRETCH_POINT's value may depend upon other STRETCH_POINT values. The FRESH component states whether the FINAL component holds the actual value of the STRETCH_POINT. Whenever a STRETCH_POINT's value is modified, all of the STRETCH_POINTs in the system have their FRESH value set FALSE. When computing a STRETCH_POINT's value, the FRESH component is examined. If FRESH is TRUE, the FINAL component hold the value. If FRESH is FALSE, the system recomputes the final value. The FINAL value is set to

the INITIAL value, and the FRESH component is set TRUE. The XFRM is then evaluated, and the resulting value is stored in the FINAL component.

COORDINATEs are used to express equations in the system. A COORDINATE may state, for example, that a certain feature be positioned with a Y-coordinate of 5 lambdas above the 'Y1' STRETCH_POINT. This equation is stated as follows.

```
'Y1'\P 5
```

The datatypes associated with COORDINATEs are listed here.

```
TYPE    COORDINATE= EITHER
            INTEGER=    INT
            STRETCH=    STRETCH_POINT
            OP=         [OP:COORDINATE_OP  A,B:COORDINATE]
            NEGATE=     COORDINATE
            IF=         [REL:IF_RELATION  C,A,B:COORDINATE]
                ENDOR;

        COORDINATES= { COORDINATE };

        COORDINATE_OP= SCALAR(ADD,SUB,MUL,DIV,MIN,MAX);

        IF_RELATION= SCALAR(ZERO,NZERO,NEG,NNEG,POS,NPOS,EVEN,ODD);
```

In the simplest case, a COORDINATE may be an INTeger. A STRETCH_POINT may also be a COORDINATE. A COORDINATE may be a simple function of two other COORDINATES: A OP B, where A and B are coordinates and OP is either ADD, SUB, MUL, DIV, MIN, or MAX. A COORDINATE may be the inverse of another COORDINATE, and finally, a COORDINATE may be an IF...THEN...ELSE...FI equation: the C COORDINATE is compared with relation REL. If the comparison is TRUE, the value of A is returned. Otherwise, the value of B is returned.

Using the above definition of COORDINATE, definitions for wires, boxes (VBOXES), and polygons (SSXY) were defined. These primitives were not associated with mask layers, but all the primitives of a single layer (within a single cell), were collected into a single MASK_LAYER.

```
TYPE    XY_PAIR= [X,Y:COORDINATE];

        SXY= { XY_PAIR };
```

```
SSXY= { SXY };

WIRE= [WIDTH:INT  PATH:SSXY];

WIRES= { WIRE };

VBOX= [LOW,HIGH:XY_PAIR];

VBOXES= { VBOX };

MASK_LAYER= [COLOR:COLOR  WIRES:WIRES  BOXES:VBOXES  POLYS:SSXY];

MASK_SET= { MASK_LAYER };

DMASK_SET= a swappable version of MASK_SET ;
```

A collection of MASK LAYERs formed a MASK SET, which was the complete set of geometric primitives for a particular representation. The PICTURE datatype described one representation.

```
TYPE    VIEW= SCALAR(LAYOUT,STICKS,TRANS,BLOCK,LOGIC);

        PICTURE= [VIEW:VIEW  MASKS:DMASK_SET];

        PICTURES= { PICTURE };
```

The next set of datatype definitions described connection points. Connection points could be kept with the artwork, swapped out in the disk file. Connection points contain a name, positions, signal direction (into or out of the cell), connection type (control connection, pad connection, etc.), buffer type or pad type, connection edge (north, south, east, or west), timing information, layer information, and the associated microcode function.

```
TYPE    CONNECT=[
            NAME:          SC
            FROM,TO:       XYPAIR
            DIRECTION:     SCALAR(IN,OUT,IO,ANY)
            TYPE:          SCALAR(CONTROL,PAD,CONDITION,....)
            BUFFER:        SCALAR(PHI_1,PHI_2,P1MUX,P2MUX,P1INV,P2INV,VDD,GND,
                               BUFIN,BUFOUT,BUFINV)
            PAD:           SCALAR(IN,OUT,DOWN,IO,IO_DOWN,ENABLE,OUT_ENABLED,
                               DOWN_ENABLED,BOTH,IN_PULL,DOWN_PULL,
                               ENABLED_PULL,IO_PULL,BOTH_PULL)
            EDGE:          SCALAR(NORTH,EAST,SOUTH,WEST)
            VALID:         SCALAR(PHI1,PHI2,BOTH,NONE)
            COLOR:         COLOR
            UCODE:         DECODE_COLUMN
            ....];

        CONNECTS= { CONNECT };

        DECODE_COLUMN= [
            TYPE:              SCALAR(UCODE,SOURCE,COMPLIMENT,PAD,WIRE,BLANK)
            COLUMN,LENGTH:     INT
```

```
        Q0,Q1,D0,D1:    SI];
```

Next, we have the BLOCK definition. A BLOCK is the basic cell in Bristle Blocks. It contains a name, some layout information (pictures), calls to subblocks, connection points, and a bounding box. Recall that many of the BLOCKs for a particular chip are computed by a program. These BLOCKs have enough flexibility, however, that many of the datapath cells can be represented as BLOCKs rather than programs.

```
TYPE    BLOCK= [
            NAME:       SC
            PICTURES:   PICTURES
            CALLS:      CALLS
            INTERFACE:  CONNECTS
            MBB:        VBOX
            ....];

        BLOCKS= { BLOCK };

        DBLOCK= a swappable version of BLOCK ;
```

Lastly, we have definitions for CALLs. A CALL is a reference to a subBLOCK.

```
TYPE    CALL= EITHER
            BLOCK=      DBLOCK
            TRANSLATE=  [C:CALL  T:XY_PAIR]
            ROTATE=     [C:CALL  R:SCALAR(R90,R180,R270)]
            MIRROR=     [C:CALL  M:SCALAR(MIRX,MIRY,BOTH)]
            STRING=     [C:CALL  S:SXY]
            VECTOR=     [C:CALL  V:[I:XY_PAIR  N:COORDINATE]]
            CALLS=      CALLS
            WITH_MASKS= [C:CALL  M:MASK_MAKERS]
            PASS_MASKS= [C:CALL  N:SI]
            MASKED=     [C:CALL  N:INT]
            IF=         [REL:IF_RELATION  C:COORDINATE  A,B:CALL]
            ....
                ENDOR;

        CALLS= { CALL };

        MASK_MAKER= EITHER
            ALWAYS= DUMMY
            ROTATE= SB
            EXTEND= SB
            FIXED=  SB
                      ENDOR;

        MASK_MAKERS= { MASK_MAKER };
```

The first four types of CALLs are fairly straightforward. A STRING CALL places a subCALL at each point in the list of XY PAIRs (SXY). A VECTOR CALL evaluates the V.N COORDINATE to determine an iteration count. The V.I XY PAIR specifies a step distance. The VECTOR CALL will return a row of the subCALLs, each offset from

the previous instance by the step distance. The total number of instances in the row is given by the iteration count. The CALLS CALL allows a BLOCK to refer to several subBLOCKs. The next three types of CALLs specify masks. Each of the iteration type CALLs (STRING, VECTOR, CALLS) can be masked: only a few of the specified subCALLs will be returned. The WITH_MASKS CALL adds masks to a global list of masks. PASS_MASKS reorders the masks in the global list, and MASKED extracts one mask from the list and applied the mask to the subCALL. Finally, the IF CALL returns one of its subCALLs depending upon the correspondence of its COORDINATE and relation (similar to the IF type COORDINATE).

These datatype definitions were arrived at through many iterations and trials. They are not as general or easy to use as straight procedural cells, but they sufficed with the implementation restrictions that existed.

10.2: The Future

In the future, there are four areas of improvement needed in Bristle Blocks. The first area has to do with the implementation concessions using the current ICL implementation. Secondly, the floorplan of Bristle Blocks needs to have a greater flexibility, which would allow more efficient implementations of many datapath chips. Thirdly, more work has to be done with the simulation aspects of the chips. Finally, I need to address the user specification issues. What languages are suitable for the specification of Bristle Blocks chips?

The main implementation concession in the current Bristle Blocks programs has to do with the address space limitations. Because of the 18 bit limit, the datapath cell programs have had a split personality. Portions of cells are data structures kept in disk files, while the remaining portions exist as programs compiled into the Bristle Blocks system. In the new ICL system, code is swappable, so that the cells can be entirely represented as programs without exceeding the address space of the machine.

The second improvement to Bristle Blocks modifies the floorplan of the compiler. In addition to allowing a greater number of buses in the datapath, I would like to add greater flexibility in the instruction decode portion of the chip. The most

logical way to enhance the instruction decoder is to perform a fusion of Bristle Blocks with the RELAY compiler, allowing the user to design chips which are hierarchical compositions of register transfer units and finite state machines. The datapath portion of the compiler would generate the efficient register transfer circuitry and the PLA portion of the compiler would generate the random logic and state machine mechanisms. The proposed compiler will interconnect the various datapaths and PLAs using a hierarchical general interconnection system.

Thirdly, I need simulation procedures in Bristle Blocks. Each version of Bristle Blocks has had hooks for linking simulators to the compiler, both register transfer simulators and timing simulators. Due to the press of time, these simulators have only been dreams. When I have the added flexibility of the Bristle Blocks/RELAY fusion, simulation will become a very important aspect of the design. I do not plan to do electrical model simulations of the entire chip. The simulation will be performed in much the same manner as the layouts are generated. Since the user provides a very high level specification of the design in the well defined design language, RT simulations and timing information can be generated directly from the high level specification, without having to generate the artwork and examine the resulting layout.

Finally, I need to develop languages for specifying Bristle Blocks chips which also capture the random logic/state machine information. These languages should feel natural to the designer, so that the designer can easily express his desires, and so that the user can intuitively grasp the meaning of expressions in the language. A lower bound exists on the information content required in a chip specification. Appropriate languages can capture the information in a clear, concise form.

Appendicies

Appendix 1: ICL Summary and ICLIC Reference Guide

This appendix summarizes some of the language features of ICL and lists the ICLIC functions used in this thesis for describing integrated circuit layouts. For a more detailed description of ICL, refer to the ICL appendix of Ron Ayres' thesis [3]. A more complete description of ICLIC is given in the ICLIC manual [4].

A1.1: ICL Summary

For the purposes of understanding the code examples presented in this thesis, ICL is very similar to PASCAL, with the following exceptions.

Pointers: ICL makes use of pointers in its memory management scheme, like PASCAL. However, the pointers are implicit in ICL, whereas the user must explicitly state when pointers are to be used in PASCAL.

Strings: ICL does not have a mechanism for building arrays. Instead, ICL allows the user to build strings. Most languages allow text strings to be arbitrarily long. In ICL, the user may build structures which are arbitrarily long strings of any desired datatype. Strings are generated in ICL by enclosing the string elements in curly brackets, {}. The elements of the string are separated by semicolons. Elements can be appended to the front of an existing string using the <$ operator, and elements can be appended to the end of an existing string using the $> operator. The $$ operator concatenates two strings. Elements of a string can be examined by indexing into the string. The ith element of string S is accessed by writting S[i]. The tail of a string (all elements from a specified index to the end of the string) is accessed by writting S[i-]. Quantifiers can be used to sequentially access elements in a string without indexing into the string.

Record Generation: ICL has record constructs similar to PASCAL's. There are differences between the record generation processes of the two languages. In PASCAL, one must explicitly request a chuck of memory for the record, the sequentially fill each component of the record. In ICL, one never requests chucks of memory. Instead, one merely specifies the record template with the desired values for each component.

Points: POINT is a basic datatype in ICL, just like integers and reals. A POINT contains two real values, which are usually interpreted as X and Y coordinates of a point in two-space. Points are generated using the binary operator #. 3#4 is the point whose x-coordinate is 3 and whose y-coordinate is 4. The x-coordinate of a POINT P is accessed by writting P.X.

Polymorphic Functions: In ICL, the user can specify any number of functions (procedures) with the same name. There is no ambiguity if the set of input parameters and return parameters uniquely determine the proper function to apply. For instance, the user may have a WRITE(INTeger) function, a WRITE(REAL) function, and a WRITE(CHARacter) function. For each call to a WRITE function, ICL selects the appropriate function based upon the parameter types. If the user writes WRITE(5), the WRITE(INTeger) routine is called; if the user writes WRITE(5.), the WRITE(REAL) routine is called.

Coercions: In most languages, there are predefined arithmetic coercions. If the user assigns an INTeger value to a REAL variable, the compiler automatically calls a routine which translates INTegers to REALs. In ICL, the user may declare coercions between any datatypes. ICL will implicitly apply coercions to satisfy datatype requirements.

Infix Operators: Math operators, such as + and -, are infix operators: one writes A + B rather than +(A,B). Binary function definitions (functions which take two parameters and return one value) typically do not use infix format: f(A,B), not A f B. In ICL, any binary function may use the infix format when the function name is preceded by the \ operator. f(A,B) can be written A \f B.

Quantifiers: Virtually every language has constructs for generating loops in the program control flow. These loops may be arithmetic loops (FOR loops) or conditional loops (WHILE loops or REPEAT loops). In addition to these standard loop generators (quantifiers), ICL has mechanisms for sequencing through strings (FOR element $E string;). ICL also has unary and binary operators which apply to quantifiers. The && operator forces two loops to iterate together; the !! operator steps one quantifier for each iteration of the other quantifier. Unary operators may eliminate some iterations of the quantifier or perform some actions before each iteration of the quantifier.

Suspendable Functions: The suspendable function mechanism in ICL allows the user to assign function call references to variables. A reference to function X may be assigned to the variable Y by writting Y:= // X \\;. Later, function X may be envoked by writting <*Y*>.

A1.2: ICLIC Reference Guide

The datatype definitions used in ICLIC are listed here:

```
TYPE    SP= { POINT };

        WIRE= [WIDTH:REAL  PATH:SP];

        RG= EITHER
            POLY=   SP
            WIRE=   WIRE
            BOX=    BOX
            UNION=  MRGS
            MATRIX= [DISPLACE:MRG  BY:MATRIX]
            POINT=  [DISPLACE:MRG  BY:POINT]
            COLOR=  [COLOR:MRG  WITH:COLOR]
              .
              .
            DISK=   .....
            ENDOR;

        MRG= [RG:RG  VMBB:BOX  .....];

        MRGS= { MRG };

        COLOR= SCALAR(RED,BLUE,GREEN,YELLOW,BLACK,GLASS,BROWN,VIOLET,BURIED);

        MATRIX= [A, B, C,
                 D, E, F: REAL];
```

These definitions declare that SP (String of Points) is an indefinite list of points, a WIRE contains a width and a path, and a BOX is two points. An RG (ReGion) may either be a POLYgon, represented by an SP, a WIRE, a BOX, an arbitrary list of MRGs, an MRG whose points are transformed, a displaced MRG, an MRG with an associated COLOR, or other types which are not used in this thesis. An MRG contains an RG along with a Virtual bounding box and other internal data.

There are functions to aid in the generation of MRGs. The basic functions are first defined:

```
DEFINE TO(A,B:POINT)=BOX: .............    ENDDEFN
DEFINE AT(M:MRG  P:POINT)=MRG:  ........    ENDDEFN
DEFINE ROT(M:MRG  ANGLE:REAL)=MRG:  ....    ENDDEFN
DEFINE MIRX(M:MRG)=MRG:  ..............    ENDDEFN
DEFINE MIRY(M:MRG)=MRG:  ..............    ENDDEFN
DEFINE PAINTED(M:MRG  C:COLOR)=MRG:  ...    ENDDEFN
DEFINE UNION(A,B:MRG)=MRG:  ............    ENDDEFN
```

The TO function takes two points and makes a box. AT takes an MRG and a POINT and generates a new MRG identical to the first MRG with all features displaced by the amount specified by the point. ROT takes an MRG and a REAL and generates a new MRG identical to the first but rotated counterclockwise the number of degrees specified by the REAL. Similarly, MIRX and MIRY mirror about the X and Y axis, respectively. PAINTED applies the given COLOR to the given MRG, and UNION takes two MRGs and merges them. To generate an array of identical MRGs, the following routine can be used:

```
TYPE ARRAY_OF_DOTS= [IX,IY:REAL  NX,NY:INT];
DEFINE AT(M:MRG  A:ARRAY_OF_DOTS)=MRG:  .....   ENDDEFN
```

IX and IY specify the distance between columns and rows, and NX and NY specify the number of columns and rows. To easily generate colored geometric primitives, the following routines have been defined:

```
DEFINE WIRE(C:COLOR  W:REAL  P:SP)=MRG:   ....    ENDDEFN
DEFINE WIRE(C:COLOR  P:SP)=MRG: ....    ENDDEFN
DEFINE BOX(C:COLOR  B:BOX)=MRG: ....    ENDDEFN
DEFINE POLYGON(C:COLOR  SP:SP)=MRG: ....    ENDDEFN
DEFINE DISK(M:MRG)=MRG: ....    ENDDEFN
```

The second wire function does not require a width parameter: it uses the default width for the given color. The DISK function configures the MRG so that it can swap to a disk file with the virtual memory system in ICL. The color interpretation for NMOS is as follows:

```
GREEN   diffusion
RED     polysilicon
BLUE    metal (first layer)
```

```
YELLOW   implant
BLACK    contacts
GLASS    overglassing
BROWN    metal-to-metal contact
VIOLET   second layer metal
BURIED   buried contacts
```

There are globally defined MRGs for each of the feedthroughs in the NMOS technology:

```
GCB      Green-to-Blue feedthrough (Green-Contact-Blue)
RCB      Red-Contact-Blue
GRCBU    Butting contact, Red 'UP' (Green-Red-Contact-Blue-Up)
GRCBL    Butting contact, Red 'Left'
GRCBD    Butting contact, Red 'Down'
GRCBR    Butting contact, Red 'Right'
GCBCB    Green-to-Metal2 (Green-Contact-Blue-Contact-Blue)
RCBCB    Red-to-Metal2
BCB      Metal1-to-Metal2
```

Global variables and routines:

```
LAMBDA=REAL
    The basic dimension for describing layouts
WIDTH(REAL)=REAL
    The width of metal wire required to supply power to the given number
    of squares of pullup.  For example, to supply 100 minimum size
    inverters whose pullups are each 1/4 squares wide, the metal wire
    should be WIDTH(100*.25) wide.
WIDTH(COLOR)=REAL
    The default width of features for the given layer
SPACING(COLOR,COLOR)=REAL
    The spacing between feature edges of the two colors
CENTER_TO_CENTER(COLOR,COLOR)=REAL
    The center-to-center spacing for wires of default sizes on the two
    layers
Q_LOAD=REAL
    The capacitive load for the minimum size transistor
LOAD(COLOR,BOX)=REAL
    The capacitive load for the box
LOAD(WIRE)=REAL
    The capacitive load for the wire
```

There are routines for input/output of MRGs:

```
PLOT(PICTURE,PLOTTER);
    where PICTURE may be one of:
        an MRG
        AIF(file-name)
        AIF(file-name,list_of_colors)
        ...
    and PLOTTER may be one of:
        HP_7221A
        SCREEN
        HP_2649
        HP_1302
```

```
        AIF(file-name)
        AIF(file-name,list_of_colors)
        ...
MBB(MRG)=BOX     the minimum bounding box of the MRG
CIF2_OUT(MRG,file-name);     produces a CIF file
CIF2_IN(file-name)=MRG       reads a CIF file
```

Appendix 2: Imbedded Language Example

The code listed here generates the parameterized shift register cell presented in chapter 3. There are several parameters used in the routines below. The following table lists these parameters and states what information each parameter represents:

Parameter	Meaning
PU	The length of the pullup transistor in lambda
PD	The length of the pulldown transistor in lambda
HP	The width of a power line which supplies half a row of cells
SP	The power line width for a whole row
DP	The power line width for two rows
TP	The power line width for the entire array
NR	The number of shift register bits in a row
RB	The number of rows for each shift register (always an odd number, which indicates how many times the long shift register is folded)
NB	The number of shift registers in the array
NL	The number of bits in the last row of the shift register
TB	The total number of bits in each shift register

The first set of routines generate a single bit of the shift register. There are six routines: each generates layouts with one of the six aspect ratios. The first five cell layouts generate only one layout for the cell, but the last generates different layouts for adjacent bits. By alternating the two layouts, the total array size is less. For this reason, the SHIFT_CELL datatype is defined, which can contain two MRGs. The first five routines only use the ODD component of the SHIFT_CELL, while the last routine uses both.

```
TYPE    SHIFT_CELL= [EVEN,ODD:MRG];

DEFINE SHIFT1_CELL(PU,PD,DP:REAL)=SHIFT_CELL:
    BEGIN   VAR CVDD=REAL;
    DO  CVDD:=8+DP/2 MAX 4+PU;
    GIVE [ODD:( (GRCBO\AT 5#-5.;
                 GRCBU\AT 12#1;
                 RCBCB\AT 5#3;
                 GCB\AT (5#-12.-DP/2;.#CVDD);
                 WIRE(RED,(4#3;.#.5;1#-2.5;.#-3.));
                 WIRE(GREEN,(0#0;1.5#.;4#-2.5;.#-4.));
                 WIRE(RED,(5.5#-8.;9.5#-12.;15#.));
                 WIRE(GREEN,(6#-13.-DP/2;.#-14.;12#-8.;.#CVDD;6#.));
                 IF PD>=3
                   THEN POLYGON(GREEN,(6#-15.;8+PD#.;.#-9.;9.5#.;6#-12.5))
                   ELSE NIL FI;
                 BOX(RED,9#2\TO 15#2+PU);
                 BOX(YELLOW,9#0\TO 15#4+PU))\AT (0#0;14#0);
               WIRE(VIOLET,(5#3;.#9.5));
               WIRE(VIOLET,(19#3;.#-3.5)))]
```

```
    END
ENDDEFN

DEFINE SHIFT2_CELL(PU,PD,DP:REAL)=SHIFT_CELL:
    BEGIN   VAR CVDD=REAL;
    DO  CVDD:=11+DP/2+PU;
    GIVE [ODD:{ {GRCBD\AT 4#-7.;
                 GRCBU\AT 10#1;
                 RCBCB\AT 4#6+PU;
                 GCB\AT {4#-14.-DP/2;10#CVDD};
                 WIRE(RED,{4#6+PU;.#.5;1#-2.5;.#-3.});
                 WIRE(GREEN,{0#0;1.5#.;4#-2.5;.#-6.});
                 WIRE(RED,{4.5#-10.;8.5#-14.;12#.});
                 WIRE(GREEN,{5#-14.-DP/2;.#-14.;10#-10.;.#CVDD});
                 IF PD>=3
                   THEN POLYGON(GREEN,{5#-16.;6+PD#.;.#-12.;.-1#.+1;
                                       7.5#.;5#-13.5})
                   ELSE NIL FI;
                 BOX(RED,7#2\TO 13#2+PU);
                 BOX(YELLOW,7#.5\TO 12.5#4+PU)}\AT {0#0;12#0};
                WIRE(VIOLET,{4#6+PU;.#12.5+PU});
                WIRE(VIOLET,{16#6+PU;.#PU-.5})}]
    END
ENDDEFN

DEFINE SHIFT3_CELL(PU,PD,DP:REAL)=SHIFT_CELL:
    [ODD:{ {RCBCB\AT 4#4;
            GRCBR\AT {11#4;24#.};
            GCB\AT {16#-1.-DP/2;22+PU#9+DP/2};
            WIRE(RED,{5#4;.#-2.});
            WIRE(GREEN,{0#0;10#.;.#3});
            WIRE(RED,{14#5;.#8;20#.;.#-1.});
            WIRE(GREEN,{17#-1.-DP/2;.#1});
            WIRE(GREEN,{28+PU#0;23#.;.#5;23+PU#.;.#9+DP/2});
            IF PD>2 THEN BOX(GREEN,16#0\TO 23#PD) ELSE NIL FI;
            BOX(RED,25#2\TO 26+PU#8);
            BOX(YELLOW,23#2\TO 26+PU#10)}\AT {0#0;26+PU#0};
          WIRE(VIOLET,{4#4;.#9+DP/2});
          WIRE(VIOLET,{30+PU#4;.#-1.-DP/2})}]
ENDDEFN

DEFINE SHIFT4_CELL(PU,PD,SP,DP:REAL)=SHIFT_CELL:
    BEGIN   VAR M=MRG;
    DO  M:={RCBCB\AT 1#4;
            GRCBR\AT {8#4;21#4};
            GCB\AT {13#-1.-SP/2;19+PU#9+SP/2};
            WIRE(RED,{11#4;.#7;17#.;.#-2.});
            WIRE(GREEN,{14#-1.-SP/2;.#0;20#.;.#5;20+PU#.;.#9+SP/2});
            IF PD>2 THEN BOX(GREEN,13#-1.\TO 21#PD-1) ELSE NIL FI;
            BOX(RED,22#2\TO 23+PU#8);
            BOX(YELLOW,20#2\TO 23+PU#10)};
    GIVE [ODD:{M;
               M\MIRX\AT 13#-2.-SP;
               WIRE(RED,{2#4;.#-2.});
               WIRE(GREEN,{0#0;7#.;.#3});
               WIRE(RED,{16#-5.-SP;.#.+1.5;21#.+2.5;22#.});
               WIRE(GREEN,{20#0;.#-5.-SP});
               WIRE(GREEN,{33#-2.-SP;.#0;26+PU#.})}]
    END
ENDDEFN
```

```
DEFINE SHIFT5_CELL(PU,PD,SP,DP:REAL)=SHIFT_CELL:
    BEGIN   VAR M=MRG;Y1,Y2=REAL;
    DO  Y1:=-20+SP MAX 26+PD;
        Y2:= Y1+ (9+DP/2 MAX 6+PU);
        M:=(GRCBU\AT (0#10;.#Y1+2);
            BCB\AT 0#Y1+2;
            GCBCB\AT 0#-3.;
            WIRE(VIOLET,(0#-3.;.#Y1+2));
            WIRE(RED,(0#13;.#21.5;2#23.5;.#24+PD));
            POLYGON(GREEN,(2#20;5#.;.#23+PD;-2.#.;.#24));
            WIRE(GREEN,(-1.#23+PD;.#Y1+1;0#.;.#Y2-1;4#.));
            BOX(RED,-3.#Y1+3\TO 3#Y1+PU+3);
            BOX(YELLOW,-3.#Y1+1\TO 3#Y1+PU+5));
    GIVE [ODD:(M\AT 7#0;
               M\MIRY\AT 15#0;
               GCB\AT (11#19;.#Y2);
               RCBCB\AT (7#-17.;14#-10.);
               WIRE(GREEN,(0#-1.;.#.5;6#6.5;.#9));
               WIRE(GREEN,(8#-1.;.#.5;14#6.5;.#9));
               WIRE(RED,(6#-16.;3#.;.#.5;0#3.5));
               WIRE(RED,(13#-9.;11#.;.#.5;8#3.5)))]
    END
ENDDEFN

DEFINE SHIFT6_CELL(PU,PD,SP,DP,HP:REAL)=SHIFT_CELL:
    BEGIN   VAR M,ML=MRG;Y1,Y2=REAL;
    DO  Y1:= 23+HP MAX 29+PD;
        Y2:= Y1+ (9+DP/2 MAX 6+PU);
        M:=(GRCBU\AT (0#13;.#Y1+2);
            BCB\AT 0#Y1+2;
            WIRE(VIOLET,(0#10;.#Y1+2));
            WIRE(RED,(0#16;.#24.5;2#26.5;.#27+PD));
            POLYGON(GREEN,(2#23;5#.;.#26+PD;-2.#.;.#27));
            WIRE(GREEN,(-1.#26+PD;.#Y1+1;0#.;.#Y2-1;4#.));
            WIRE(GREEN,(0#0;.#12));
            BOX(RED,-3.#Y1+3\TO 3#Y1+PU+3);
            BOX(YELLOW,-3.#Y1+1\TO 3#Y1+PU+5));
        ML:=(GRCBD\AT 8#-1.;
             GCBCB\AT 3#-10.;
             RCBCB\AT 7#-17.;
             WIRE(GREEN,(3#-10.;.#-19.));
             WIRE(RED,(7#-17.;.#-4.));
             WIRE(VIOLET,(0#10;2.5#-9.));
             WIRE(VIOLET,(3#-28.;6.5#-18.)));
    GIVE [ODD:(M;
              M\MIRX\AT 3#-18.;
              ML;
              GCB\AT (4#22;.#Y2;7#-40.;.#-18.-Y2);
              RCB\AT (4#6;7#-24.);
              WIRE(RED,(-2.#9;3#.;.#7));
              WIRE(RED,(1#-27.;6#.;.#-25.)))
          EVEN:(M\MIRY\AT 8#0;
                M\ROT 180\AT 11#-18.;
                ML\AT 8#0;
                WIRE(RED,(10#9;5#.;.#7));
                WIRE(RED,(13#-27.;8#.;.#-25.)))]
    END
ENDDEFN
```

We would like a series of routines which would take the shift register bits from the routines above and generate complete arrays. As one might expect, much of the work in generating these arrays is independent of which type of aspect ratio one is using, and as one might also expect, there are some differences. Therefore, we have a routine FINISH which contains the code which can be common for each of the different cell types and individual routines for generating the type-specific data. The datatype SHIFT_ROW was created to contain the information which must be transferred between each of the SHIFTn_ROW routines and the FINISH routine.

```
TYPE    SHIFT_ROW= [FIRST,MIDDLE,LAST,ALT,ONLY:MRG
                    DOWN,UP,TOP,BOTTOM,REDGE:REAL];

DEFINE FINISH(R:SHIFT_ROW  RB,NB:INT  TP:REAL)=MRG:
    BEGIN   VAR CTC,TOP,BOTTOM=REAL;M=MRG;
    DO  CTC:=R.UP-R.DOWN;
        TOP:=R.UP+R.TOP;
        BOTTOM:=R.DOWN-R.BOTTOM;
        M:=IF RB>1 THEN
               (R.FIRST;
                IF RB>4 THEN
                   R.MIDDLE\AT 0#2*CTC\AT [NX:1  NY:RB/2-1 IY:2*CTC]
                ELSE NIL FI;
                R.LAST\AT 0#CTC*(RB-1);
                R.ALT\AT [NX:1  NY:RB/2  IY:2*CTC])
            ELSE R.ONLY FI;
    GIVE   M\AT [NX:1  NY:(NB+1)/2  IY:2*CTC*RB];
           M\MIRX\AT 0#2*RB*CTC+2*R.DOWN\AT [NX:1  NY:NB/2  IY:2*CTC*RB];
           WIRE(VIOLET,(-TP+1.5#BOTTOM+1.5;-3.#.;
                        .#CTC*(RB*NB-1)+TOP-1.5;-TP+1.5#.));
           WIRE(VIOLET,(R.REDGE+TP-1.5#BOTTOM+1.5;R.REDGE+3#.;
                        .#CTC*(RB*NB-1)+TOP-1.5;R.REDGE+TP-1.5#.));
           BOX(BLUE,-TP#BOTTOM\TO 0#CTC*(RB*NB-1)+TOP);
           BOX(BLUE,R.REDGE#BOTTOM\TO R.REDGE+TP#CTC*(RB*NB-1)+TOP)
                \AT TP#-BOTTOM
    END
ENDDEFN

DEFINE SHIFT1_ROW(PU,PD,DP,TP:REAL  NR,RB,NL:INT)=SHIFT_ROW:
    BEGIN   VAR M,P,R=MRG;LEDGE,REDGE,CVDD,CTC=REAL;
    DO  LEDGE:=IF RB>1 THEN 7 ELSE 0 FI;
        REDGE:=28*NR+LEDGE+3;
        CVDD:=8+DP/2 MAX 4+PU;
        CTC:=12+DP/2+CVDD;
        M:=SHIFT1_CELL(PU,PD,DP).OOD;
        P:=(BOX(BLUE,-1.#-12.-DP\TO REDGE-3#-12.);
            BOX(BLUE,3#8\TO REDGE+1#CVDD+DP/2);
            WIRE(VIOLET,(-3.#9.5;REDGE-3#.));
            WIRE(VIOLET,(3#-3.5;REDGE+3#.)));
        R:=M\AT LEDGE#0\AT [IX:28 NX:NR NY:1];
    GIVE [DOWN:-12.-DP/2
          UP:CVDD
          TOP:DP/2
          BOTTOM:DP/2
          REDGE:REDGE
```

```
            FIRST: {R;P;WIRE(GREEN,{1-TP#0;LEDGE#.})}
            MIDDLE: {R;P}
            LAST: {M\AT LEDGE#0\AT [IX:28 NX:NL NY:1];
                   P;WIRE(GREEN,{28*NL+LEDGE#0;REDGE+TP-1#.})}
            ALT: {R\ROT 180\AT REDGE#2*CVDD;
                  P\MIRX\AT 0#2*CVDD;
                  WIRE(GREEN,{4#2*CVDD;-1.#.;.#2*CTC;LEDGE#.});
                  WIRE(GREEN,{REDGE-4#0;.+5#.;.#2*CVDD;REDGE-LEDGE#.})}
            ONLY: {R;P;WIRE(GREEN,{1-TP#0;LEDGE#.});
                   WIRE(GREEN,{REDGE-3#0;REDGE+TP-1#.})}]
    END
ENDDEFN

DEFINE SHIFT2_ROW(PU,PD,DP,TP:REAL  NR,RB,NL:INT)=SHIFT_ROW:
    BEGIN   VAR M,P,R=MRG;LEDGE,REDGE,CVDD,CTC=REAL;
    DO  LEDGE:=IF RB>1 THEN 6 ELSE 1 FI;
        REDGE:=24*NR+LEDGE+3;
        CVDD:=11+DP/2+PU;
        CTC:=14+DP/2+CVDD;
        M:=SHIFT2_CELL(PU,PD,DP).ODD;
        P:={BOX(BLUE,-1.#-14.-DP\TO REDGE-3#-14.);
            BOX(BLUE,3#11+PU\TO REDGE+1#CVDD+DP/2);
            WIRE(VIOLET,{-3.#PU+12.5;REDGE-3#.});
            WIRE(VIOLET,{3#PU-.5;REDGE+3#.})};
        R:=M\AT LEDGE#0\AT [IX:24 NX:NR NY:1];
    GIVE {DOWN:-14.-DP/2
          UP:CVDD
          TOP:DP/2
          BOTTOM:DP/2
          REDGE:REDGE
          FIRST: {R;P;WIRE(GREEN,{1-TP#0;LEDGE#.})}
          MIDDLE: {R;P}
          LAST: {M\AT LEDGE#0\AT [IX:24 NX:NL NY:1];
                 P;WIRE(GREEN,{24*NL+LEDGE#0;REDGE+TP-1#.})}
          ALT: {R\ROT 180\AT REDGE#2*CVDD;
                P\MIRX\AT 0#2*CVDD;
                WIRE(GREEN,{4#2*CVDD;-1.#.;.#2*CTC;LEDGE#.});
                WIRE(GREEN,{REDGE-4#0;.+5#.;.#2*CVDD;REDGE-LEDGE#.})}
          ONLY: {R;P;WIRE(GREEN,{1-TP#0;LEDGE#.});
                 WIRE(GREEN,{REDGE-3#0;REDGE+TP-1#.})}]
    END
ENDDEFN

DEFINE SHIFT3_ROW(PU,PD,DP,TP:REAL  NR,RB,NL:INT)=SHIFT_ROW:
    BEGIN   VAR M,P,R=MRG;LEDGE,REDGE,CTC=REAL;
    DO  LEDGE:=IF RB>1 THEN PU-5 MAX 1 ELSE 1 FI;
        REDGE:=(52+2*PU)*NR+ IF RB>1 THEN 2+ABS(PU-6) ELSE 2 FI;
        CTC:=10+DP;
        M:=SHIFT3_CELL(PU,PD,DP).ODD;
        P:={BOX(BLUE,-1.#-1.-DP\TO REDGE-3#-1.);
            BOX(BLUE,3#9\TO REDGE+1#9+DP);
            WIRE(VIOLET,{-3.#9+DP/2;REDGE-3#.});
            WIRE(VIOLET,{3#-1.-DP/2;REDGE+3#.})};
        R:=M\AT LEDGE#0\AT [IX:52+2*PU NX:NR NY:1];
    GIVE {DOWN:-1.-DP/2
          UP:9+DP/2
          TOP:DP/2
          BOTTOM:DP/2
          REDGE:REDGE
          FIRST: {R;P;WIRE(GREEN,{1-TP#0;LEDGE#.})}
          MIDDLE: {R;P}
```

```
            LAST: {M\AT LEDGE#0\AT [IX:52+2*PU NX:NL NY:1];
                  P;WIRE(GREEN,{(52+2*PU)*NL+LEDGE#0;.#4;REDGE+TP-1#.;.#0})}
            ALT: {R\ROT 180\AT REDGE#18+DP;
                  P\MIRX\AT 0#18+DP;
                  WIRE(GREEN,{6-PU MAX 0#18+DP;-1.#.;.#2*CTC;LEDGE#.});
                  WIRE(GREEN,{REDGE-(6-PU MAX 0)#0;REDGE+1#.;.#18+DP;
                              REDGE-LEDGE#.})}
            ONLY: {R;P;WIRE(GREEN,{1-TP#0;LEDGE#.});
                  WIRE(GREEN,{REDGE-1#0;.#4;REDGE+TP-1#.;.#0})}]
    END
ENDDEFN

DEFINE SHIFT4_ROW(PU,PD,SP,DP,TP:REAL  NR,RB,NL:INT)=SHIFT_ROW:
    BEGIN   VAR M,P,R=MRG;REDGE,CTC=REAL;
    DO  REDGE:=(26+PU)*NR+15;
        CTC:=20+2*SP;
        M:=SHIFT4_CELL(PU,PD,SP,DP).ODD;
        P:={BOX(BLUE,-1.#-1.-SP\TO REDGE-3#-1.);
            BOX(BLUE,3#9\TO REDGE+1#9+SP);
            BOX(BLUE,3#-11.-2*SP\TO REDGE+1#-11.-SP);
            WIRE(VIOLET,{-3.#4;REDGE-3#.});
            WIRE(VIOLET,{3#-6.-SP;REDGE+3#.})};
        R:=M\AT 4#0\AT [IX:26+PU NX:NR NY:1];
    GIVE [DOWN:-11.-3.*SP/2.
          UP:9+SP/2
          TOP:SP/2
          BOTTOM:SP/2
          REDGE:REDGE
          FIRST: {R;P;WIRE(GREEN,{1-TP#0;4#.})}
          MIDDLE: {R;P}
          LAST: {M\AT 4#0\AT [IX:26+PU NX:NL NY:1];
                 P;WIRE(GREEN,{(26+PU)*NL+4#0;.#4;REDGE+TP-1#.;.#0})}
          ALT: {R\ROT 180\AT REDGE#18+SP;
                P\MIRX\AT 0#18+SP;
                WIRE(GREEN,{11#18+SP;-1.#.;.#2*CTC;4#.});
                WIRE(GREEN,{REDGE-11#0;.+10#.;.#18+SP;REDGE-4#.})}
          ONLY: {R;P;WIRE(GREEN,{1-TP#0;4#.});
                 WIRE(GREEN,{REDGE-11#0;.#4;REDGE+TP-1#.;.#0})}]
    END
ENDDEFN

DEFINE SHIFT5_ROW(PU,PD,SP,DP,TP:REAL  NR,RB,NL:INT)=SHIFT_ROW:
    BEGIN   VAR M,P,R=MRG;REDGE,CTC,Y1,Y2=REAL;
    DO  REDGE:=16*NR+2;
        Y1:= 21+SP MAX 27+PD;
        Y2:= Y1+ (9+DP/2 MAX 6+PU);
        CTC:=Y2+16;
        M:=SHIFT5_CELL(PU,PD,SP,DP).ODD;
        P:={BOX(BLUE,-1.#18\TO REDGE-3#Y1-3);
            BOX(BLUE,3#Y1+9\TO REDGE+1#Y2+DP/2);
            WIRE(VIOLET,{-3.#-16.;REDGE-3#.});
            WIRE(VIOLET,{3#-9.;REDGE+3#.})};
        R:=M\AT -2.#1\AT [IX:16 NX:NR NY:1];
    GIVE [DOWN:-16.
          UP:Y2
          TOP:DP/2
          BOTTOM:2
          REDGE:REDGE
          FIRST: {R;P;WIRE(GREEN,{1-TP#0;-2.#.})}
          MIDDLE: {R;P}
          LAST: {M\AT -2.#1\AT [IX:16 NX:NL NY:1];
```

```
                P;WIRE(GREEN,(16*NL-2#0;REDGE+TP-1#.)))
            ALT:(R\ROT 180\AT REDGE#2*Y2;
                 P\MIRX\AT 0#2*Y2;
                 WIRE(GREEN,(4#2*Y2;-6.#.;.#2*CTC;-2.#.));
                 WIRE(GREEN,(REDGE-4#0;.+10#.;.#2*Y2;REDGE+2#.)))
            ONLY:(R;P;WIRE(GREEN,(1-TP#0;-2.#.));
                 WIRE(GREEN,(REDGE-4#0;REDGE+TP-1#.)))]
    END
ENDDEFN

DEFINE SHIFT6_ROW(PU,PD,SP,DP,TP,HP:REAL  NR,RB,NL:INT)=SHIFT_ROW:
    BEGIN   VAR ME,MO,P,R=MRG;REDGE,CTC,Y1,Y2=REAL;
    DO  REDGE:=8*NR+18.5;
        Y1:=23+HP MAX 29+PD;
        Y2:=Y1+ (9+SP/2 MAX 6+PU);
        CTC:=2*Y2+18;
        [ODD:MO  EVEN:ME]:=SHIFT6_CELL(PU,PD,SP,DP,HP);
        P:=(BOX(BLUE,-1.#20\TO REDGE-3#Y1-3);
            BOX(BLUE,-1.#-15.-Y1\TO REDGE-3#-38.);
            BOX(BLUE,3#Y1+9\TO REDGE+1#Y2+SP/2);
            BOX(BLUE,3#-18.-Y2-SP/2\TO REDGE+1#-27.-Y1);
            WIRE(BLUE,(REDGE-5#6;5#.));
            WIRE(BLUE,(5#-24.;REDGE-5#.;.#-33.));
            BCB\AT (5#6;REDGE-5#-33.);
            WIRE(VIOLET,(-3.#6;5#.));
            WIRE(VIOLET,(REDGE-5#-33.;REDGE+3#.)));
        R:=(MO\AT 11.5#0\AT [IX:16 NX:(NR+1)/2 NY:1];
            ME\AT 11.5#0\AT [IX:16 NX:NR/2 NY:1]);
    GIVE [DOWN:-18.-Y2
          UP:Y2
          TOP:SP/2
          BOTTOM:SP/2
          REDGE:REDGE
          FIRST:(R;P;WIRE(GREEN,(1-TP#0;11.5#.)))
          MIDDLE:(R;P)
          LAST:(MO\AT 11.5#0\AT [IX:16 NX:(NL+1)/2 NY:1];
                ME\AT 11.5#0\AT [IX:16 NX:NL/2 NY:1];
                P;WIRE(GREEN,(8*NL+11.5#0;REDGE+TP-1#.)))
          ALT:(R\ROT 180\AT REDGE#2*Y2;
               P\MIRX\AT 0#2*Y2;
               WIRE(GREEN,(6#2*Y2;2#.;.#2*CTC;11.5#.));
               WIRE(GREEN,(REDGE-6#0;.+2#.;.#2*Y2;REDGE-11.5#.)))
          ONLY:(R;P;WIRE(GREEN,(1-TP#0;11.5#.));
               WIRE(GREEN,(REDGE-14#0;REDGE+TP-1#.)))]
    END
ENDDEFN
```

