The University of Calgary

Integrated Environmental Support for Silicon Compilation of Digital Filters

by

Wallace I. Kroeker

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

Department of Computer Science

Calgary, Alberta

March, 1986

© Wallace I. Kroeker, 1986.

Permission has been granted to the National Library of Canada to microfilm this thesis and to lend or sell copies of the film.

The author (copyright owner) has reserved other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without his/her written permission.

L'autorisation a été accordée à la Bibliothèque nationale du Canada de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

L'auteur (titulaire du droit d'auteur) se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation écrite.

ISBN Ø-315-29966-5

The University Of Calgary

Faculty Of Graduate Studies

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled, "Integrated Environmental Support for Silicon Compilation of Digital Filters" submitted by Wallace I. Kroeker in partial fulfillment of the requirements for the degree of Master of Science.

Supervisor,
Dr. G. M. Birtwistle
Department of Computer Science

Dr. E. J. M. Kendall
Department of Computer Science

R. V. Vasudevan
Department of Computer Science

Dr. L. E. Turner
Department of Electrical Engineering

March 26, 1986

Abstract

Designing digital systems for integrated circuits currently requires the specification of the circuit in the behavioural, structural, and mask layout domains. Typically each specification domain has an independent representation, and translation between domains is done manually. This leads to problems with consistency between domains, making it difficult to verify that the chip fabricated was the one originally intended.

Silicon compilation is a technique which automatically maps from the behavioural or structural domains to the mask layout domain. To be successful, silicon compilation techniques must be supported by an integrated design environment in which all the design tools work from a central specification. Furthermore, silicon compilation is only practical for a properly chosen application domain.

This dissertation illustrates these ideas by presenting EFIDO (Electric's FIlter Design Organizer), an integrated environment which supports a silicon compiler for digital filters. EFIDO allows a filter structure to be specified in an intermediate language. This specification is then used by a complete range of validation tools and finally by a silicon compiler to generate a mask level description.

Acknowledgements

I would like to thank Dr. G.M. Birtwistle for his encouragement and patience during the development of this thesis.

I am indebted to Jeff Joyce, Greg Lomow, and Craig Thompson for taking the time to review earlier drafts of this work. Also, special thanks to Dr. L. Turner, Erwin Liu, and Richard Lyon for their help in understanding the areas of digital filter design and bit-serial architectures; Rick Schediwy, Bill Coates, and Simon Williams for the basic cell design; Mike Bonham for his help with *troff*; and the faculty, staff, and graduate students in the Computer Science department for their help and useful discussions.

I would like to thank my wife Paula for her love, patience and editing skills that made this work possible. Finally, a special thanks to Anthony Sean Kroeker for having the foresight to arrive on the scene two days after my defence.

Table of Contents

List of Figures

CHAPTER 1

Introduction

Designing a Very Large Scale Integrated (VLSI) circuit currently requires designers to be experts not only in a specific application area, but also in logic level design and chip level mask layout. Even for expert designers circuit design is a slow process requiring the designer to manually move from each stage of the design process to the next with no automatic way of preserving correctness between the levels of design specification. What is needed are tools which help the designer move from specification to implementation in a consistent manner.

Silicon compilation is one tool which automatically bridges the gaps between domain specifications. However, for silicon compilation to be successful it must be supported by an integrated design environment and have a well-chosen application domain. This thesis presents the objectives, design and implementation of one such environment for digital filter design, called EFIDO (Electric's FIlter Design Organizer).

The EFIDO environment provides a basic set of tools to allow digital filter designers to specify, verify, and implement a digital filter. The circuit is specified using a graphical data flow representation of a digital filter. This is the central specification which is used to automatically generate input to a functional simulator and an internal intermediate bit-serial form of the digital filter. The intermediate bit-serial representation is then used to provide input to a bit-serial simulator and a bit-serial silicon compiler.

This chapter presents an overview of VLSI design by introducing design methodology, design specification languages, data base design, layout editors, user interface issues, circuit functionality concerns and silicon compilation. This is followed by an outline of the thesis.

1.1. VLSI Design: An Overview

During the last 25 years the complexity of integrated circuits (IC) has increased from 10 or 20 transistors on a chip to more than 500,000. To make the design process more manageable new circuit design methodologies, data base techniques, and Computer Aided Design (CAD) tools have been developed. The intent of these new techniques is to allow circuit designers to devote more time to higher-level structural investigations, since global optimization has now become a major concern as system size increases.

The purpose of CAD is to free designers from tedious, error-prone tasks which can easily be automated. This allows circuit designers to handle larger designs, and enables non-expert designers to obtain functional circuits without having to learn the intricacies of design.

1.1.1. Design Methodologies

Circuit design methodologies have been developed for technologies such as Transistor Transistor Logic (TTL) [Zissos 1979], which stress minimizing chip count as opposed to improving system functionality. With new high density technologies, such as Complementary Metal Oxide Semiconductor (CMOS), realizing the logic circuitry is no longer a problem; rather the problem is the communication between functional blocks of a circuit system. Mead and Conway [Mead and Conway 1980] have established a

methodology for Large and Very Large Scale Integrated circuits (LSI and VLSI), which stresses hierarchical design and correctness by construction, allowing abstraction of design problems at several levels of detail. Their methodology has been used for developing circuits in industry and university environments and has had a strong influence on the design of CAD systems.

1.1.2. Design Data Base Concerns

Currently, most design automation tools consist of collections of independent programs. Each program usually addresses a specific design problem, such as compaction of mask level layouts [Mosteller 1981] or routing [Breuer and Carter 1984], with the only communication between programs being the Caltech Intermediate Form (CIF) [Sproull and Lyon 1980]. CIF was initially designed as a standard interface language between early design tools and silicon foundries. The most notable shortcomings of CIF are that it does not maintain structural or behavioural information for a design and it is not easily extended to include non-graphical design information required by tools such as simulators.