Each shift array function is now trivial: They each call their corresponding SHIFTn_ROW function and the FINISH function. Also note that each of the SHIFTn_ROW functions requires a subset of the total list of parameters, but that the SHIFTn_ARRAY functions require all parameters, but do not use all of the parameters. This is done so that other programs do not have to be aware of the differences in the parameter requirements.

```
DEFINE SHIFT1_ARRAY(PU,PD,SP,DP,TP,HP:REAL  NR,RB,NB,NL:INT)=MRG:
    FINISH(SHIFT1_ROW(PU,PD,DP,TP,NR,RB,NL),RB,NB,TP)
```

```
ENDDEFN

DEFINE SHIFT2_ARRAY(PU,PD,SP,DP,TP,HP:REAL  NR,RB,NB,NL:INT)=MRG: .
    FINISH(SHIFT2_ROW(PU,PD,DP,TP,NR,RB,NL),RB,NB,TP)
ENDDEFN

DEFINE SHIFT3_ARRAY(PU,PD,SP,DP,TP,HP:REAL  NR,RB,NB,NL:INT)=MRG:
    FINISH(SHIFT3_ROW(PU,PD,DP,TP,NR,RB,NL),RB,NB,TP)
ENDDEFN

DEFINE SHIFT4_ARRAY(PU,PD,SP,DP,TP,HP:REAL  NR,RB,NB,NL:INT)=MRG:
    FINISH(SHIFT4_ROW(PU,PD,SP,DP,TP,NR,RB,NL),RB,NB,TP)
ENDDEFN

DEFINE SHIFT5_ARRAY(PU,PD,SP,DP,TP,HP:REAL  NR,RB,NB,NL:INT)=MRG:
    FINISH(SHIFT5_ROW(PU,PD,SP,DP,TP,NR,RB,NL),RB,NB,TP)
ENDDEFN

DEFINE SHIFT6_ARRAY(PU,PD,SP,DP,TP,HP:REAL  NR,RB,NB,NL:INT)=MRG:
    FINISH(SHIFT6_ROW(PU,PD,SP,DP,TP,HP,NR,RB,NL),RB,NB,TP)
ENDDEFN
```

To choose between the various possible cell types and configurations, we need to know the sizes of all arrays. Since we want to try many configurations, but we will only use one, we don't want to perform the expensive computation of generating the arrays until we know which one we want. The SIZE function takes the pertinent parameters and computes what the array size would be if we were to actually generate that array. This computation is very cheap both in terms of time and memory space. The SIZE function returns a POINT whose x coordinate is the horizontal size of the array and whose y coordinate is the vertical size. The SIZE function also returns a Suspendable Function. The suspendable function is generated inside the //: \\ characters. This function is not executed, but is a freeze-dried function call. In this usage, all of the parameters for the call to the SHIFTn_ARRAY functions are evaluated, but the SHIFTn_ARRAY function is not called. At any time in the future we may, if we wish, actually perform the function call and receive the resulting layout. The datatype SHIFT_MAKER is our freeze-dried function call, and SHIFT_RESULT is the datatype which SIZE returns, containing both the array size and the suspendable function.

```
TYPE    SHIFT_MAKER=//MRG\\;

        SHIFT_RESULT=[SIZE:POINT  SS:SHIFT_MAKER];

DEFINE SIZE(NB,TB:INT  POWER:REAL  CLASS,RB:INT)=SHIFT_RESULT:
    BEGIN   VAR PU,PD,SP,DP,TP,HP=REAL;NR,NL=INT;
    DO  PU:=2./POWER MAX 16./3.;
        PD:=32./PU MAX 2.;
        NR:=(TB+RB-1)/RB;
        NL:=TB-(RB-1)*NR;
        SP:=WIDTH(POWER*NR);
```

```
        DP:=WIDTH(2*POWER*NR);
        TP:=WIDTH(TB*NB*POWER);
        HP:=WIDTH(POWER*NR/2);
    GIVE IF CLASS=1 THEN
            [SIZE: 28*NR+ IF RB>1 THEN 10 ELSE 3 FI +2*TP #
                  ((8+DP/2 MAX 4+PU)+12+DP/2)*NB*RB+DP
             SS://:SHIFT1_ARRAY[PU,PD,SP,DP,TP,HP,NR,RB,NB,NL]\\]
         EF CLASS=2 THEN
            [SIZE: 24*NR+ IF RB>1 THEN 9 ELSE 4 FI +2*TP #
                  (25+DP+PU)*RB*NB+DP
             SS://:SHIFT2_ARRAY[PU,PD,SP,DP,TP,HP,NR,RB,NB,NL]\\]
         EF CLASS=3 THEN
            [SIZE: (52+2*PU)*NR+ IF RB>1 THEN 2+ABS(PU-6) ELSE 2 FI +2*TP #
                  (10+DP)*RB*NB+DP
             SS://:SHIFT3_ARRAY[PU,PD,SP,DP,TP,HP,NR,RB,NB,NL]\\]
         EF CLASS=4 THEN
            [SIZE: (26+PU)*NR+15+2*TP # (20+2*SP)*RB*NB+SP
             SS://:SHIFT4_ARRAY[PU,PD,SP,DP,TP,HP,NR,RB,NB,NL]\\]
         EF CLASS=5 THEN
            [SIZE: 16*NR+2+2*TP #
                  (16+(21+SP MAX 27+PD)+(9+DP/2 MAX 6+PU))*RB*NB+DP/2+2
             SS://:SHIFT5_ARRAY[PU,PD,SP,DP,TP,HP,NR,RB,NB,NL]\\]
         ELSE
            [SIZE: 8*NR+18.5+2*TP #
                  (2*((23+HP MAX 29+PD) + (9+SP/2 MAX 6+PU))+18)*RB*NB+SP
             SS://:SHIFT6_ARRAY[PU,PD,SP,DP,TP,HP,NR,RB,NB,NL]\\] FI
    END
ENDDEFN
```

The SHIFT_CELL function is our actual shift cell. We call it passing the number of shift registers required, the number of bits per register, the power requirements, the desired area, and the oversize costs. This function generates several candidates by calling the SIZE function and returns the array which best matches the desired size. If there are candidates which fit within the desired area, the one with the closest match to the area is chosen. If no candidates match, the amount of oversize in both x and y for all candidates is multiplied by the weights and the candidate with the smallest resulting cost is used.

```
DEFINE SHIFT_CELL(NB,TB:INT  POWER:REAL  SIZE,WEIGHT:POINT)=MRG:
    BEGIN   VAR I,J=INT;
            DEFINE BEST(A,B:SHIFT_RESULT)=SHIFT_RESULT:
                IF A.SIZE<SIZE THEN
                    IF (B.SIZE<SIZE)&(DIST(B.SIZE,SIZE)<DIST(A.SIZE,SIZE))
                    THEN B ELSE A FI
                EF B.SIZE<SIZE THEN B
                EF ABS(((A.SIZE-SIZE)\SCALED_BY WEIGHT) MAX 0#0 ) <
                   ABS(((B.SIZE-SIZE)\SCALED_BY WEIGHT) MAX 0#0 ) THEN A
                ELSE B FI
            ENDDEFN
    <*(\BEST SIZE(NB,TB,POWER,I,J) FOR I FROM 1 TO 6;!!
                                   FOR J FROM 1 TO 21 BY 2;).SS*>
    END
ENDDEFN
```

When the user has specific size requirements for the shift array, a direct call on the SHIFT_CELL function is used. Most of the time, however, the user can make tradeoffs of chip area between various units. In these cases, the user may wish to see the sizes of the various candidates. The GRAPH function will plot a graph of all candidates within a maximum size limit while the TABLE function prints a table of this same data. Given this information, the user can see what the possible areas are for the arrays, which will aid in the planning of other circuit sizes. These functions take the number of shift registers, the number of bits per register, the power required, the maximum number of folds used (although the SHIFT_CELL as written always uses a maximum of 21), and the maximum candidate size which filters the output.

```
DEFINE GRAPH(NB,TB:INT  POWER:REAL  N:INT  MAX:POINT)=MRG:
    BEGIN   VAR M=MRG;CLASS,RB=INT;P,Q=POINT;SPS=SPS;SP=SP;
    DO  SPS:=(COLLECT
                (COLLECT SIZE(NB,TB,POWER,CLASS,RB).SIZE
                 FOR RB FROM 1 TO N BY 2;)
              FOR CLASS FROM 1 TO 6;);
        P:= MAX IF P<MAX THEN P ELSE 0#0 FI FOR (P) $E SPS;;
        SPS:=(COLLECT
                (COLLECT Q.X*500/P.X # Q.Y*500/P.Y FOR Q $E SP;)
              FOR SP $E SPS;);
    GIVE (COLLECT WIRE(BLUE,0,(COLLECT Q FOR Q $E SP;WITH Q<500#500;))
          FOR SP $E SPS;;
          COLLECT (COLLECT M\AT Q FOR Q $E SP;WITH Q<500#500;)
          FOR SP $E SPS;&& FOR M $E
            ( BOX(RED,-5.#-5.\TO 5#5) ;
              POLYGON(RED,(-5.#-4.;5#.;0#4)) ;
              POLYGON(RED,(0#5;-3.#-4.;5#2.5;-5.#.;3#-4.)) ;
               (WIRE(RED,0,(5#5;-5.#-5.));WIRE(RED,0,(-5.#5;5#-5.))) ;
              POLYGON(RED,(5#0;0#5;-5.#0;0#-5.));
              POLYGON(RED,(2#5;-2.#.;-5.#2;.#-2.;-2.#-5.;2#.;5#-2.;.#2)) );;
          WIRE(GREEN,0,(0#500;0#0;500#0));
          SC(P.X)\PAINTED BLACK\SCALED_BY 2#2\AT 500#10;
          SC(P.Y)\PAINTED BLACK\SCALED_BY 2#2\AT 10#472;
          'NB:'$$SC(NB)\PAINTED BLACK\SCALED_BY 2#2\AT 500#130;
          'TB:'$$SC(TB)\PAINTED BLACK\SCALED_BY 2#2\AT 500#90;
          'POWER:'$$SC(POWER)\PAINTED BLACK\SCALED_BY 2#2\AT 500#50)
    END
ENDDEFN

DEFINE TABLE(NB,TB:INT  POWER:REAL  N:INT  MAX:POINT):
    BEGIN   VAR CLASS,RB=INT;P=POINT;
    FOR CLASS FROM 1 TO 6;!! FOR RB FROM 1 TO N BY 2; DO
        P:=SIZE(NB,TB,POWER,CLASS,RB).SIZE;
        IF P<MAX THEN
            WRITE('CLASS:');WRITE(CLASS);TAB;
            WRITE('ROWS/BIT:');WRITE(RB);TAB;
            WRITE('SIZE:');WRITE(P);CRLF; FI
    END
    END
ENDDEFN
```

Appendix 3: River Routers

This appendix discusses the design of a river router and illustrates some of the extensions which augment the usefulness of river routers. River routers are used to interconnect the connectors along the adjacent edges of two cells. The following restrictions apply to the connectors, and can be thought of as the definition of a river route:

1) There must be a one-to-one mapping between connectors of the two cells.

2) Corresponding connectors must be on the same mask layer.

3) Each set of connectors must satisfy the design rules for minimum width wires.

4) Adjacent connector pairs on dependent mask layers must not cross.

The first condition simply states that the two sets of connectors be of the same length. We will connect the first connector of one list to the first connector of the other list; the second connectors will be interconnected, etc. The second condition assures us that we can route a single wire between the two connectors without changing mask layers. The third condition assures us that we can indeed route wires to all the connectors without violating the design rules. The fourth condition assures us that we do not have to cross wires. If wires had to cross, we would have to change layers, and we do not wish to change layers with our wires (see condition 2). Dependent layers are layers that produce undesirable side-effects when wires cross. For instance, in NMOS design, when diffusion and polysilicon cross, a transistor is formed. Hence, diffusion and polysilicon are dependent layers. On the other hand, the metal layer is independent of polysilicon and diffusion since metal wires may freely cross wires of these other layers. Notice that every layer is dependent with itself: If two wires of the same layer cross, they short together.

Based upon these conditions, there are a few properties of river routes which can be used. One of these properties has already been mentioned: the interconnection between two connectors will be a single wire on a single mask layer. A second property is that independent layers can be routed independently. We have noticed that, in NMOS, metal wires can arbitrarily cross polysilicon or diffusion wires. Therefore, we can route all of the metal wires as a group, and then route all of the

polysilicon and diffusion wires as a group. This also allows connector pairs to cross, provided the connector pairs are on independent layers.

We can also divide the routing task for each set of dependent layers into groups. We will define a group to be all adjacent connector-pairs on dependent layers which route in the same direction. Using figure A3-1 as an example, we see that the first three connector pairs have wires slanting to the left as we go from top to bottom. The next three connectors slant to the right, and the final three connectors slant to the left. We can divide the connector pairs into groups and route each group independently. This is possible because each wire drawn will only move horizontally in one direction, towards its destination. We can also route these independent groups as if they were dependent. This allows us to separate the connectors into two groups: those that tend to the left and those that tend to the right (any wires which need only be vertical can belong in either group).

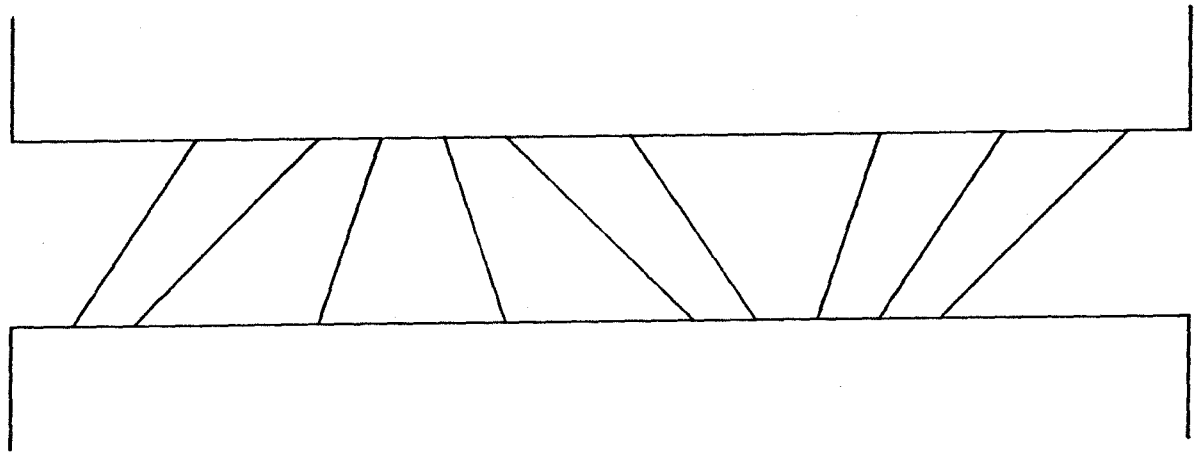


Fig. A3-1: Connector Pairs

Another property we will use is that each wire depends only upon one other wire in the route: its adjacent neighbor in its direction of travel. If every wire maintains proper distance from its neighbor in its direction of travel, we will not have design rule violations between wires. We will use this property to determine the order of routing wires. In the left-going group, we will route the left most wire first, followed by the next-to-the-left most wire, etc. We will route the right-going wires starting with the right most wire. The first wire in each group will move directly over to its destination connector's x coordinate and wait. The second wire in each group now only needs to avoid this one wire as it heads toward its

destination. In a like manner, each wire will only have to consider the previous wire as it is generating its path.

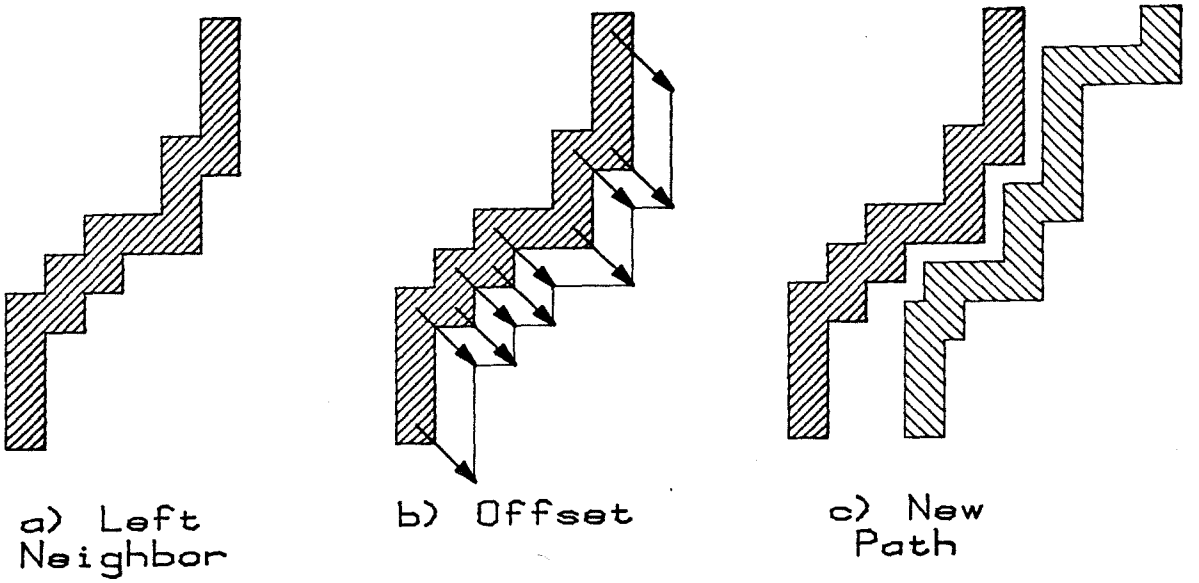


Fig. A3-2: Computing New Path

The final property we will use is that the design rule spacing between wires is uniform in both directions. This allows us to compute the majority of a wire's path by simply shifting the points from the previous path. In figure A3-2a we see the path of one wire. If we shift the points of this path over in x and down in y, each time by the minimum design rule spacing for the two layers in question, we have the path of the wire which is as close to the given wire as possible. Given this new path, we need only fix the ends of this path to have the route for the next wire. We will remove any points of this path which lie beyond the destination and will append segments to the front of the wire which connect to the starting connector (fig A3-2c). This efficiently generates each wire given the neighbor's wire. As we have already stated, the first wire is trivial to implement: We move from the initial connector over to the final connector's x coordinate, then down. We can now prove that each wire only draws in one direction. The first wire draws only in one direction, as shown in the previous statement. The central portion of every wire follows its neighbor's path until the destination coordinate is reached. Hence, once the central portion of the wire is reached, the wire only heads in the direction of its destination. To complete the proof, we must show that the start of the wire does not move in the opposite direction. The end of the shifted portion of the wire is at minimum design rule spacing from the neighbor's wire. For the initial

segment of the wire to run in the opposite direction, the starting connector must be closer to the neighbor's wire then design rules allow. This is a violation of condition 3. Therefore, every wire draws in one direction, which completes the inductive proof. Given this, we can then prove that the extend of a wire is limited by the x coordinates of its two connectors. If the wire ever extended beyond one of the two connectors, it could never connect to the connector since it would have to change directions. Therefore, wire extents are limited, and we can separate the routes into groups.

The following code is the basic river router routine. We will discuss the Forbidden Zones later, for now assume that they are identity functions. The River Node contains the coordinates of the two connectors, and the common color of the connectors. The river routing routine returns a River Return, which contains the layout and the height, which is the height of the completed route. The river routing routine calls a routine to route the individual sets of dependent layers. This routing, GROUP_ROUTE, also returns a RIVER_RETURN, but this routine uses the DONE component. The Done component contains all of the river nodes at the end of the route. Since we can not state how tall the river route will be until the route is completed, we do not know how long to make each of the final wire segments until we have finished the rest of the route. We put each of the nodes into the Done component when we are finished jogging them, and we look at these nodes after all the wires are jogged to add the final segments to the wires.

```
TYPE    RIVER_NODE= [FROM,TO:POINT  COLOR:COLOR];
        RIVER_NODES= { RIVER_NODE };
        RIVER_RETURN= [LAYOUT:MRG  HEIGHT:REAL  DONE:RIVER_NODES];
        RIVER_RETURNS= { RIVER_RETURN };
        FORBIDDEN_ZONE= //WIRE(SP,REAL,COLOR)\\;

DEFINE SORT(OLD:RIVER_NODES)=RIVER_NODES:
    BEGIN   VAR NEW=RIVER_NODES;N1,N2=RIVER_NODE;I,J=INT;
    DO  NEW:=NIL;
        WHILE DEFINED(OLD); DO
            N1:=OLD[1];
            I:=1;
            (FOR N2 $E OLD[2-];&& FOR J FROM 2 BY 1;)WITH N2.FROM.X>N1.FROM.X;
            DO  I:=J;
                N1:=N2;
            END
```

```
            NEW::= N1<$;
            OLD[I]:=NIL;
        END
    GIVE NEW
    END
ENDDEFN

DEFINE GROUP_ROUTE(LIST:RIVER_NODES  MIN,TOP,BOT:REAL
                   FZ1,FZ2:FORBIDDEN_ZONE)=RIVER_RETURN:
    BEGIN   VAR RIGHT,DONE=RIVER_NODES;LAST_PATH=SP;
                N,LAST_NODE=RIVER_NODE;LAST_COLOR=COLOR;COUNT=INT;
                L1,L2=MRGS;LOW,SPACE=REAL;P=POINT;
            DEFINE ADD(C:COLOR):
                BEGIN   VAR P=POINT;
                [PATH:LAST_PATH  WIDTH:LOW]:=<*FZ1*>(LAST_PATH,LOW,C);
                L2::= WIRE(C,(<*FZ2*>(LAST_PATH,LOW,C)).PATH) <$;
                COUNT::=+1;
                IF COUNT>40 THEN
                    L1::= DISK(L2) <$;
                    L2:=NIL;
                    COUNT:=0; FI
                END
            ENDDEFN
    DO  RIGHT:=NIL;
        DONE:=NIL;
        LAST_PATH:=NIL;
        LAST_NODE:=[FROM:-999999.#999999  COLOR:RED];
        LIST::=\SORT;
        COUNT:=0;
        L1:=NIL;
        L2:=NIL;
        LOW:=TOP-MIN;
        LAST_COLOR:=RED;
        FOR N $E LIST; DO
            IF N.FROM.X<N.TO.X THEN RIGHT::= N <$;
            ELSE    SPACE:=CENTER_TO_CENTER(N.COLOR,LAST_COLOR);
                    LAST_COLOR:=N.COLOR;
                    IF N.TO.X-SPACE>=LAST_NODE.FROM.X THEN
                        LAST_PATH:=(N.FROM;
                                    IF N.FROM.Y\IS_CLOSE_TO TOP
                                    THEN NIL ELSE .#TOP FI;
                                    N.TO.X#.);
                    ELSE
                        LAST_PATH:=(COLLECT P+SPACE#-SPACE
                                    FOR P $E LAST_PATH;
                                    WITH (P.Y=<TOP)&(P.X+SPACE>=N.TO.X););
                        IF LAST_PATH[1].X<N.FROM.X THEN
                            LAST_PATH::=(N.FROM;
                                         IF N.FROM.Y\IS_CLOSE_TO TOP
                                         THEN NIL ELSE .#TOP FI;
                                         LAST_PATH[1].X#.)$$;
                        ELSE LAST_PATH::= N.FROM<$; FI
                        P:=REVERSE(LAST_PATH)[1];
                        IF -(P.X\IS_CLOSE_TO N.TO.X)
                        THEN LAST_PATH::= $> N.TO.X#P.Y; FI
                        LOW::= MIN P.Y;
                        LAST_PATH:=REFRESH(LAST_PATH); FI
                    ADD(N.COLOR);
                    DONE::=[FROM:N.TO.X#REVERSE(LAST_PATH)[1].Y
                            TO:N.TO
                            COLOR:N.COLOR]<$;
```

```
                    LAST_NODE:=N; FI END
        LAST_NODE:=[FROM:999999#999999 COLOR:RED];
        FOR N $E RIGHT; DO
            SPACE:=CENTER_TO_CENTER(LAST_COLOR,N.COLOR);
            LAST_COLOR:=N.COLOR;
            IF N.TO.X+SPACE=<LAST_NODE.FROM.X THEN
                LAST_PATH:={N.FROM;IF N.FROM.Y\IS_CLOSE_TO TOP
                            THEN NIL ELSE .#TOP FI;N.TO.X#.};
            ELSE    LAST_PATH:={COLLECT P-SPACE#SPACE FOR P $E LAST_PATH;
                                WITH (P.Y=<TOP)&(P.X-SPACE=<N.TO.X);};
                    IF LAST_PATH[1].X>N.FROM.X THEN
                        LAST_PATH::= {N.FROM;
                                      IF N.FROM.Y\IS_CLOSE_TO TOP
                                      THEN NIL ELSE .#TOP FI;
                                      LAST_PATH[1].X#.}$$;
                    ELSE LAST_PATH::= N.FROM<$; FI
                    P:=REVERSE(LAST_PATH)[1];
                    IF -(P.X\IS_CLOSE_TO N.TO.X)
                    THEN LAST_PATH::= $> N.TO.X#P.Y; FI
                    LOW::= MIN P.Y;
                    LAST_PATH:=REFRESH(LAST_PATH); FI
            ADD(N.COLOR);
            DONE::=[FROM:N.TO.X#REVERSE(LAST_PATH)[1].Y
                    TO:N.TO
                    COLOR:N.COLOR]<$;
            LAST_NODE:=N; END
        IF COUNT>0 THEN L1::= DISK(L2) <$; FI
    GIVE [LAYOUT:DISK(L1) HEIGHT:LOW DONE:DONE]
    END
ENDDEFN

DEFINE RIVER_ROUTE(LIST:RIVER_NODES  MIN,TOP,BOT:REAL
                   FZ1,FZ2:FORBIDDEN_ZONE)=RIVER_RETURN:
    BEGIN   VAR N=RIVER_NODE;LISTS=RIVER_RETURNS;CLASS=INT;LOW=REAL;
                R=RIVER_RETURN;
    DO  LISTS:=NIL;
        WHILE DEFINED(LIST); DO
            CLASS:=CLASS(LIST[1].COLOR);
            LISTS::= GROUP_ROUTE({COLLECT N FOR N $E LIST;
                                  WITH N.COLOR\CLASS=CLASS;},MIN,TOP,BOT,
                                 FZ1,FZ2)<$;
            LIST:={COLLECT N FOR N $E LIST;WITH N.COLOR\CLASS<>CLASS;};
        END
        LOW:= MIN R.HEIGHT FOR R $E LISTS;;
    GIVE [LAYOUT:DISK({COLLECT R.LAYOUT FOR R $E LISTS;;
                       COLLECT WIRE(N.COLOR,
                           (<*FZ2*>{N.FROM;.#LOW+N.TO.Y-BOT},0,N.COLOR).PATH))
                       FOR [DONE:{N}] $E LISTS;})
          HEIGHT:LOW]
    END
ENDDEFN
```

The RIVER ROUTE routine takes the list of connector pairs and routes between them. The route is assumed to be horizontal. To generate vertical routes, the connector positions can be rotated 270 degrees, and the resulting layout rotates 90 degrees. The MIN parameter is used to specify a minimum width for the route. We can not state maximum width of the route, but we may wish to state a minimum width for

the route. (For example, we may wish to run some horizontal metal wires over the route, so we would require the route to be tall enough to allow all of the metal wires to fit between the cells.) In some cases, the connectors do not lie on the perimeter of the cell, but rather lie inside the cell's boundary. To connect to the point, we either have to examine the entire set of geometry contained in the cell or we have to have conventions for connecting to the cell. We will use the convention that if a point lies within the cell boundary, we may draw a minimum width wire from the connector straight to the edge of the cell. The TOP and BOT parameters indicate the boundries of the two cells. If a node's FROM point has a Y value greater than TOP, a wire is drawn from the point straight down to TOP, before the river route begins. Similarly, if the TO point has a Y value less than BOT, a wire is drawn. The FZ1 parameter is used to jog the wire, and the FZ2 parameter is used to translate the wire. These operations are discussed later.

The RIVER_ROUTE routine takes all of the connectors and separates them into groups, based upon the color of the connectors. The CLASS routine in ICLIC is used to determine the dependence of the layers. Dependent layers have the same class. RIVER_ROUTE calls GROUP_ROUTE will all of the connectors in each class. Once GROUP_ROUTE has been called for each group, RIVER_ROUTE determines the height of the route, extends all of the wires, and returns the layout.

GROUP_ROUTE routes all of the wires which slope to the left first, then it routes all of the wires which slope to the right. For each wire, it determines the design rule spacing between this wire and the previous wire. It then checks to see if the previous wire is outside the range of the current wire, in which case it can immediately draw the current wire connecting directly to its desired location. If the previous wire was in range, all of the points in the previous wire are diagonally shifted by the design rule spacing, and the two ends of the wire are adjusted to fit the TO and FROM points of the current wire. Given the current wire, the ADD routine is called. The ADD routine passes the wire to the first FORBIDDEN_ZONE, which may jog the wire. The result of the jogs becomes the official path of the wire, which the neighboring wires must avoid. This is also passed to the second FORBIDDEN_ZONE, which may arbitrarily map the wire from the river route coordinate system to the chip coordinate system. For standard river routes, these two FORBIDDEN_ZONEs are indentity functions. The following code facilitates calling standard river routes.

```
DEFINE WIRE=FORBIDDEN_ZONE:
    //(SP:SP  R:REAL  C:COLOR)[PATH:SP]\\
ENDDEFN

DEFINE IDENTITY=FORBIDDEN_ZONE:
    //(SP:SP  R:REAL  C:COLOR)[WIDTH:R  PATH:SP]\\
ENDDEFN

DEFINE RIVER_ROUTE(LIST:RIVER_NODES  MIN,TOP,BOT:REAL)=RIVER_RETURN:
    RIVER_ROUTE(LIST,MIN,TOP,BOT,IDENTITY,WIRE)
ENDDEFN
```

This new RIVER_ROUTE routine does not require the two FORBIDDEN_ZONEs, but uses the two default routines.

The first FORBIDDEN_ZONE is used to jog the wires. Due to global concerns, there may be obstacles to the river route. The FORBIDDEN_ZONEs allow the user to specify a routine which will modify the path of a wire in the river router. When the river router wants to route a wire through one of these obstacles, the user's routine may deflect the path of the wire. In figure A3-3a we see a wire which runs through an obstacle. The wire's path may be deflected to lie outside the obstacle (fig. A3-3b), and the river router will route all future wires to the new path (fig. A3-3c).

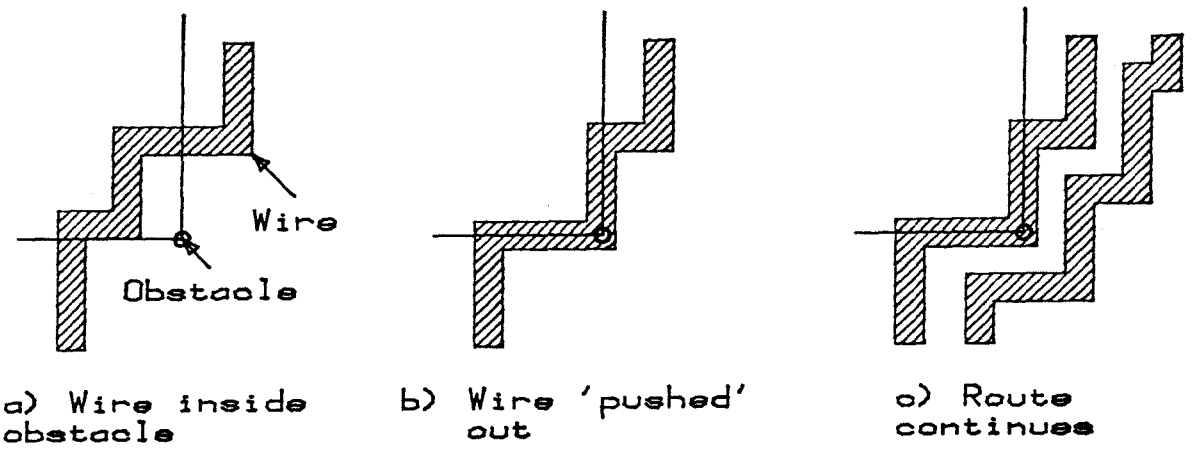


Fig A3-3: Jogging the Path of a Wire

We will define obstacles to be a collection of colored points. For an Upper Left obstacle we state that if a wire path begins to the right of the point, the path may not contain any points above and to the left of the obstacle. Figure A3-3 illustrated an Upper Left obstacle. Similarly, we may have Upper Right obstacles. These two sets of obstacles can be used to describe features of the upper cell which must be

avoided in the river route. We would also like to avoid features of the lower cell. We can not, however, just 'push' the wires outside of the obstacle points, as we did for the upper obstacles. If we did push the wires, they would run into neighboring wires. Instead, we push the lower cell down so that the wire path lies outside the obstacle, as shown in figure A3-4.

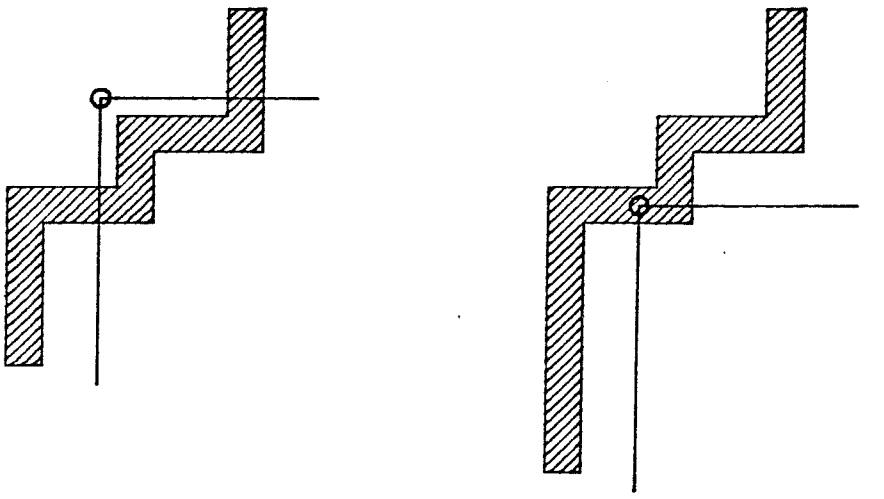


Fig. A3-4: Moving Lower Cell

The following COLOR_LIMIT datatypes are used to describe the obstacles, and the LIMIT function will move a path (SP) to remain outside the obstacles.

```
TYPE    COLOR_LIMIT= [COLOR:COLOR  LIMITS:SP];

        COLOR_LIMITS= { COLOR_LIMIT };

DEFINE LIMIT(SP:SP  LOW:REAL  COLOR:COLOR  UL,UR,LL,LR:COLOR_LIMITS)=WIRE:
    BEGIN   VAR CL=COLOR_LIMIT;P,Q=POINT;X1,X2=REAL;W=WIRE;
    DO  X1:=SP[1].X;
        X2:=REVERSE(SP)[1].X;
        IF X1<X2 THEN
            IF THERE_IS CL.COLOR=COLOR FOR CL $E UR;
            THEN SP:=RCLIP(SP,CL.LIMITS,X1,X2); FI
            IF THERE_IS CL.COLOR=COLOR FOR CL $E LL;
            THEN LOW::= MIN LMOVE(SP,CL.LIMITS,X1,X2); FI
        ELSE    IF THERE_IS CL.COLOR=COLOR FOR CL $E UL;
                THEN SP:=LCLIP(SP,CL.LIMITS,X2,X1); FI
                IF THERE_IS CL.COLOR=COLOR FOR CL $E LR;
                THEN LOW::= MIN RMOVE(SP,CL.LIMITS,X2,X1); FI FI
    GIVE [WIDTH:LOW  PATH:SP]
```

```
    END
ENDDEFN
```

The LIMIT function takes the current path (SP) and computes a new path (result.PATH). Since this routine may need to push the lower cell down, we must also return the new separation of the cells. The LOW input parameter is the previous spacing. We return the new spacing in the WIDTH component of the result. The LIMIT function also requires the wire's color, and the list of obstacles. The routine determines whether the line slopes to the left or right, and calls the appropriate CLIP and MOVE routines. The CLIP routines are used for the upper limits to jog the wires, while the MOVE routines are used for the lower limits to move the lower cell. The MOVE and CLIP routines are listed here.

```
DEFINE LMOVE(PATH,CORNERS:SP  LX,HX:REAL)=REAL:
    BEGIN   VAR MIN=REAL;P,Q=POINT;
    DO  MIN:=999999;
        FOR P $E CORNERS;WITH (P.X>LX)&(P.X=<HX); DO
            IF THERE_IS Q.X>=P.X FOR Q $E PATH; THEN MIN::= MIN Q.Y-P.Y; FI
        END
    GIVE MIN
    END
ENDDEFN

DEFINE RMOVE(PATH,CORNERS:SP  LX,HX:REAL)=REAL:
    BEGIN   VAR MIN=REAL;P,Q=POINT;
    DO  MIN:=999999;
        FOR P $E CORNERS;WITH (P.X>=LX)&(P.X<HX); DO
            IF THERE_IS Q.X=<P.X FOR Q $E PATH; THEN MIN::= MIN Q.Y-P.Y; FI
        END
    GIVE MIN
    END
ENDDEFN

DEFINE LCLIP(PATH,CORNERS:SP  LX,HX:REAL)=SP:
    BEGIN   VAR Y=REAL;P,Q=POINT;NEW=SP;FLAG=BOOL;
    DO  Y:=PATH[1].Y;
        FOR P $E CORNERS;WITH (P.X>LX)&(P.X=<HX)&(P.Y<Y); DO
            FOR Q $E PATH;
            FIRST_DO    NEW:={Q};
                        FLAG:=Q.X=<P.X;;
            OTHER_DO    IF Q.X>P.X THEN NEW::= Q<$;
                        EF Q.Y<P.Y THEN
                            IF FLAG THEN NEW::= Q.X#P.Y<$; FI
                            FLAG:=FALSE;
                            NEW::= Q <$;
                        EF -FLAG THEN NEW::= {P;P.X#Q.Y}$$;FLAG:=TRUE; FI;
            FINALLY_DO    IF FLAG THEN NEW::= LX#P.Y<$; FI;
            DO NOTHING; END
            PATH:=REVERSE(NEW);
        END
    GIVE PATH
    END
ENDDEFN
```

```
DEFINE RCLIP(PATH,CORNERS:SP  LX,HX:REAL)=SP:
    BEGIN   VAR Y=REAL;P,Q=POINT;NEW=SP;FLAG=BOOL;
    DO  Y:=PATH[1].Y;
        FOR P $E CORNERS;WITH (P.X>=LX)&(P.X<HX)&(P.Y<Y); DO
            FOR Q $E PATH;
            FIRST_DO    NEW:=(Q);
                        FLAG:=Q.X>=P.X;;
            OTHER_DO    IF Q.X<P.X THEN NEW::= Q<$;
                        EF Q.Y<P.Y THEN
                            IF FLAG THEN NEW::= Q.X#P.Y<$; FI
                            FLAG:=FALSE;
                            NEW::= Q <$;
                        EF -FLAG THEN NEW::= (P;P.X#Q.Y)$$;FLAG:=TRUE; FI;
            FINALLY_DO    IF FLAG THEN NEW::= HX#P.Y<$; FI;
            DO NOTHING; END
            PATH:=REVERSE(NEW);
        END
    GIVE PATH
    END
ENDDEFN
```

The MOVE routines look through the list of obstacles (CORNERS) for points which lie within the limits of the wire (PATH). For each obstacle point within the wire's limits, the routine computes the offset required to move the lower cell. The largest offset is returned by the routine. The CLIP routines take each obstacle which lies within the span of the wire, and moves all wire points which lie inside the obstacle.

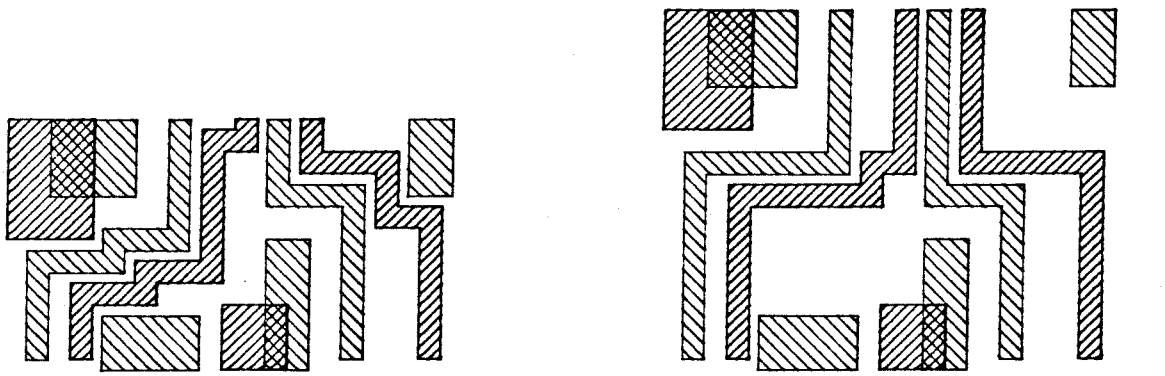


Fig. A3-5: River Route Comparison

To use this LIMIT routine in the river router, we need only compute the obstacles and pass this routine as the first FORBIDDEN_ZONE. In figure A3-5, we show a river route that uses the LIMIT routine and one that does not. The routine that uses LIMIT can route some of the wires inside the cell's boundry, while the route that does not use limit must remain outside of the cell's boundry. In many cases, the

program can compute these obstacles, so that more efficient routes can be used.

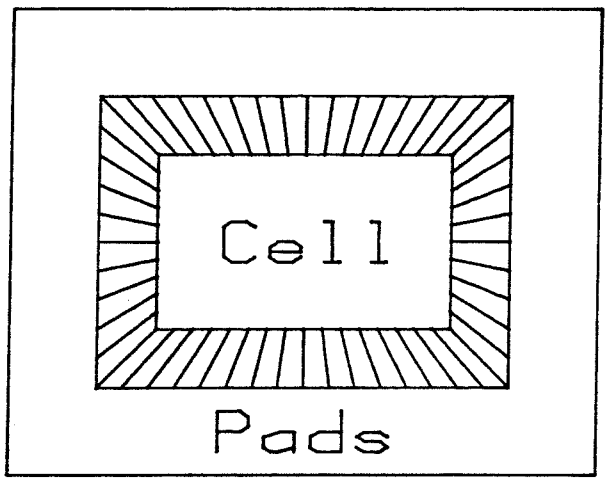


Fig. A3-6: River Routing to Pads

Another interesting use of the river router is to route wires to pads. In figure A3-6, we show a cell surrounded by pads. Between the cell and the pads, we need to route wires. A river route could be used, except for one thing: a river route is a single channel, whereas the pad route routes around a box.

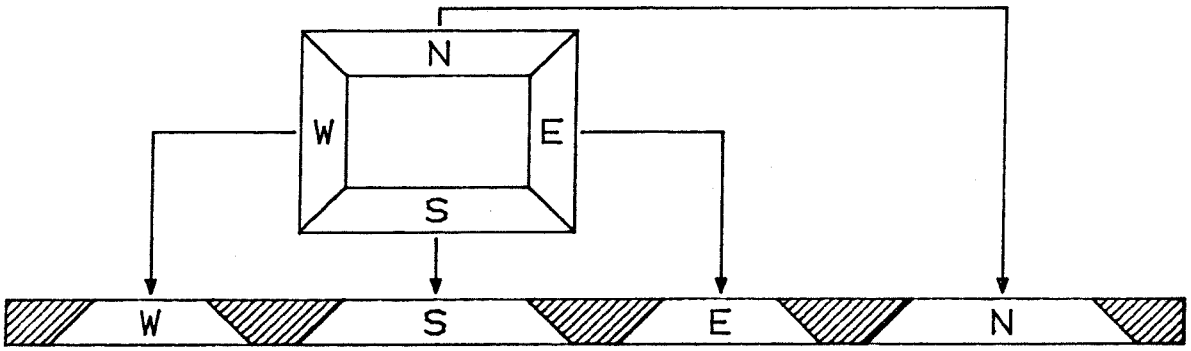


Fig. A3-7: Unfolding the Box

We can still use the river router, if we can convert the box route into a channel route, perform the river route, the convert the result back into a box route. In figure A3-7, we show the mapping from a box route to the linear route and back. We cut the box into four trapezoids and unfold the box into a single strip. The shaded portions of the strip are cut out of the river route when the trapezoids are folded back into a box. Because the shaded portions are removed, we can not have

any wires jogging inside the shaded regions. For this reason, the suspendable functions are called FORBIDDEN ZONEs: it is forbidden for the wires to jog inside the shaded regions. We will write a procedure, TRAPEZOID, which will constrain wires to jog outside of these forbidden zones. Figure A3-8 shows two cases of wires which jogged inside these shaded areas and were pushed outside of the region. In the following code, we describe TRAPEZOIDs as a left point and a right point, along with a left slope (SLEFT) and a right slope (SRIGHT). The TRAPEZOID function takes a series of these trapezoids and assures that each wire lies outside of the trapezoid. Notice that here we have reversed the polarity of the trapezoids. These trapezoids are the shaded regions, no corners may exist within the trapezoid.

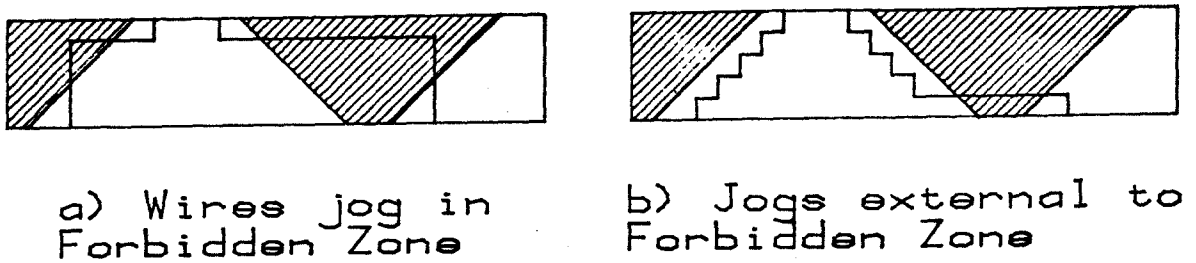


Fig. A3-8: Constraining Jogs

```
TYPE    TRAPEZOID= [LEFT,RIGHT,SLEFT,SRIGHT:POINT  CENTER:REAL];

        TRAPEZOIDS= { TRAPEZOID };

        JOG_SIZE= //REAL(COLOR)\\;

VAR TRAPEZOID_JOG=JOG_SIZE;TRAPEZOID_EDGE=REAL;

TRAPEZOID_JOG:=//(C:COLOR) CENTER_TO_CENTER(C,C)\\;

DEFINE TRAPEZOID(SP:SP  LOW:REAL  COLOR:COLOR  TS:TRAPEZOIDS)=WIRE:
    BEGIN   VAR T=TRAPEZOID;P,Q=POINT;X1,X2,R=REAL;NEW=SP;FLAG=BOOL;
    DO  X1:=SP[1].X;
        X2:=REVERSE(SP)[1].X;
        X1#X2:= (X1 MIN X2) # (X1 MAX X2);
        FOR T $E TS;WITH (T.CENTER>X1)&(T.CENTER<X2); DO
            NEW:=(SP[1]);
            FLAG:=FALSE;
            FOR (P;Q) $C SP; DO
                IF Q\INSIDE T THEN
                    IF -FLAG THEN
                        FLAG:=TRUE;
                        WHILE Q\INSIDE T; DO
                            R:= IF Q.X<P.X THEN P\RIGHT_EDGE T
                                ELSE P\LEFT_EDGE T FI;
                            NEW::= R#Q.Y<$;
                            P:=R#P.Y-<*TRAPEZOID_JOG*>(COLOR);
                            NEW::= P <$;
```

```
                        Q.Y:=P.Y;
                        LOW::= MIN Q.Y;
                    END FI
                ELSE    FLAG:=FALSE; FI
                NEW::= Q<$;
            END
            SP:=REVERSE(NEW);
        END
    GIVE [WIDTH:LOW  PATH:SP]
    END
ENDDEFN

DEFINE INSIDE(P:POINT  T:TRAPEZOID)=BOOL:
    IF P.X=< P\LEFT_EDGE T THEN FALSE ELSE P.X< P\RIGHT_EDGE T FI
ENDDEFN

DEFINE RIGHT_EDGE(P:POINT  T:TRAPEZOID)=REAL:
    T.RIGHT.X-T.SRIGHT.X*(T.RIGHT.Y-P.Y)/T.SRIGHT.Y+TRAPEZOID_EDGE
ENDDEFN

DEFINE LEFT_EDGE(P:POINT  T:TRAPEZOID)=REAL:
    T.LEFT.X-T.SLEFT.X*(T.LEFT.Y-P.Y)/T.SLEFT.Y-TRAPEZOID_EDGE
ENDDEFN

DEFINE TRAPEZOID(TS:TRAPEZOIDS)=FORBIDDEN_ZONE:
    //:TRAPEZOID(SP,REAL,COLOR) [TS]\\
ENDDEFN
```

We use the second FORBIDDEN_ZONE in the river router to map the wire from the river route coordinate system to the chip coordinate system. We can use this function to map from the linear strip into the box. In the following section of code, we have a datatype REGION which describes one of the four regions of the route. For each region, we have the trapezoid in the linear space which corresponds to one section of the box. Additionally, we have transformations from the chip coordinates to the linear coordinates and back. If we transform the connectors' locations by the MAP_TO matrix, the new locations correspond to locations along the strip. When we transform each point in the wire paths by the MAP_FROM matrix, the resulting path has the correct coordinates for the chip coordinate system. We may need to add points to the wires when mapping them back to the chip coordinate system. If figure A3-9a, we show a route in the river route coordinate system. The wire travels from one trapezoid to another, which is valid since the wire does not jog withing the shaded area. If we just transformed the four points in the wire, we would get the layout shown in figure A3-9b, which has one wire cutting across our cell. We need to add a point on the edge between the two trapezoids when we do the mapping, resulting in the layout shown in figure A3-9c.

The REGION function takes two corner points and two slopes and computes the corresponding region. The REGIONS function takes the wire in the river route's

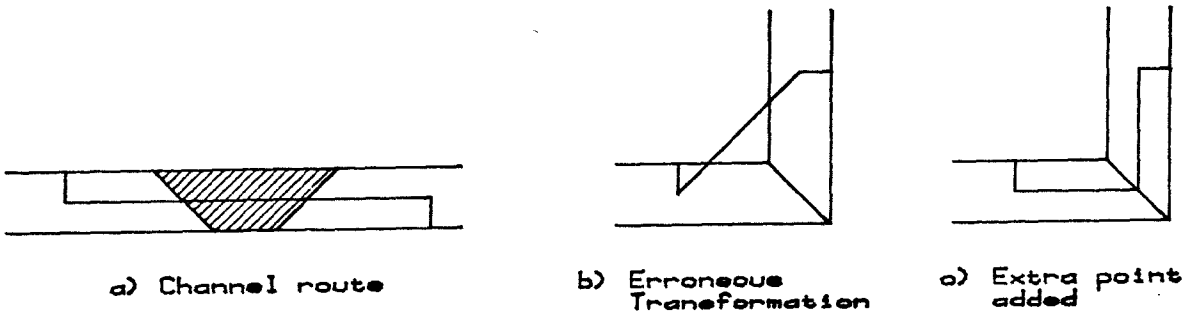


Fig. A3-9: Mapping Wires into Box

coordinates and computes the path in the chip's coordinates, adding the points where needed.

```
TYPE    REGION= [INSIDE:TRAPEZOID  MAP_TO,MAP_FROM:MATRIX
                 CORNER,SLOPE:POINT  MINX,MAXX:REAL];

        REGIONS= { REGION };

DEFINE REGION(UL,UR,LL,LR:POINT  GROUP:INT)=REGION:
    BEGIN   VAR A=REAL;
    DO  A:=(UR-UL)\ANGLE;
        GROUP::=*100000;
    GIVE [INSIDE:[LEFT:GROUP#0  RIGHT:GROUP#0+((UR-UL)\ROTATED_BY -A)
                  SLEFT:LL\ROTATED_BY -A  SRIGHT:LR\ROTATED_BY -A]
          MAP_TO:DISPLACEMENT(GROUP#0)\ROTATED_BY -A\AT -UL
          MAP_FROM:DISPLACEMENT(UL)\ROTATED_BY A\AT -(GROUP#0)
          CORNER:UR  SLOPE:LR  MINX:GROUP-50000  MAXX:GROUP+50000]
    END
ENDDEFN

DEFINE REGIONS(RS:REGIONS)=FORBIDDEN_ZONE:
    //:REGIONS(SP,REAL,COLOR)[RS]\\
ENDDEFN

DEFINE REGIONS(SP:SP  BOT:REAL  COLOR:COLOR  RGS:REGIONS)=WIRE:
    BEGIN   VAR NEW=SP;P=POINT;I,J,K=INT;
    DO  I:=FIXR(SP[1].X/100000);
        NEW:={SP[1]\AT RGS[I].MAP_FROM};
        FOR P $E SP[2-]; DO
            J:=FIXR(P.X/100000);
            IF I<J THEN
                DO NEW::= RGS[K].CORNER-RGS[K].SLOPE*P.Y <$;
                FOR K FROM I TO J-1;
            EF J<I THEN
                DO NEW::= RGS[K].CORNER-RGS[K].SLOPE*P.Y <$;
                FOR K FROM I-1 TO J;  FI
            NEW::= P\AT RGS[J].MAP_FROM <$;
            I:=J;
        END
    GIVE [PATH:NEW]
    END
ENDDEFN
```

There are a few other concerns before we have completed the box router. First, consider figure A3-10. We have a wire that starts on the NORTH and ends on the WEST. In the river route space, this wire extends from the far right to the far left, shorting out every other wire in the route. To solve this, we may move the WEST trapezoid to be to the right of the NORTH trapezoid, but then we would have the same problem with WEST/SOUTH wires. Instead, we may have a second WEST region, W', which is to the right of the NORTH region. We have two WEST regions now. NORTH/WEST wires use W', while WEST/SOUTH wires use the original WEST region. WEST/WEST wires can use either region.

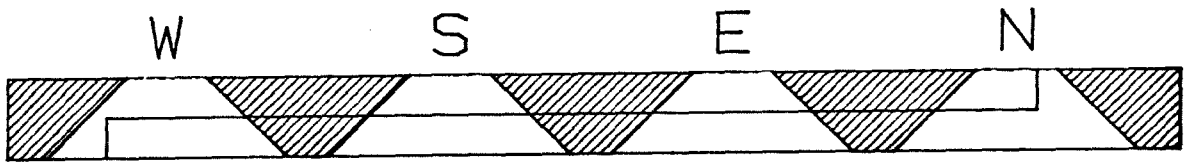


Fig. A3-10: Erroneous Wire Wrap-Around

Unfortunately, this causes another problem. We now have two independent WEST regions in the river route space, but there is only one WEST region in the chip space. In figure A3-11, we show two wires, one a SOUTH/WEST wire, the other a WEST/NORTH wire. Since these are in the independent regions of the river route, they independently route, which causes trouble in the chip space. What we need to do is to make the two WEST regions independent. We have noticed above that wires can be routed independently if they run in opposite directions. The two wires in figure A3-11 run in the same direction, so they are not independent. We will make a new SOUTH region, S', to the right of W', and move the wire AB into the W'/S' regions. We continue this process until the left-most wire in the river route runs in the opposite direction of the right-most wire. (We can also stop the circulation of wires when the wire spans do not overlap.) We must check for the condition that all wires run in the same direction, and signal an error if this occurs.

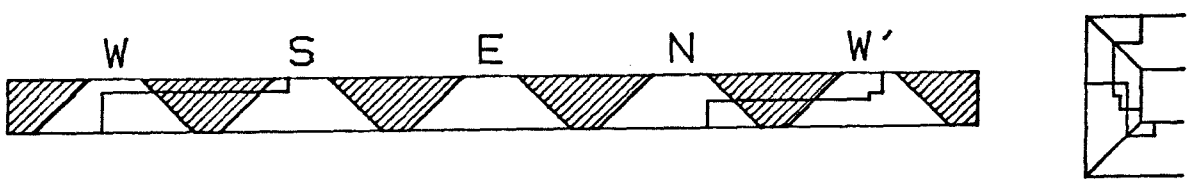


Fig. A3-11: Non-Independent Wires

Another potential problem occurs near the edges of the trapezoids. Given two neighboring trapezoids, the adjacent edges in the river route coordinates represent the same line in the chip coordinates. Wires jogging close to these lines may short together in the chip space while quite far apart in the river space, as shown in figure A3-12. To combat this problem, we just bloat the trapezoids by half the maximum design rule spacing. This assures that wires remain far enough apart.

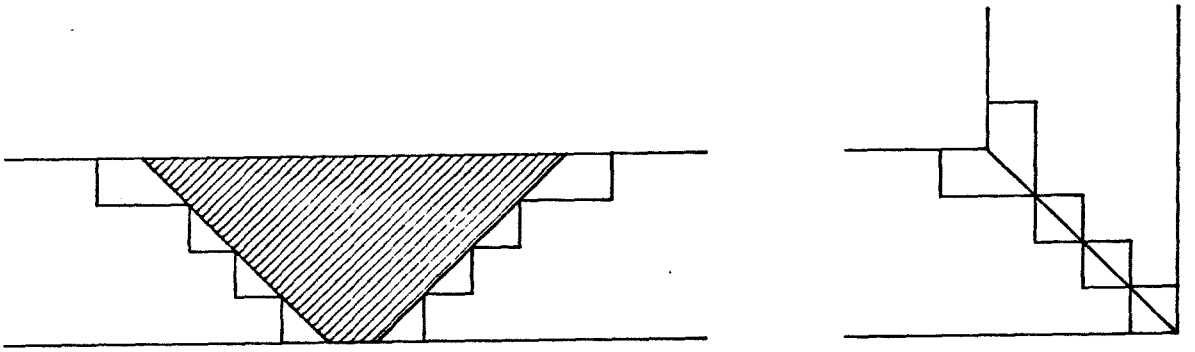


Fig. A3-12: Boundary Interference

The remaining code describes the connection points for box routes, which need to know the side on which the connector resides. Also, the routines for implementing the route are listed. The NORMALS routine is used for generating the trapezoids given the outline of the cell. The OUTSIDE routine is used to invert the polarity of the trapezoids. The first ROTO_ROUTE function is used to reorder the pads to shorten the wire lengths. The final ROTO_ROUTE routine is the river router which routes around the outside of the cell. Figure A3-13 shows a river route around a rectangular cell, while figure A3-14 shows a river route around a hexagonal cell.

```
TYPE     CONNECT2= [FROM,TO:POINT  COLOR:COLOR  FEDGE,TEDGE:INT];

         CONNECT2S= { CONNECT2 };

DEFINE NORMALIZED(A,B:POINT)=POINT:
    A*(DIST(B,0#0)/DOT(A,B))
ENDDEFN

DEFINE NORMALS(SP:SP)=SP:
    BEGIN   VAR NORMALS=SP;P,Q=POINT;
    DO  NORMALS:={COLLECT (Q-P)\NORMAL FOR {P;*Q} $C SP;};
        NORMALS:=REVERSE(NORMALS);
        NORMALS[2-]:=REVERSE(NORMALS[2-]);
    GIVE {COLLECT NORMALIZED(P+Q,P) FOR {P;*Q} $C NORMALS;}
    END
ENDDEFN
```

```
DEFINE OUTSIDE(RGS:REGIONS)=TRAPEZOIDS:
    BEGIN   VAR P,Q=REGION;
    {COLLECT
        [LEFT:P.INSIDE.RIGHT  RIGHT:Q.INSIDE.LEFT
         SLEFT:P.INSIDE.SRIGHT  SRIGHT:Q.INSIDE.SLEFT
         CENTER: (P.INSIDE.LEFT.X+Q.INSIDE.LEFT.X)/2.]
    FOR {P;Q} $C RGS;}
    END
ENDDEFN

DEFINE ROTO_ROUTE(RNS:RIVER_NODES  J:INT)=RIVER_NODES:
    BEGIN   VAR RN=RIVER_NODE;TO=SP;CGF,CGT=REAL;P=POINT;
    DO  TO:={COLLECT RN.TO FOR RN $E RNS;};
        CGF:=+ RN.FROM.X FOR RN $E RNS;;
        CGT:=+ RN.TO.X FOR RN $E RNS;;
        WHILE ABS(CGF-CGT)>.55*J; DO
            IF CGT>CGF THEN
                RN:=RNS[1];
                RN.FROM.X::=+J;
                RNS:=RNS[2-]$>RN;
                CGF::=+J;
            ELSE    TO:=TO[2-]$> TO[1].X+J#TO[1].Y;
                    CGT::=+J; FI
        END
    GIVE {COLLECT    DO  RN.TO:=P;
                     GIVE RN
          FOR RN $E RNS;&& FOR P $E TO;}
    END
ENDDEFN

DEFINE ROTO_ROUTE(CS:CONNECT2S  MIN,TOP,BOT:REAL  OUTLINE:SP  ROTO:BOOL)=
                                                        RIVER_RETURN:
    BEGIN   VAR NORMALS=SP;P,Q,R,S=POINT;RGS=REGIONS;RG=REGION;C=CONNECT2;
                RNS=RIVER_NODES;RN=RIVER_NODE;I,J=INT;
    DO  NORMALS:=OUTLINE\NORMALS;
        TRAPEZOID_EDGE:=MAX_SPACING/2.;
        RGS:={COLLECT REGION(P,Q,R,S,I)
              FOR {P;*Q} $C OUTLINE$$OUTLINE;&&
              FOR {R;*S} $C NORMALS$$NORMALS;&&
              FOR I FROM 1 BY 1;};
        J:=+1 FOR P $E OUTLINE;;
        RNS:={COLLECT
              [FROM:C.FROM\AT
                    RGS[C.FEDGE+ IF C.TEDGE<1 THEN J ELSE 0 FI].MAP_TO
               TO:C.TO\AT RGS[C.TEDGE+ IF C.TEDGE<1 THEN J ELSE 0 FI].MAP_TO
               COLOR:C.COLOR]
              FOR C $E CS;};
        WHILE RNS[1].FROM.X>RNS[2].FROM.X; DO RNS:=RNS[2-]$>RNS[1]; END
        J::=*100000;
        IF ROTO THEN RNS::=\ROTO_ROUTE J; FI
        RN:=REVERSE(RNS)[1];
        WHILE (RN.FROM.X<RN.TO.X)=(RNS[1].FROM.X<RNS[1].TO.X);&&
        FOR I FROM 1 TO 1000; DO
            RN:=RNS[1];
            RN.FROM.X::=+J;
            RN.TO.X::=+J;
            RNS:=RNS[2-]$>RN;
        END
        IF I>=1000 THEN WRITE('ROTO_ROUTE: CIRCULAR');CRLF;HELP; FI
    GIVE RIVER_ROUTE(RNS,MIN,TOP,BOT,
                     RGS\OUTSIDE\TRAPEZOID,RGS\REGIONS)
```

```
    END
ENDDEFN
```

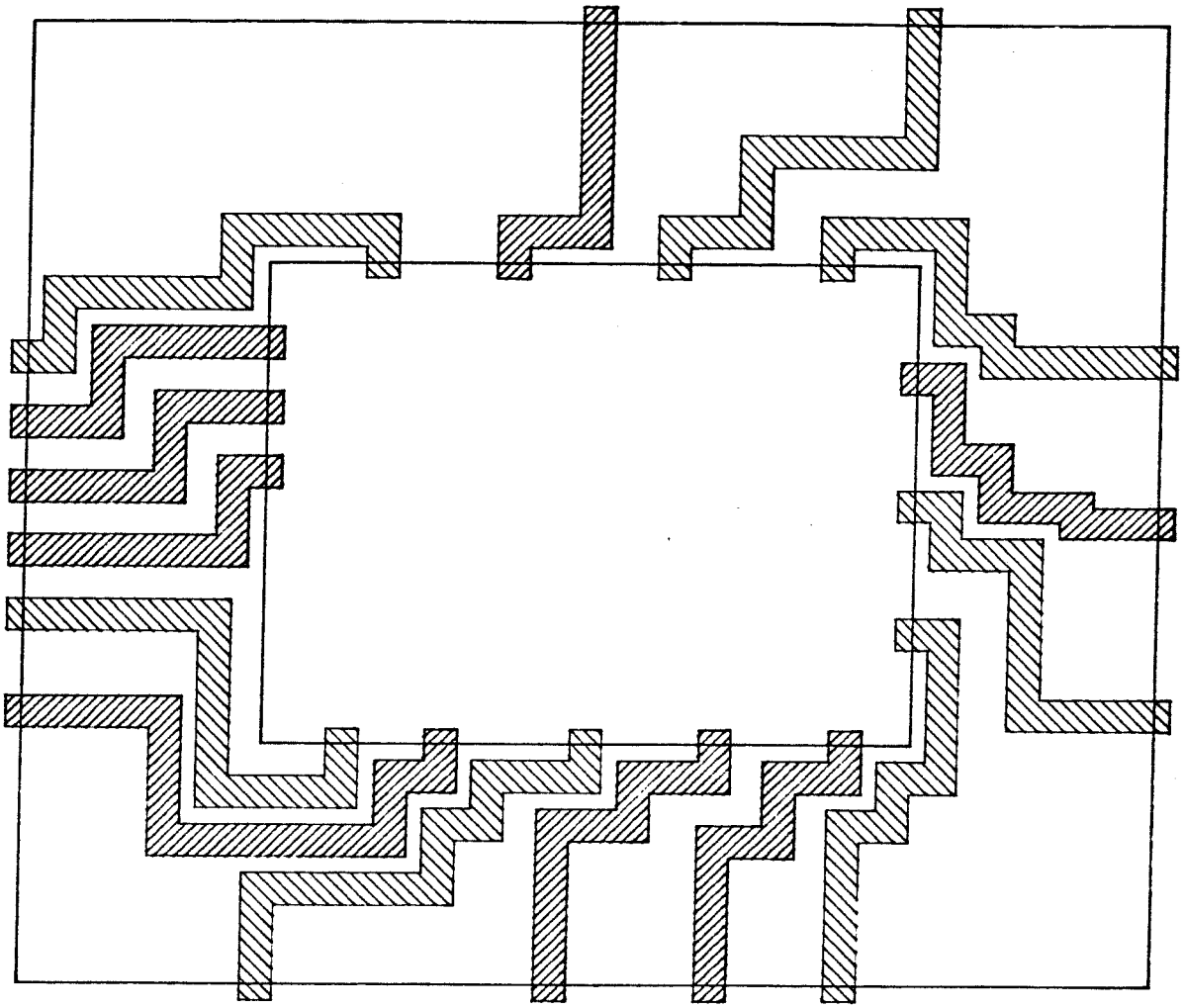


Fig. A3-13: Box Route

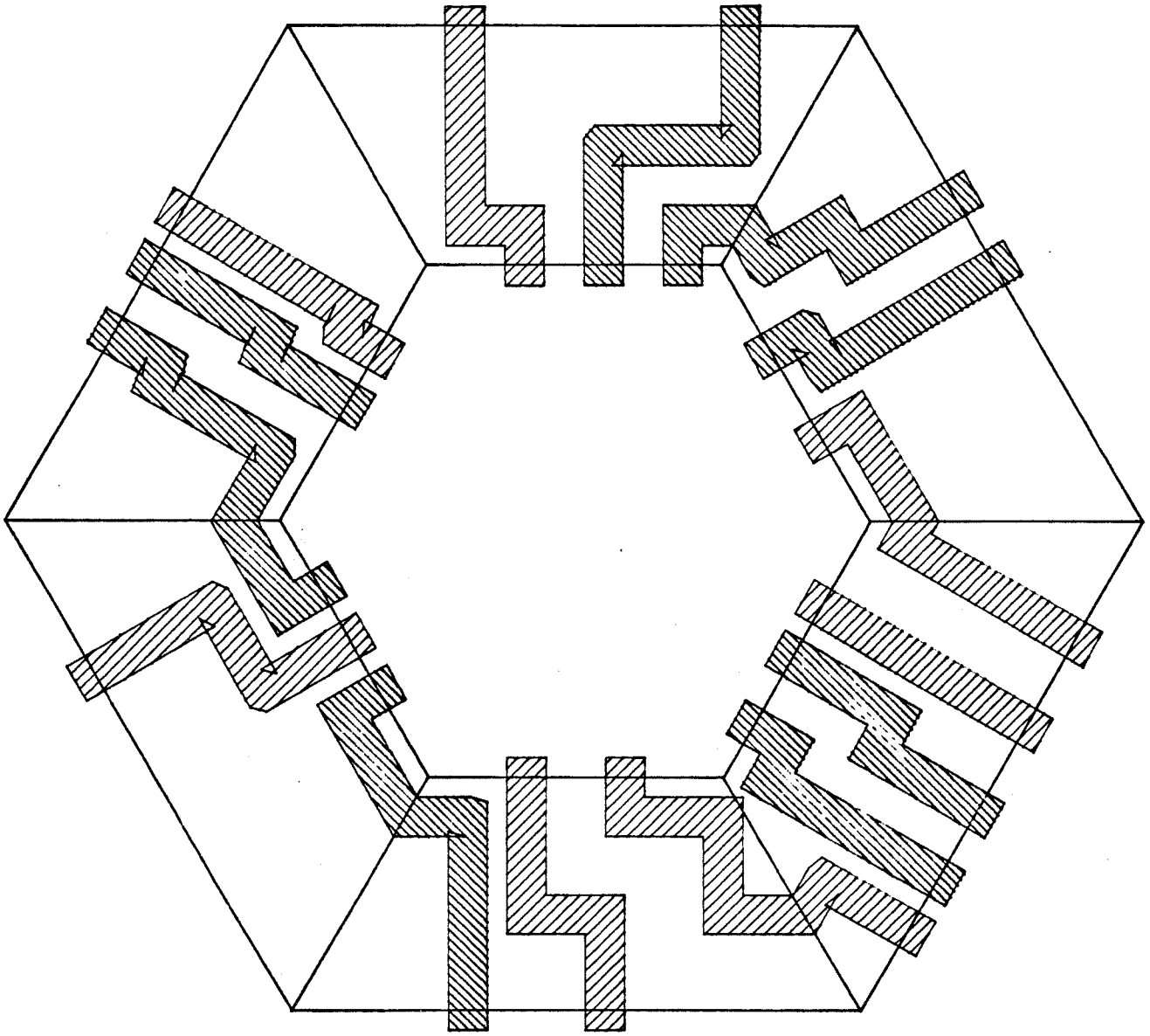


Fig. A3-14: Hexagon Route

Appendix 4: The RLC Compiler

The appendix contains the complete code listings for the Random Logic Compiler described in Chapter 5. In some cases, Chapter 5 used approximations for the data structures and routines, so there may be a few differences between the code implied by Chapter 5 and the code listed here.