The shortcomings of the CIF standard have been addressed by SCALE [Buchanan 1982], EDIF (Electronic Design Interchange Format) [Edif 1985], and SHIFT (a Structured Hierarchical Intermediate Form for VLSI design Tools) [Liblong 1984]. These intermediate forms attempt to capture the structure of the design within the representation and are extendable to allow specification of new information as needed. Design exchange formats are useful in transporting information between various existing design tools, however they should not serve as design data bases.

A somewhat antiquated view is that every design tool can parse in EDIF, for example, and write out EDIF allowing communication between tools. This is not feasible for large designs where one wants to move between tools fairly quickly. To address this problem new design data bases are being developed for powerful personal workstations, such as the LISP machine [Moon and Weinreb 1981]. Design data bases are not easily handled by conventional techniques due to the wide range of domains (eg. behavioral, structural, physical etc.) and the diverse requirements within each domain. Each component in a design requires a representation in each domain, and consistency must be maintained between representations. Problems arise when critical changes in one domain are not properly reflected in other domains.

Configuration management [Katz 1985] deals with the problem of consistency between the various domain representations. If a designer changes the structural specification of a component, the layout specification may need to be updated. Work is now going on in this area [Katz 1985] which should aid in configuration management for design data bases.

1.1.3. Layout Design Tools

The acceptance of any design tool often depends on the friendliness of the user interface. A working environment should give users the tools required to do the job or at least the capabilities to extend the tools easily to suit the users and their needs. An excellent example of such a tool is Emacs [Stallman 1981], a text editor, which was given to a community of users who expanded its capabilities to suit their individual needs. Many of the extensions were incorporated in new releases of Emacs, thus producing a

powerful tool through an evolutionary process.

Graphical editors and circuit design tools have not met with the same success. The user community of such tools often lack the programming experience to make extensions. The tools were not usually designed with extensibility in mind and have differing interfaces with which users must learn to cope. Nevertheless, more recent developments such as Electric [Rubin 1983] have recognized that this is important and provide extendable interfaces.

Interfaces to current design tools are either textual, such as LAP (a circuit LAyout Package) [Locanthi 1978], or graphical, as in Electric [Rubin 1983]. There have been attempts at mixed modes where designs can be entered textually or graphically [Trimberger 1981]. Current production systems use graphical interfaces, since design is essentially a topological problem.

Layout editors help automate the specification of photolithographic masks which are used in the fabrication of integrated circuits. Originally this specification was done by hand, on plastic sheets using colored pens, requiring huge investment of time. Even minor modifications were impractical. Layout editors have evolved from simple CIF editors like Caesar [Ousterhout 1981] to symbolic editors such as Mulga [Weste 1981]. Caesar allows the manipulation of mask geometry but has no concept of connectivity of a circuit. Caesar does provide minimal hierarchy but only in the sense that any number of mask objects can be grouped together to form larger objects called cells. Caesar has provided a basis for LSI and VLSI designs in universities and an initial framework for commercial layout systems.

Systems such as Mulga deal with symbolic mask information. A symbolic mask representation keeps the mask information to a minimum by excluding mask layers which are implied by the existence of other mask layers. The excluded mask layers are then automatically generated when CIF is required for chip fabrication. Also, a circuit is specified by connectivity. That is, designers no longer deal with mask interaction; instead they deal with wire and transistor connectivity to specify the circuit. This nullifies the need for extractors to determine the net-list of a circuit. It also makes the interface to simulators much more practical since information the user has specified is not thrown away at the mask level design stage.

Current commercial design systems are workstation systems and come with a variety of editors for working in the layout and logic domains of a design. However, these systems provide minimal configuration management facilities and often keep independent data bases for each domain representation.

1.1.4. Simulation and Testability

There are two areas of concern when dealing with circuit correctness: functionality and testability. Functionality means that the function of the circuit designed is the one originally desired. Testability is the capacity of a circuit to be verified after it has been fabricated.

Traditionally the functionality of a design has been verified by simulators, which vary in levels of detail. Spice [Nagel 1975], for example, gives a very detailed analysis of a circuit, taking into account numerous characteristics for each transistor. This, however, limits the number of transistors that Spice can handle within an acceptable amount of time. To

allow larger circuits to be simulated, switch level simulators such as MOSSIM [Bryant 1981] have been developed.

Establishing a circuit's correctness through proofs is currently being studied as another method to verify the functionality of a circuit design. A number of complex circuits, such as a "computer" [Gordon 1983] and a "network controller" [Birtwistle *et al.* 1986] have been proven. Proving the correctness of a design improves the chances of providing a functionally correct circuit. However, current circuit proof techniques can not take into account the large number of fabrication factors which can cause a circuit to malfunction.

Unfortunately, testability of a circuit has long been an afterthought in the design process. When dealing with simple circuits, test patterns and procedures could easily be formulated and applied; however, with the complexity of designs increasing this is no longer true. New technologies require new testing techniques as certain faults can go undetected unless given specific inputs. Methodologies have already been proposed which address testability from the outset allowing for effective test pattern generation [Williams 1983; Fung and Hirschhorn 1986].

1.1.5. Focusing Design Automation

VLSI design covers a broad spectrum of problem areas, ranging from design methodologies to mask level layout. To fully automate and integrate the design process is a formidable task. However, by choosing a well-understood application domain and hardware architecture, silicon compilation can automate part of this process.