The PHYSICAL_WIRE datatypes are defined as shown in Chapter 5. In addition, we declare a type GATE_PRODUCER which is a suspendable function. The input and output parameters for this function match the requirements of the NAND, NOR, and INVERT functions. We will use instances of this datatype to refer to virtual routines for generating the gate layouts. The user at any time may reassign new routines to these variables, which will modify the layout produced.

```
TYPE    PHYSICAL_WIRE= [HEIGHT,LEFT,RIGHT:REAL  NAME:QS];
        PHYSICAL_WIRES= { PHYSICAL_WIRE };
        GATE_PRODUCER= //MRG(PHYSICAL_WIRES,PHYSICAL_WIRE,REAL)\\;
```

We can now define routines for generating gates in a number of technologies. We will have global variables set to one group of these functions, which indicate the current technology. Currently, we support NMOS, 2-layer metal NMOS, CMOS, and 2-layer metal CMOS. In addition to these actual technologies, we have a few pseudo-technologies: NMOS sticks, 2-layer metal NMOS sticks, Logic diagrams, and NMOS transistor diagrams. The gate producing functions for these technologies are listed here.

```
DEFINE NMOS_PULLUP(OUTPUT:PHYSICAL_WIRE  X:REAL)=MRG:
    DO  CONNECT(OUTPUT,X-2);
        POWER::=+.25;
    GIVE    {BOX(RED,X-16#0\TO X-5#6);
             BOX(YELLOW,X-16#-2.\TO X-5#9);
             WIRE(GREEN,2,(X-13#YVDD;.#3;X-8#.;.#.-5;.+5#.;.#OUTPUT.HEIGHT));
             GCB\AT (X-12#YVDD;X-2#OUTPUT.HEIGHT);
             GRCBU\AT X-7#-1.}
ENDDEFN

DEFINE NMOS_NAND(INPUTS:PHYSICAL_WIRES
                 OUTPUT:PHYSICAL_WIRE  X:REAL)=MRG:
    BEGIN   VAR IN=PHYSICAL_WIRE;NUMBER=INT;X2=REAL;
    DO  NUMBER:= +1 FOR IN $E INPUTS;;
        X2:=X-10-2*NUMBER;
        DO CONNECT(IN,X2); FOR IN $E INPUTS;
        CWIDTH:=X2-5;
```

```
    GIVE {GCB\AT X-8#YGND;
          BOX(GREEN,X2+3#YGND-2\TO X-7#-1.);
          COLLECT {RCB\AT X2#IN.HEIGHT;
                   WIRE(RED,2,{X2#IN.HEIGHT;X-6#.})} FOR IN $E INPUTS;;
          NMOS_PULLUP(OUTPUT,X)}
    END
ENDDEFN

DEFINE NMOS_NOR(INPUTS:PHYSICAL_WIRES
                OUTPUT:PHYSICAL_WIRE  X:REAL)=MRG:
    BEGIN   VAR IN=PHYSICAL_WIRE;
    DO  DO CONNECT(IN,X-16); FOR IN $E INPUTS;
        CWIDTH:=X-24;
    GIVE {GCB\AT X-19#YGND;
          WIRE(GREEN,2,{X-20#YGND;.# MAX IN.HEIGHT FOR IN $E INPUTS;+4});
          WIRE(GREEN,2,{X-8# MIN IN.HEIGHT FOR IN $E INPUTS;+4;.#-2.});
          COLLECT {RCB\AT X-16#IN.HEIGHT;
                   WIRE(RED,2,{X-15#IN.HEIGHT+1;X-11#.;.#.+5});
                   WIRE(GREEN,2,{X-20#IN.HEIGHT+4;X-8#.})}
          FOR IN $E INPUTS;;
          NMOS_PULLUP(OUTPUT,X)}
    END
ENDDEFN

DEFINE NMOS_INVERT(INPUTS:PHYSICAL_WIRES
                   OUTPUT:PHYSICAL_WIRE  X:REAL)=MRG:
    BEGIN   VAR IN=PHYSICAL_WIRE;
    DO  IN:=INPUTS[1];
        CONNECT(IN,X-12);
        CWIDTH:=X-17;
    GIVE {GCB\AT X-8#YGND;
          BOX(GREEN,X-9#YGND-2\TO X-7#-1.);
          RCB\AT X-12#IN.HEIGHT;
          WIRE(RED,2,{X-12#IN.HEIGHT;X-6#.});
          NMOS_PULLUP(OUTPUT,X)}
    END
ENDDEFN


DEFINE METAL2_NAND(INPUTS:PHYSICAL_WIRES
                   OUTPUT:PHYSICAL_WIRE  X:REAL)=MRG:
    BEGIN   VAR IN=PHYSICAL_WIRE;NUMBER=INT;X2=REAL;
    DO  NUMBER:=+1 FOR IN $E INPUTS;;
        X2:= -14. MIN -9.-2*NUMBER;
        DO CONNECT(IN,X+X2+2); FOR IN $E INPUTS;
        CONNECT(OUTPUT,X+X2+9);
        CWIDTH:=X+X2;
        POWER::=+.25;
    GIVE {GCB\AT {7#YGND;8#YVDD};
          GRCBU\AT 2#-1.;
          BCB\AT {2#-1.;9#OUTPUT.HEIGHT};
          WIRE(VIOLET,3,{2#-1.;9#.;.#OUTPUT.HEIGHT});
          BOX(RED,0#0\TO 11#6);
          BOX(YELLOW,0#-2.\TO 11#8);
          WIRE(GREEN,2,{8#YVDD;.#3;3#.;.#-2.;6#.;.#YGND});
          BOX(GREEN,5#YGND-2\TO 5+2*NUMBER#-4.);
          COLLECT {RCB\AT 2#IN.HEIGHT;
                   WIRE(RED,2,{2#IN.HEIGHT;6+2*NUMBER#.})}
          FOR IN $E INPUTS;}\AT X+X2#0
    END
ENDDEFN
```

```
DEFINE METAL2_NOR(INPUTS:PHYSICAL_WIRES
                  OUTPUT:PHYSICAL_WIRE  X:REAL)=MRG:
    BEGIN   VAR IN=PHYSICAL_WIRE;
    DO  DO CONNECT(IN,X-12); FOR IN $E INPUTS;
        CONNECT(OUTPUT,X-5);
        CWIDTH:=X-17;
        POWER::=+.25;
    GIVE {GCB\AT {2#YGND;6#YVDD};
          GRCBU\AT 12#-1.;
          BCB\AT {12#-1.;.#OUTPUT.HEIGHT};
          WIRE(VIOLET,3,{12#-1.;.#OUTPUT.HEIGHT});
          BOX(RED,3#0\TO 14#6);
          BOX(YELLOW,3#-2.\TO 14#8);
          WIRE(GREEN,2,{1#YGND;.# MAX IN.HEIGHT FOR IN $E INPUTS;-4});
          WIRE(GREEN,2,{13# MIN IN.HEIGHT FOR IN $E INPUTS;-4;.#0});
          WIRE(GREEN,2,{6#YVDD;.#3;11#.;.#0});
          COLLECT {RCB\AT 5#IN.HEIGHT;
                   WIRE(RED,2,{6#IN.HEIGHT-1;10#.;.#.-5});
                   WIRE(GREEN,2,{1#IN.HEIGHT-4;13#.})}
          FOR IN $E INPUTS;}\AT X-17#0
    END
ENDDEFN

DEFINE METAL2_INVERT(INPUTS:PHYSICAL_WIRES
                     OUTPUT:PHYSICAL_WIRE  X:REAL)=MRG:
    BEGIN   VAR IN=PHYSICAL_WIRE;
    DO  IN:=INPUTS[1];
        CONNECT(IN,X-12);
        CONNECT(OUTPUT,X-5);
        CWIDTH:=X-14;
        POWER::=+.25;
    GIVE {GCB\AT {7#YGND;8#YVDD};
          GRCBU\AT 2#-1.;
          BCB\AT {2#-1.;9#OUTPUT.HEIGHT};
          WIRE(VIOLET,3,{2#-1.;9#.;.#OUTPUT.HEIGHT});
          BOX(RED,0#0\TO 11#6);
          BOX(YELLOW,0#-2.\TO 11#8);
          WIRE(GREEN,2,{8#YVDD;.#3;3#.;.#-2.;6#.;.#YGND});
          BOX(GREEN,5#YGND-2\TO 7#-4.);
          RCB\AT 2#IN.HEIGHT;
          WIRE(RED,2,{2#IN.HEIGHT;8#.})}\AT X-14#0
    END
ENDDEFN


DEFINE LOGICAL_NAND(INPUTS:PHYSICAL_WIRES
                    OUTPUT:PHYSICAL_WIRE  X:REAL)=MRG:
    BEGIN   VAR IN=PHYSICAL_WIRE;NUMBER=INT;Y=REAL;
    DO  NUMBER:= +1 FOR IN $E INPUTS;;
        DO CONNECT(IN,X+Y); FOR IN $E INPUTS;&&
                            FOR Y FROM 10./NUMBER-25. BY 20./NUMBER;
        CONNECT(OUTPUT,X);
        CWIDTH:=X-30;
    GIVE {WIRE(BLUE,0,{-5.#0;-25.#.;-25.#15;-24.5#18;-23.#21;-21.#23;-18.#24.5;
                       -15.#25;-14.#25.2;-12.9#25.9;-12.2#27;-12.#28;
                       -12.2#29;-12.9#30.1;-14.#30.8;-15.#31;-16.#30.8;
                       -17.1#30.1;-17.8#29;-18.#28;-17.8#27;-17.1#25.9;
                       -16.#25.2;-15.#25;-12.#24.5;-9.#23;-7.#21;-5.5#18;
                       -5.#15;-5.#0});
          WIRE(GREEN,0,{-15.#31;.#33;0#.;.#OUTPUT.HEIGHT});
          COLLECT WIRE(GREEN,0,{Y#0;.#IN.HEIGHT})
```

```
        FOR IN $E INPUTS;&& FOR Y FROM 10./NUMBER-25. BY 20./NUMBER;;
        OUTPUT.NAME\PAINTED RED\ROT 90\AT -8.#37}\AT X#0
    END
ENDDEFN

DEFINE LOGICAL_NOR(INPUTS:PHYSICAL_WIRES
                   OUTPUT:PHYSICAL_WIRE  X:REAL)=MRG:
    BEGIN   VAR IN=PHYSICAL_WIRE;NUMBER=INT;Y=REAL;
    DO  NUMBER:= +1 FOR IN $E INPUTS;;
        DO CONNECT(IN,X+Y); FOR IN $E INPUTS;&&
                            FOR Y FROM 10./NUMBER-25. BY 20./NUMBER;
        CONNECT(OUTPUT,X);
        CWIDTH:=X-30;
    GIVE {WIRE(BLUE,0,{-5.#0;-9.#1;-13.#1.7;-17.#.;-21.#1;-25.#0;
                       .#7;-24.7#10;-23.#15;-20.5#20;-18.#22.8;-16.#24.2;
                       -15.#25;-14.#25.2;-12.9#25.9;-12.2#27;-12.#28;
                       -12.2#29;-12.9#30.1;-14.#30.8;-15.#31;-16.#30.8;
                       -17.1#30.1;-17.8#29;-18.#28;-17.8#27;-17.1#25.9;
                       -16.#25.2;-15.#25;-14.#24.2;-12.#22.8;-9.5#20;-7.#15;
                       -5.3#10;-5.#7;.#0});
          WIRE(GREEN,0,{-15.#31;.#33;0#.;.#OUTPUT.HEIGHT});
          COLLECT WIRE(GREEN,0,{Y#IF Y<-21. THEN (25+Y)/4
                                EF Y<-17. THEN 1+.7*(21+Y)/4
                                EF Y<-13. THEN 1.7
                                EF Y<-9. THEN 1-.7*(9+Y)/4
                                ELSE 1-(9+Y)/4 FI;.#IN.HEIGHT})
          FOR IN $E INPUTS;&& FOR Y FROM 10./NUMBER-25. BY 20./NUMBER;;
          OUTPUT.NAME\PAINTED RED\ROT 90\AT -8.#37}\AT X#0
    END
ENDDEFN

DEFINE LOGICAL_INVERT(INPUTS:PHYSICAL_WIRES
                      OUTPUT:PHYSICAL_WIRE  X:REAL)=MRG:
    BEGIN   VAR IN=PHYSICAL_WIRE;NUMBER=INT;Y=REAL;
    DO  IN:=INPUTS[1];
        CONNECT(IN,X-15);
        CONNECT(OUTPUT,X);
        CWIDTH:=X-30;
    GIVE {WIRE(BLUE,0,{-5.#0;-25.#.;
                       -15.#25;-14.#25.2;-12.9#25.9;-12.2#27;-12.#28;
                       -12.2#29;-12.9#30.1;-14.#30.8;-15.#31;-16.#30.8;
                       -17.1#30.1;-17.8#29;-18.#28;-17.8#27;-17.1#25.9;
                       -16.#25.2;-15.#25;-5.#0});
          WIRE(GREEN,0,{-15.#31;.#33;0#.;.#OUTPUT.HEIGHT});
          WIRE(GREEN,0,{-15.#0;.#IN.HEIGHT});
          OUTPUT.NAME\PAINTED RED\ROT 90\AT -8.#37}\AT X#0
    END
ENDDEFN


DEFINE CMOS_NAND(INPUTS:PHYSICAL_WIRES
                 OUTPUT:PHYSICAL_WIRE  X:REAL)=MRG:
    BEGIN   VAR IN=PHYSICAL_WIRE;MN=REAL;
    DO  DO CONNECT(IN,X-13);CONNECT(IN,X-21); FOR IN $E INPUTS;
        CONNECT(OUTPUT,X-2);
        CWIDTH:=X-33;
        MN:= MIN IN.HEIGHT FOR IN $E INPUTS;;
    GIVE   {COLLECT {RCB\AT {X-21#IN.HEIGHT;X-13#.};
                    WIRE(RED,2,{X-22#IN.HEIGHT-1;.-4#.;.#.-5});
                    WIRE(RED,2,{X-12#IN.HEIGHT;.+5#.});
                    WIRE(GREEN,2,{X-29#IN.HEIGHT-4;X-17#.})}
```

```
            FOR IN $E INPUTS;;
            GCB\AT {X-28#YVDD;X-16#YVDD-7;.+6#.;.+6#.;X-2#OUTPUT.HEIGHT;
                    X-10#YGND};
            WIRE(BLUE,3,{X-16#YVDD-7;.+12#.});
            WIRE(GREEN,2,{X-29#YVDD;.#MN-4});
            WIRE(GREEN,2,{X-17#MN-4;.#YVDD-7});
            WIRE(GREEN,2,{X-9#YVDD-7;.#YGND});
            WIRE(GREEN,2,{X-3#YVDD-7;.#OUTPUT.HEIGHT});
            BOX(YELLOW,X-32#YGND-3\TO X-13#YVDD-4)}
    END
ENDDEFN

DEFINE CMOS_NOR(INPUTS:PHYSICAL_WIRES
                OUTPUT:PHYSICAL_WIRE  X:REAL)=MRG:
    BEGIN   VAR IN=PHYSICAL_WIRE;MX=REAL;
    DO  DO CONNECT(IN,X-13);CONNECT(IN,X-21); FOR IN $E INPUTS;
        CONNECT(OUTPUT,X-2);
        CWIDTH:=X-33;
        MX:= MAX IN.HEIGHT FOR IN $E INPUTS;;
    GIVE    {COLLECT {RCB\AT {X-21#IN.HEIGHT;X-13#.};
                      WIRE(RED,2,{X-22#IN.HEIGHT-1;.-4#.;.#.-5});
                      WIRE(RED,2,{X-12#IN.HEIGHT;.+5#.});
                      WIRE(GREEN,2,{X-29#IN.HEIGHT-4;X-17#.})}
            FOR IN $E INPUTS;;
            GCB\AT {X-28#YGND;X-16#YGND+7;.+6#.;.+6#.;X-2#OUTPUT.HEIGHT;
                    X-10#YVDD};
            WIRE(BLUE,3,{X-16#YGND+7;.+12#.});
            WIRE(GREEN,2,{X-29#YGND;.#MX-4});
            WIRE(GREEN,2,{X-17#MX-4;.#YGND+7});
            WIRE(GREEN,2,{X-9#YGND+7;.#YVDD});
            WIRE(GREEN,2,{X-3#YGND+7;.#OUTPUT.HEIGHT});
            BOX(YELLOW,X-13#YGND+4\TO X-6#YVDD+3)}
    END
ENDDEFN

DEFINE CMOS_INVERT(INPUTS:PHYSICAL_WIRES
                   OUTPUT:PHYSICAL_WIRE  X:REAL)=MRG:
    BEGIN   VAR IN=PHYSICAL_WIRE;
    DO  IN:=INPUTS[1];
        CONNECT(IN,X-12);
        CONNECT(OUTPUT,X-2);
        CWIDTH:=X-18;
    GIVE    {RCB\AT X-12#IN.HEIGHT;
             WIRE(RED,{X-6#IN.HEIGHT;X-11#.;.#YVDD+1});
             GCB\AT {X-2#OUTPUT.HEIGHT;X-7#YGND;.#YVDD-7;.#YVDD;X-15#YVDD-7};
             WIRE(BLUE,3,{X-15#YVDD-7;X-7#.});
             WIRE(GREEN,2,{X-8#YVDD-1;X-14#.;.#YVDD-6});
             WIRE(GREEN,2,{X-8#YGND;.#YVDD-8;X-3#.;.#OUTPUT.HEIGHT+1});
             BOX(YELLOW,X-18#YVDD-10\TO X-11#YVDD+2);
             BOX(YELLOW,X-12#YVDD-3.5\TO X-4#YVDD+2)}
    END
ENDDEFN


DEFINE CMOS_2_NAND(INPUTS:PHYSICAL_WIRES
                   OUTPUT:PHYSICAL_WIRE  X:REAL)=MRG:
    BEGIN   VAR IN=PHYSICAL_WIRE;MN=REAL;
    DO  DO CONNECT(IN,X-9);CONNECT(IN,X-17); FOR IN $E INPUTS;
        CONNECT(OUTPUT,X-2);
        CWIDTH:=X-25;
        MN:= MIN IN.HEIGHT FOR IN $E INPUTS;;
```

```
    GIVE    {COLLECT {RCB\AT {X-9#IN.HEIGHT;X-17#.};
                      WIRE(RED,2,{X-8#IN.HEIGHT-1;X-4#.;.#.-5});
                      WIRE(RED,2,{X-18#IN.HEIGHT-1;X-23#.});
                      WIRE(GREEN,2,{X-1#IN.HEIGHT-4;X-13#.})}
             FOR IN $E INPUTS;;
             GCB\AT {X-2#YVDD;X-20#YVDD-7;.#YGND};
             GCBCB\AT X-14#YVDD-7;
             BCB\AT X-2#OUTPUT.HEIGHT;
             WIRE(BLUE,3,{X-14#YVDD-7;X-20#.});
             WIRE(VIOLET,3,{X-14#YVDD-7;X-2#.;.#OUTPUT.HEIGHT});
             WIRE(GREEN,2,{X-1#YVDD;.#MN-4});
             WIRE(GREEN,2,{X-13#YVDD-8;.#MN-4});
             WIRE(GREEN,2,{X-21#YVDD-8;.#YGND});
             BOX(YELLOW,X+1.5#MN-7\TO X-17#YVDD+2)}
    END
ENDDEFN

DEFINE CMOS_2_NOR(INPUTS:PHYSICAL_WIRES
                  OUTPUT:PHYSICAL_WIRE  X:REAL)=MRG:
    BEGIN   VAR IN=PHYSICAL_WIRE;MX=REAL;
    DO  DO CONNECT(IN,X-9);CONNECT(IN,X-17); FOR IN $E INPUTS;
        CONNECT(OUTPUT,X-2);
        CWIDTH:=X-25;
        MX:= MAX IN.HEIGHT FOR IN $E INPUTS;;
    GIVE    {COLLECT {RCB\AT {X-9#IN.HEIGHT;X-17#.};
                      WIRE(RED,2,{X-8#IN.HEIGHT-1;X-4#.;.#.-5});
                      WIRE(RED,2,{X-18#IN.HEIGHT-1;X-23#.});
                      WIRE(GREEN,2,{X-1#IN.HEIGHT-4;X-13#.})}
             FOR IN $E INPUTS;;
             GCB\AT {X-2#YGND;X-20#YGND+7;.#YVDD};
             GCBCB\AT X-14#YGND+7;
             BCB\AT X-2#OUTPUT.HEIGHT;
             WIRE(BLUE,3,{X-14#YGND+7;X-20#.});
             WIRE(VIOLET,3,{X-14#YGND+7;X-2#.;.#OUTPUT.HEIGHT});
             WIRE(GREEN,2,{X-1#YGND;.#MX-4});
             WIRE(GREEN,2,{X-13#YGND+8;.#MX-4});
             WIRE(GREEN,2,{X-21#YGND+8;.#YVDD});
             BOX(YELLOW,X-17#YGND+4\TO X-23.5#YVDD+2)}
    END
ENDDEFN

DEFINE CMOS_2_INVERT(INPUTS:PHYSICAL_WIRES
                     OUTPUT:PHYSICAL_WIRE  X:REAL)=MRG:
    BEGIN   VAR IN=PHYSICAL_WIRE;
    DO  IN:=INPUTS[1];
        CONNECT(IN,X-9);
        CONNECT(OUTPUT,X-2);
        CWIDTH:=X-15;
    GIVE    {RCB\AT X-9#IN.HEIGHT;
             WIRE(RED,2,{X-8#IN.HEIGHT+1;.#.+2;X-1#.});
             WIRE(RED,2,{X-6#IN.HEIGHT+3;.#YVDD+1});
             GCB\AT {X-3#YGND;X-2#YVDD;X-10#YVDD-7};
             GCBCB\AT X-2#YVDD-7;
             BCB\AT X-2#OUTPUT.HEIGHT;
             WIRE(BLUE,3,{X-10#YVDD-7;X-2#.});
             WIRE(VIOLET,3,{X-2#YVDD-7;.#OUTPUT.HEIGHT});
             WIRE(GREEN,2,{X-3#YGND+1;.#YVDD-8});
             WIRE(GREEN,2,{X-9#YVDD-6;.#YVDD-1;X-3#.});
             BOX(YELLOW,X-13.5#YVDD-10.5\TO X-6#YVDD+2);
             BOX(YELLOW,X-7#YVDD-3.5\TO X#YVDD+2)}
    END
```

```
ENDDEFN


VAR SWW=REAL;   "STICKS WIRE WIDTH"

    SCON=MRG;   "STICKS_CONTACT"

SWW:=.25;

SCON:=BOX(BLACK,-1.#-1.\TO 1#1);

DEFINE NMOS_STICKS_NAND(INPUTS:PHYSICAL_WIRES
                        OUTPUT:PHYSICAL_WIRE  X:REAL)=MRG:
    BEGIN   VAR IN=PHYSICAL_WIRE;
    DO  DO CONNECT(IN,X-10); FOR IN $E INPUTS;
        CONNECT(OUTPUT,X-2);
        CWIDTH:=X-12;
    GIVE {COLLECT {SCON\AT X-10#IN.HEIGHT;
                   WIRE(RED,SWW,{X-10#IN.HEIGHT;.#.-4;X-4#.})}
          FOR IN $E INPUTS;;
          WIRE(GREEN,SWW,{X-6#YGND+2;.#YVDD-2});
          WIRE(GREEN,SWW,{X-6#0;X-2#.;.#OUTPUT.HEIGHT});
          WIRE(RED,SWW,{X-6#0;X-10#.;.#6;X-2#.});
          BOX(YELLOW,X-8#4\TO X-4#8);
          SCON\AT {X-6#YGND+2;.#0;.#YVDD-2;X-2#OUTPUT.HEIGHT}}
    END
ENDDEFN

DEFINE NMOS_STICKS_NOR(INPUTS:PHYSICAL_WIRES
                       OUTPUT:PHYSICAL_WIRE  X:REAL)=MRG:
    BEGIN   VAR IN=PHYSICAL_WIRE;
    DO  DO CONNECT(IN,X-10); FOR IN $E INPUTS;
        CONNECT(OUTPUT,X-2);
        CWIDTH:=X-16;
    GIVE {COLLECT {SCON\AT X-10#IN.HEIGHT;
                   WIRE(RED,SWW,{X-10#IN.HEIGHT;.#.-6});
                   WIRE(GREEN,SWW,{X-14#IN.HEIGHT-4;X-6#.})}
          FOR IN $E INPUTS;;
          WIRE(GREEN,SWW,{X-14#YGND+2;.# MAX IN.HEIGHT FOR IN $E INPUTS;-4});
          WIRE(GREEN,SWW,{X-6# MIN IN.HEIGHT FOR IN $E INPUTS;-4;.#YVDD-2});
          WIRE(GREEN,SWW,{X-6#0;X-2#.;.#OUTPUT.HEIGHT});
          WIRE(RED,SWW,{X-6#0;X-10#.;.#6;X-2#.});
          BOX(YELLOW,X-8#4\TO X-4#8);
          SCON\AT {X-14#YGND+2;X-6#0;.#YVDD-2;X-2#OUTPUT.HEIGHT}}
    END
ENDDEFN

DEFINE NMOS_STICKS_INVERT(INPUTS:PHYSICAL_WIRES
                          OUTPUT:PHYSICAL_WIRE  X:REAL)=MRG:
    BEGIN   VAR IN=PHYSICAL_WIRE;
    DO  IN:=INPUTS[1];
        CONNECT(IN,X-10);
        CONNECT(OUTPUT,X-2);
        CWIDTH:=X-12;
    GIVE {WIRE(RED,SWW,{X-10#IN.HEIGHT;.#.-4;X-4#.});
          WIRE(GREEN,SWW,{X-6#YGND+2;.#YVDD-2});
          WIRE(GREEN,SWW,{X-6#0;X-2#.;.#OUTPUT.HEIGHT});
          WIRE(RED,SWW,{X-6#0;X-10#.;.#6;X-2#.});
          BOX(YELLOW,X-8#4\TO X-4#8);
          SCON\AT {X-10#IN.HEIGHT;X-6#YGND+2;.#0;.#YVDD-2;X-2#OUTPUT.HEIGHT}}
    END
```

```
ENDDEFN


DEFINE METAL2_STICKS_NAND(INPUTS:PHYSICAL_WIRES
                          OUTPUT:PHYSICAL_WIRE  X:REAL)=MRG:
    BEGIN   VAR IN=PHYSICAL_WIRE;
    DO  DO CONNECT(IN,X-10); FOR IN $E INPUTS;
        CONNECT(OUTPUT,X-2);
        CWIDTH:=X-12;
    GIVE {COLLECT {SCON\AT X-10#IN.HEIGHT;
                   WIRE(RED,SWW, {X-10#IN.HEIGHT;.#.-4;X-4#.})}
          FOR IN $E INPUTS;;
          WIRE(GREEN,SWW, {X-6#YGND+2;.#YVDD-2});
          WIRE(VIOLET,SWW, {X-6#0;X-2#.;.#OUTPUT.HEIGHT});
          WIRE(RED,SWW, {X-6#0;X-10#.;.#6;X-2#.});
          BOX(YELLOW,X-8#4\TO X-4#8);
          SCON\AT {X-6#YGND+2;.#0;.#YVDD-2;X-2#OUTPUT.HEIGHT}}
    END
ENDDEFN

DEFINE METAL2_STICKS_NOR(INPUTS:PHYSICAL_WIRES
                         OUTPUT:PHYSICAL_WIRE  X:REAL)=MRG:
    BEGIN   VAR IN=PHYSICAL_WIRE;
    DO  DO CONNECT(IN,X-10); FOR IN $E INPUTS;
        CONNECT(OUTPUT,X-2);
        CWIDTH:=X-16;
    GIVE {COLLECT {SCON\AT X-10#IN.HEIGHT;
                   WIRE(RED,SWW, {X-10#IN.HEIGHT;.#.-6});
                   WIRE(GREEN,SWW, {X-14#IN.HEIGHT-4;X-6#.})}
          FOR IN $E INPUTS;;
          WIRE(GREEN,SWW, {X-14#YGND+2;.# MAX IN.HEIGHT FOR IN $E INPUTS;-4});
          WIRE(GREEN,SWW, {X-6# MIN IN.HEIGHT FOR IN $E INPUTS;-4;.#YVDD-2});
          WIRE(VIOLET,SWW, {X-6#0;X-2#.;.#OUTPUT.HEIGHT});
          WIRE(RED,SWW, {X-6#0;X-10#.;.#6;X-2#.});
          BOX(YELLOW,X-8#4\TO X-4#8);
          SCON\AT {X-14#YGND+2;X-6#0;.#YVDD-2;X-2#OUTPUT.HEIGHT}}
    END
ENDDEFN

DEFINE METAL2_STICKS_INVERT(INPUTS:PHYSICAL_WIRES
                            OUTPUT:PHYSICAL_WIRE  X:REAL)=MRG:
    BEGIN   VAR IN=PHYSICAL_WIRE;
    DO  IN:=INPUTS[1];
        CONNECT(IN,X-10);
        CONNECT(OUTPUT,X-2);
        CWIDTH:=X-12;
    GIVE {WIRE(RED,SWW, {X-10#IN.HEIGHT;.#.-4;X-4#.});
          WIRE(GREEN,SWW, {X-6#YGND+2;.#YVDD-2});
          WIRE(VIOLET,SWW, {X-6#0;X-2#.;.#OUTPUT.HEIGHT});
          WIRE(RED,SWW, {X-6#0;X-10#.;.#6;X-2#.});
          BOX(YELLOW,X-8#4\TO X-4#8);
          SCON\AT {X-10#IN.HEIGHT;X-6#YGND+2;.#0;.#YVDD-2;X-2#OUTPUT.HEIGHT}}
    END
ENDDEFN


VAR TRANQ,TRANGND,TRANPULL=MRG;

TRANQ:={WIRE(BLACK,0, {0#0;2#.});
        WIRE(BLACK,0, {2#-2.;.#2});
        WIRE(BLACK,0, {2.5#-3.;.#3});
```

```
        WIRE(BLACK,0,{2.5#2;4#.});
        WIRE(BLACK,0,{2.5#-2.;4#.})};

TRANGND:={WIRE(BLACK,0,{-2.#0;2#.});
          WIRE(BLACK,0,{-1.2#-.8;1.2#.});
          WIRE(BLACK,0,{-.4#-1.6;.4#.})};

TRANPULL:={TRANQ\AT -4.#6;
           WIRE(BLACK,0,{-4.#6;.#2;0#.});
           WIRE(BLACK,0,{0#0;.#4});
           WIRE(BLACK,0,{0#8;.#14});
           WIRE(BLACK,0,{-.6#12;0#14;.6#12})};

DEFINE TRANS_NAND(INPUTS:PHYSICAL_WIRES
                  OUTPUT:PHYSICAL_WIRE  X:REAL)=MRG:
    BEGIN   VAR IN=PHYSICAL_WIRE;SR1,SR2=SR;R,S=REAL;
    DO  DO CONNECT(IN,X-10); FOR IN $E INPUTS;
        CONNECT(OUTPUT,X-2);
        SR2:={COLLECT IN.HEIGHT FOR IN $E INPUTS;};
        SR1:=NIL;
        WHILE DEFINED(SR2); DO
            S:= MAX R FOR R $E SR2;;
            SR1::= S <$;
            SR2:={COLLECT R FOR R $E SR2;WITH R<>S;};
        END
        CWIDTH:=X-12;
    GIVE {COLLECT {TRANQ\AT X-10#IN.HEIGHT-4;
                   WIRE(BLACK,0,{X-10#IN.HEIGHT;.#.-4})}
          FOR IN $E INPUTS;;
          COLLECT WIRE(BLACK,0,{X-6#R-2;.#S-6})
          FOR {R;S} $C YGND+2 <$ SR1 $> 6;;
          TRANGND\AT X-6#YGND;
          TRANPULL\AT X-6#0;
          WIRE(BLACK,0,{X-6#0;X-2#.;.#OUTPUT.HEIGHT})}
    END
ENDDEFN

DEFINE TRANS_NOR(INPUTS:PHYSICAL_WIRES
                 OUTPUT:PHYSICAL_WIRE  X:REAL)=MRG:
    BEGIN   VAR IN=PHYSICAL_WIRE;
    DO  DO CONNECT(IN,X-10); FOR IN $E INPUTS;
        CONNECT(OUTPUT,X-2);
        CWIDTH:=X-16;
    GIVE {COLLECT {TRANQ\ROT 270\AT X-10#IN.HEIGHT;
                   WIRE(BLACK,0,{X-14#IN.HEIGHT-4;.+2#.});
                   WIRE(BLACK,0,{X-8#IN.HEIGHT-4;.+2#.})}
          FOR IN $E INPUTS;;
          TRANGND\AT X-14#YGND;
          WIRE(BLACK,0,{X-14#YGND;.# MAX IN.HEIGHT FOR IN $E INPUTS;-4});
          WIRE(BLACK,0,{X-6# MIN IN.HEIGHT FOR IN $E INPUTS;-4;.#0;X-2#.;
                        .#OUTPUT.HEIGHT});
          TRANPULL\AT X-6#0}
    END
ENDDEFN

DEFINE TRANS_INVERT(INPUTS:PHYSICAL_WIRES
                    OUTPUT:PHYSICAL_WIRE  X:REAL)=MRG:
    BEGIN   VAR IN=PHYSICAL_WIRE;
    DO  IN:=INPUTS[1];
        CONNECT(IN,X-10);
        CONNECT(OUTPUT,X-2);
```

```
        CWIDTH:=X-12;
    GIVE {TRANQ\AT X-10#IN.HEIGHT-4;
          WIRE(BLACK,0,{X-10#IN.HEIGHT;.#.-4});
          WIRE(BLACK,0,{X-6#YGND;.#IN.HEIGHT-6});
          TRANGND\AT X-6#YGND;
          TRANPULL\AT X-6#0;
          WIRE(BLACK,0,{X-6#IN.HEIGHT-2;.#0;X-2#.;.#OUTPUT.HEIGHT})}
    END
ENDDEFN
```