The notion of a silicon compiler was introduced by Johannsen [Johannsen 1979]. He used the term to define two software packages, Bristle Blocks and a Random Logic Compiler (RLC), that used a high-level description of a circuit to automatically generate the mask level layout. For example, the Bristle Blocks compiler accepted a high-level description of a data path chip and generated the circuit layout. Bristle Blocks led to the development of the first commercial silicon compiler, which has met with some success, most notably the " Vax on a Chip " [Rose 1985]. This silicon compiler allowed the design and testing of a chip set for the MicroVax to be completed in seven months as opposed to two years using conventional techniques. A complete discussion of silicon compilation is presented in Chapter 3.

1.2. Scope and Structure of the Thesis

This thesis supports the argument that silicon compilation must be supported by an integrated design environment. The silicon compiler and its environment must also be directed towards a well-chosen application domain.

Chapter 2 introduces the notion of an integrated environment for software and hardware development. This includes a detailed explanation of Electric [Rubin 1983] which provides the underlying support structure for EFIDO's tools.

Chapter 3 introduces in greater detail the concept of silicon compilation. A detailed definition of a silicon compiler is given, a suitable high-level specification form is presented and the basic design of a silicon compiler is outlined. This is followed by a brief history of silicon compiler development.

Chapter 4 presents an analogy between software and silicon compilation. It is argued that silicon compilation must be supported by an integrated environment and that choosing a proper application domain is essential. To illustrate these ideas the application of silicon compilation techniques to the problem of digital filter design is considered.

Chapter 5 gives a rudimentary introduction to digital filter design. There is a discussion on the implementation of digital filters using various hardware architectures. This is followed by a detailed description of bit-serial hardware architecture and how it can be used to implement digital filters.

Chapter 6 discusses implementation considerations and design of the EFIDO system, which consists of a data flow representation of digital filters, an interface to the Digicap [Turner 1984] simulator, the generation of an intermediate bit-serial form, a bit-serial simulator, and a bit-serial silicon compiler. The design of the EFIDO interfaces is presented, and implementation of the bit-serial cell library is outlined. Detailed descriptions of the digicap interface, bit-serial simulator and bit-serial silicon compiler are then given.

Chapter 7 shows the use of EFIDO using an example of the design of a Low Pass Filter (LPF). Also, to demonstrate EFIDO's ability to handle other types of filter structures a Lossless Discrete Integrator (LDI) filter is implemented.

Chapter 8 summarizes the work done by giving an account of the system capabilities and restrictions, accomplishments and future research possibilities.

CHAPTER 2

Integrated Environments

An integrated environment provides the tools required to move from an initial problem description to a solution in a consistent and efficient manner. This type of environment requires that all tools work from a central representation and have a common interface. By working from a central representation, consistency is maintained throughout the development process. Also, the use of a common interface allows users to move from tool to tool with ease since all tools work from the same representation.

The development of integrated environments in the software domain has been an evolutionary process. This chapter presents examples of integrated environments for software development and integrated environments for VLSI design. This includes a detailed discussion of Electric, which provides a basic framework for CAD tool development and is used as the development environment for EFIDO. This is followed by a discussion of the differences between software and hardware environments.

2.1. Integrated Environments For Software

The LISP Machine environment [Moon and Weinreb 1981] is an integrated environment for the development of large and complex software systems. This environment simplifies the implementation of large software systems by providing tools such as a text editor, incremental compilation, and good debugging facilities. The text editor is specific to LISP code development providing such things as parenthesis balancing and function

search, which automatically edits a function's source code. Incremental compilation is the compilation of individual functions within the editor allowing the implementation and testing of low-level functions which can then be used to build higher-level functions. The debugging facilities include a debugger and a data structure inspector. The debugger provides stack traces, variable checking, minor fixes such as variable binding, and restart of execution. The data structure inspector allows the programmer to inspect and modify all instantiated data types. All these tools work together to allow a programmer to quickly implement a working prototype.

A programmer's productivity is improved by giving a quicker response, a more consistent view of the environment (i.e. a common interface to all tools), and incremental improvement capabilities. Programmers are able to focus on the problem rather than having to deal with the environment.

2.2. Integrated Environments For VLSI Design

Integrated environments are emerging for VLSI design. These environments are being developed to unify the diverse areas of the design process, allowing designers to better explore the solution space of a design.

2.2.1. Palladio

Palladio [Brown, Tong and Foyster 1983] is an exploratory environment for circuit design tools and languages. Palladio was developed on the following observations:

- circuit design is a process of incremental refinement;
- it is an exploratory process in which design specification and and design goals co-evolve; and
- most important, circuit designers need an integrated design environment that provides compatible design tools ranging from simulators to layout generators, ...

These observations led to the development of a design environment which provides:

- an interactive graphics editor,
- a behavioural rule editor,
- an event-driven simulator that uses the behavioural and structural specifications of a circuit to simulate it,
- a frame-based mechanism for assigning multiple perspectives to components,
- a protocol for creating new structural and behavioural perspectives based on Palladio's object-oriented paradigm,
- mechanisms for implementing rule-based, expert-system design aids.

A design in Palladio consists of perspectives which provide a conceptual model of the behaviour or structure of a circuit. The association between perspectives is left to the designer's discretion and allows any number of behavioural perspectives to be associated with a structural perspective and vice versa. This allows designers to have a very abstract view of the behaviour or structure of a circuit early in the design process and a more detailed perspective later.

Palladio uses an object-oriented paradigm to define all circuit components and tools. Each component or tool is thought of being an object which can have associated with it various attributes. For example, an *end-point* object may have static spatial coordinate attributes and a move operation, which is active. Also, objects may be required to have certain attributes which allows all objects to be treated identically. For example, *wire* and *transistor* objects would both have a *draw* attribute which in turn simplifies the display of objects as they already know how to do this when called. Object-oriented programming helps to abstract the problem simply and in terms which are familiar to the programmer.

2.2.2. The Electric CAD Environment

Electric [Rubin 1983] is an integrated framework for VLSI Computer Aided Design (CAD) tool development. At Electric's core is a general data base representation and standard user interface. The tool developer uses these two features by defining technologies, which are graphical representations of the information manipulated by the user, and design aids which work on the data base representation of the technology information. For example, a Complementary Metal Oxide Semiconductor (CMOS) layout technology is used by the circuit designer to define the mask level layout of an integrated circuit. The circuit can then be design rule checked using the design rule checking analysis aid which highlights any design rule violations.

The addition of a new technology requires the specification of a set of basic graphics primitives consisting of nodes and arc prototypes. A node prototype such as a transistor consists of port, graphical, and electrical layer descriptions. Arc prototypes are defined in a similar manner, however an arc only requires a layer description and which layers may be connected. The node and arc prototypes can now be used to define instances of the prototypes. These instances are placed inside complex nodes which are prototypes inside of a library.

The actual manipulation of a technology's prototypes is done by analysis aids. Analysis aids are often technology sensitive, such as the design rule checker, so they can be aimed at particular design needs. For example, the user aid allows the graphical manipulation of the technologies, or a design rule checking aid will check for mask level design rule violations within a complex node prototype.

The implementation of an analysis aid requires the specification of the routines outlined in Figure 2-1. Each routine defines a major data base action, allowing a particular analysis aid to continually monitor the design process, or allowing an analysis aid to be invoked on a specific section of a design.

The analysis aids can be active all the time or only invoked at the request of the designer. If an analysis aid is active, the appropriate routine in Figure 2-1 is called to handle a data base change. For example, if the design rule checker is on and the designer places a new arc instance (eg. a metal wire in CMOS technology), then the *newarcinst* routine of the design rule checker is called to check the arc for possible design rule violations within the current cell.

```
aidname_init()                {}
aidname_set()                 {}
aidname_examinecell()         {}
aidname_slice()               {}
aidname_done()                {}
aidname_startbatch()          {}
aidname_endbatch()            {}
aidname_newnodeinst()         {}
aidname_killnodeinst()        {}
aidname_modifynodeinst()      {}
aidname_newarcinst()          {}
aidname_killarcinst()         {}
aidname_modifyarcinst()       {}
aidname_newportproto()        {}
aidname_killportproto()       {}
aidname_modifyportproto()     {}
aidname_newnodeproto()        {}
aidname_killnodeproto()       {}
aidname_modifynodeproto()     {}
```

Figure 2-1 Electric Analysis Aid Routines

If an analysis aid is turned off it can be invoked on the cell currently being edited. For example, a designer requiring a Spice simulation can simply invoke the Spice simulation aid on a cell after it is complete.

Electric provides a framework for an integrated design environment by providing a common data base representation of a design and a common user interface to all design tools. Just as the LISP Machine environment is tailored towards LISP code development, tailoring an environment such as Electric will allow the development of a reasonable environment for a particular circuit design problem.

2.2.3. Discussion of Environments

It has been shown [Buchanan 1980; Walker and Thomas 1985] that VLSI design can be broken into three domains: behavioural, structural, and physical. Palladio addresses the behavioural and structural domains, however it includes the physical domain in the structural specification. Electric, on the other hand, concentrates on the structural and physical domains and ignores the behavioural domains. Both Electric and Palladio can be extended to make up for these deficiencies, however new approaches which closely integrate all three domains are being proposed and implemented [Birtwistle *et al.* 1984; Zippel *et al.* 1984; Lusky *et al.* 1985].

2.3. Summary

Integrated environments for software development have been models for integrated VLSI design environments. Software environments focus on the support of a particular language, not all languages. On the other hand current VLSI environments attempt to support the whole design process.

The next chapter introduces silicon compilation which focuses on the automation of part of the design process.

CHAPTER 3

Silicon Compilation

Integrated circuit design encompasses a large number of interrelated tasks such as problem specification, logic design, simulation, testability, floorplanning, component design and routing. Silicon compilation assists circuit designers by transforming a high-level specification to a mask level layout thus tying together different levels of the design process, such as design specification, floor planning, and routing.

Silicon compilation allows chip designers to explore several design alternatives without a large commitment of time for mask level layout. Also, silicon compilation permits experts in application areas, such as digital filter and data path design, to realize hardware solutions to their problems without becoming expert designers.

This chapter defines what a silicon compiler is, and proposes criteria an application area suitable for silicon compilation techniques. The basic requirements for a usable silicon compiler are then outlined followed by a detailed discussion of current applications for silicon compilation and a brief survey of previous work.

3.1. Silicon Compilation

Silicon compilation maps a high-level specification onto a target hardware architecture. The form of the high-level specification defines a silicon compiler to be either behavioural or structural. A behavioural specification is

an expression of the relationship between the inputs and outputs. On the other hand, a structural specification is an expression of a circuit's subcircuits and their connections [Goldberg, Hirschhorn and Lieberherr 1985].

3.1.1. Behavioural versus Structural Specification

The strengths of behavioural specification lie in the ability to express a problem in terms specific to a particular application area, and the ability to abstract such an expression easily. For example,

$$x = a * b \qquad \{3.1\}$$

is a behavioural expression where the output x is the product of a and b. This type of expression would be well-suited to a mathematical application and would allow circuits to be designed by someone with little knowledge of circuit design principles. Abstraction is then simply a case of using previously defined variables to make more complex expressions such as

$$z = 2 * x \qquad \{3.2\}$$

where the value of x has been defined by Equation 3.1.