In addition to the gate producing routines, each technology requires a function which will draw the final signal wires on the chip. This wire function accepts a chip as the input parameter and produces an MRG as the output parameter. Since we want the user to be able to change the wire drawing routine at will, this too will be a global variable which is a suspendable function. The following type declaration declares the type. The wire drawing routines are listed after the type declaration.

```
TYPE    CHIP_TO_MRG= //MRG(CHIP)\\;

DEFINE NMOS_WIRES(C:CHIP)=MRG:
    BEGIN   VAR S=SIGNAL_WIRE;LEFT,RIGHT,PWIDTH=REAL;
    DO  LEFT:=CWIDTH+5;
        RIGHT:=-2.;
        PWIDTH:=WIDTH(POWER) MAX 4;
    GIVE    {COLLECT WIRE(BLUE,3,{S.PHYSICAL.LEFT#S.PHYSICAL.HEIGHT;
                                  S.PHYSICAL.RIGHT#.})
             FOR S $E C.SIGNALS;
             EACH_DO @(S.PHYSICAL).LEFT::= MAX LEFT;
                     @(S.PHYSICAL).RIGHT::= MIN RIGHT;;;
             BOX(BLUE,CWIDTH+3#YVDD-3\TO 4#YVDD+(PWIDTH-3 MAX 2));
             BOX(BLUE,CWIDTH-1#YGND+2-PWIDTH\TO 0#YGND+2)}
    END
ENDDEFN

DEFINE METAL2_WIRES(C:CHIP)=MRG:
    BEGIN   VAR S=SIGNAL_WIRE;LEFT,RIGHT,PWIDTH=REAL;
    DO  LEFT:=CWIDTH+2;
        RIGHT:=-5.;
        PWIDTH:=WIDTH(POWER) MAX 4;
    GIVE    {COLLECT WIRE(BLUE,3,{S.PHYSICAL.LEFT#S.PHYSICAL.HEIGHT;
                                  S.PHYSICAL.RIGHT#.})
             FOR S $E C.SIGNALS;
             EACH_DO @(S.PHYSICAL).LEFT::= MAX LEFT;
                     @(S.PHYSICAL).RIGHT::= MIN RIGHT;;;
             BOX(BLUE,CWIDTH#YVDD-3\TO 1#YVDD+(PWIDTH-3 MAX 2));
             BOX(BLUE,CWIDTH-4#YGND+2-PWIDTH\TO -3.#YGND+2)}
    END
ENDDEFN

DEFINE LOGICAL_WIRES(C:CHIP)=MRG:
    BEGIN   VAR S=SIGNAL_WIRE;LEFT,RIGHT=REAL;
    DO  LEFT:=CWIDTH-2;
        RIGHT:=5;
    GIVE    {COLLECT WIRE(GREEN,0,{S.PHYSICAL.LEFT#S.PHYSICAL.HEIGHT;
                                   S.PHYSICAL.RIGHT#.})
             FOR S $E C.SIGNALS;
```

```
            EACH_DO @(S.PHYSICAL).LEFT::= MAX LEFT;
                    @(S.PHYSICAL).RIGHT::= MIN RIGHT;;}
    END
ENDDEFN

DEFINE CMOS_WIRES(C:CHIP)=MRG:
    BEGIN   VAR S=SIGNAL_WIRE;LEFT,RIGHT=REAL;
    DO  LEFT:=CWIDTH+12;
        RIGHT:=-2.;
    GIVE {COLLECT WIRE(BLUE,3,{S.PHYSICAL.LEFT#S.PHYSICAL.HEIGHT;
                               S.PHYSICAL.RIGHT#.})
          FOR S $E C.SIGNALS;
          EACH_DO @(S.PHYSICAL).LEFT::= MAX LEFT;
                  @(S.PHYSICAL).RIGHT::= MIN RIGHT;;;
          WIRE(BLUE,4,{CWIDTH+4#YVDD;2#.});
          WIRE(BLUE,4,{CWIDTH#YGND;-2.#.})}
    END
ENDDEFN

DEFINE CMOS_2_WIRES(C:CHIP)=MRG:
    BEGIN   VAR S=SIGNAL_WIRE;LEFT,RIGHT=REAL;
    DO  LEFT:=CWIDTH+8;
        RIGHT:=-2.;
    GIVE {COLLECT WIRE(BLUE,3,{S.PHYSICAL.LEFT#S.PHYSICAL.HEIGHT;
                               S.PHYSICAL.RIGHT#.})
          FOR S $E C.SIGNALS;
          EACH_DO @(S.PHYSICAL).LEFT::= MAX LEFT;
                  @(S.PHYSICAL).RIGHT::= MIN RIGHT;;;
          WIRE(BLUE,4,{CWIDTH+5#YVDD;2#.});
          WIRE(BLUE,4,{CWIDTH#YGND;-2.#.})}
    END
ENDDEFN

DEFINE NMOS_STICKS_WIRES(C:CHIP)=MRG:
    BEGIN   VAR S=SIGNAL_WIRE;LEFT,RIGHT=REAL;
    DO  LEFT:=CWIDTH+2;
        RIGHT:=-2.;
    GIVE    {COLLECT WIRE(BLUE,SWW,{S.PHYSICAL.LEFT#S.PHYSICAL.HEIGHT;
                                    S.PHYSICAL.RIGHT#.})
             FOR S $E C.SIGNALS;
             EACH_DO @(S.PHYSICAL).LEFT::= MAX LEFT;
                     @(S.PHYSICAL).RIGHT::= MIN RIGHT;;;
             WIRE(BLUE,SWW,{LEFT#YVDD-2;2#.});
             WIRE(BLUE,SWW,{CWIDTH-2#YGND+2;RIGHT#.})}
    END
ENDDEFN

"METAL-2 sticks uses the NMOS_STICKS_WIRES routine"

DEFINE TRANS_WIRES(C:CHIP)=MRG:
    BEGIN   VAR S=SIGNAL_WIRE;LEFT,RIGHT=REAL;
    DO  LEFT:=CWIDTH+2;
        RIGHT:=-2.;
    GIVE    {COLLECT WIRE(BLACK,0,{S.PHYSICAL.LEFT#S.PHYSICAL.HEIGHT;
                                   S.PHYSICAL.RIGHT#.})
             FOR S $E C.SIGNALS;
             EACH_DO @(S.PHYSICAL).LEFT::= MAX LEFT;
                     @(S.PHYSICAL).RIGHT::= MIN RIGHT;;}
    END
ENDDEFN
```

In addition to the wire drawing routine, each technology has a routine for initializing the global coordinates and a routine for calculating wire positions in the wiring channel. The first routine requires the chip as an input parameter, and produces no output. The second routine takes a channel index, an INTEGER, and returns the channel position, a REAL. These two routines will be CHIP_CONSUMERs and INT_TO_REALs.

```
TYPE     CHIP_CONSUMER= //(CHIP)\\;

         INT_TO_REAL= //REAL(INT)\\;

DEFINE NMOS_SETUP(C:CHIP):
    BEGIN   VAR S=SIGNAL_WIRE;
    YGND:= -9.*(MAX S.VHEIGHT FOR S $E C.SIGNALS;)-6;
    YVDD:=9;
    END
ENDDEFN

"METAL2 uses NMOS_SETUP"

"LOGICAL uses NMOS_SETUP"

DEFINE CMOS_SETUP(C:CHIP):
    BEGIN   VAR S=SIGNAL_WIRE;
    YVDD:=0;
    YGND:=-9.*(MAX S.VHEIGHT FOR S $E C.SIGNALS;)-19;
    END
ENDDEFN

"CMOS_2 uses CMOS_SETUP"

DEFINE NMOS_STICKS_SETUP(C:CHIP):
    BEGIN   VAR S=SIGNAL_WIRE;
    YGND:= -10.*(MAX S.VHEIGHT FOR S $E C.SIGNALS;)-6;
    YVDD:=12;
    END
ENDDEFN

"METAL2_STICKS uses NMOS_STICKS_SETUP"

DEFINE TRANS_SETUP(C:CHIP):
    BEGIN   VAR S=SIGNAL_WIRE;
    YGND:= -10.*(MAX S.VHEIGHT FOR S $E C.SIGNALS;)-8;
    END
ENDDEFN


DEFINE NMOS_WIRE_HEIGHTS(I:INT)=REAL:   1-9*I   ENDDEFN

"METAL2 uses NMOS_WIRE_HEIGHTS"

"LOGICAL uses NMOS_WIRE_HEIGHTS"
```

```
DEFINE CMOS_WIRE_HEIGHTS(I:INT)=REAL:    -5.-9*I ENDDEFN

"CMOS_2 uses CMOS_WIRE_HEIGHTS"

DEFINE NMOS_STICKS_WIRE_HEIGHTS(I:INT)=REAL:    6-10*I    ENDDEFN

"METAL2_STICKS uses NMOS_STICKS_WIRE_HEIGHTS"

DEFINE TRANS_WIRE_HEIGHTS(I:INT)=REAL:    6-10*I    ENDDEFN
```

The final technology dependent routines in RLC concern wire packing and gate sorting. For each technology, we may desire to have a routine which will pack the wires in the wiring channel. Similarly, we may desire sorting routines which sort the gates to achieve higher performance or smaller area. These routines are similar to the SETUP routines: They require a CHIP as an input parameter, and they return no output data.

With these considerations in mind, we can declare a TECHNOLOGY datatype which contains all of the technology-dependent information. We can define new technologies and add them to the technology list at any time, and can then output our circuits in any of the available technologies.

```
TYPE     TECHNOLOGY=
             [NAND,NOR,INVERT: GATE_PRODUCER
              WIRES:           CHIP_TO_MRG
              PACK,SORT,SETUP: CHIP_CONSUMER
              WIRE_HEIGHT:     INT_TO_REAL
              VDD,GND:         REAL
              NAME:            QS];

         TECHNOLOGIES= { TECHNOLOGY };

VAR TECHNOLOGIES= TECHNOLOGIES;

VAR NMOS,METAL2,LOGICAL,CMOS,CMOS2,NMOS_STICKS,METAL2_STICKS,
    TRANSISTOR=TECHNOLOGY;

NMOS:=[NAND://:NMOS_NAND(PHYSICAL_WIRES,PHYSICAL_WIRE,REAL)\\
       NOR://:NMOS_NOR(PHYSICAL_WIRES,PHYSICAL_WIRE,REAL)\\
       INVERT://:NMOS_INVERT(PHYSICAL_WIRES,PHYSICAL_WIRE,REAL)\\
       WIRES://:NMOS_WIRES(CHIP)\\
       PACK://:NMOS_PACK_2(CHIP)\\
       SORT://:NO_SORT(CHIP)\\
       SETUP://:NMOS_SETUP(CHIP)\\
       WIRE_HEIGHT://:NMOS_WIRE_HEIGHTS(INT)\\
       VDD:3
       GND:0
       NAME:'NMOS'];

METAL2:=[NAND://:METAL2_NAND(PHYSICAL_WIRES,PHYSICAL_WIRE,REAL)\\
         NOR://:METAL2_NOR(PHYSICAL_WIRES,PHYSICAL_WIRE,REAL)\\
         INVERT://:METAL2_INVERT(PHYSICAL_WIRES,PHYSICAL_WIRE,REAL)\\
         WIRES://:METAL2_WIRES(CHIP)\\
```

```
            PACK://:NMOS_PACK_2(CHIP)\\
            SORT://:NO_SORT(CHIP)\\
            SETUP://:NMOS_SETUP(CHIP)\\
            WIRE_HEIGHT://:NMOS_WIRE_HEIGHTS(INT)\\
            VDD:3
            GND:0
            NAME:'METAL2'];

LOGICAL:=[NAND://:LOGICAL_NAND(PHYSICAL_WIRES,PHYSICAL_WIRE,REAL)\\
           NOR://:LOGICAL_NOR(PHYSICAL_WIRES,PHYSICAL_WIRE,REAL)\\
           INVERT://:LOGICAL_INVERT(PHYSICAL_WIRES,PHYSICAL_WIRE,REAL)\\
           WIRES://:LOGICAL_WIRES(CHIP)\\
           PACK://:NMOS_PACK_2(CHIP)\\
           SORT://:NO_SORT(CHIP)\\
           SETUP://:NMOS_SETUP(CHIP)\\
           WIRE_HEIGHT://:NMOS_WIRE_HEIGHTS(INT)\\
           NAME:'LOGICAL'];

CMOS:=[NAND://:CMOS_NAND(PHYSICAL_WIRES,PHYSICAL_WIRE,REAL)\\
       NOR://:CMOS_NOR(PHYSICAL_WIRES,PHYSICAL_WIRE,REAL)\\
       INVERT://:CMOS_INVERT(PHYSICAL_WIRES,PHYSICAL_WIRE,REAL)\\
       WIRES://:CMOS_WIRES(CHIP)\\
       PACK://:NMOS_PACK_2(CHIP)\\
       SORT://:NO_SORT(CHIP)\\
       SETUP://:CMOS_SETUP(CHIP)\\
       WIRE_HEIGHT://:CMOS_WIRE_HEIGHTS(INT)\\
       VDD:3
       GND:-1.
       NAME:'CMOS'];

CMOS2:=[NAND://:CMOS_2_NAND(PHYSICAL_WIRES,PHYSICAL_WIRE,REAL)\\
        NOR://:CMOS_2_NOR(PHYSICAL_WIRES,PHYSICAL_WIRE,REAL)\\
        INVERT://:CMOS_2_INVERT(PHYSICAL_WIRES,PHYSICAL_WIRE,REAL)\\
        WIRES://:CMOS_2_WIRES(CHIP)\\
        PACK://:NMOS_PACK_2(CHIP)\\
        SORT://:NO_SORT(CHIP)\\
        SETUP://:CMOS_SETUP(CHIP)\\
        WIRE_HEIGHT://:CMOS_WIRE_HEIGHTS(INT)\\
        VDD:3
        GND:-1.
        NAME:'CMOS2'];

NMOS_STICKS:=
     [NAND://:NMOS_STICKS_NAND(PHYSICAL_WIRES,PHYSICAL_WIRE,REAL)\\
      NOR://:NMOS_STICKS_NOR(PHYSICAL_WIRES,PHYSICAL_WIRE,REAL)\\
      INVERT://:NMOS_STICKS_INVERT(PHYSICAL_WIRES,PHYSICAL_WIRE,REAL)\\
      WIRES://:NMOS_STICKS_WIRES(CHIP)\\
      PACK://:NMOS_PACK_2(CHIP)\\
      SORT://:NO_SORT(CHIP)\\
      SETUP://:NMOS_STICKS_SETUP(CHIP)\\
      WIRE_HEIGHT://:NMOS_STICKS_WIRE_HEIGHTS(INT)\\
      VDD:2
      GND:-2.
      NAME:'NMOS_STICKS'];

METAL2_STICKS:=
     [NAND://:METAL2_STICKS_NAND(PHYSICAL_WIRES,PHYSICAL_WIRE,REAL)\\
      NOR://:METAL2_STICKS_NOR(PHYSICAL_WIRES,PHYSICAL_WIRE,REAL)\\
      INVERT://:METAL2_STICKS_INVERT(PHYSICAL_WIRES,PHYSICAL_WIRE,REAL)\\
      WIRES://:NMOS_STICKS_WIRES(CHIP)\\
      PACK://:NMOS_PACK_2(CHIP)\\
```

```
      SORT://:NO_SORT(CHIP)\\
      SETUP://:NMOS_STICKS_SETUP(CHIP)\\
      WIRE_HEIGHT://:NMOS_STICKS_WIRE_HEIGHTS(INT)\\
      VDD:1
      GND:-1.
      NAME:'METAL2_STICKS'];

TRANSISTOR:=[NAND://:TRANS_NAND(PHYSICAL_WIRES,PHYSICAL_WIRE,REAL)\\
             NOR://:TRANS_NOR(PHYSICAL_WIRES,PHYSICAL_WIRE,REAL)\\
             INVERT://:TRANS_INVERT(PHYSICAL_WIRES,PHYSICAL_WIRE,REAL)\\
             WIRES://:TRANS_WIRES(CHIP)\\
             PACK://:NMOS_PACK_2(CHIP)\\
             SORT://:NO_SORT(CHIP)\\
             SETUP://:TRANS_SETUP(CHIP)\\
             WIRE_HEIGHT://:TRANS_WIRE_HEIGHTS(INT)\\
             VDD:1
             GND:-1.
             NAME:'TRANSISTOR'];

DEFINE NMOS=MRG:     COMPILE(CHIP,NMOS)  ENDDEFN

DEFINE METAL2=MRG:   COMPILE(CHIP,METAL2)     ENDDEFN

DEFINE LOGICAL=MRG:     COMPILE(CHIP,LOGICAL)  ENDDEFN

DEFINE CMOS=MRG:     COMPILE(CHIP,CMOS)  ENDDEFN

DEFINE CMOS2=MRG:    COMPILE(CHIP,CMOS2) ENDDEFN

DEFINE NMOS_STICKS=MRG:.    COMPILE(CHIP,NMOS_STICKS)  ENDDEFN

DEFINE METAL2_STICKS=MRG:     COMPILE(CHIP,METAL2_STICKS)  ENDDEFN

DEFINE TRANSISTOR=MRG:     COMPILE(CHIP,TRANSISTOR)  ENDDEFN

DEFINE PUT_NMOS:     PUT(CHIP,NMOS);      ENDDEFN

DEFINE PUT_METAL2:   PUT(CHIP,METAL2);        ENDDEFN

DEFINE PUT_LOGICAL:     PUT(CHIP,LOGICAL);      ENDDEFN

DEFINE PUT_CMOS:     PUT(CHIP,CMOS);      ENDDEFN

DEFINE PUT_CMOS2:    PUT(CHIP,CMOS2);     ENDDEFN

DEFINE PUT_NMOS_STICKS:     PUT(CHIP,NMOS_STICKS);      ENDDEFN

DEFINE PUT_METAL2_STICKS:     PUT(CHIP,METAL2_STICKS);      ENDDEFN

DEFINE PUT_TRANSISTOR:     PUT(CHIP,TRANSISTOR);      ENDDEFN

TECHNOLOGIES:= {NMOS;METAL2;LOGICAL;CMOS;CMOS2;NMOS_STICKS;METAL2_STICKS;
                TRANSISTOR};
```

Now that we have our basic technologies defined, we will present the data structure definitions for representing the chip. These definitions, which follow

the definitions in Chapter 5, represent the wires and gates of the chip. In addition, the definition for the CHIP datatype is given. The DCHIP type is a swappable CHIP, which means that an instance of type DCHIP can be swapped into the virtual memory by the system. ICL allows the user to specify what datatypes are swappable, because the user can do a much better job of describing conceptual units than a program can.

```
TYPE    SIGNAL_WIRE= [FROM:GATE
                      TO:GATES
                      NAME:QS
                      PHYSICAL:PHYSICAL_WIRE
                      INPUT,OUTPUT:BOOL
                      VLEFT,VRIGHT,VHEIGHT:INT
                      VINVERT:SIGNAL_WIRE];

        SIGNAL_WIRES= { SIGNAL_WIRE };

        GATE= [INPUTS:SIGNAL_WIRES
               OUTPUT:SIGNAL_WIRE
               TYPE:GATE_TYPE
               INDEX:INT
               RINDEX:REAL];

        GATES= { GATE };

        GATE_TYPE= SCALAR(NAND,NOR,INVERT);

        CHIP= [GATES:GATES
               SIGNALS:SIGNAL_WIRES
               SIGNAL_COUNT:INT
               NAME,DESCRIPTION:QS];

        DCHIP= PRIVATE DISK_NODE;

VAR YVDD,YGND,POWER,CWIDTH=REAL;

    CHIP=CHIP;

LET DCHIP BECOME CHIP BY MACRO-10('INCOR$')

DEFINE DISK(C:CHIP)=DCHIP:  MACRO-10('DSKIZ$')

DEFINE MODIFIED(D:DCHIP):   MACRO-10('DMOD$')

DEFINE PUT(D:DCHIP  N:QS):
    BEGIN   LET DCHIP BECOME GLS24 BY MACRO-10('IDENT$')
    PUT(D,N,'DCHIP  1/2/81');
    END
ENDDEFN
```

```
DEFINE PUT(C:CHIP): PUT(DISK(C),C.NAME);     ENDDEFN

DEFINE GET(N:QS)=DCHIP:
    BEGIN   LET GLS24 BECOME DCHIP BY MACRO-10('IDENT$')
    GET(N,'DCHIP  1/2/81')
    END
ENDDEFN
```

This next section of code is the actual compiler, which closely follows the code in Chapter 5. Because of the similarity of the code, no additional comments will be given here.

```
DEFINE PHYSICAL(SW:SIGNAL_WIRE)=PHYSICAL_WIRE:  SW.PHYSICAL ENDDEFN

DEFINE PHYSICAL(SWS:SIGNAL_WIRES)=PHYSICAL_WIRES:
    BEGIN   VAR S=SIGNAL_WIRE;
    {COLLECT S\PHYSICAL FOR S $E SWS;}
    END
ENDDEFN

DEFINE INPUTS(C:CHIP)=SIGNAL_WIRES:
    BEGIN   VAR S=SIGNAL_WIRE;
    {COLLECT S FOR S $E C.SIGNALS;WITH S.INPUT;}
    END
ENDDEFN

DEFINE OUTPUTS(C:CHIP)=SIGNAL_WIRES:
    BEGIN   VAR S=SIGNAL_WIRE;
    {COLLECT S FOR S $E C.SIGNALS;WITH S.OUTPUT;}
    END
ENDDEFN

DEFINE CONNECT(WIRE:PHYSICAL_WIRE  X:REAL):
    @(WIRE).LEFT::= MIN X;
    @(WIRE).RIGHT::= MAX X;
ENDDEFN

DEFINE INITIALIZE_WIRES(C:CHIP  T:TECHNOLOGY):
    BEGIN   VAR S=SIGNAL_WIRE;
    FOR S $E C.SIGNALS; DO
        @(S).PHYSICAL:=[LEFT:IF S.INPUT THEN -999999. ELSE 999999 FI
                        RIGHT:IF S.OUTPUT THEN 999999 ELSE -999999. FI
                        HEIGHT:<*T.WIRE_HEIGHT*>(S.VHEIGHT)
                        NAME:S.NAME];
    END
    END
ENDDEFN

DEFINE DRAW_CELLS(C:CHIP  T:TECHNOLOGY)=MRG:
    BEGIN   VAR X=REAL;G=GATE;
    {COLLECT <* CASE G.TYPE OF
                NOR:    T.NOR
                NAND:   T.NAND
                INVERT: T.INVERT
                ENDCASE *> (G.INPUTS\PHYSICAL,G.OUTPUT\PHYSICAL,CWIDTH)\DISK
     FOR G $E REVERSE(C.GATES);}
    END
ENDDEFN
```

```
DEFINE LOAD(S:SIGNAL_WIRE)=REAL:
    BEGIN   VAR G=GATE;T=SIGNAL_WIRE;
    (+ CASE G.TYPE OF
         NOR:    1
         INVERT: 1
         NAND:   +1 FOR T $E G.INPUTS;
       ENDCASE FOR G $E S.TO;)*Q_LOAD +
    LOAD(BLUE,WIDTH(BLUE),S.PHYSICAL.RIGHT-S.PHYSICAL.LEFT)
    END
ENDDEFN

DEFINE COMPILE(C:CHIP  T:TECHNOLOGY)=MRG:
   BEGIN   VAR M=MRG;
    DO  CWIDTH:=0;
        POWER:=0;
        <*T.SORT*>(C);
        <*T.PACK*>(C);
        <*T.SETUP*>(C);
        INITIALIZE_WIRES(C,T);
        M:=DRAW_CELLS(C,T);
    GIVE {M;<*T.WIRES*>(C)}\DISK
    END
ENDDEFN

DEFINE PUT(C:CHIP  T:TECHNOLOGY):
    BEGIN   VAR M=MRG;G=GATE;S=SIGNAL_WIRE;
    M:=COMPILE(C,T);
    PUT([NAME:C.NAME
         DESCRIPTION:C.DESCRIPTION
         LAYOUT:M
         VDD:T.VDD#YVDD-2
         GND:CWIDTH+T.GND#YGND+2
         POWER:POWER
         PORTS:{COLLECT  [NAME:{S.NAME}
                          AT:{{[COLOR:BLUE
                                EDGE:WEST
                                AT:S.PHYSICAL.LEFT#S.PHYSICAL.HEIGHT]}}
                          LOAD: LOAD(S)]
                FOR S $E C\INPUTS;;
                COLLECT [NAME:{S.NAME}
                          AT:{{[COLOR:BLUE
                                EDGE:EAST
                                AT:S.PHYSICAL.RIGHT#S.PHYSICAL.HEIGHT]}}
                          DRIVE:1
                          LOAD: LOAD(S)]
                FOR S $E C\OUTPUTS;}]\DISK,C.NAME);
    END
ENDDEFN


DEFINE EQ(A,B:GATE)=BOOL:    MACRO-10('LSPEQ$')

DEFINE EQ(A,B:SIGNAL_WIRE)=BOOL:    MACRO-10('LSPEQ$')

DEFINE LINK_INPUT(G:GATE  S:SIGNAL_WIRE):
    @(S).TO::= G <$;
    @(G).INPUTS::= S <$;
ENDDEFN

DEFINE LINK_OUTPUT(G:GATE  S:SIGNAL_WIRE):
    @(G).OUTPUT:=S;
```

```
    @(S).FROM:=G;
ENDDEFN

DEFINE UNLINK_INPUT(G:GATE  S:SIGNAL_WIRE):
    BEGIN   VAR Q=GATE;R=SIGNAL_WIRE;
    @(S).TO:={COLLECT Q FOR Q $E S.TO;WITH -(Q\EQ G);};
    @(G).INPUTS:={COLLECT R FOR R $E G.INPUTS;WITH -(R\EQ S);};
    END
ENDDEFN

DEFINE UNLINK_OUTPUT(G:GATE  S:SIGNAL_WIRE):
    @(S).FROM:=NIL;
    @(G).OUTPUT:=NIL;
ENDDEFN

DEFINE ELIMINATE(G:GATE):
    BEGIN   VAR Q=GATE;
    CHIP.GATES:={COLLECT Q FOR Q $E CHIP.GATES;WITH -(Q\EQ G);};
    END
ENDDEFN

DEFINE ELIMINATE(S:SIGNAL_WIRE):
    BEGIN   VAR R=SIGNAL_WIRE;
    CHIP.SIGNALS:={COLLECT R FOR R $E CHIP.SIGNALS;WITH -(R\EQ S);};
    IF DEFINED(S.VINVERT) THEN @(S.VINVERT).VINVERT:=NIL; FI
    END
ENDDEFN

DEFINE VINVERT(A,B:SIGNAL_WIRE):
    IF DEFINED(A.VINVERT) THEN @(A.VINVERT).VINVERT:=NIL; FI
    IF DEFINED(B.VINVERT) THEN @(B.VINVERT).VINVERT:=NIL; FI
    @(A).VINVERT:=B;
    @(B).VINVERT:=A;
ENDDEFN

DEFINE FUSE(A,B:SIGNAL_WIRE):
    BEGIN   VAR G=GATE;C=CHAR;
    IF DEFINED(B.FROM)!B.INPUT THEN
        IF DEFINED(A.FROM)!A.INPUT THEN HELP;
        ELSE    @(A).INPUT:=B.INPUT;
                G:=B.FROM;
                IF DEFINED(G) THEN
                    UNLINK_OUTPUT(G,B);
                    LINK_OUTPUT(G,A); FI FI FI
    IF ALWAYS C\DIGIT FOR C $E A.NAME; THEN @(A).NAME:=B.NAME; FI
    IF DEFINED(B.VINVERT) THEN VINVERT(A,B.VINVERT); FI
    @(A).OUTPUT::=!B.OUTPUT;
    FOR G $E B.TO; DO
        UNLINK_INPUT(G,B);
        LINK_INPUT(G,A);
    END
    ELIMINATE(B);
    END
ENDDEFN


LET QS BECOME SIGNAL_WIRE BY
    BEGIN   VAR S=SIGNAL_WIRE;
    IF THERE_IS S.NAME\EQ QS FOR S $E CHIP.SIGNALS; THEN S
    ELSE    DO  S:=[NAME:QS];
                CHIP.SIGNALS::= S <$;
```

```
            GIVE S  FI
    END;

DEFINE NEW_SIGNAL=SIGNAL_WIRE:  SC((CHIP.SIGNAL_COUNT::=+1;))    ENDDEFN

DEFINE SET(S:SIGNAL_WIRE  G:GATE):  LINK_OUTPUT(G,S);    ENDDEFN

LET GATE BECOME SIGNAL_WIRE BY
    BEGIN   VAR S=SIGNAL_WIRE;
    DO  S:=NEW_SIGNAL;
        LINK_OUTPUT(GATE,S);
    GIVE S
    END;

DEFINE INPUT(S:SIGNAL_WIRE):     @(S).INPUT:=TRUE;    ENDDEFN

DEFINE OUTPUT(S:SIGNAL_WIRE):    @(S).OUTPUT:=TRUE;   ENDDEFN

DEFINE INPUTS(SQS:SQS):
    BEGIN   VAR QS=QS;
    DO INPUT(QS); FOR QS $E SQS;
    END
ENDDEFN

DEFINE OUTPUTS(SQS:SQS):
    BEGIN   VAR QS=QS;
    DO OUTPUT(QS); FOR QS $E SQS;
    END
ENDDEFN

DEFINE NAME(QS:QS): CHIP.NAME:=QS;  ENDDEFN

DEFINE DESCRIPTION(QS:QS):  CHIP.DESCRIPTION:=QS;    ENDDEFN

DEFINE NEW_CHIP:     CHIP:=NIL;  ENDDEFN

DEFINE FINISH:
    CHIP.GATES:=REVERSE(CHIP.GATES);
ENDDEFN

DEFINE NEW_GATE(SWS:SIGNAL_WIRES  TYPE:GATE_TYPE)=GATE:
    BEGIN   VAR GATE=GATE;SW=SIGNAL_WIRE;
    DO  GATE:=[TYPE:TYPE];
        CHIP.GATES::= GATE <$;
        DO LINK_INPUT(GATE,SW); FOR SW $E SWS;
    GIVE GATE
    END
ENDDEFN

DEFINE NAND(SWS:SIGNAL_WIRES)=GATE: NEW_GATE(SWS,NAND)   ENDDEFN

DEFINE NOR(SWS:SIGNAL_WIRES)=GATE:  NEW_GATE(SWS,NOR)    ENDDEFN

DEFINE INVERT(SW:SIGNAL_WIRE)=GATE: NEW_GATE((SW),INVERT)    ENDDEFN

DEFINE AND(SWS:SIGNAL_WIRES)=GATE:  SWS\NAND\INVERT      ENDDEFN

DEFINE OR(SWS:SIGNAL_WIRES)=GATE:   SWS\NOR\INVERT       ENDDEFN
```

```
DEFINE NAND(A,B:SIGNAL_WIRE)=GATE:  NAND({A;B})      ENDDEFN

DEFINE NOR(A,B:SIGNAL_WIRE)=GATE:  NOR({A;B})      ENDDEFN

DEFINE AND(A,B:SIGNAL_WIRE)=GATE:  AND({A;B})      ENDDEFN

DEFINE OR(A,B:SIGNAL_WIRE)=GATE:  OR({A;B})      ENDDEFN

DEFINE XOR(A,B:SIGNAL_WIRE)=GATE:
    BEGIN   VAR C=SIGNAL_WIRE;
    DO  C:=NAND(A,B);
    GIVE NAND(A\NAND C,B\NAND C)
    END
ENDDEFN
```

The following code lists the optimizers defined in the RLC. These optimizers look at the logical structure of the chip, replacing gates while preserving functionality. The GET_INVERT function is a utility function which generates the inverse of its input signal, using existing inverters if they exist. The REMOVE_INVERTERS function removes extra inverters from the chip's logic. REMOVE_REDUNDANCIES looks for redundant gates, removing those which don't add to the functionality of the chip. The DE_MORGAN function will convert a NAND gate into a NOR implementation and turn a NOR gate into a NAND implementation. This function is used by REMOVE_NANDS and REMOVE_NORS, which eliminate all instances of their respective gates. REMOVE_NANDS is used to turn a NAND circuit into a NOR circuit, while REMOVE_NORS does the inverse transformation. DE_MORGAN_COST is a function which computes the relative cost of a NAND or NOR gate in the chip's logic equations. The DE_MORGAN function calls this cost routine to determine which gates to replace. If the cost of converting a particular gate into its dual gate is negative, which means we would use fewer gates to implement the chip, the DE_MORGAN function will perform the transformation. The UNIQUE_INPUTS function removes extra inputs to NAND and NOR gates. If a particular gate has more than one connection to a signal, all but one of those connections are removed. Finally, the MERGE function moves signals which connect to strings of NAND or NOR gates.

```
DEFINE GET_INVERT(S:SIGNAL_WIRE)=SIGNAL_WIRE:
    BEGIN   VAR T=SIGNAL_WIRE;G=GATE;
    IF S.FROM.TYPE=INVERT THEN
        GIVING S.FROM.INPUTS[1]
        DO  IF -(DEFINED(S.TO)!S.OUTPUT) THEN
                G:=S.FROM;
                UNLINK_OUTPUT(G,S);
                UNLINK_INPUT(G,G.INPUTS[1]);
                ELIMINATE(G);
                ELIMINATE(S); FI
        END
    EF DEFINED(S.VINVERT) THEN S.VINVERT
    EF THERE_IS G.TYPE=INVERT FOR G $E S.TO; THEN G.OUTPUT
```

```
    ELSE INVERT(S) FI
    END
ENDDEFN

DEFINE REMOVE_INVERTERS:
    BEGIN   VAR G=GATE;S,T=SIGNAL_WIRE;
    FOR G $E CHIP.GATES;WITH G.TYPE=INVERT;WITH DEFINED(G.OUTPUT); DO
        S:=G.OUTPUT;
        T:=G.INPUTS[1];
        UNLINK_OUTPUT(G,S);
        UNLINK_INPUT(G,T);
        ELIMINATE(G);
        FUSE(T\GET_INVERT,S);
    END
    END
ENDDEFN

DEFINE REMOVE_REDUNDANCIES:
    BEGIN   VAR G,G1,G2=GATE;LIST=GATES;S,S1,S2=SIGNAL_WIRE;I=INT;
    LIST:=NIL;
    ((FOR G1 $E CHIP.GATES;&& FOR I FROM 2 BY 1;)WITH DEFINED(G1.OUTPUT);
     !! FOR G2 $E CHIP.GATES[I-];WITH DEFINED(G2.OUTPUT);)
     WITH   IF G1.TYPE<>G2.TYPE THEN FALSE
            ELSE    ALWAYS THERE_IS S1\EQ S2 FOR S2 $E G2.INPUTS;
                        FOR S1 $E G1.INPUTS; &
                    ALWAYS THERE_IS S2\EQ S1 FOR S1 $E G1.INPUTS;
                        FOR S2 $E G2.INPUTS; FI; DO
        S2:=G2.OUTPUT;
        S1:=G1.OUTPUT;
        FOR S $E G2.INPUTS; DO
            UNLINK_INPUT(G2,S);
        END
        UNLINK_OUTPUT(G2,S2);
        LIST::= G2 <$;
        FUSE(S1,S2);
    END
    DO ELIMINATE(G); FOR G $E LIST;
    END
ENDDEFN

DEFINE DE_MORGAN(G:GATE):
    BEGIN   VAR TYPE=GATE_TYPE;S,T=SIGNAL_WIRE;SWS=SIGNAL_WIRES;N=GATE;
    TYPE:= CASE G.TYPE OF
           NAND:    NOR
           NOR:     NAND
           ELSE:    NIL
           ENDCASE;
    IF DEFINED(TYPE) THEN
        SWS:=NIL;
        FOR S $E G.INPUTS; DO
            UNLINK_INPUT(G,S);
            SWS::= S\GET_INVERT <$;
        END
        N:=NEW_GATE(SWS,TYPE);
        S:=G.OUTPUT;
        UNLINK_OUTPUT(G,S);
        ELIMINATE(G);
        IF THERE_IS G.TYPE=INVERT FOR G $E S.TO;
        THEN    T:=G.OUTPUT;
                UNLINK_OUTPUT(G,T);
                LINK_OUTPUT(N,T);
```

```
                UNLINK_INPUT(G,S);
                IF DEFINED(S.TO)!S.OUTPUT
                THEN    LINK_INPUT(G,T);
                        LINK_OUTPUT(G,S);
                ELSE    ELIMINATE(G);
                        ELIMINATE(S); FI
        ELSE    LINK_OUTPUT(INVERT(N),S);   FI FI
    END
ENDDEFN

DEFINE REMOVE_NANDS:
    BEGIN   VAR G=GATE;
    CHIP.GATES:=REVERSE(CHIP.GATES);
    DO DE_MORGAN(G); FOR G $E CHIP.GATES;WITH G.TYPE=NAND;
    FINISH;
    END
ENDDEFN

DEFINE REMOVE_NORS:
    BEGIN   VAR G=GATE;
    CHIP.GATES:=REVERSE(CHIP.GATES);
    DO DE_MORGAN(G); FOR G $E CHIP.GATES;WITH G.TYPE=NOR;
    FINISH;
    END
ENDDEFN

DEFINE DE_MORGAN_COST(G:GATE)=INT:
    BEGIN   VAR S=SIGNAL_WIRE;N=GATE;
    IF G.TYPE=NAND ! G.TYPE=NOR THEN
        IF NEVER N.TYPE=INVERT FOR N $E G.OUTPUT.TO; THEN 1
        EF -(DEFINED(G.OUTPUT.TO[2-])!G.OUTPUT.OUTPUT) THEN -1
        ELSE 0 FI +
        + IF DEFINED(S.VINVERT) THEN 0
          EF S.FROM.TYPE=INVERT THEN
              IF DEFINED(S.TO[2-])!S.OUTPUT THEN 0 ELSE -1 FI
          EF THERE_IS N.TYPE=INVERT FOR N $E S.TO; THEN 0 ELSE 1 FI
        FOR S $E G.INPUTS;
    ELSE 999999 FI
    END
ENDDEFN

DEFINE DE_MORGAN:
    BEGIN   VAR G=GATE;
    CHIP.GATES:=REVERSE(CHIP.GATES);
    FOR G $E CHIP.GATES;WITH DE_MORGAN_COST(G)<0; DO DE_MORGAN(G); END
    FINISH;
    END
ENDDEFN

DEFINE UNIQUE_DESTINATION(S:SIGNAL_WIRE)=BOOL:
    IF S.OUTPUT THEN FALSE ELSE -DEFINED(S.TO[2-]) FI
ENDDEFN

DEFINE UNIQUE_INPUTS:
    BEGIN   VAR G=GATE;S1,S2=SIGNAL_WIRE;I=INT;SWS=SIGNAL_WIRES;
    FOR G $E CHIP.GATES; DO
        SWS:=NIL;
        FOR S1 $E G.INPUTS;&& FOR I FROM 2 BY 1; DO
            IF THERE_IS S2\EQ S1 FOR S2 $E G.INPUTS[I-]; &
               NEVER S1\EQ S2 FOR S2 $E SWS; THEN SWS::= S1 <$; FI
        END
```

```
        FOR S1 $E SWS; DO
            UNLINK_INPUT(G,S1);
            LINK_INPUT(G,S1);
        END
    END
    END
ENDDEFN

DEFINE MERGE:
    BEGIN   VAR LIST=GATES;G,H,I=GATE;S,T,U=SIGNAL_WIRE;
    LIST:=NIL;
    (FOR G $E CHIP.GATES;WITH DEFINED(G.OUTPUT);WITH G.TYPE=NAND!G.TYPE=NOR;)!!
    (FOR S $E G.INPUTS;WITH (I:=S.FROM;).TYPE=INVERT;
        WITH S\UNIQUE_DESTINATION;
        WITH (H:=(T:=S.FROM.INPUTS[1];).FROM;).TYPE=G.TYPE;
        WITH T\UNIQUE_DESTINATION;) DO
            FOR U $E H.INPUTS; DO
                UNLINK_INPUT(H,U);
                LINK_INPUT(G,U);
            END
            UNLINK_OUTPUT(H,T);
            UNLINK_INPUT(I,T);
            UNLINK_OUTPUT(I,S);
            UNLINK_INPUT(G,S);
            ELIMINATE(T);
            ELIMINATE(S);
            LIST::= {H;I} $$;
    END
    DO ELIMINATE(G); FOR G $E LIST;
    END
ENDDEFN
```