The weaknesses of behavioural specification are the lack of designer control and the size of the possible solution space. For example, the multiplication in Equation 3.1 requires some restrictions to the word size of variables a and b to avoid overflow problems. Also, the hardware implementation of Equation 3.2 could add another multiplier to the circuit. However, implementing a shift left, or using the multiplier, which was defined for Equation 3.1, could result in a savings in circuit area. Such problems can be overcome by supplementing the behavioural description with structural information. The structural information not only gives the designer more

control; it can also be used by the silicon compiler for circuit optimizations.

The advantage of a purely structural specification is the ability to control the final hardware implementation. For example, one possible structural specification implementing Equation 3.1 is:

```
procedure mult-section (x, y):
begin
        define a bit-serial multiplier section from subcircuits
end

procedure mult (n, input1, input2, output):
begin
        x ← 0;
        y ← 0;
        QUEUE ← 0;
        for i ← 1 until n do
        begin
                concatenate mult-section(x,y) to end of QUEUE
                x ← x + width of mult-section;
        end
        return(QUEUE);
end

mult-1 = mult(12, a, b, x);
```

This specification describes a circuit that is bit-serial consisting of 12 stages. The designer requires extra knowledge, in this case about bit-serial architectures, to implement a circuit. However, the designer has complete control of the structure of the circuit with the behaviour imbedded in the specification.

The weaknesses of structural specification are the large amount of circuit design knowledge required by the user, and the concealment of the behavioural specification. More design control places the onus of correct circuit structure on the designer. This requires a better understanding of the underlying circuit architecture, thus limiting the use of structural silicon compilers to circuit design experts. In a structural description, the actual

behaviour of the circuit may not be entirely obvious. It is often necessary to expand the specification in some way to clarify the circuit's behaviour.

3.1.2. Specifications Using Data Flow Graphs

The input form for both structural and behavioural specifications is usually textual, requiring users to learn a new language for expressing their application problems. An alternative is to use graphical input in the form of data flow graphs. It has been noted that [Mostow 1985]

- dataflow graphs abstract away extraneous details found in program-like textual representation ...
- new information can be added to an annotated dataflow graph in small increments ...
- and a data flow graph can be executed on symbolic and test-case data.

The use of data flow graphs for silicon compilation was first suggested by [Keller, Lindstrom and Patil 1980]. They are already used in areas such as digital filter design and microprocessor design to describe system interactions [Rabiner and Gold 1975; Siewiorek, Bell and Newell 1982]. By using graphical data flow specifications a larger community of system designers could immediately use silicon compilation to realize physical implementations of their systems.

Data flow graphs can be used to capture both behavioural and structural specifications. A data flow graph uses nodes to represent functions performed on the data, and directed arcs to indicate the direction of data flow. Structural aspects of the underlying digital circuit system are implied by the interconnection of the nodes, while the behaviour is represented by the node functions. For example, Equation 3.1 can be represented by the data flow graph in Figure 3-1. Depending on the form of the data flow representation

and its extendability Equation 3.2 can be represented by either of the data flow graphs in Figure 3-2. The final form of the data flow description is dependent on the application domain of a silicon compiler and the level of detail required in the specification.

3.1.3. Application Domain Specification

The choice of the application area is important to silicon compilation [Johannsen 1979; Siskind, Southard and Crouch 1981; Bergmann 1983]. If the domain encompasses too wide an area, practical solutions will be difficult to find. Likewise, if the domain is too narrow the solutions will be trivial and can be handled using existing design techniques. The main criteria for

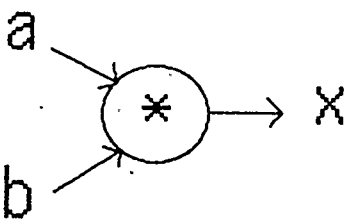


Figure 3-1 Data Flow Description of Equation 3.1

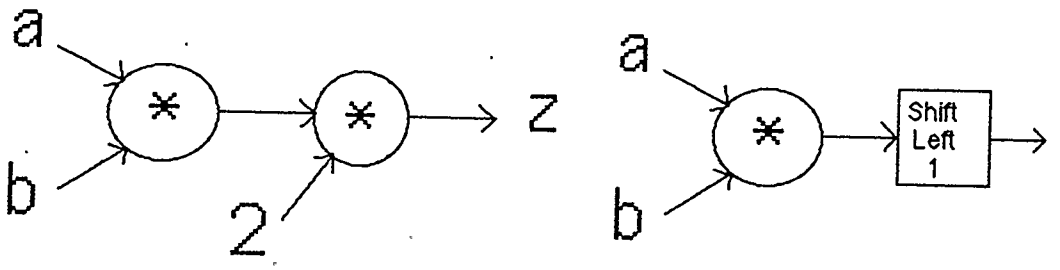


Figure 3-2 Data Flow Description of Equation 3.2

choosing an application domain are that:

- it have a theoretical foundation,
- and a strong structural relationship between the problem specification domain and the architecture.

A theoretical foundation implies that there already exists a formal way of expressing the problem, such as boolean equations for logic or z-transform and difference equations for digital filters. A strong structural relationship between the specification and architecture ensures that the mapping to the hardware implementation is realistic. For example, random logic can be mapped onto a number of hardware architectures, such as programmable logic arrays, gate arrays, or read only memories, making the choice of implementation style more difficult. On the other hand, a digital filter can be specified as a collection of adders, time delays, and multipliers [Jackson, Kaiser and McDonald 1968], which can be provided as architectural building blocks.

3.1.4. The Basics of a Silicon Compiler

The mechanics of silicon compilation can be generalized as follows:

- transformation of the specification to an intermediate form,
- virtual placement and routing of circuit components,
- generation of the mask level layout descriptions of the components,
- physical routing of internal signals,
- and insertion into pad frame.

A central representation provides consistency during design. More importantly "an intermediate form is needed that retains all of the original structure inherent in the design" [Liblong 1984]. Consistency is maintained by requiring the user to update a single specification and use only this specification to derive input for the design verification tools and the silicon

compiler.

There is a strong interaction between the placement of components and the routing between these objects. For example, if a wiring channel is used for interconnects, the width of the channel cannot be determined until the routing is complete, and the routing cannot be determined until the placement of components is complete. To solve this dilemma, virtual routing and placement are used to assign relative locations to objects and then relative wiring tracks. This allows trade-offs to be made before actually committing to true physical positions.

Just as a software compiler is directed towards a particular instruction level, a silicon compiler must be directed towards a particular hardware architecture. This allows the development of one or more appropriate floor plans to allow efficient implementation of the designs being proposed [Johannsen 1979; Bergmann 1983] Once a hardware architecture is chosen, issues such as basic component design and automatic macro cell generation can be addressed.

The form of the basic components library will be affected by the hardware architecture and the implementation style of a silicon compiler. Basic component design will be directed by the hardware architecture. For example, a carry look ahead adder is useful in a parallel hardware architecture but is of little use in a bit-serial hardware architecture. The implementation of the basic components will depend on a compiler's implementation style. For example, Johannsen's Random Logic Compiler (RLC) [Johannsen 1981] implements basic components as parameterized procedures, whereas the FIRST [Bergmann 1983] compiler uses CIF templates. A library of carefully

selected and properly implemented basic components will simplify the compiler development task.

Macro cell generation helps to minimize the size of the basic component library. With a well-defined set of building blocks in the component library it is possible to parameterize the specification of larger circuit sections used in the design. These procedures should be implemented as process-independent as possible, to enable the transition between process technologies (i.e. CMOS to NMOS) to be simply a matter of using a new basic component library.

There are various types of routing, such as river routing, or channel routing, and the routing used by a silicon compiler will depend on the floor plan. For example, the FIRST [Bergmann 1983] compiler provides a routing channel which simplifies routing to be the connection of signal end-points within this channel. With routing complete the circuit can be placed inside a pad frame for fabrication [Johannsen 1981; Bergmann 1983; Kroeker, Birtwistle and Esau 1984].

3.2. Previous Work

The term silicon compiler was coined by David Johannsen [Johannsen 1979] to describe two silicon design tools, the Random Logic Compiler and the Bristle Blocks compiler. When using RLC, circuits are specified as collections of NAND, NOR, and INVERTER gates. Once a design has been specified, RLC allows it to be simulated at the gate level, or have the mask level layouts generated for a number of technologies (NMOS, CMOS etc.). Bristle Blocks deals with the compilation of data path chips. It accepts a high-level structural language description of the register transfer functions for a given data path and compiles this into the mask level layouts. Both RLC

and Bristle Blocks were written in ICL (Integrated Circuit Language) [Ayer 1979], a graphical layout language developed at Caltech.

The development of Siclops [Hedges *et al.* 1982] at Caltech was a follow-up to the work done by Johannsen. The two most notable design improvements in Siclops are using a portable programming language (MAINSAIL) for implementation, and making each piece of the compiler (eg. data path compiler, Programmable Logic Array (PLA) generator etc.) a stand-alone tool. Since MAINSAIL is available on a wide range of computer systems Siclops is available to a larger user community than was Bristle Blocks. The division of the compiler into stand-alone pieces allows the compiler implementation to be distributed among a number of programmers and allows pieces to be modified or updated without affecting other sections of the design system.

The MacPitts [Siskind, Southard and Crouch 1981] silicon compiler developed at the Massachusetts Institute of Technology (MIT) was designed to handle data path chips. The compiler uses an algorithmic definition in a LISP-like language called MacPitts. A chip definition for the MacPitts compiler is a functional specification of a number of processes that can be executed in parallel. If the chip specification passes syntactic and semantic checks the compiler then proceeds to extract the data path, sequencer, and control information into a well-defined intermediate form. The data path information determines what registers and operators are required to implement the data path. The sequencer is the control unit of each of the finite state machines. This control unit can be in one of three forms: a sequencer (logic control), a sequencer with a counter, or a sequencer with a

counter and a stack. The control extraction simply converts source code to boolean equations describing the interaction between the data path, the sequencers, and the peripheral chips. The intermediate form can then be used to drive the simulator or the physical layout tools (Router, Weinberger Array generator etc.).

The FIRST (Fast Implementation of Real-Time Signal Transforms) silicon compiler [Bergmann 1983] is designed for a specific problem area and hardware architecture. The application area is confined to Real-Time Signal transforms and the designer is constrained to a bit-serial architecture with a fixed floor plan.

The designer must generate a FIRST structural language description of the filter, that is then compiled giving a listing and an intermediate form description. Syntactic errors are highlighted in the listing and can be used to correct the filter description for recompilation. The intermediate form is then used as input to either a simulator, to check for bit-serial timing errors, or a layout program, to generate the mask geometry.

3.3. Summary

The design of VLSI chips is now pushing current design systems and methodologies to the limit. With the introduction each year of larger chip sizes and denser circuits on these chips, handling the complexity becomes a major chip design problem. Just as the introduction of compilers for high-level languages allowed programmers to handle more complex software systems, silicon compilers will allow designers to handle larger circuit systems for specific problems effectively and easily by allowing them to concentrate on global structure.

A silicon compiler allows designers to work at a high level, preferably with symbols and terminology that are familiar. This allows some detail to be temporarily ignored while dealing with global optimization of the system. This is particularly important for VLSI, since [Mead and Conway 1980]

> "...the area occupied on the silicon surface by circuitry is far more a function of topological properties of the circuit interconnections than it is of the number of logic gates implemented."