The ANNOTATE function is used to label plots. All of the input and output signals of the chip have their names drawn on the plot. This function has the technology as a parameter, so that any technology's layout can be annotated.

```
DEFINE ANNOTATE(C:CHIP  T:TECHNOLOGY)=MRG:
    BEGIN   VAR M=MRG;LENGTH=INT;S=SIGNAL_WIRE;X=REAL;SCALE=POINT;
    DO  M:=COMPILE(C,T);
        LENGTH:= MAX LENGTH(S.NAME) FOR S $E C\INPUTS;;
        X:=CWIDTH-30*LENGTH/7.-4;
        SCALE:=.8*(<*T.WIRE_HEIGHT*>(1)-<*T.WIRE_HEIGHT*>(2))/14.*(1#1);
    GIVE {M;
          COLLECT S.NAME\SCALED_BY SCALE\AT X#S.PHYSICAL.HEIGHT-2.5
                    \PAINTED VIOLET  FOR S $E C\INPUTS;;
          COLLECT S.NAME\SCALED_BY SCALE\AT 6#S.PHYSICAL.HEIGHT-2.5
                    \PAINTED VIOLET  FOR S $E C\OUTPUTS;}
    END
ENDDEFN
```

To allow the use of macro definitions, RLC allows the user to expand a previously declared CHIP into the current chip. The user specifies the set of interconnections via a set of SIGNAL_VALUEs, each of which state which signal of the current CHIP connects to the ports of the expanding CHIP. The inputs of the expanding CHIP may be tied to TRUE or FALSE signals. The FIXED_HIGH and FIXED_LOW routines

eliminate these fixed value signals, and in doing so may eliminate the gates they connect to. The EXPAND function takes a CHIP_INSTANCE, which states which CHIP to expand and how to interconnect the signals, and adds the equations to the current chip.

```
TYPE     POSSIBLE_SIGNAL= EITHER
             FIXED=  BOOL
             VAR=    SIGNAL_WIRE
                          ENDOR;

         SIGNAL_VALUE= [NAME:QS  FROM:POSSIBLE_SIGNAL];

         SIGNAL_VALUES= { SIGNAL_VALUE };

         CHIP_INSTANCE= [CHIP:CHIP  NAME:QS  VALUES:SIGNAL_VALUES];

DEFINE FIXED_HIGH(S:SIGNAL_WIRE):
    BEGIN   VAR G=GATE;T=SIGNAL_WIRE;J=INT;
            DEFINE ZAP(G:GATE):
                T:=G.OUTPUT;
                UNLINK_OUTPUT(G,T);
                ELIMINATE(G);
                FIXED_LOW(T);
            ENDDEFN
    FOR G $E S.TO; DO
        UNLINK_INPUT(G,S);
        CASE G.TYPE OF
        INVERT: ZAP(G);
        NAND:   J:=+1 FOR T $E G.INPUTS;;
                IF J=0 THEN ZAP(G);
                EF J=1 THEN @(G).TYPE:=INVERT; FI
        NOR:    DO UNLINK_INPUT(G,T); FOR T $E G.INPUTS;
                ZAP(G);
        ENDCASE
    END
    ELIMINATE(S);
    END
ENDDEFN

DEFINE FIXED_LOW(S:SIGNAL_WIRE):
    BEGIN   VAR G=GATE;T=SIGNAL_WIRE;J=INT;
            DEFINE ZAP(G:GATE):
                T:=G.OUTPUT;
                UNLINK_OUTPUT(G,T);
                ELIMINATE(G);
                FIXED_HIGH(T);
            ENDDEFN
    FOR G $E S.TO; DO
        UNLINK_INPUT(G,S);
        CASE G.TYPE OF
        INVERT: ZAP(G);
        NOR:    J:=+1 FOR T $E G.INPUTS;;
                IF J=0 THEN ZAP(G);
                EF J=1 THEN @(G).TYPE:=INVERT; FI
        NAND:   DO UNLINK_INPUT(G,T); FOR T $E G.INPUTS;
                ZAP(G);
```

```
        ENDCASE
    END
    ELIMINATE(S);
    END
ENDDEFN

DEFINE COPY(C:CHIP  N:QS)=CHIP:
    BEGIN   VAR CHIP=CHIP;G,H=GATE;S,T=SIGNAL_WIRE;I=INT;
    DO  DO @(G).INDEX:=I; FOR G $E C.GATES;&& FOR I FROM 1 BY 1;
        DO @(S).VHEIGHT:=I; FOR S $E C.SIGNALS;&& FOR I FROM 1 BY 1;
        CHIP:=[GATES:[COLLECT [TYPE:G.TYPE] FOR G $E C.GATES;]
               SIGNAL_COUNT: C.SIGNAL_COUNT
               SIGNALS:[COLLECT [NAME: N$$S.NAME] FOR S $E C.SIGNALS;]
               NAME: C.NAME
               DESCRIPTION: C.DESCRIPTION];
        FOR G $E CHIP.GATES;&& FOR H $E C.GATES; DO
            IF DEFINED(H.OUTPUT) THEN
                LINK_OUTPUT(G,CHIP.SIGNALS[H.OUTPUT.VHEIGHT]); FI
            DO LINK_INPUT(G,CHIP.SIGNALS[S.VHEIGHT]); FOR S $E H.INPUTS;
        END
        DO IF DEFINED(S.VINVERT)
           THEN @(T).VINVERT:=CHIP.SIGNALS[S.VINVERT.VHEIGHT]; FI
        FOR S $E C.SIGNALS;&& FOR T $E CHIP.SIGNALS;
    GIVE CHIP
    END
ENDDEFN

DEFINE EXPAND(C:CHIP_INSTANCE):
    BEGIN   VAR SV=SIGNAL_VALUE;HIGH,LOW=SIGNAL_WIRES;Q=CHIP;
                S=SIGNAL_WIRE;PS=POSSIBLE_SIGNAL;N=QS;
    HIGH:=NIL;
    LOW:=NIL;
    Q:=C.CHIP\COPY C.NAME;
    CHIP.GATES:=REFRESH(CHIP.GATES $$ Q.GATES);
    CHIP.SIGNALS:=REFRESH(CHIP.SIGNALS $$ Q.SIGNALS);
    FOR SV $E C.VALUES; DO
        N:=C.NAME $$ SV.NAME;
        S:=IF THERE_IS S.NAME=N FOR S $E Q.SIGNALS; THEN S ELSE NIL FI;
        PS:=SV.FROM;
        CASE PS OF
        VAR:    FUSE(PS,S);
        FIXED:  IF PS THEN HIGH ELSE LOW FI ::= S <$;
        ENDCASE
    END
    FOR S $E HIGH; DO FIXED_HIGH(S); END
    FOR S $E LOW; DO FIXED_LOW(S); END
    END
ENDDEFN
```

When the chip expanders and chip optimizers have been used upon a chip, the logic equations of the chip are changed, although the function of the chip has remained constant. To allow the user to see what the new logic equations are, the UNPARSE function is used. This function displays the logic of the chip in the same format as the parser reads chip definitions.

```
DEFINE LOCAL(S:SIGNAL_WIRE)=BOOL:
    -(S.INPUT!S.OUTPUT!UNIQUE_DESTINATION(S))
```

```
ENDDEFN

DEFINE UNPARSE(C:CHIP):
    BEGIN   VAR SW=SIGNAL_WIRE;B=BOOL;
        CRLF;
        WRITE('DEFINE '$$CHIP.NAME$$'(');
        IF THERE_IS SW.INPUT FOR SW $E C.SIGNALS; THEN
            WRITE('INPUTS:');
            FOR SW $E C.SIGNALS;WITH SW.INPUT; OTHER_DO WRITE(',');;
            DO WRITE(SW.NAME);
            END
            B:=TRUE;
        ELSE B:=FALSE; FI
        IF THERE_IS SW.OUTPUT FOR SW $E C.SIGNALS; THEN
            IF B THEN WRITE('  '); FI
            WRITE('OUTPUTS:');
            FOR SW $E C.SIGNALS;WITH SW.OUTPUT; OTHER_DO WRITE(',');;
            DO WRITE(SW.NAME);
            END
            B:=TRUE;  FI
        IF THERE_IS SW\LOCAL FOR SW $E C.SIGNALS; THEN
            IF B THEN WRITE('  '); FI
            WRITE('LOCALS:');
            FOR SW $E C.SIGNALS;WITH SW\LOCAL; OTHER_DO WRITE(',');;
            DO WRITE(SW.NAME);
            END FI
        WRITE('):');CRLF;
        FOR SW $E C.SIGNALS; WITH SW.OUTPUT ! SW\LOCAL; DO UNPARSE(SW); END
        WRITE('ENDDEFN');CRLF;
    END
ENDDEFN

DEFINE UNPARSE(SW:SIGNAL_WIRE):
    WRITE('    '$$SW.NAME$$' = ');
    UNPARSE(SW,TRUE);
    CRLF;
ENDDEFN

DEFINE UNPARSE(SW:SIGNAL_WIRE  B:BOOL):
    BEGIN   VAR S=SIGNAL_WIRE;G=GATE;
        IF -B & (SW.INPUT!LOCAL(SW)!SW.OUTPUT) THEN WRITE(SW.NAME);
        ELSE    G:=SW.FROM;
                CASE G.TYPE OF
                INVERT: WRITE('-');UNPARSE(G.INPUTS[1],FALSE);
                NAND:   IF -B THEN WRITE('('); FI
                        FOR S $E G.INPUTS;OTHER_DO WRITE(' & ');;
                        DO UNPARSE(S,FALSE);
                        END
                        IF -B THEN WRITE(')'); FI
                NOR:    IF -B THEN WRITE('('); FI
                        FOR S $E G.INPUTS;OTHER_DO WRITE(' ! ');;
                        DO UNPARSE(S,FALSE);
                        END
                        IF -B THEN WRITE(')'); FI
                ENDCASE FI
    END
ENDDEFN
```

```
DEFINE UNPARSE: UNPARSE(CHIP);    ENDDEFN
```

In conjunction with the unparsing of logic equations, the user might like a quick summary of the size of the chip. This allows the user to judge the usefulness of various optimizations which can be applied to the chip. The STATS function will list the area of the chip, the number of gates, and the number of wiring channels, as a function of the technology and the current chip.

```
DEFINE STATS(T:TECHNOLOGY):
    BEGIN   VAR G=GATE;S=SIGNAL_WIRE;
    CRLF;
    WRITE('Technology:');
    WRITE(T.NAME);
    TAB;
    WRITE('Size: ');
    WRITE(COMPILE(CHIP,T)\MBB);
    CRLF;
    WRITE('Number of gates: ');
    WRITE(+1 FOR G $E CHIP.GATES;);
    TAB;
    WRITE('Number of channels:');
    WRITE(MAX S.VHEIGHT FOR S $E CHIP.SIGNALS;);
    CRLF;
    END
ENDDEFN
```

As mentioned above in the technology definition, we have routines to pack the interconnection wires. The packing routines attempt to have wires share channels, so that the number of channels (and the size of the chip) is minimized. There are two packers presented here. The first, NMOS_PACK_1, does not 'know' about the internals of a cell. It assumes that every wire which connects to a cell consumes the channel for the entire width of the cell. This packer is more general for new technologies. The second packer, NMOS_PACK_2, knows enough about the internals of the cells to allow the output wire to share a channel with one of the input wires, under certain circumstances. Since this packer knows about the implementation of cells, it is not as general as the first packer, but it does a better job of packing the wires for the currently defined technologies.

```
DEFINE SORT(SWS:SIGNAL_WIRES)=SIGNAL_WIRES:
    BEGIN   VAR OUT=SIGNAL_WIRES;W=SIGNAL_WIRE;I,J,K=INT;
    DO  OUT:=NIL;
        WHILE DEFINED(SWS); DO
            I:=-1;
            FOR W $E SWS;&& FOR J FROM 1 BY 1; DO
                IF W.VLEFT>I THEN
                    I:=W.VLEFT;
                    K:=J; FI
            END
            OUT::= SWS[K] <$;
```

```
            SWS[K]:=NIL;
        END
    GIVE OUT
    END
ENDDEFN

DEFINE NMOS_PACK_1(C:CHIP):
    BEGIN   VAR SWS=SIGNAL_WIRES;H=INT;G=GATE;S=SIGNAL_WIRE;
            DEFINE DRAW_WIRE(LEFT:INT):
                BEGIN   VAR W=SIGNAL_WIRE;I=INT;
                IF THERE_IS W.VLEFT>LEFT FOR W $E SWS;&& FOR I FROM 1 BY 1;
                THEN    SWS[I]:=NIL;
                        @(W).VHEIGHT:=H;
                        DRAW_WIRE(W.VRIGHT); FI
                END
            ENDDEFN
    FOR G $E C.GATES;&& FOR H FROM 1 BY 1;DO @(G).INDEX:=H; END
    FOR S $E C.SIGNALS; DO
        @(S).VLEFT:= IF S.INPUT THEN 0
                     EF DEFINED(S.TO)
                     THEN S.FROM.INDEX MIN MIN G.INDEX FOR G $E S.TO;
                     ELSE S.FROM.INDEX FI;
        @(S).VRIGHT:= IF S.OUTPUT THEN 999999
                      ELSE S.FROM.INDEX MAX MAX G.INDEX FOR G $E S.TO; FI;
    END
    SWS:=C.SIGNALS\SORT;
    WHILE DEFINED(SWS);&& FOR H FROM 1 BY 1; DO DRAW_WIRE(-1); END
    END
ENDDEFN

DEFINE NMOS_PACK_2(C:CHIP):
    BEGIN   VAR SWS=SIGNAL_WIRES;H=INT;G=GATE;S=SIGNAL_WIRE;
            DEFINE DRAW_WIRE(LEFT:INT):
                BEGIN   VAR W=SIGNAL_WIRE;I=INT;
                IF THERE_IS W.VLEFT>LEFT FOR W $E SWS;&& FOR I FROM 1 BY 1;
                THEN    SWS[I]:=NIL;
                        @(W).VHEIGHT:=H;
                        DRAW_WIRE(W.VRIGHT); FI
                END
            ENDDEFN
    FOR G $E C.GATES;&& FOR H FROM 1 BY 2;DO @(G).INDEX:=H; END
    FOR S $E C.SIGNALS; DO
        @(S).VLEFT:= IF S.INPUT THEN 0
                     EF DEFINED(S.TO)
                     THEN S.FROM.INDEX+1 MIN MIN G.INDEX FOR G $E S.TO;
                     ELSE S.FROM.INDEX+1 FI;
        @(S).VRIGHT:= IF S.OUTPUT THEN 999999
                      ELSE S.FROM.INDEX+1 MAX MAX G.INDEX FOR G $E S.TO; FI;
    END
    SWS:=C.SIGNALS\SORT;
    WHILE DEFINED(SWS);&& FOR H FROM 1 BY 1; DO DRAW_WIRE(-1); END
    END
ENDDEFN
```

In addition to the packers, we have sorters. The sorters may reorder the gates in attempts to minimize wire lengths or minimize the number of wiring channels. The first 'sorter', NO_SORT, does nothing. The SMALL_SORT routine rebuilds the chip from left to right, each time adding the gate which will add the fewest wiring

channels. This is a local optimization, which means that it will not necessarily (and, in fact, rarely) produce the smallest chip. The RELAXATION_SORT is an iterative routine. Each time it is executed, it 'averages' each gates position. For each gate, the routines averages the indexes of the gates input and output gates. It then sorts the gates by these averages. Presumably, if this routine is executed a few times, gates will tend to be near the gates they connect to.

```
DEFINE NO_SORT(C:CHIP): NOTHING;     ENDDEFN

DEFINE SMALL_SORT(C:CHIP):
    BEGIN   DEFINE ACTIVES(SWS:SIGNAL_WIRES  GS:GATES)=SIGNAL_WIRES:
                BEGIN   VAR G=GATE;S,T=SIGNAL_WIRE;
                {COLLECT S FOR S $E SWS;WITH
                    IF S.OUTPUT THEN TRUE
                    ELSE  THERE_IS (G.OUTPUT\EQ S !
                                    THERE_IS T\EQ S FOR T $E G.INPUTS;)
                        FOR G $E GS; FI;}
                END
            ENDDEFN
            DEFINE UNIQUE(S1,S2:SIGNAL_WIRES)=SIGNAL_WIRES:
                BEGIN   VAR A,B=SIGNAL_WIRE;
                DO  FOR A $E S1; DO
                        IF NEVER A\EQ B FOR B $E S2; THEN S2::= A <$; FI
                    END
                GIVE S2
                END
            ENDDEFN
            VAR ACTIVE,L1,L2=SIGNAL_WIRES;OLD,NEW=GATES;S1,S2=SIGNAL_WIRE;
                G,G1=GATE;I,J,K,L=INT;
    OLD:=C.GATES;
    NEW:=NIL;
    ACTIVE:={COLLECT S1 FOR S1 $E CHIP.SIGNALS; WITH S1.INPUT;};
    WHILE DEFINED(OLD); DO
        I:=999999;
        FOR G $E OLD;&& FOR J FROM 1 BY 1; DO
            L2:=ACTIVES(UNIQUE(G.OUTPUT<$G.INPUTS,ACTIVE),
                        {COLLECT G1  (FOR G1 $E OLD;&& FOR K FROM 1 BY 1;)
                                    WITH K<>J;});
            K:=+1 FOR S1 $E L2;;
            IF K<I THEN
                I:=K;
                L:=J;
                L1:=L2; FI
        END
        NEW::= OLD[L] <$;
        OLD[L]:=NIL;
        ACTIVE:=L1;
    END
    @(C).GATES:=REVERSE(NEW);
    END
ENDDEFN

DEFINE RELAXATION_SORT(C:CHIP):
    BEGIN   VAR OLD,NEW=GATES;G,H=GATE;S=SIGNAL_WIRE;I,N=INT;R=REAL;
    OLD:=C.GATES;
```

```
    N:= +1 FOR G $E OLD;+1;
    FOR G $E OLD;&& FOR I FROM 1 BY 1; DO @(G).INDEX:=I; END
    FOR G $E OLD;DO @(G).RINDEX:=
        (+ IF S.INPUT THEN 0 ELSE S.FROM.INDEX FI FOR S $E G.INPUTS; +
         IF G.OUTPUT.OUTPUT THEN N ELSE 0 FI +
         + H.INDEX FOR H $E G.OUTPUT.TO;)/
        (+1 FOR S $E G.INPUTS; + +1 FOR H $E G.OUTPUT.TO; +
         IF G.OUTPUT.OUTPUT THEN 1 ELSE 0 FI);  END
    NEW:=NIL;
    WHILE DEFINED(OLD); DO
        R:=-1.;
        (FOR G $E OLD;&& FOR I FROM 1 BY 1;) WITH G.RINDEX>R; DO
            R:=G.RINDEX;
            N:=I;
            H:=G;
        END
        NEW::= H <$;
        OLD[N]:=NIL;
    END
    @(C).GATES:=NEW;
    END
ENDDEFN
```

Next, we have the parser. The parser accepts a series of function definitions and generates a CHIP for each function. The following input is an example of the parser's input.

```
DEFINE DFLOP(INPUTS:DATA,CLOCK,RESET,SET OUTPUTS:OUT,BAR LOCALS:X1,X2,X3):
    X3 = X2 & RESET & DATA
    X2 = X1 & CLOCK & X3
    X1 = RESET & CLOCK & ( X3 & SET & X1 )
    BAR = OUT & X2 & RESET
    OUT = BAR & X1 & SET
    OUT~BAR
ENDDEFN

DEFINE EQ(INPUTS:A,B,CIN  OUTPUTS:COUT):
    COUT = (A & -B)*(-A & B)*CIN
ENDDEFN

DEFINE GE(INPUTS:A,B,CIN  OUTPUTS:COUT):
    COUT= (-A&B)*CIN
ENDDEFN

DEFINE COUNTER(INPUTS:RESET,EI,CLOCK  OUTPUTS:OUT,BAR,EO):
    <DFLOP(DATA:BAR  SET:.TRUE.
           RESET:RESET CLOCK:(-CLOCK!-EI) OUT:OUT BAR:BAR)>
    EO=-EI!BAR
ENDDEFN

DEFINE ONE_BIT(INPUTS:SYNC,SHIFT,DATA,LOAD_ADR,LOAD_VAL,ONI,EQI,CNTI
               OUTPUTS:VALUE,ONO,EQO,CNTO
               LOCALS:VAL,ADR,COUNT):
    <DFLOP(DATA:DATA SET:.TRUE. RESET:.TRUE. CLOCK:SHIFT OUT:VALUE)>
    <DFLOP(DATA:VALUE SET:.TRUE. RESET:.TRUE.
           CLOCK:(-LOAD_VAL!-SHIFT) OUT:VAL)>
    <DFLOP(DATA:VALUE SET:.TRUE. RESET:.TRUE.
           CLOCK:(-LOAD_ADR!-SHIFT) OUT:ADR)>
```

```
    <COUNTER(RESET:SYNC EI:CNTI CLOCK:SHIFT OUT:COUNT EO:CNTO)>
    <GE(A:COUNT B:VAL CIN:ONI COUT:ONO)>
    <EQ(A:ADR B:VALUE CIN:EQI COUT:EQO)>
ENDDEFN
```

This input will produce five CHIPs in the virtual memory. The final two CHIPs have expansions of the previously defined CHIPs. This parser will accept characters from a character string, a data file, or from the terminal. There is a file INCLUDE feature which uses the ICL metalanguage syntax: /*READ file;*/.

```
DEFINE PRODUCER(SC:SC)=CHAR_PRODUCER:
    //{SC;} IF DEFINED(SC) THEN GIVING SC[1] DO SC:=SC[2-]; END
          ELSE THE_CHAR(0) FI \\
ENDDEFN

DEFINE FILE_PRODUCER(FILE:FILE_SC)=CHAR_PRODUCER:
    BEGIN   VAR F=IN_CHAR_FILE;
    //[F:=FILE\OPEN;]
        BEGIN   VAR C=CHAR;
        DO  C:=F\INPUT;
            IF EOF(F) THEN CLOSE(F);C:=THE_CHAR(0); FI
        GIVE C
        END\\
    END
ENDDEFN

VAR NESTING_LEVELS=SQS;

    PRODUCER= CHAR_PRODUCER;

    PRODUCERS= { CHAR_PRODUCER };

    PUSHED_SC=SC;

    TOKEN=QS;

    INS,OUTS,LOCALS=SQS;

    CALL_NUMBER=INT;


DEFINE PARSE_SC(SC:SC):
    HOLDING NESTING_LEVELS:=NIL;
    DO  ALSO_PARSE(SC\PRODUCER);
    ENDHOLD
ENDDEFN

DEFINE PARSE_FILE(SC:SC):
    HOLDING NESTING_LEVELS:=NIL;
    DO  ALSO_PARSE(SC\FILE_PRODUCER);
    ENDHOLD
ENDDEFN

DEFINE ALSO_PARSE(CP:CHAR_PRODUCER):
    HOLDING PRODUCER:=CP;
    DO  WHILE DO VERIFY({'DEFINE';{THE_CHAR(26)}},'Definition');
              GIVE TOKEN='DEFINE';
        DO  GET_DEFINITION; END
```

```
    ENDHOLD
ENDDEFN

DEFINE GET_A_CHAR1=CHAR:
    BEGIN   VAR C=CHAR;
    IF DEFINED(PUSHED_SC) THEN GIVING PUSHED_SC[1]
                               DO PUSHED_SC:=PUSHED_SC[2-]; END
    EF -DEFINED(PRODUCER) THEN THE_CHAR(26)
    ELSE    DO  C:=<*PRODUCER*>\UPPERCASE;
                IF C=THE_CHAR(0) THEN
                    PRODUCER:=PRODUCERS[1];
                    PRODUCERS:=PRODUCERS[2-];
                    C:=GET_A_CHAR1; FI
            GIVE C  FI
    END
ENDDEFN

DEFINE GET_A_CHAR2=CHAR:
    BEGIN   VAR C=CHAR;
    DO  C:=GET_A_CHAR1;
        IF C='"' THEN
            WHILE GET_A_CHAR1<>'"'; DO NOTHING; END
            C:=GET_A_CHAR1; FI
    GIVE C
    END
ENDDEFN

DEFINE GET_A_CHAR=CHAR:
    BEGIN   VAR C=CHAR;
    DO  C:=GET_A_CHAR2;
        IF C='/' THEN
            C:=GET_A_CHAR2;
            IF C='*' THEN C:=METALANGUAGE;
            ELSE PUSHED_SC:=(C);
                 C:='/'; FI FI
    GIVE C
    END
ENDDEFN

DEFINE IS_BLANK(C:CHAR)=BOOL:   C\IN_SET (SPACE;TAB;CR;LF)  ENDDEFN

DEFINE IS_ID_CHAR(C:CHAR)=BOOL: LETTER(C)!DIGIT(C)!(C='_')  ENDDEFN

DEFINE GET_TOKEN=QS:
    BEGIN   VAR C=CHAR;SC=SC;
    DO  WHILE (C:=GET_A_CHAR;)\IS_BLANK; DO NOTHING; END
        IF C\IS_ID_CHAR THEN
            SC:=(C);
            WHILE (C:=GET_A_CHAR;)\IS_ID_CHAR; DO SC::= C <$; END
            PUSHED_SC::= C <$;
            SC:=REVERSE(SC);
        ELSE SC:=(C); FI
    GIVE (TOKEN:=SC;)
    END
ENDDEFN

DEFINE ERROR(A:QS):
    CRLF;
    WRITE('ERROR: Expected '$$A$$', got '$$TOKEN);
    CRLF;
    HELP;
```

```
ENDDEFN

DEFINE VERIFY(SQS:SQS  B:QS):
    BEGIN   VAR QS,TOKEN=QS;
    TOKEN:=GET_TOKEN;
    IF NEVER QS=TOKEN FOR QS $E SQS; THEN ERROR(B); FI
    END
ENDDEFN

DEFINE VERIFY(Q:QS):    VERIFY({Q},Q);  ENDDEFN

DEFINE CHECK_TOKEN(SQS:SQS)=BOOL:
    BEGIN   VAR QS,TOKEN=QS;
    DO  TOKEN:=GET_TOKEN;
    GIVE    IF THERE_IS QS=TOKEN FOR QS $E SQS; THEN TRUE
            ELSE DO PUSHED_SC::= TOKEN $$; GIVE FALSE FI
    END
ENDDEFN

DEFINE CHECK_TOKEN(Q:QS)=BOOL:  CHECK_TOKEN({Q})    ENDDEFN

DEFINE METALANGUAGE=CHAR:
    BEGIN   VAR SC=SC;C=CHAR;
            DEFINE FILE_DOES_NOT_EXIST(SC:SC)=SC:
                DO  CRLF;
                    WRITE('File '$$SC$$
                          ' does not exist.  Reset SC to new name.');
                    CRLF;
                    HELP;
                GIVE SC
            ENDDEFN
            DEFINE METAHELP:
                CRLF;
                WRITE('Error in metalanguage termination.');
                CRLF;
                HELP;
            ENDDEFN
    DO  VERIFY('READ');
        SC:=GET_TOKEN;
        IF CHECK_TOKEN('.') THEN SC::= $$ '.'<$GET_TOKEN;
        ELSE SC::= $$ '.RLC'; FI
        IF GET_A_CHAR2<>';' THEN METAHELP; FI
        IF GET_A_CHAR2<>'*' THEN METAHELP; FI
        IF GET_A_CHAR2<>'/' THEN METAHELP; FI
        WHILE IF DEFINED(SC) THEN -EXISTS(FILE_SC::SC) ELSE FALSE FI;
        DO SC::=\FILE_DOES_NOT_EXIST; END
        IF DEFINED(SC) THEN
            PRODUCERS::= PRODUCER <$;
            PRODUCER:=SC\FILE_PRODUCER; FI
    GIVE GET_A_CHAR
    END
ENDDEFN

DEFINE GET_DEFINITION:
    BEGIN   VAR NAME,NEST=QS;
    HOLDING INS:=NIL;
            OUTS:=NIL;
            LOCALS:=NIL;
            CHIP:=NIL;
            CALL_NUMBER:=0;
    DO  NAME:=GET_TOKEN;
```

```
        NAME($$ NEST FOR NEST $E REVERSE(NESTING_LEVELS); $$ NAME);
        NESTING_LEVELS::= (NAME$>'_') <$;
        GET_HEADER;
        WHILE GET_TOKEN<>'ENDDEFN'; DO
            IF TOKEN='DEFINE' THEN GET_DEFINITION;
            EF TOKEN={THE_CHAR(26)} THEN
                CRLF;
                WRITE('End of file encountered inside DEFINE');
                CRLF;
                HELP;
            EF TOKEN='<' THEN GET_CALL;
            ELSE GET_EQUATION(TOKEN); FI
        END
        FINISH;
        NESTING_LEVELS:=NESTING_LEVELS[2-];
        PUT(CHIP);
        CRLF;
        WRITE('DEFINED:'$$CHIP.NAME);
        CRLF;
    ENDHOLD
    END
ENDDEFN

DEFINE GET_HEADER:
    BEGIN   VAR GROUP,SIG=QS;SQS=SQS;
    VERIFY('(');
    WHILE DO  VERIFY({'INPUTS';'OUTPUTS';'LOCALS';')'},'Signal type');
          GIVE TOKEN<>')';
    DO  GROUP:=TOKEN;
        VERIFY(':');
        SQS:={COLLECT GET_TOKEN UNTIL -CHECK_TOKEN(',');};
        IF GROUP='INPUTS' THEN INS::= SQS $$;
                               INPUTS(SQS);
        EF GROUP='OUTPUTS' THEN OUTS::= SQS $$;
                                OUTPUTS(SQS);
        ELSE LOCALS::= SQS $$; FI
    END
    VERIFY(':');
    END
ENDDEFN

DEFINE GET_POSSIBLE=POSSIBLE_SIGNAL:
    IF CHECK_TOKEN('.') THEN
        DO  VERIFY({'TRUE';'FALSE'},'.TRUE. or .FALSE.');
        GIVE GIVING TOKEN='TRUE'
             DO VERIFY('.'); END
    ELSE GET_RHS1 FI
ENDDEFN

DEFINE GET_CALL:
    BEGIN   VAR NAME,NEST,SIG=QS;C=CHIP;SV=SIGNAL_VALUES;
                S=SIGNAL_WIRE;I=INT;
    IF CHECK_TOKEN('@') THEN
        NAME:=GET_TOKEN;
    ELSE    NAME:=GET_TOKEN;
            IF DEFINED(NESTING_LEVELS) THEN
                IF THERE_IS
                    $$ NEST FOR NEST $E REVERSE(NESTING_LEVELS[I-]);
                    $$ NAME \VM_EXISTS_AS 'DCHIP  1/2/81'
                  FOR I FROM 1 TO 1++1 FOR NEST $E NESTING_LEVELS;;
                THEN NAME::= $$ NEST
```

```
                                FOR NEST $E REVERSE(NESTING_LEVELS[I-]);$$;
                FI FI FI
    IF NAME\VM1_EXISTS_AS 'DCHIP  1/2/81' THEN
        CALL_NUMBER::=+1;
        C:=GET(NAME);
        SV:=NIL;
        VERIFY('(');
        WHILE (SIG:=GET_TOKEN;)<>')'; DO
            IF THERE_IS S.NAME=SIG FOR S $E C.SIGNALS;WITH S.INPUT!S.OUTPUT;
            THEN    VERIFY(':');
                    SV::= [NAME:SIG  FROM:GET_POSSIBLE] <$;
            ELSE    CRLF;
                    WRITE('Chip '$$NAME$$' does not have a port named '$$SIG);
                    CRLF;
                    HELP; FI
        END
        VERIFY('>');
        EXPAND([CHIP:C  NAME:'..'$$SC(CALL_NUMBER)  VALUES:SV]);
    ELSE CRLF;
         WRITE('There is no CHIP named '$$NAME);
         CRLF;
         HELP; FI
    END
ENDDEFN

DEFINE GET_EQUATION(QS:QS):
    BEGIN   VAR Q=QS;
    IF THERE_IS Q=QS FOR Q $E OUTS$$LOCALS; THEN
        VERIFY({'=';'~'},'Equation');
        IF TOKEN='=' THEN FUSE(QS,GET_RHS1); ELSE VINVERT(QS,GET_TOKEN); FI
    ELSE CRLF;
         WRITE('There is no local or output named '$$QS);
         CRLF;
         HELP; FI
    END
ENDDEFN

DEFINE GET_RHS1=SIGNAL_WIRE:
    BEGIN   VAR S=SIGNAL_WIRE;
    DO  S:=GET_RHS2;
        WHILE CHECK_TOKEN('XOR'); DO S:=XOR(S,GET_RHS2); END
    GIVE S
    END
ENDDEFN

DEFINE GET_RHS2=SIGNAL_WIRE:
    BEGIN   VAR SWS=SIGNAL_WIRES;
    DO  SWS:={GET_RHS3};
        WHILE CHECK_TOKEN('!'); DO SWS::= GET_RHS3 <$; END
    GIVE IF DEFINED(SWS[2-]) THEN NOR(SWS) ELSE SWS[1] FI
    END
ENDDEFN

DEFINE GET_RHS3=SIGNAL_WIRE:
    BEGIN   VAR SWS=SIGNAL_WIRES;
    DO  SWS:={GET_RHS4};
        WHILE CHECK_TOKEN('+'); DO SWS::= GET_RHS4 <$; END
    GIVE IF DEFINED(SWS[2-]) THEN OR(SWS) ELSE SWS[1] FI
    END
ENDDEFN
```

```
DEFINE GET_RHS4=SIGNAL_WIRE:
    BEGIN   VAR SWS=SIGNAL_WIRES;
    DO  SWS:={GET_RHS5};
        WHILE CHECK_TOKEN('&'); DO SWS::= GET_RHS5 <$; END
    GIVE IF DEFINED(SWS[2-]) THEN NAND(SWS) ELSE SWS[1] FI
    END
ENDDEFN

DEFINE GET_RHS5=SIGNAL_WIRE:
    BEGIN   VAR SWS=SIGNAL_WIRES;
    DO  SWS:={GET_RHS6};
        WHILE CHECK_TOKEN('☆'); DO SWS::= GET_RHS6 <$; END
    GIVE IF DEFINED(SWS[2-]) THEN AND(SWS) ELSE SWS[1] FI
    END
ENDDEFN

DEFINE GET_RHS6=SIGNAL_WIRE:
    IF CHECK_TOKEN('-') THEN INVERT(GET_RHS7) ELSE GET_RHS7 FI
ENDDEFN

DEFINE GET_RHS7=SIGNAL_WIRE:
    BEGIN   VAR Q,X=QS;S,IF=SIGNAL_WIRE;ALL,IFS=SIGNAL_WIRES;
            DEFINE POSSIBLE(SWS:SIGNAL_WIRES):
                BEGIN   VAR P=POSSIBLE_SIGNAL;
                P:=GET_POSSIBLE;
                CASE P OF
                FIXED:  IF P THEN ALL::= NAND(SWS) <$; FI
                VAR:    ALL::= NAND(SWS$>P) <$;
                ENDCASE
                END
            ENDDEFN
    DO  IF CHECK_TOKEN('(') THEN
            S:=GET_RHS1;
            VERIFY(')');
        EF CHECK_TOKEN('IF') THEN
            IF:=GET_RHS1;
            ALL:=NIL;
            VERIFY('THEN');
            POSSIBLE({IF});
            IFS:={GET_INVERT(IF)};
            WHILE DO VERIFY({'EF';'ELSE'},'EF or ELSE');
                  GIVE TOKEN='EF';
            DO  IF:=GET_RHS1;
                VERIFY('THEN');
                POSSIBLE(IFS$>IF);
                IFS::= GET_INVERT(IF) <$;
            END
            POSSIBLE(IFS);
            VERIFY('FI');
            S:=NAND(ALL);
        ELSE    Q:=GET_TOKEN;
                IF THERE_IS X=Q FOR X $E INS$$OUTS$$LOCALS; THEN S:=Q;
                ELSE CRLF;
                     WRITE('There is no signal named '$$Q);
                     CRLF;
                     HELP; FI FI
    GIVE S
    END
ENDDEFN
```