The automation of the layout phase is the key to silicon compilation. The layout phase, when done by hand, is prone to human design errors and is often inflexible, not allowing minor changes without complete redesign of major portions of the chip. Silicon compilation requires only the hand design of the basic components of the system. Once designed and checked, the placement and connection of the components are handled by the compiler, which produces a layout which is correct by construction.

The next chapter pursues the analogy between software and silicon compilation.

CHAPTER 4

Software and Silicon Compilation: An Analogy

Silicon compilation techniques were motivated by the same objectives that motivated software compilation twenty years ago. Software compilers, and the high-level languages which they use as input, were developed to allow programmers to think of and generate solutions to their problems at a higher level. A higher-level description could then be compiled into the machine code of the computer being used, automating the assembly level programming phase, allowing the detection of errors, and generating messages about these errors in terminology familiar to the programmer. Initially this approach resulted in inefficient assembly code which often gave erroneous results, but after twenty years software compilation now produces very reliable and efficient assembly code. The efficiency of compiler generated assembly code is now to the point where writing machine code (in a large software project) is only considered under very special circumstances. This has allowed the development of large complex software systems and the ability to use the same high level code on a number of different machines and different generations of the same machine.

This chapter presents an analogy between silicon and software compilation. This is followed by a discussion of an environment for silicon compilation.

4.1. Compilation

Languages used by software and silicon compilers allow users to deal with their problems at a higher-level. As stated by [Whitehead 1911]:

> "By relieving the brain of all unnecessary work, a good notation sets it free to concentrate on more advanced problems ..."

Giving the user of a software compiler a good language [Goldberg, Hirschhorn and Lieberherr 1985]:

> " ... can lead the programmer to the correct solution of a problem in a natural and easy manner."

Likewise, a silicon compiler with a good language moves designers away from device-level considerations allowing them to deal with architectural concerns [Goldberg, Hirschhorn and Lieberherr 1985; Johannsen 1981].

Many of the software techniques used for building a software compiler apply to silicon compiler development.

> "For example, source-to-source transformation techniques are very useful for translating a user-oriented input language to its lower level, processing-oriented intermediate forms [Goldberg, Hirschhorn and Lieberherr 1985]."

These techniques are well understood for software compilation and can be applied immediately to silicon compilation [Johannsen 1981; Siskind, Southard and Crouch 1981; Bergmann 1983].

Both software and silicon compilation make use of templates. Johannsen noted [Johannsen 1981]:

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

Similarly, silicon compilers use templates called floorplans which map certain language constructs into a particular mask level specification.

The output of a compiler is correct by construction [Johannsen 1981]. During the development of a software compiler attention must be paid to ensure that the machine code produced is correct. Once implemented though, a programmer will never check whether the machine code produced is correct. Similarly, a user of a silicon compiler never needs to design rule check a layout generated by a silicon compiler.

Design consistency is maintained in software and silicon compilation since [Goldberg, Hirschhorn and Lieberherr 1985]:

> "In both cases, the interaction is done through the compiler input language. "

This implies that all changes to a design are made to the source code. For example, in the early days of computing, compiled code was sometimes modified. Now the developer of a large software system would never consider such action since the changes may cause side effects which could never be properly traced down and corrected. Likewise, a VLSI design implemented using a silicon compiler should only be modified in the source code [Johannsen 1981; Goldberg, Hirschhorn and Lieberherr 1985].

Software and hardware design are processes of refinement. Johannsen notes that [Johannsen 1981]:

> "... compilers make changes affordable, giving the designer freedom to explore many design strategies. "

A software designer is unwilling to discard large pieces of painstakingly handcrafted machine code. On the other hand a half day's worth of high-level code implementation which was used to explore a new idea can be discarded with little concern to the time invested. Similarly a hardware designer is unwilling to discard handcrafted mask level layout. Silicon

compilation, however, allows the exploration of new ideas without an excessive input of time.

The design and implementation of large software systems has shown the need for integrated environments which support the software compilation process [Moon and Weinreb 1981; Teitelman and Masinter 1981; Weekly 1985]. These environments provide programmers with tools, such as debuggers and data structure inspectors, to implement correct specifications of their problems.

Analogously, silicon compilation must be supported by an integrated environment. Silicon compilation without the benefit of editing tools, simulators, test pattern generators, etc., would be the same as trying to develop a large software system with no access to powerful editing tools, debuggers, source and object code management facilities, etc. Building on the experiences of software compilation, silicon compilation must be properly supported to allow designers to handle large and complex hardware designs.

Furthermore, a language for a silicon compiler must focus on a specific application domain, just as a software language usually addresses a particular application need. For example, LISP addresses *list* processing and Fortran addresses *numerical* processing [Ledgard and Marcotty 1981]. This helps to target a silicon compiler environment to the application domain, giving designers better tools with which to attack a problem.

4.2. An Environment for Silicon Compilation

An environment for silicon compilation should have:

- a consistent user interface,
- a single design specification format,

- and verifiers.

A consistent user interface implies that the interface to all tools should be similar. This may be impractical since some tools may not even run on the same machine as the compiler environment (eg. special simulation hardware). Also, it may be impractical to rewrite an existing tool to fit into the design environment. In cases such as these the inconvenience to the user should be minimized by generating input files for the design tool from the silicon compiler's intermediate form.

A single design specification used by all tools maintains consistency throughout the design process. A single specification can be used as a repository of information where design data generated in one phase of the design process (eg. cell placement) can be used later by other tools such as routers.

A silicon compiler must be supported by verification tools. Silicon compilation currently requires simulators to verify a circuit's functionality. If a silicon compiler uses a very high level of specification, a number of simulators may be required. At the highest level a functional simulator may be required to verify the specification, while at the lowest level a device level simulator may be required for basic cell characterization.