There is a tau-model simulator built into RLC. The MAKE_SIMULATOR function will take the current CHIP and construct a SIM_CHIP, which is the simulator representation of the chip. The user then defines the pulse trains which drive the input wires, using the CLOCK and WAVEFORM functions. Following this, the RUN(time) function is called, which actually runs the simulation from t=0 to t=time. RUN will initialize all of the nodes in the circuit. In some cases, like for cross-coupled circuits, RUN will ask the user whether a node should be initiallized high or low. Once the simulation is complete, the user may plot waveforms of any of the nodes using the PLOT functions. The simulator saves the waveforms of each node so that many plots can be generated from a single simulation run.

```
TYPE    SIM_GATE= [INPUTS:SIM_WIRES
                   OUTPUT:SIM_WIRE
                   TYPE:GATE_TYPE
                   GATE:GATE];

        SIM_GATES= { SIM_GATE };

        SIM_WIRE= [NAME:QS
                   FROM:SIM_GATE
                   TO:SIM_GATES
                   WIRE:SIGNAL_WIRE
                   VALUE,NEW,SET:BOOL
                   TAU:REAL
                   TRACE:SP];

        SIM_WIRES= { SIM_WIRE };

        SIM_CHIP= [WIRES:SIM_WIRES  GATES:SIM_GATES];


VAR SIM_CHIP=SIM_CHIP;

DEFINE MAKE_SIMULATOR:
    BEGIN   VAR G=GATE;I=INT;S,T=SIGNAL_WIRE;Q=SIM_GATE;
    DO @(G).INDEX:=I ; FOR G $E CHIP.GATES;&& FOR I FROM 1 BY 1;
    SIM_CHIP:=[GATES:{COLLECT [TYPE:G.TYPE  GATE:G] FOR G $E CHIP.GATES;}];
    DO @(S).VHEIGHT:=I; FOR S $E CHIP.SIGNALS;&& FOR I FROM 1 BY 1;
    SIM_CHIP.WIRES:={COLLECT
        [NAME:S.NAME
         FROM:SIM_CHIP.GATES[S.FROM.INDEX]
         TO:{COLLECT SIM_CHIP.GATES[G.INDEX] FOR G $E S.TO;}
         WIRE:S
         SET:FALSE
         TAU: + CASE G.TYPE OF
                NOR:    1
                INVERT: 1
                NAND:   +1 FOR T $E G.INPUTS;
                ENDCASE FOR G $E S.TO;]
                     FOR S $E CHIP.SIGNALS;};
    DO @(Q).OUTPUT:=SIM_CHIP.WIRES[Q.GATE.OUTPUT.VHEIGHT];
       @(Q).INPUTS:={COLLECT SIM_CHIP.WIRES[S.VHEIGHT]
```

```
                    FOR S $E Q.GATE.INPUTS;};
    FOR Q $E SIM_CHIP.GATES;
    END
ENDDEFN


TYPE    EVENT= EITHER
            WIRE=   SIM_WIRE
            SS=     SS
               ENDOR;

        EVENTS= { EVENT };

        TIME_SLOT= [TIME:REAL  EVENTS:EVENTS];

        TIME_LINE= { TIME_SLOT };

        GATE_SIMULATION= //(SIM_GATE)\\;


VAR     TIME=REAL;

        TIME_LINE= TIME_LINE;

        NAND_SIMULATION, NOR_SIMULATION,
        INVERT_SIMULATION= GATE_SIMULATION;

        ABORT_SIMULATION= BOOL;


DEFINE CLEAR_SIMULATION:
    BEGIN   VAR S=SIM_WIRE;
    TIME_LINE:=NIL;
    TIME:=0;
    DO @(S).TRACE:=NIL;
       @(S).SET:=FALSE; FOR S $E SIM_CHIP.WIRES;
    END
ENDDEFN

DEFINE SIMULATE(GS:SIM_GATES):
    BEGIN   VAR G=SIM_GATE;
    FOR G $E GS; DO
        CASE G.TYPE OF
        INVERT: <*INVERT_SIMULATION*>(G);
        NOR:    <*NOR_SIMULATION*>(G);
        NAND:   <*NAND_SIMULATION*>(G);
        ENDCASE
    END
    END
ENDDEFN

DEFINE SIMULATE(E:EVENT):
    CASE E OF
    WIRE:   @(E).VALUE:=E.NEW;
            @(E).TRACE::= TIME# IF E.VALUE THEN 1 ELSE 0 FI <$;
            SIMULATE(E.TO);
    SS:     <*E*>;
    ENDCASE
ENDDEFN
```

```
DEFINE SIMULATE(T:TIME_SLOT):
    BEGIN   VAR E=EVENT;
    TIME:=T.TIME;
    WHILE -ABORT_SIMULATION;&& FOR E $E T.EVENTS; DO SIMULATE(E); END
    END
ENDDEFN

DEFINE SIMULATE:
    HOLDING ABORT_SIMULATION:=FALSE;
    DO  WHILE -ABORT_SIMULATION; DO
            SIMULATE(GIVING TIME_LINE[1]
                    DO TIME_LINE:=TIME_LINE[2-]; END);
        END
    ENDHOLD
ENDDEFN

DEFINE EQ(A,B:SIM_WIRE)=BOOL:   MACRO-10('LSPEQ$')

DEFINE EQ(A,B:EVENT)=BOOL:
    CASE A OF
    WIRE:   CASE B OF
            WIRE:   A\EQ B
            ELSE:   FALSE
            ENDCASE
    ELSE:   FALSE
    ENDCASE
ENDDEFN

DEFINE HOLD_UNTIL(E:EVENT  R:REAL):
    BEGIN   VAR TS=TIME_SLOT;I=INT;V=EVENT;
    I:=0;
    IF DEFINED(TIME_LINE) THEN
        FOR TS $E TIME_LINE;WITH TS.TIME-EPSILON=<R;&& FOR I FROM 1 BY 1; DO
            IF TS.TIME\IS_CLOSE_TO R THEN
                IF NEVER E\EQ V FOR V $E TS.EVENTS; THEN
                    @(TS).EVENTS::= E <$; FI
                I:=-1; FI
        END
        IF I>0 THEN TIME_LINE[I+1-]:=[TIME:R EVENTS:{E}]
                                        <$ TIME_LINE[I+1-];
        EF I=0 THEN TIME_LINE::= [TIME:R  EVENTS:{E}] <$; FI
    ELSE TIME_LINE:={[TIME:R  EVENTS:{E}]}; FI
    END
ENDDEFN

DEFINE HOLD(E:EVENT  R:REAL):   HOLD_UNTIL(E,TIME+R);   ENDDEFN


TYPE    GATE_EVALUATOR=//BOOL(SIM_WIRES)\\;

DEFINE GATE_SIMULATOR(G:SIM_GATE  GE:GATE_EVALUATOR):
    BEGIN   VAR R=BOOL;
    R:=<*GE*>(G.INPUTS);
    IF R<>G.OUTPUT.NEW THEN
        @(G.OUTPUT).NEW:=R;
        HOLD(G.OUTPUT,.31*G.OUTPUT.TAU); FI
    END
ENDDEFN

DEFINE NAND(WS:SIM_WIRES)=BOOL:
    BEGIN   VAR S=SIM_WIRE;
```

```
    THERE_IS -S.VALUE FOR S $E WS;
    END
ENDDEFN

DEFINE NOR(WS:SIM_WIRES)=BOOL:
    BEGIN   VAR S=SIM_WIRE;
    NEVER S.VALUE FOR S $E WS;
    END
ENDDEFN

DEFINE INVERT(WS:SIM_WIRES)=BOOL:
    -WS[1].VALUE
ENDDEFN

NAND_SIMULATION:=//:GATE_SIMULATOR(SIM_GATE)[//:NAND(SIM_WIRES)\\]\\;

NOR_SIMULATION:=//:GATE_SIMULATOR(SIM_GATE)[//:NOR(SIM_WIRES)\\]\\;

INVERT_SIMULATION:=//:GATE_SIMULATOR(SIM_GATE)[//:INVERT(SIM_WIRES)\\]\\


DEFINE INITIALIZE(G:SIM_GATE):
    BEGIN   VAR W=SIM_WIRE;Q=SIM_GATE;
            DEFINE INIT(B:BOOL):
                PRESET(G.OUTPUT,B);
                DO INITIALIZE(Q); FOR Q $E G.OUTPUT.TO;
            ENDDEFN
    IF -G.OUTPUT.SET THEN
        CASE G.TYPE OF
        INVERT: IF G.INPUTS[1].SET THEN
                    INIT(-G.INPUTS[1].VALUE); FI
        NAND:   IF THERE_IS W.SET & -W.VALUE FOR W $E G.INPUTS; THEN
                    INIT(TRUE);
                EF ALWAYS W.SET & W.VALUE FOR W $E G.INPUTS; THEN
                    INIT(FALSE); FI
        NOR:    IF THERE_IS W.SET & W.VALUE FOR W $E G.INPUTS; THEN
                    INIT(FALSE);
                EF ALWAYS W.SET & -W.VALUE FOR W $E G.INPUTS; THEN
                    INIT(TRUE); FI
        ENDCASE FI
    END
ENDDEFN

DEFINE INITIALIZE:
    BEGIN   VAR G=SIM_GATE;W=SIM_WIRE;
    DO  INITIALIZE(G); FOR G $E SIM_CHIP.GATES;
    IF THERE_IS -W.SET FOR W $E SIM_CHIP.WIRES; THEN
        CWRITE('/NInitialize node '$$W.NAME$$'.  High(1) or Low(0)?');
        PRESET(W,GET_RESPONSE('10')='1');
        INITIALIZE; FI
    END
ENDDEFN

DEFINE RUN(T:REAL):
    BEGIN   VAR SW=SIM_WIRE;
    TIME:=0;
    HOLD_UNTIL(//ABORT_SIMULATION:=TRUE;\\,T);
    INITIALIZE;
    SIMULATE;
    CRLF;
    WRITE('Simulation terminated at time=');
```

```
    WRITE(TIME);
    CRLF;
    FOR SW $E SIM_CHIP.WIRES; DO
        @(SW).TRACE:=REVERSE(TIME#IF SW.VALUE THEN 1 ELSE 0 FI <$ SW.TRACE);
    END
    END
ENDDEFN

DEFINE PRESET(W:SIM_WIRE  V:BOOL):
    @(W).VALUE:=V;
    @(W).NEW:=V;
    @(W).SET:=TRUE;
    @(W).TRACE:={0#IF V THEN 1 ELSE 0 FI};
ENDDEFN

DEFINE PRESET(N:QS  V:BOOL):
    BEGIN   VAR W=SIM_WIRE;
    IF THERE_IS W.NAME\EQ N FOR W $E SIM_CHIP.WIRES; THEN PRESET(W,V);
    ELSE    CRLF;
            WRITE('There is no wire named ');
            WRITE(N);
            CRLF;
            HELP; FI
    END
ENDDEFN

DEFINE PRESET_HIGH(SQS:SQS):
    BEGIN   VAR QS=QS;
    DO PRESET(QS,TRUE); FOR QS $E SQS;
    END
ENDDEFN

DEFINE PRESET_LOW(SQS:SQS):
    BEGIN   VAR QS=QS;
    DO PRESET(QS,FALSE); FOR QS $E SQS;
    END
ENDDEFN


TYPE    CLOCK= [PHASE,HIGH,LOW:REAL  VALUE:BOOL  WIRE:SIM_WIRE  INPUT:Q

        WAVEFORM= [VALUE:BOOL  DELTAS:SR  WIRE:SIM_WIRE  INPUT:QS];

DEFINE NEXT_CLOCK(C:CLOCK):
    @(C.WIRE).VALUE:=(C.VALUE::=-;);
    @(C.WIRE).TRACE::= TIME#IF C.VALUE THEN 1 ELSE 0 FI <$;
    SIMULATE(C.WIRE.TO);
    HOLD(//:NEXT_CLOCK[C]\\,IF C.VALUE THEN C.HIGH ELSE C.LOW FI);
ENDDEFN

DEFINE CLOCK(C:CLOCK):
    BEGIN   VAR W=SIM_WIRE;
    IF THERE_IS W.NAME\EQ C.INPUT FOR W $E SIM_CHIP.WIRES;WITH W.WIRE.INPUT;
    THEN    PRESET(W,C.VALUE);
            C.WIRE:=W;
            HOLD_UNTIL(//:NEXT_CLOCK[C]\\,C.PHASE);
    ELSE    CRLF;
            WRITE('There is no input named ');
            WRITE(C.INPUT);
            CRLF;
            HELP; FI
```

```
    END
ENDDEFN

DEFINE NEXT_WAVEFORM(W:WAVEFORM):
    @(W.WIRE).VALUE:=(W.VALUE::=-;);
    @(W.WIRE).TRACE::= TIME# IF W.VALUE THEN 1 ELSE 0 FI <$;
    SIMULATE(W.WIRE.TO);
    @(W).DELTAS:=W.DELTAS[2-];
    IF DEFINED(W.DELTAS) THEN
        HOLD_UNTIL(//:NEXT_WAVEFORM[W]\\,W.DELTAS[1]); FI
ENDDEFN

DEFINE WAVEFORM(W:WAVEFORM):
    BEGIN   VAR I=SIM_WIRE;
    IF THERE_IS I.NAME\EQ W.INPUT FOR I $E SIM_CHIP.WIRES;
                                  WITH I.WIRE.INPUT;
    THEN    PRESET(I,W.VALUE);
            W.WIRE:=I;
            HOLD_UNTIL(//:NEXT_WAVEFORM[W]\\,W.DELTAS[1]);
    ELSE    CRLF;
            WRITE('There is no input named ');
            WRITE(W.INPUT);
            CRLF;
            HELP; FI
    END
ENDDEFN


DEFINE PLOT(PATH:SP  NAME:QS  START,SCALEX:REAL)=MRG:
    BEGIN   VAR P,Q=POINT;
     {NAME\PAINTED RED\SCALED_BY .5*(1#1);
      WIRE(BLUE,0,PATH[1].X#PATH[1].Y*7 <$ $$ {Q.X*SCALEX#P.Y*7;.#Q.Y*7}
                                          FOR {P;Q} $C PATH;)\AT START#0}
    END
ENDDEFN

DEFINE PLOT(SQS:SQS  PLT:SIZABLE_COLOR_PLOTTER  SCALEX:REAL):
    BEGIN   VAR QS=QS;X,Y=REAL;SW=SIM_WIRE;
    X:=6* MAX LENGTH(SC::QS) FOR QS $E SQS; + 4;
    PLOT(MRG:: {COLLECT IF THERE_IS SW.NAME\EQ QS FOR SW $E SIM_CHIP.WIRES;
                    THEN PLOT(SW.TRACE,SW.NAME,X,SCALEX)\AT 0#Y
                    ELSE NIL FI   FOR QS $E SQS;&&FOR Y FROM 0 BY -12.;},PLT);
    END
ENDDEFN

DEFINE PLOT(SQS:SQS  PLT:SIZABLE_COLOR_PLOTTER):
    PLOT(SQS,PLT,1);
ENDDEFN
```

Finally, the RLC has a Run Time System (RTS) which interacts with the user. The user types commands to the RTS, which then calls the appropriate routine. We want the user to be able to add new routines (such as sorters or packers) at any time, just as new technologies can be added. This requires the use of suspendable functions. We will name these functions, so users may call them by name. The NAMED_SS datatype holds functions which require no parameters, while NAMED_CHIP_CONSUMERs hold functions which require a CHIP as its single input

parameter. We then define global lists of these functions, and assign the existing routines to the list.

```
TYPE     NAMED_SS= [NAME:QS  FUNCTION:SS];

         NAMED_SSS= { NAMED_SS };

         NAMED_CHIP_CONSUMER= [NAME:QS  CONSUMER:CHIP_CONSUMER];

         NAMED_CHIP_CONSUMERS= { NAMED_CHIP_CONSUMER };

VAR OPTIMIZERS= NAMED_SSS;

    SORTERS,PACKERS= NAMED_CHIP_CONSUMERS;

OPTIMIZERS:=
     { [NAME:'REMOVE_INVERTERS'  FUNCTION://:REMOVE_INVERTERS\\] ;
       [NAME:'REMOVE_REDUNDANCIES'  FUNCTION://:REMOVE_REDUNDANCIES\\] ;
       [NAME:'REMOVE_NANDS'  FUNCTION://:REMOVE_NANDS\\] ;
       [NAME:'REMOVE_NORS'  FUNCTION://:REMOVE_NORS\\] ;
       [NAME:'DE_MORGAN'  FUNCTION://:DE_MORGAN\\] ;
       [NAME:'UNIQUE_INPUTS'  FUNCTION://:UNIQUE_INPUTS\\] ;
       [NAME:'MERGE'  FUNCTION://:MERGE\\] };

PACKERS:=
     { [NAME:'NMOS_PACK_1'  CONSUMER://:NMOS_PACK_1(CHIP)\\] ;
       [NAME:'NMOS_PACK_2'  CONSUMER://:NMOS_PACK_2(CHIP)\\] };

SORTERS:=
     { [NAME:'SMALL_SORT'  CONSUMER://:SMALL_SORT(CHIP)\\] ;
       [NAME:'NO_SORT'  CONSUMER://:NO_SORT(CHIP)\\] ;
       [NAME:'RELAXATION_SORT'  CONSUMER://:RELAXATION_SORT(CHIP)\\] };
```

The following section is the RLC run time system. The user types commands to the RTS, which then calls the appropriate routine. When typing a command, the user need only type enough to make the command unambiguous. Question marks can be typed at any point to list the current options.

```
DEFINE RLC_SYSTEM:
    SYSOUT(SAVE_INDEPENDENT);
    RLC;
    SYSOUT(NO_SAVE);
ENDDEFN

DEFINE RLC:
    BEGIN   VAR GO=BOOL;
    GO:=TRUE;
    WHILE GO; DO
        JRST(MENU('?',{'GET chip';'PUT chip';'READ file';'PARSE input';
                       'SIMULATE';'EDIT logic';'PLOT chip';'FILE plot';
                       'SORT gates';'DIRECTORY';'UNPARSE';'STATS';'QUIT'}))
        0=> CRLF;
```

```
        1=> RTS_GET;
        2=> RTS_PUT;
        3=> RTS_READ;
        4=> RTS_PARSE;
        5=> RTS_SIMULATE;
        6=> RTS_EDIT;
        7=> RTS_PLOT;
        8=> RTS_FILE;
        9=> RTS_SORT;
        10=> CRLF;
             VM_DIR('*','DCHIP  1/2/81');
             CRLF;
        11=> UNPARSE;
        12=> RTS_STATS;
        13=> GO:=FALSE;
        ENDJRST
    END
    END
ENDDEFN

DEFINE RTS_GET:
    BEGIN   VAR V=VM_DIRECTORY_ELEMENT;
    CRLF;
    V:=MENU('CHIP name?','*','DCHIP  1/2/81');
    IF DEFINED(V) THEN CHIP:=GET(V.NAME); FI
    END
ENDDEFN

DEFINE RTS_PUT:
    CRLF;
    WRITE('Putting '$$CHIP.NAME$$' into virtual memory...');
    CRLF;
    PUT(CHIP);
ENDDEFN

DEFINE RTS_READ:
    BEGIN   VAR FILE=SC;
    CRLF;
    FILE:=GET_SC('Enter file name',CR);
    WHILE IF DEFINED(FILE) THEN -EXISTS(FILE) ELSE FALSE FI; DO
        CRLF;
        WRITE('The file '$$FILE$$' does not exist.  ');
        FILE:=GET_SC('Enter new file name',CR);
    END
    IF DEFINED(FILE) THEN PARSE_FILE(FILE); FI
    END
ENDDEFN

DEFINE RTS_PARSE:
    BEGIN   VAR SC=SC;
    CRLF;
    SC:=GET_SC('Enter RLC source:',BELL);
    IF DEFINED(SC) THEN PARSE_SC(SC); FI
    END
ENDDEFN

DEFINE RTS_SIMULATE:CRLF; ENDDEFN

DEFINE RTS_EDIT:
    BEGIN   VAR GO=BOOL;NSS=NAMED_SS;I=INT;
    GO:=TRUE;
```

```
    CRLF;
    WHILE GO; DO
        I:=MENU('EDIT>',{'DONE';COLLECT NSS.NAME FOR NSS $E OPTIMIZERS;});
        IF I<2 THEN GO:=FALSE;
        ELSE    <*OPTIMIZERS[I-1].FUNCTION*>; FI
        CRLF;
    END
    END
ENDDEFN

VAR RLC_MPICTURE=MPICTURE;

DEFINE RTS_STATS:
    BEGIN   VAR T=TECHNOLOGY;I=INT;
    CRLF;
    I:=MENU('Enter Technology:',{COLLECT T.NAME FOR T $E TECHNOLOGIES;});
    CRLF;
    IF I>0 THEN STATS(TECHNOLOGIES[I]); FI
    END
ENDDEFN

DEFINE RTS_PLOT:
    BEGIN   VAR T=TECHNOLOGY;I=INT;
    CRLF;
    I:=MENU('Enter Technology:',{COLLECT T.NAME FOR T $E TECHNOLOGIES;});
    CRLF;
    IF I>0 THEN
        RLC_MPICTURE:=COMPILE(CHIP,TECHNOLOGIES[I]);
        RTS_PLOTTER; FI
    END
ENDDEFN

DEFINE RTS_FILE:
    BEGIN   VAR SC=SC;
    CRLF;
    SC:=GET_SC('Enter AIF file name:',CR);
    CRLF;
    IF DEFINED(SC) THEN
        RLC_MPICTURE:=SC\AIF;
        RTS_PLOTTER; FI
    END
ENDDEFN

DEFINE RTS_PLOTTER:
    BEGIN   VAR I=INT;SC=SC;
        JRST(MENU('Enter Plotter:',{'HP7221A';'HP1302';'HP2649';'SCREEN';
                                    'FILE';'AREA_HP7221A';'AREA_HP2649'}))
        0=> NOTHING;
        1=> PLOT(RLC_MPICTURE,HP_7221A);
        2=> PLOT(RLC_MPICTURE,HP1302);
        3=> NOTHING;
        4=> PLOT(RLC_MPICTURE,SCREEN);
        5=> CRLF;
            SC:=GET_SC('Enter file name:',CR);
            IF DEFINED(SC) THEN PLOT(RLC_MPICTURE,SC\AIF); FI
        6=> PLOT(RLC_MPICTURE,AREA_HP_7221A_NO);
        7=> NOTHING;
        ENDJRST
        CRLF;
    END
ENDDEFN
```

```
DEFINE RTS_SORT:
    BEGIN   VAR I=INT;NCC=NAMED_CHIP_CONSUMER;
    CRLF;
    I:=MENU('Sort routine?',{COLLECT NCC.NAME FOR NCC $E SORTERS;});
    IF I>0 THEN <*SORTERS[I].CONSUMER*>(CHIP); FI
    CRLF;
    END
ENDDEFN
```

Appendix 5: Bristle Blocks Elements

The following elements are available for use in Bristle Blocks. The type of each element is given, followed by the required and optional parameters for element of the given type.

A5.1: Registers

There are four basic styles of registers in Bristle Blocks. The first type is the standard scratchpad register. It may read or write data from the two data buses. Its internal value may refresh, and it may load with a constant. The second type of register acts like the scratchpad register, but its value may be driven into the instruction decoder. The third register type acts like the scratchpad register, but it may also load selected bits from the instruction decoder. The fourth register type is a combination of the second and third types: the register may drive the instruction decoder, and the register may load from the instruction decoder. In the second and fourth types, the LATCH parameter controls the loading of the register, which occurs during PHI 2.

(1) Element: **REGISTER**
Required Parameters:
    Keyword: OPTIONS Type: REGISTER
Optional Parameters: NONE

(2) Element: **DATA TO CONTROL**
Required Parameters:
    Keyword: REGISTER Type: REGISTER
    Keyword: MAP Type: SOURCES
Optional Parameters: NONE

(3) Element: **CONTROL TO DATA**
Required Parameters:
    Keyword: REGISTER Type: REGISTER
    Keyword: MAP Type: DESTS
    Keyword: LATCH Type: EQUATION
Optional Parameters: NONE

(4) Element: **CONTROL TO DATA AND BACK**
Required Parameters:
    Keyword: REGISTER Type: REGISTER
    Keyword: TO_CONTROL Type: SOURCES
    Keyword: TO_DATA Type: DESTS
    Keyword: LATCH Type: EQUATION
Optional Parameters: NONE

A5.2: Simple Arithmetic Elements

There are four simple arithmetic elements in Bristle Blocks: Incrementers, Decrementers, Adders, and Subtracters. The incrementer and decrementer each have an input register and an optional output register. If the output register is specified, the output of the incrementer/decrementer will load the register. If the output register is not specified, the incrementer/decrementer will load the input register. The LOAD equation states when the load should occur. The carry output is available, if desired, to drive the instruction decoder or an output pad.

The adder and subtracter have two input registers and an optional output register. If the output register is specified, the results of the operation are stored in the output register. If the output register is not specified, the result of the operation is stored in the INPUT_A register. For the subtracter, INPUT_B is subtracted from INPUT_A. The LOAD equation again controls when the register is to be loaded. The LATCH equation transfers data from the input registers into internal nodes, and this happens during PHI_1. The user may specify a carry input and may use the carry output. **Notice that these signals are inverted.**

```
(5) Element: INCREMENTER
Required Parameters:
    Keyword: INPUT_REGISTER     Type: REGISTER
    Keyword: LOAD               Type: EQUATION
Optional Parameters:
    Keyword: OUTPUT_REGISTER    Type: REGISTER
    Keyword: PRECHARGE          Type: EQUATION    Default: ALWAYS
    Keyword: CARRY_OUT          Type: OUTPUT

(6) Element: DECREMENTER
Required Parameters:
    Keyword: INPUT_REGISTER     Type: REGISTER
    Keyword: LOAD               Type: EQUATION
Optional Parameters:
    Keyword: OUTPUT_REGISTER    Type: REGISTER
    Keyword: PRECHARGE          Type: EQUATION    Default: ALWAYS
    Keyword: CARRY_OUT          Type: OUTPUT
```

(7) Element: **ADDER**

```
Required Parameters:
    Keyword: INPUT_A              Type: REGISTER
    Keyword: INPUT_B              Type: REGISTER
    Keyword: LOAD                 Type: EQUATION
Optional Parameters:
    Keyword: OUTPUT_REGISTER      Type: REGISTER
    Keyword: PRECHARGE            Type: EQUATION    Default: ALWAYS
    Keyword: CARRY_OUT_BAR        Type: OUTPUT
    Keyword: LATCH                Type: EQUATION    Default: ALWAYS
    Keyword: CARRY_IN_BAR         Type: EQUATION    Default: NEVER
```

(8) Element: **SUBTRACTER**

```
Required Parameters:
    Keyword: INPUT_A              Type: REGISTER
    Keyword: INPUT_B              Type: REGISTER
    Keyword: LOAD                 Type: EQUATION
Optional Parameters:
    Keyword: OUTPUT_REGISTER      Type: REGISTER
    Keyword: PRECHARGE            Type: EQUATION    Default: ALWAYS
    Keyword: CARRY_OUT_BAR        Type: OUTPUT
    Keyword: LATCH                Type: EQUATION    Default: ALWAYS
    Keyword: CARRY_IN_BAR         Type: EQUATION    Default: NEVER
```

A5.3: Arithmetic/Logic Units

There are three versions of ALUs in Bristle Blocks. The differences have to do with the flag logic. In the first case, the flags are valid during the PHI_2 that the ALU is operating, so they may control an operation occurring the next PHI_1. In the second case, these flags may load a flag register, which sits on the buses like any other register. The flag bits from this register may drive the instruction decoder. The third type of ALU has a complex flag unit that allows selectable loading/testing/modifying of any bit in the flag register.

Each of the ALUs has two input registers and either one or two output registers. Equations control when the two output registers are to be loaded from the ALU. In addition, the flags from the ALU are immediately available in the instruction decoder, or to pads. The carry output and carry into the MSB are inverted polarity logic. Overflow is detected by exclusive-oring these two output signals. Additionally, the MSB and the ZERO flag are available.

There are several operations which the ALUs will perform. The basic arithmetic operations are ADD, SUBTRACT, SUBTRACT_REV, NEGATE_A, and NEGATE_B. The subtract operation subtracts INPUT_B from INPUT_A, while subtract reversed does the opposite. Each of these operations assumes there is no carry (or borrow) input. Corresponding to each of these operations is an operation which forces a carry or

borrow on the input. These operations are ADD_W_CARRY, SUB_W_BORROW, SUBR_W_BORROW, NEG_A_W_BORROW, and NEG_B_W_BORROW, respectively. Similarly, the increment/decrement operations are available: INCREMENT_A, INCREMENT_B, DECREMENT_A, and DECREMENT_B. These operations force a carry or borrow input. The operations which assume no carry or borrow input are just SETA, SETB, SETA, and SETB, respectively.

There are operations which set the output of the ALU to a constant value or to one of the input values. These operations are SETZ (or ZERO), SETO (or ONES), SETA, SETB, SETCA, and SETCB. SETA sets the ALU output to be the value in the INPUT_A register, while SETCA sets the output to be the compliment of this value. Additionally, the ALU can do AND and OR operations on either the input data or its compliment. These operations are AND, ANDCA, ANDCB (or TEST), ANDC (or NOR),OR, ORCA, ORCB, and ORC (or NAND). The basic AND and OR functions perform the obvious operation. The -CA suffix indicates that the operation is performed using the compliment of the INPUT_A value, while -CB indicates that the compliment of the INPUT_B value is used. -C indicates that compliments of both input values are used. The exclusive-or operations are also available: XOR and EQV (or XNOR).

The ALU can perform single bit left shift operations: SHIFT_A, SHIFT_B, SHIFT_A_W_LSB, and SHIFT_B_W_LSB. The SHIFT_A and SHIFT_B operations shift a zero into the least significant bit, while the remaining operations shift a one into the LSB.

The remaining operations include MASK operations and Find-First-One (or zero). The MASK_AB and MASK_BA instructions are used to generate masks. With the MASK_AB operation, the ALU output will be high between the least significant high bit in A and the next high bit in B, and between the next high bit in A and the next high bit in B, etc. High bits in A generate carrys while high bits in B kill the carry. The FFO_A instruction produces an output which is low in every bit position except the first low bit in A. This is the Find First Zero in A instruction. Similarly, the FF1_A, FFO_B, and FF1_B instructions exist.

The DONT_CARE instruction is also listed. This operation states that the particular instruction is an undefined opcode, so Bristle Blocks can fill this with any instruction.

(9) Element: **ALU**

Required Parameters:

Keyword: INPUT_A		Type: REGISTER	
Keyword: INPUT_B		Type: REGISTER	
Keyword: OUTPUT_1		Type: REGISTER	
Keyword: DECODE		Type: DECODE	
Operations:			
ADD	ADD_W_CARRY	MASK_AB	MASK_BA
SUBTRACT	SUB_W_BORROW	SUBTRACT_REV	SUBR_W_BORROW
NEGATE_A	NEG_A_W_BORROW	NEGATE_B	NEG_B_W_BORROW
INCREMENT_A	INCREMENT_B	DECREMENT_A	DECREMENT_B
SHIFT_A	SHIFT_A_W_LSB	SHIFT_B	SHIFT_B_W_LSB
FF0_A	FF0_B	FF1_A	FF1_B
SETZ	ANDC	ANDCB	SETCA
ANDCA	SETCB	XOR	ORC
AND	EQV	SETB	ORCA
SETA	ORCB	OR	SETO
ZERO	ONES	NAND	NOR
XNOR	TEST	DONT_CARE	

Optional Parameters:

Keyword: OUTPUT_2	Type: REGISTER	
Keyword: PRECHARGE	Type: EQUATION	Default: ALWAYS
Keyword: CARRY_OUT_BAR	Type: OUTPUT	
Keyword: CARRY_INTO_MSB_BAR	Type: OUTPUT	
Keyword: MSB	Type: OUTPUT	
Keyword: ZERO	Type: OUTPUT	
Keyword: WRITE_OUTPUT_1	Type: EQUATION	Default: ALWAYS
Keyword: WRITE_OUTPUT_2	Type: EQUATION	Default: NEVER

(10) Element: ALU WITH FLAGS

Required Parameters:

Keyword: INPUT_A		Type: REGISTER	
Keyword: INPUT_B		Type: REGISTER	
Keyword: OUTPUT_1		Type: REGISTER	
Keyword: DECODE		Type: DECODE	
Operations:			
ADD	ADD_W_CARRY	MASK_AB	MASK_BA
SUBTRACT	SUB_W_BORROW	SUBTRACT_REV	SUBR_W_BORROW
NEGATE_A	NEG_A_W_BORROW	NEGATE_B	NEG_B_W_BORROW
INCREMENT_A	INCREMENT_B	DECREMENT_A	DECREMENT_B
SHIFT_A	SHIFT_A_W_LSB	SHIFT_B	SHIFT_B_W_LSB
FF0_A	FF0_B	FF1_A	FF1_B
SETZ	ANDC	ANDCB	SETCA
ANDCA	SETCB	XOR	ORC
AND	EQV	SETB	ORCA
SETA	ORCB	OR	SETO
ZERO	ONES	NAND	NOR
XNOR	TEST	DONT_CARE	

Optional Parameters:

Keyword: OUTPUT_2	Type: REGISTER	
Keyword: PRECHARGE	Type: EQUATION	Default: ALWAYS
Keyword: CARRY_OUT_BAR	Type: OUTPUT	
Keyword: CARRY_INTO_MSB_BAR	Type: OUTPUT	
Keyword: MSB	Type: OUTPUT	
Keyword: ZERO	Type: OUTPUT	
Keyword: WRITE_OUTPUT_1	Type: EQUATION	Default: ALWAYS
Keyword: WRITE_OUTPUT_2	Type: EQUATION	Default: NEVER
Keyword: FLAGS	Type: REGISTER	Default: [REFRESH:ALWAYS]
Keyword: LOAD_FLAGS	Type: EQUATION	Default: NEVER
Keyword: TO_CONTROL	Type: SOURCES	

This element is similar to the ALU element, with the addition of a flag register. The flag register will load from the ALU when the load flags equation is true. Bit 1 of the register loads with the carry output, bit 2 loads with the MSB, bit width/2 + 1 loads with zero, and bit width loads with the LSB. If the datapath width is 8, bit 5 loads with zero and bit 8 loads with LSB. The remaining bits are unaltered by the load flags control. The to control specification allows these flag bits to drive lines of the instruction decoder.

(11) Element: **ALU WITH FULL FLAGS**

Required Parameters:

Keyword: INPUT_A		Type: REGISTER	
Keyword: INPUT_B		Type: REGISTER	
Keyword: OUTPUT_1		Type: REGISTER	
Keyword: DECODE		Type: DECODE	
Operations:			
ADD	ADD_W_CARRY	MASK_AB	MASK_BA
SUBTRACT	SUB_W_BORROW	SUBTRACT_REV	SUBR_W_BORROW
NEGATE_A	NEG_A_W_BORROW	NEGATE_B	NEG_B_W_BORROW
INCREMENT_A	INCREMENT_B	DECREMENT_A	DECREMENT_B
SHIFT_A	SHIFT_A_W_LSB	SHIFT_B	SHIFT_B_W_LSB
FF0_A	FF0_B	FF1_A	FF1_B
SETZ	ANDC	ANDCB	SETCA
ANDCA	SETCB	XOR	ORC
AND	EQV	SETB	ORCA
SETA	ORCB	OR	SETO
ZERO	ONES	NAND	NOR
XNOR	TEST	DONT_CARE	
Keyword: MASK		Type: REGISTER	
Keyword: FLAGS		Type: REGISTER	
Keyword: FLAG_ACCUMULATER		Type: REGISTER	
Keyword: FLAG_SELECT		Type: FIELD	
Keyword: FALSE_FALSE		Type: EQUATION	Default: NEVER
Keyword: FALSE_TRUE		Type: EQUATION	Default: NEVER
Keyword: TRUE_FALSE		Type: EQUATION	Default: NEVER
Keyword: TRUE_TRUE		Type: EQUATION	Default: NEVER
Keyword: OPERATION		Type: DECODE	
Operations:			
DONT_CARE	LOAD_ALL		
LOAD_MASKED	TEST_SELECTED		
SET_SELECTED	CLR_SELECTED		
CMP_SELECTED	LOAD_SELECTED		

Optional Parameters:

Keyword: OUTPUT_2	Type: REGISTER	
Keyword: PRECHARGE	Type: EQUATION	Default: ALWAYS
Keyword: CARRY_OUT_BAR	Type: OUTPUT	
Keyword: CARRY_INTO_MSB_BAR	Type: OUTPUT	
Keyword: MSB	Type: OUTPUT	
Keyword: ZERO	Type: OUTPUT	
Keyword: WRITE_OUTPUT_1	Type: EQUATION	Default: ALWAYS
Keyword: WRITE_OUTPUT_2	Type: EQUATION	Default: NEVER
Keyword: OLD_FLAG	Type: EQUATION	Default: NEVER
Keyword: FLAG	Type: OUTPUT	

In addition to the operations available with the standard ALU, this ALU includes a wide variety of flag operations. The FLAGS register holds the values of the flags, the MASK register may select which of the FLAGS register's bits should load, and the FLAG_ACCUMULATOR register is used to accumulate flag values. A function block (see #29 in section A5.8) exists between the FLAGS register and the FLAG_ACCUMULATOR to implement the flag accumulations. The LOAD_ALL operation loads all flags from the ALU into the FLAGS register. The LOAD_MASKED operation only loads those bits whose corresponding MASK register bits are high. TEST_SELECTED will load the FLAG bit (MSB of FLAGS) with the FLAGS bit selected by the

FLAG_SELECT field. SET_SELECTED will set the bit which FLAG_SELECT indicates, and CLR_SELECTED will clear that bit. CMP_SELECTED complements the selected bit. LOAD_SELECTED transfers from the FLAG bit to the selected bit.

The bits in the flag register have the following values. The MSB is carry out, the next bit is carry into the MSB, the next bit is MSB, the next bit is overflow, the next is greater than or equal, the next is higher, the next is greater than, the next is zero, the next is the value of OLD_FLAG (an optional input), and the LSB is LSB. Bits 10-15 are not used. **This element can only be used in datapaths than are 16 bits wide.**

A5.4: Ports

The port units are used for data communication with off-chip circuitry. The INPUT_PORT has a register which will load data from off chip when the LOAD equation is TRUE. The OUTPUT_PORT will always drive the data in its register off chip unless the DRIVE equation is present, in which case the port only drives when the equation is TRUE. The IO_PORT incorporates features of both the input ports and the output ports. When the LOAD equation is TRUE, the off chip data are loaded into the input register. When the DRIVE equation is TRUE, data in the output register are driven off chip. If the INPUT_REGISTER is not specified, the port will have only a single register, which is uses for both types of data transfer.

In each of these ports, the LOAD and DRIVE equations have variable timing, which means that the timing requirements of the control line buffers may be given by the user. These operations will occur during PHI_2 by default, but the user may state either PHI_1 timing or asynchronous timing should be used. Each of these ports has an optional mask, which can be used to indicate which bits of the register(s) actually connect to pads. Bits of a register which do not connect to a pad will be unaffected by a LOAD operation.

(12) Element: **INPUT PORT**
Required Parameters:
    Keyword: REGISTER   Type: REGISTER
    Keyword: LOAD   Type: EQUATION   Variable Timing
Optional Parameters:
    Keyword: MASK   Type: MASK

(13) Element: **OUTPUTPORT**
Required Parameters:
    Keyword: REGISTER   Type: REGISTER
Optional Parameters:
    Keyword: DRIVE   Type: EQUATION   Variable Timing
    Keyword: MASK   Type: MASK

(14) Element: **IOPORT**
Required Parameters:
    Keyword: OUTPUT_REGISTER   Type: REGISTER
    Keyword: LOAD   Type: EQUATION   Variable Timing
    Keyword: DRIVE   Type: EQUATION   Variable Timing
Optional Parameters:
    Keyword: INPUT_REGISTER   Type: REGISTER
    Keyword: MASK   Type: MASK

A5.5: Constants

The ROM (Read Only Memory) functions in Bristle Blocks are used to drive constant data onto the data buses. The value(s) contained in these ROMs can drive each bit of the data bus(es) high or low or not affect the value on the bus. The enable functions control the gating of the fixed value onto the bus. The LOWER_ROM function drives the lower data bus, the UPPER_ROM function drives the upper data bus, while the ROM and ROM_PAIR functions drive both buses. The ROM_PAIR function is logically equivalent to two ROM functions, but requires less chip area.

(15) Element: **LOWER_ROM**
Required Parameters:
    Keyword: VALUE   Type: MASK
    Keyword: ENABLE   Type: EQUATION
Optional Parameters: NONE

(16) Element: **UPPER_ROM**
Required Parameters:
    Keyword: VALUE   Type: MASK
    Keyword: ENABLE   Type: EQUATION
Optional Parameters: NONE

(17) Element: **ROM**
Required Parameters:
    Keyword: UPPER   Type: MASK
    Keyword: LOWER   Type: MASK
    Keyword: ENABLE   Type: EQUATION
Optional Parameters: NONE

(18) Element: **ROM PAIR**

```
Required Parameters:
    Keyword: LEFT_ENABLE      Type: EQUATION
    Keyword: RIGHT_ENABLE     Type: EQUATION
Optional Parameters:
    Keyword: LEFT_UPPER       Type: MASK
    Keyword: LEFT_LOWER       Type: MASK
    Keyword: RIGHT_UPPER      Type: MASK
    Keyword: RIGHT_LOWER      Type: MASK
```

A5.6: Barrel Shifters

The barrel shifters are capable of performing multiple-bit shifts in a single clock cycle. These shifters have two input words: the Most Significant Word (MSW) and the Least Significant Word (LSW). The output register may load from almost any contiguous set of bits in the combined MSW-LSW register. The shift constant indicates how many bits from the most significant end of the LSW are to appear in the output, with the remaining bits coming from the least significant end of the MSW. The width of the shift constant field must be at least log base two of the datapath width. In the SIMPLE_SHIFTER, the user specifies registers for the MSW, LSW, and the output, along with the shift constant and a LOAD equation, which controls the loading of the output register. The MASKED_SHIFTER has an additional mask register which can be used to control the loading of the output register. The two load signals, LOAD_IF_0 and LOAD_IF_1, specify the polarity of the mask bits. When the LOAD_IF_0 line is high, the only bits of the output register than are loaded from the shift operation are those bits whose corresponding mask bits are low. Similarly, the LOAD_IF_1 line controls loading the output register's bits whose corresponding mask bits are high. If both control lines are high, all of the output register bits are loaded. The BARREL_SHIFTER does not have an explicit MSW register or LSW register. Instead, two input registers are provided, along with circuitry which multiplexes various values into the MSW and LSW of the shifter. The MSW can be loaded from either of the two input registers or from the constants 0, 1, -1, and -2. The LSW can be loaded from either of the two input registers or from the constants 0 and -1. Given these possibilities, any of the arithmetic or logical shifts and rotates can be performed with the shifter. The following table lists the MSW and LSW values for the various OPERATIONs of the BARREL_SHIFTER.

Operation	MSW	LSW	
ROTATE_A	A	A	
ROTATE_B	B	B	
SHIFT_AB	A	B	
SHIFT_BA	B	A	
SLA	A	0	
SLB	B	0	
SRA_LOGICAL	0	A	
SRB_LOGICAL	0	B	
UNARY	0	-1	
UNARY_BAR	-1	0	
SRA_ZERO	0	A	(see SRA_LOGICAL)
SRB_ZERO	0	B	(see SRB_LOGICAL)
SRA_ONE	-1	A	
SRB_ONE	-1	B	
DECODE	1	0	
DECODE_BAR	-2	-1	

The most significant bits of the two input registers are available to drive the instruction decoder, which is useful for computing the sign-extention constants for arithmetic shifts. The BARREL_SHIFTER also has a mask register.

(19) Element: **SIMPLE_SHIFTER**

Required Parameters:

Keyword: MOST_SIGNIFICANT_WORD	Type: REGISTER
Keyword: LEAST_SIGNIFICANT_WORD	Type: REGISTER
Keyword: OUTPUT_REGISTER	Type: REGISTER
Keyword: SHIFT_CONSTANT	Type: FIELD
Keyword: LOAD	Type: EQUATION

Optional Parameters: NONE

(20) Element: **MASKED_SHIFTER**

Required Parameters:

Keyword: MOST_SIGNIFICANT_WORD	Type: REGISTER
Keyword: LEAST_SIGNIFICANT_WORD	Type: REGISTER
Keyword: OUTPUT_REGISTER	Type: REGISTER
Keyword: MASK_REGISTER	Type: REGISTER
Keyword: SHIFT_CONSTANT	Type: FIELD
Keyword: LOAD_IF_0	Type: EQUATION
Keyword: LOAD_IF_1	Type: EQUATION

Optional Parameters: NONE

(21) Element: **BARREL SHIFTER**
Required Parameters:

Keyword: INPUT_A		Type: REGISTER
Keyword: INPUT_B		Type: REGISTER
Keyword: OUTPUT_REGISTER		Type: REGISTER
Keyword: SHIFT_CONSTANT		Type: FIELD
Keyword: LOAD_IF_0		Type: EQUATION
Keyword: LOAD_IF_1		Type: EQUATION
Keyword: OPERATION		Type: DECODE
Operations:		
ROTATE_A	ROTATE_B	
SHIFT_AB	SHIFT_BA	
SLA	SLB	
SRA_LOGICAL	SRB_LOGICAL	
UNARY	UNARY_BAR	
SRA_ZERO	SRB_ZERO	
SRA_ONE	SRB_ONE	
DECODE	DECODE_BAR	

Optional Parameters:

Keyword: MASK_REGISTER	Type: REGISTER
Keyword: A_MSB	Type: OUTPUT
Keyword: B_MSB	Type: OUTPUT

A5.7: Bus Precharge Elements

The bus precharge elements are used to precharge the data buses. Each of the data processing elements in Bristle Blocks (except for the ROM cells) only drives the data buses low. To transmit a high value, the data processing elements do not affect the bus, assuming that the bus originally had every bus line high. In order to transmit data, therefore, the buses must be precharged. These elements precharge one or both of the buses during PHI_2. The data buses can be used to store data from one cycle to the next, if the clocks run fast enough, and if no other element writes on the bus. The first three elements simply precharge the buses. The remaining two functions not only precharge the bus, but they 'break' the bus. The bus to the left is terminated, and a new bus begins to the right (this new bus must be precharged by a different bus precharge element). This allows Bristle Blocks to compile chips with more than two data buses, although only two data buses may pass any element.

(22) Element: **PRECHARGE LOWER**
Required Parameters: NONE
Optional Parameters:
Keyword: PRECHARGE Type: EQUATION Default: ALWAYS

(23) Element: **PRECHARGE UPPER**
Required Parameters: NONE
Optional Parameters:
Keyword: PRECHARGE Type: EQUATION Default: ALWAYS

(24) Element: **PRECHARGE BOTH**
Required Parameters: NONE
Optional Parameters:
    Keyword: PRECHARGE   Type: EQUATION   Default: ALWAYS

(25) Element: **PRECHARGE AND BREAK LOWER**
Required Parameters: NONE
Optional Parameters:
    Keyword: PRECHARGE   Type: EQUATION   Default: ALWAYS

(26) Element: **PRECHARGE AND BREAK UPPER**
Required Parameters: NONE
Optional Parameters:
    Keyword: PRECHARGE   Type: EQUATION   Default: ALWAYS

A5.8: Random Simple Elements

There are a few simple elements which do not fit in the categories presented above. These elements are described here.

(27) Element: **BUS CAM**
Required Parameters:
    Keyword: VALUE   Type: MASK
    Keyword: OUTPUT   Type: OUTPUT
Optional Parameters:
    Keyword: LATCH   Type: EQUATION   Default: ALWAYS

The BUS CAM element will monitor data flow across the lower bus. When the sampled data matches the fixed value wired into the CAM, the output signal will go high. The LATCH equation controls the sampling of the bus. The VALUE mask states the comparison value for the CAM. When all the bus bits corresponding to O bits in the mask are low and when all the bus bits corresponding to I bits in the mask are high, the output signal goes high.

(28) Element: **CAM**
Required Parameters:
    Keyword: REGISTER   Type: REGISTER
    Keyword: VALUE   Type: MASK
    Keyword: OUTPUT   Type: OUTPUT
Optional Parameters: NONE

This element is similar to the BUS CAM but that the CAM monitors the value contained in its register. Whenever the register's value matches the CAM's value, the output signal goes high. There is no LOAD signal, since the CAM always monitors the register's value.

(29) Element: **FUNCTION_BLOCK**

```
Required Parameters:
    Keyword: INPUT_A          Type: REGISTER
    Keyword: INPUT_B          Type: REGISTER
Optional Parameters:
    Keyword: OUTPUT           Type: REGISTER
    Keyword: PRECHARGE        Type: EQUATION    Default: ALWAYS
    Keyword: LOAD_INPUT_A     Type: EQUATION    Default: NEVER
    Keyword: LOAD_OUTPUT      Type: EQUATION    Default: ALWAYS
    Keyword: FALSE_FALSE      Type: EQUATION    Default: NEVER
    Keyword: FALSE_TRUE       Type: EQUATION    Default: NEVER
    Keyword: TRUE_FALSE       Type: EQUATION    Default: NEVER
    Keyword: TRUE_TRUE        Type: EQUATION    Default: NEVER
```

The FUNCTION_BLOCK element is used to perform boolean operations between values. The function block takes data from the two input registers, and can store data into the INPUT_A register and the OUTPUT register. The FALSE_FALSE (FF), FALSE_TRUE (FT), TRUE_FALSE (TF), and TRUE_TRUE (TT) lines control the function of the element. If the FF line is high, all bits of the output which correspond to low bits in both input registers will be high. Similarly, the TT line controls the output bits corresponding to high bits in both registers. If TF is high, all output bits which correspond to high bits in INPUT_A and low bits in INPUT_B will be high. The FT control is similar to the TF control. An alternative statement of the FUNCTION_BLOCK operation is that each pair of input bits selects which control line drives the corresponding output bit. For example, if the MSB of INPUT_A is high and the MSB of INPUT_B is low, the MSB of the output will be the value of the TRUE_FALSE control. If TT, TF, and FT are high and FF is low, the function block performs an OR operation, while if TT is the only high control, an AND function is performed. The PRECHARGE equation controls the loading of data from the input registers to internal nodes.

(30) Element: **LEFT_RIGHT_SHIFT**

```
Required Parameters:
    Keyword: INPUT_REGISTER      Type: REGISTER
    Keyword: SHIFT_LEFT          Type: EQUATION
    Keyword: SHIFT_RIGHT         Type: EQUATION
Optional Parameters:
    Keyword: OUTPUT_REGISTER     Type: REGISTER
    Keyword: INPUT               Type: EQUATION    Default: NEVER
    Keyword: MSB                 Type: OUTPUT
    Keyword: PRECHARGE           Type: EQUATION    Default: ALWAYS
```

The LEFT_RIGHT_SHIFT element is a bi-directional, single-bit shifter. When the SHIFT_LEFT control is high, the data in the INPUT_REGISTER are shifted one bit toward the MSB and loaded into the OUTPUT_REGISTER. If the OUTPUT_REGISTER is not specified, the data are loaded into the INPUT_REGISTER. The LSB of the output

register is loaded with value of the INPUT equation. The SHIFT_RIGHT control shifts data toward the LSB, with the MSB receiving data from INPUT. The PRECHARGE equation loads the input register's data into internal nodes. The MSB of the input register is available to drive the instruction decoder.

(31) Element: **STACK**

```
Required Parameters:
    Keyword: DEPTH       Type: INTEGER
    Keyword: TOP         Type: REGISTER
    Keyword: POP         Type: EQUATION
    Keyword: PUSH        Type: EQUATION
Optional Parameters:
    Keyword: MIDDLE      Type: REGISTER    Default: [REFRESH:ALWAYS]
    Keyword: BOTTOM      Type: REGISTER    Default: [REFRESH:ALWAYS]
    Keyword: REFRESH     Type: EQUATION    Default: ALWAYS
```

The STACK element implements a stack in the datapath. The stack is consists of a TOP register followed by DEPTH-1 MIDDLE registers, followed by a BOTTOM register. Between adjacent register pairs lie circuitry for transfering data between the registers. When the PUSH control is TRUE, data is moved away from the TOP register: The TOP register's data loads the first MIDDLE register, while the first MIDDLE register's data are loading the second MIDDLE register, etc. When the POP control is TRUE, data are moved towards the TOP register. The PUSH and POP controls should not both be high, nor should POP be high while the TOP register is writing onto a data bus.

A5.9: Compound IR Elements

The following cells combine the DATA_TO_CONTROL circuitry with another simple element function. The DATA_TO_CONTROL function is useful for implementing Instruction Registers (IR) because the function of an IR is to turn data values into control values. In the INCREMENTING_IR example, the IR's data can be incremented. Alternatively, one may think of the incrementer's output driving the instruction decoder. The operation of each of these units can be found by comparing the functions of the DATA_TO_CONTROL element (2) and the simple element which is fused with the IR.

(32) Element: **INCREMENTING IR**

```
Required Parameters:
    Keyword: MAP            Type: SOURCES
    Keyword: REGISTER       Type: REGISTER
    Keyword: LOAD           Type: EQUATION
Optional Parameters:
    Keyword: PRECHARGE      Type: EQUATION     Default: ALWAYS
    Keyword: CARRY_OUT      Type: OUTPUT
****** see (2) and (5)
```

(33) Element: **DECREMENTINGR**

```
Required Parameters:
    Keyword: MAP            Type: SOURCES
    Keyword: REGISTER       Type: REGISTER
    Keyword: LOAD           Type: EQUATION
Optional Parameters:
    Keyword: PRECHARGE      Type: EQUATION     Default: ALWAYS
    Keyword: CARRY_OUT      Type: OUTPUT
****** see (2) and (6)
```

(34) Element: **SHIFTINGR**

```
Required Parameters:
    Keyword: SHIFT_LEFT     Type: EQUATION
    Keyword: SHIFT_RIGHT    Type: EQUATION
    Keyword: MAP            Type: SOURCES
Optional Parameters:
    Keyword: INPUT          Type: EQUATION     Default: NEVER
    Keyword: MSB            Type: OUTPUT
    Keyword: PRECHARGE      Type: EQUATION     Default: ALWAYS
    Keyword: REGISTER       Type: REGISTER     Default: [REFRESH:ALWAYS]
****** see (2) and (30)
```

(35) Element: **SWAPPINGR**

```
Required Parameters:
    Keyword: ACTIVE     Type: REGISTER
    Keyword: MAP        Type: SOURCES
Optional Parameters:
    Keyword: BACKUP     Type: REGISTER     Default: [REFRESH:ALWAYS]
    Keyword: SAVE       Type: EQUATION     Default: NEVER
    Keyword: REFRESH    Type: EQUATION     Default: ALWAYS
    Keyword: RESTORE    Type: EQUATION     Default: NEVER
****** see (2) and (31), also section A5.11
```

This element is a depth=1 stack. One of the registers (ACTIVE) is connected to the IR, the other (BACKUP) is a backup register. SAVE moves the data from ACTIVE to BACKUP, RESTORE moves the data from BACKUP to ACTIVE, and if both are high, the two registers swap value.

(36) Element: **INPUT IR**
Required Parameters:
    Keyword: LOAD Type: EQUATION Variable Timing
    Keyword: MAP Type: SOURCES
Optional Parameters:
    Keyword: MASK Type: MASK
    Keyword: REGISTER Type: REGISTER Default: [REFRESH:ALWAYS]
****** see (2) and (12)

A5.10: Compound Output Port Elements

In the same manner as section A5.9 presented IR compounds with various elements, this section lists Output ports (13) fused with other simple elements.

(37) Element: **INCREMENTING PORT**
Required Parameters:
    Keyword: LOAD Type: EQUATION
Optional Parameters:
    Keyword: DRIVE Type: EQUATION Variable Timing
    Keyword: MASK Type: MASK
    Keyword: REGISTER Type: REGISTER Default: [REFRESH:ALWAYS]
    Keyword: PRECHARGE Type: EQUATION Default: ALWAYS
    Keyword: CARRY_OUT Type: OUTPUT
****** see (5) and (13)

(38) Element: **DECREMENTING PORT**
Required Parameters:
    Keyword: LOAD Type: EQUATION
Optional Parameters:
    Keyword: DRIVE Type: EQUATION Variable Timing
    Keyword: MASK Type: MASK
    Keyword: REGISTER Type: REGISTER Default: [REFRESH:ALWAYS]
    Keyword: PRECHARGE Type: EQUATION Default: ALWAYS
    Keyword: CARRY_OUT Type: OUTPUT
****** see (6) and (13)

(39) Element: **ADDING PORT**
Required Parameters:
    Keyword: LOAD Type: EQUATION
Optional Parameters:
    Keyword: DRIVE Type: EQUATION Variable Timing
    Keyword: MASK Type: MASK
    Keyword: ACCUMULATOR Type: REGISTER Default: [REFRESH:ALWAYS]
    Keyword: OFFSET Type: REGISTER Default: [REFRESH:ALWAYS]
    Keyword: PRECHARGE Type: EQUATION Default: ALWAYS
    Keyword: CARRY_OUT_BAR Type: OUTPUT
    Keyword: LATCH Type: EQUATION Default: ALWAYS
    Keyword: CARRY_IN_BAR Type: EQUATION Default: NEVER
****** see (7) and (13)

(40) Element: **SWAPPING OUTPUT PORT**

```
Required Parameters:
    Keyword: ACTIVE      Type: REGISTER
Optional Parameters:
    Keyword: BACKUP      Type: REGISTER    Default: [REFRESH:ALWAYS]
    Keyword: SAVE        Type: EQUATION    Default: NEVER
    Keyword: REFRESH     Type: EQUATION    Default: ALWAYS
    Keyword: RESTORE     Type: EQUATION    Default: NEVER
    Keyword: DRIVE       Type: EQUATION    Variable Timing
    Keyword: MASK        Type: MASK
****** see (31) and (13), also section A5.11
```

A5.11: Compound Swapping Elements

In the same manner as section A5.9 presented IR compounds with various elements, this section lists swapping registers fused with other simple elements. Swapping registers are effectively a depth=1 stack. One of the registers (ACTIVE) is connected to the simple element with which the swapper is compounded, the other (BACKUP) is a backup register. SAVE moves the data from ACTIVE to BACKUP, RESTORE moves the data from BACKUP to ACTIVE, and if both are high, the two registers swap value.

(41) Element: **SWAPPING REGISTERS**

```
Required Parameters:
    Keyword: LEFT              Type: REGISTER
    Keyword: RIGHT             Type: REGISTER
Optional Parameters:
    Keyword: RIGHT_TO_LEFT     Type: EQUATION    Default: NEVER
    Keyword: REFRESH           Type: EQUATION    Default: ALWAYS
    Keyword: LEFT_TO_RIGHT     Type: EQUATION    Default: NEVER
```

This element is just a pair of swapping registers.

(42) Element: **SWAPPING INPUT PORT**

```
Required Parameters:
    Keyword: LOAD        Type: EQUATION    Variable Timing
    Keyword: ACTIVE      Type: REGISTER
Optional Parameters:
    Keyword: MASK        Type: MASK
    Keyword: RESTORE     Type: EQUATION    Default: NEVER
    Keyword: REFRESH     Type: EQUATION    Default: ALWAYS
    Keyword: SAVE        Type: EQUATION    Default: NEVER
    Keyword: BACKUP      Type: REGISTER    Default: [REFRESH:ALWAYS]
****** see (12)
```

(40) Element: **SWAPPING OUTPUT PORT**

```
Required Parameters:
    Keyword: ACTIVE      Type: REGISTER
Optional Parameters:
    Keyword: BACKUP      Type: REGISTER    Default: [REFRESH:ALWAYS]
    Keyword: SAVE        Type: EQUATION    Default: NEVER
    Keyword: REFRESH     Type: EQUATION    Default: ALWAYS
    Keyword: RESTORE     Type: EQUATION    Default: NEVER
    Keyword: DRIVE       Type: EQUATION    Variable Timing
    Keyword: MASK        Type: MASK
****** see (13)
```

(43) Element: **SWAPPING INCREMENTER**

```
Required Parameters:
    Keyword: LOAD        Type: EQUATION
    Keyword: ACTIVE      Type: REGISTER
Optional Parameters:
    Keyword: PRECHARGE   Type: EQUATION    Default: ALWAYS
    Keyword: CARRY_OUT   Type: OUTPUT
    Keyword: RESTORE     Type: EQUATION    Default: NEVER
    Keyword: REFRESH     Type: EQUATION    Default: ALWAYS
    Keyword: SAVE        Type: EQUATION    Default: NEVER
    Keyword: BACKUP      Type: REGISTER    Default: [REFRESH:ALWAYS]
****** see (5)
```

(44) Element: **SWAPPING DECREMENTER**

```
Required Parameters:
    Keyword: LOAD        Type: EQUATION
    Keyword: ACTIVE      Type: REGISTER
Optional Parameters:
    Keyword: PRECHARGE   Type: EQUATION    Default: ALWAYS
    Keyword: CARRY_OUT   Type: OUTPUT
    Keyword: RESTORE     Type: EQUATION    Default: NEVER
    Keyword: REFRESH     Type: EQUATION    Default: ALWAYS
    Keyword: SAVE        Type: EQUATION    Default: NEVER
    Keyword: BACKUP      Type: REGISTER    Default: [REFRESH:ALWAYS]
****** see (6)
```

(35) Element: **SWAPPING R**

```
Required Parameters:
    Keyword: ACTIVE      Type: REGISTER
    Keyword: MAP         Type: SOURCES
Optional Parameters:
    Keyword: BACKUP      Type: REGISTER    Default: [REFRESH:ALWAYS]
    Keyword: SAVE        Type: EQUATION    Default: NEVER
    Keyword: REFRESH     Type: EQUATION    Default: ALWAYS
    Keyword: RESTORE     Type: EQUATION    Default: NEVER
****** see (2)
```

A5.12: Compound CAM Elements

In the same manner as section A5.9 presented IR compounds with various elements, this section lists CAM registers (28) fused with other simple elements.

(45) Element: **ADDER WITH VALUE CHECK**

Required Parameters:

Keyword: VALUE	Type: MASK	
Keyword: RESULT	Type: OUTPUT	
Keyword: INPUT_B	Type: REGISTER	
Keyword: INPUT_A	Type: REGISTER	
Keyword: LOAD	Type: EQUATION	

Optional Parameters:

Keyword: OUTPUT	Type: REGISTER	Default: [REFRESH:ALWAYS]
Keyword: PRECHARGE	Type: EQUATION	Default: ALWAYS
Keyword: CARRY_OUT_BAR	Type: OUTPUT	
Keyword: LATCH	Type: EQUATION	Default: ALWAYS
Keyword: CARRY_IN_BAR	Type: EQUATION	Default: NEVER

****** see (28) and (7)

(46) Element: **SUBTRACTER WITH VALUE CHECK**

Required Parameters:

Keyword: VALUE	Type: MASK	
Keyword: RESULT	Type: OUTPUT	
Keyword: INPUT_B	Type: REGISTER	
Keyword: INPUT_A	Type: REGISTER	
Keyword: LOAD	Type: EQUATION	

Optional Parameters:

Keyword: OUTPUT	Type: REGISTER	Default: [REFRESH:ALWAYS]
Keyword: PRECHARGE	Type: EQUATION	Default: ALWAYS
Keyword: CARRY_OUT_BAR	Type: OUTPUT	
Keyword: LATCH	Type: EQUATION	Default: ALWAYS
Keyword: CARRY_IN_BAR	Type: EQUATION	Default: NEVER

****** see (28) and (8)

(47) Element: **INCREMENTER WITH VALUE CHECK**

Required Parameters:

Keyword: VALUE	Type: MASK	
Keyword: RESULT	Type: OUTPUT	
Keyword: REGISTER	Type: REGISTER	
Keyword: LOAD	Type: EQUATION	

Optional Parameters:

Keyword: PRECHARGE	Type: EQUATION	Default: ALWAYS
Keyword: CARRY_OUT	Type: OUTPUT	

****** see (28) and (5)

(48) Element: **DECREMENTER WITH VALUE CHECK**

Required Parameters:

Keyword: VALUE	Type: MASK	
Keyword: RESULT	Type: OUTPUT	
Keyword: REGISTER	Type: REGISTER	
Keyword: LOAD	Type: EQUATION	

Optional Parameters:

Keyword: PRECHARGE	Type: EQUATION	Default: ALWAYS
Keyword: CARRY_OUT	Type: OUTPUT	

****** see (28) and (6)

(49) Element: **ACCUMULATOR WITH VALUE CHECK**

```
Required Parameters:
     Keyword: VALUE            Type: MASK
     Keyword: RESULT           Type: OUTPUT
     Keyword: INPUT            Type: REGISTER
     Keyword: LOAD             Type: EQUATION
Optional Parameters:
     Keyword: ACCUMULATOR      Type: REGISTER    Default: [REFRESH:ALWAYS]
     Keyword: PRECHARGE        Type: EQUATION    Default: ALWAYS
     Keyword: CARRY_OUT_BAR    Type: OUTPUT
     Keyword: LATCH            Type: EQUATION    Default: ALWAYS
     Keyword: CARRY_IN_BAR     Type: EQUATION    Default: NEVER
****** see (28) and (7)
```

(50) Element: **SHIFTERWITH VALUE CHECK**

```
Required Parameters:
     Keyword: SHIFT_LEFT       Type: EQUATION
     Keyword: SHIFT_RIGHT      Type: EQUATION
     Keyword: VALUE            Type: MASK
     Keyword: RESULT           Type: OUTPUT
Optional Parameters:
     Keyword: INPUT            Type: EQUATION    Default: NEVER
     Keyword: MSB              Type: OUTPUT
     Keyword: PRECHARGE        Type: EQUATION    Default: ALWAYS
     Keyword: REGISTER         Type: REGISTER    Default: [REFRESH:ALWAYS]
****** see (28) and (30)
```

A5.13: Random Compound Elements

The remaining two elements are SHIFTING_ACCUMULATOR and INCREMENTER_DECREMENTER. The SHIFTING_ACCUMULATOR is a two register adder (7) with a left-right shifter (30) on the input/output register. The INCREMENTER_DECREMENTER is a back-to-back two-register INCREMENTER (5) and DECREMENTER (6). When the LOAD_DEC line is high, the incrementer input register is loaded with one less than the value in the decrementer input register. When the LOAD_INC line is high, the decrementer input register is loaded with one more than the value in the incrementer input register.

(51) Element: **SHIFTING ACCUMULATOR**

Required Parameters:

Keyword: SHIFT_LEFT	Type: EQUATION	
Keyword: SHIFT_RIGHT	Type: EQUATION	
Keyword: ACCUMULATOR	Type: REGISTER	
Keyword: LOAD	Type: EQUATION	

Optional Parameters:

Keyword: INPUT	Type: EQUATION	Default: NEVER
Keyword: MSB	Type: OUTPUT	
Keyword: PRECHARGE_2	Type: EQUATION	Default: ALWAYS
Keyword: INPUT	Type: REGISTER	Default: [REFRESH:ALWAYS]
Keyword: PRECHARGE_1	Type: EQUATION	Default: ALWAYS
Keyword: CARRY_OUT_BAR	Type: OUTPUT	
Keyword: LATCH	Type: EQUATION	Default: ALWAYS
Keyword: CARRY_IN_BAR	Type: EQUATION	Default: NEVER

(52) Element: **INCREMENTER DECREMENTER**

Required Parameters:

Keyword: LOAD_DEC	Type: EQUATION	
Keyword: INC_INPUT	Type: REGISTER	
Keyword: DEC_INPUT	Type: REGISTER	
Keyword: LOAD_INC	Type: EQUATION	

Optional Parameters:

Keyword: PRECHARGE_DEC	Type: EQUATION	Default: ALWAYS
Keyword: CARRY_OUT_DEC	Type: OUTPUT	
Keyword: PRECHARGE_INC	Type: EQUATION	Default: ALWAYS
Keyword: CARRY_OUT_INC	Type: OUTPUT	

A5.14: Summary

The following list shows the Bristle Blocks element in alphabetical order.

(49) ACCUMULATOR WITH VALUE CHECK
(7) ADDER
(45) ADDER WITH VALUE CHECK
(39) ADDING PORT
(9) ALU
(10) ALU WITH FLAGS
(11) ALU WITH FULL FLAGS
(21) BARREL SHIFTER
(27) BUS CAM
(28) CAM
(3) CONTROL TO DATA
(4) CONTROL TO DATA AND BACK
(2) DATA TO CONTROL
(6) DECREMENTER
(48) DECREMENTER WITH VALUE CHECK
(33) DECREMENTING IR
(38) DECREMENTING PORT
(29) FUNCTION BLOCK
(5) INCREMENTER
(52) INCREMENTER DECREMENTER
(47) INCREMENTER WITH VALUE CHECK
(32) INCREMENTING IR
(37) INCREMENTING PORT
(36) INPUT IR
(12) INPUT PORT
(14) IO PORT
(30) LEFT RIGHT SHIFT
(15) LOWER ROM
(20) MASKED SHIFTER
(13) OUTPUT PORT
(25) PRECHARGE AND BREAK LOWER
(26) PRECHARGE AND BREAK UPPER
(24) PRECHARGE BOTH
(22) PRECHARGE LOWER
(23) PRECHARGE UPPER
(1) REGISTER
(17) ROM
(18) ROM PAIR
(50) SHIFTER WITH VALUE CHECK
(51) SHIFTING ACCUMULATOR
(34) SHIFTING IR
(19) SIMPLE SHIFTER
(31) STACK
(8) SUBTRACTER
(46) SUBTRACTER WITH VALUE CHECK
(44) SWAPPING DECREMENTER
(43) SWAPPING INCREMENTER
(42) SWAPPING INPUT PORT
(35) SWAPPING IR
(40) SWAPPING OUTPUT PORT
(41) SWAPPING REGISTERS
(16) UPPER ROM

References

[1] Applicon
Applicon Users Manual
Applicon, Inc. Burlington, MA.

[2] Automation Technology
"Precision Artwork Language (PAL)"
Automation Technology, Inc. 1971

[3] Ayres, R.F.
A Language Processor and a Sample Language
Ph.D. Thesis (#2276)
California Institute of Technology, 1979

[4] Ayres, R.F.
"IC Design Under ICL, Version 1"
Caltech SSP Report #1366 (revised #4031)
California Institute of Technology, 1978

[5] Ayres, R.F.
"Silicon Compilation-A Heirarchical Use of PLAs"
Proceedings of the 16th Design Automation Conference, 1979

[6] Buchanan, I.
Modelling and Verification in Structured Integrated Circuit Design
Ph.D. Thesis
University of Edinburgh, 1980

[7] Calma
GDS II Product Specification
Calma Interactive Graphics Systems Sunnyvale, CA.

[8] Fairbairn, D.G. and Rowson, J.A.
"Interactive Integrated Circuit Design on a Small Computer"
Proceedings of 1st Conference on Computer Graphics
in CAD/CAM Systems, 1979

[9] Feller, A.
"Automatic Layout of Low-Cost Quick-Turnaround Random-Logic
Custom LSI Devices"
Proceedings of the 13th Design Automation Conference, 1976

[10] Friedman, T.D.
"Methods Used in an Automatic Logic Design Generator (ALERT)"
IEEE Transactions on Computers, C-18, July 1969, p. 593-614

[11] Herrick, W.V. and Sims, J.R.
"A Successful Automated IC Design System"
Proceedings of the 13th Design Automation Conference, 1976

[12] Johannsen, D.L.
"Bristle Blocks: A Silicon Compiler"
Caltech Conference on VLSI, 1979

[13] Johannsen, D.L.
"Bristle Blocks: A Silicon Compiler"
Proceedings of the 16th Design Automation Conference, 1979

[14] Johannsen, D.L.
"Hierarchical Power Routing"
Caltech SSP Report #2069
California Institute of Technology, 1978

[15] Johannsen, D.L.
OM2 LSI Chip
Caltech Part #986
California Institute of Technology, 1978

[16] Johannsen, D.L.
"OM2"
Caltech SSP Report #1111
California Institute of Technology, 1978

[17] Johannsen, D.L.
"Our Machine, A Microcoded LSI Processor"
Proceedings of the 11th Annual Microprogramming Workshop, 1978

[18] Lattin, W.
"VLSI Design Methodology: The Problem of the 80's
for Microprocessor Design"
Caltech Conference on VLSI, 1979

[19] Locanthi, B.
"LAP: A Simula Package for IC Layout"
Caltech SSP Report #1862
California Institute of Technology, 1978

[20] Mead, C.A. and Conway, L.
Introduction to VLSI Systems
Addison-Wesly Publishing, Reading MA., 1980

[21] Mosteller, R.C.
REST -- Stick Diagram Editing System
Masters Thesis
California Institute of Technology, 1981

[22] Oestreicher, D.
"PLASYS: Final Report"
Caltech SSP Report #3655
California Institute of Technology, 1980

[23] Parker, A., et al.
"The CMU Design Automation System:
An Example of Automated Data Path Design"
Proceedings of the 16th Design Automation Conference, 1979

[24] Rowson, J.A. and Trimberger, S.
"Riot -- A Stupid Graphical Composition Tool"
Caltech SSP Technical Report #4142
California Institute of Technology, 1981

[25] Rowson, J.A.
Understanding Hierarchical Design
Ph.D. Thesis
California Institute of Technology, 1980

[26] Schorr, H.
"Computer-Aided Digital System Design and Analysis
Using a Register Transfer Language"
IEEE Transactions, Electronic Computers EC13, Dec. 1964, p. 730-737

[27] Segal, R.
Structure, Placement and Modelling
Masters Thesis (#4029)
California Institute of Technology, 1980

[28] Sequin, C.
"STIF: A Proposal for a Structured Topological Interchange Format"
University of California, Berkeley, 1980

[29] Tarolli, G.
"Towards a Working VLSI CAD Tool: A Chip Assembler"
Caltech SSP Report #3131
California Institute of Technology, 1979

[30] Trimberger, S.
"Combining Graphics and Layout Language in a Single Interactive System"
Caltech SSP Technical Report #3794
California Institute of Technology, 1980

[31] Trimberger, S.
"Nick -- A FORTRAN Layout Language Package"
Caltech SSP Report #3486
California Institute of Technology, 1980

[32] Trimberger, S.
"The Proposed Sticks Standard"
Caltech SSP Technical Report #3880,
California Institute of Technology, 1980

[33] Trimberger, S.
A Wire Oriented Mask Geometry Editor
Masters Thesis
California Institute of Technology, 1979

[34] Williams, J.D.
Sticks -- A New Approach to LSI Design
Masters Thesis
Massachusetts Institute of Technology, 1977