4.3. Summary

The analogy between software and silicon compilation presented demonstrates that silicon compilation should be supported by an integrated environment. In general, any compilation process is enhanced when supported by an integrated environment. For instance, text formatting is another type of compilation process. In the past text formatting has required preparing a

document source file (i.e. problem definition), formatting the source (i.e. compilation) and printing a hardcopy (i.e. fabrication). Now, however, on-line document preparation environments which support document compilation are emerging.

The remainder of this thesis illustrates these ideas by presenting the implementation of EFIDO, an integrated environment which supports silicon compilation for digital filters.

The next chapter introduces the area of digital filter design. Architectural alternatives for implementing digital filters are presented, as well as a description of bit-serial solutions for digital filters.

CHAPTER 5

Digital Filters

This chapter provides an overview of digital signal processing theory. First a brief history of the development of the area of digital signal processing is presented along with an outline of the function of a digital filter. The steps in specifying a filter are highlighted with an example filter design. The architectural alternatives for realizing a digital filter are presented, as well as a overview of bit-serial architecture.

5.1. History of Digital Signal Processing

Signal processing originally involved handling continuous amplitude (analog) signals. In the late 1940's and early 1950's (paralleling digital computer development) study was initiated into the use of binary numbers to represent discrete amplitude levels [Rabiner and Gold 1975]. This representation evolved into the field of digital signal processing and a solid theoretical basis quickly developed, with immediate applications in the interpretation of seismic data. However, early digital hardware could not even handle speech with sampling rates of 5 to 8 kilohertz (kHz), thus limiting real time applications development.

In the 1960's integrated circuit technology increased circuit speeds. This led to the development of digital filters for a number of low frequency applications such as speech synthesis and telecommunications (8 to 10 kHz). The increased speed and reduced size of general purpose computers in the 1970's allowed them to be used for digital signal processing. In the late 1970's

and early 1980's, VLSI technology has permitted high-bandwidth applications such as video and radar (10 megahertz (Mhz) or more) to be addressed [Rabiner and Gold 1975].

5.2. Digital Filter Applications

A digital filter is described in [Rabiner and Gold 1975; Liu 1982] as,

> "an implementation of a discrete time filtering process which executes on either special purpose digital hardware or a general purpose digital computer and a discrete time filtering process refers to any process that performs arithmetic operations on an input sequence $y(k)$".

A conceptually simpler description is that a digital filter can be viewed as a black box, in which the discrete time filter process occurs. This process can be described by the transfer function which relates the input and output signals. The effect of the transfer function on the signal is to pass some frequencies of the signals while attenuating others.

Some digital filtering application areas are [Lyon 1980]:

- Filtering and detection of voice signals
- Time varying filters for synthesis (voice and music)
- Adaptive systems for signal noise reduction improvement
- High fidelity audio, stereo, etc.
- Quantization, code conversion, compression
- Geophysical and underwater systems
- High-bandwidth systems: video and radar.

These applications cover a broad range of frequencies and will require a variety of digital filters to improve results in each area.

5.3. How A Digital Filter Works

Digital signal processing is the transformation of a digital signal by a process. A discrete time signal is obtained by sampling a continuous analog signal at fixed intervals, where T_s, is the sampling rate. "The quantization of

the magnitude of this discrete time signal transforms it into a digital signal" [Rabiner and Gold 1975; Liu 1982]. Figure 5-1 shows this transformation diagrammatically.

There are a number of non-ideal effects that must be considered such as multiplier coefficient quantization and internal signal quantization which can result in nonlinear behaviour from a filter. The end results of these effects are the deviation of the frequency response from the ideal, the possible introduction of undesirable limit cycle oscillation, and the generation of roundoff noise. A thorough discussion of these problems is beyond the scope of this thesis and the interested reader is referred to [Rabiner and Gold 1975]. The following discussion deals only with ideal filters which are single input, single output, causal and linear shift-invariant that are defined by:

$$y(m) = \sum_{k=0}^{\infty} h(k)x(m-k) \qquad \{3.1\}$$

$h(k)$ - is the unit sample or impulse response of the filter.

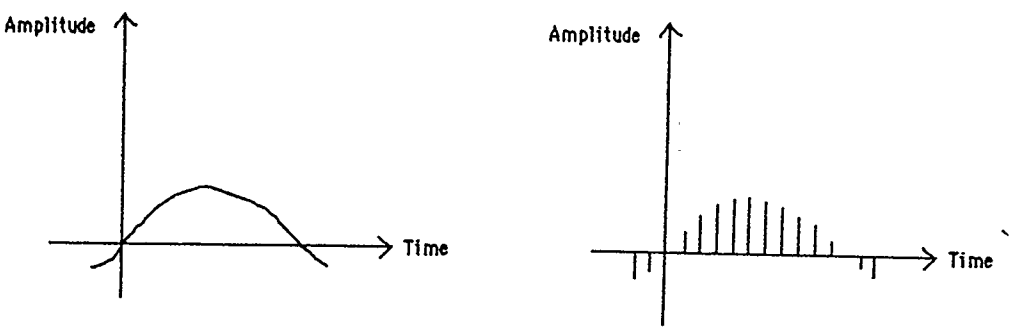


Figure 5-1 Analog Signal and Corresponding Digital Signal

Some examples of what a digital filter might do are shown in Figures 5-2 and 5-3 as plots of frequency plotted against magnitude response. Figure 5-2 shows a low pass filter (LPF), which allows only frequency components below a certain frequency F_s (stop frequency) pass. Figure 5-3 is a band stop filter which passes all frequencies within a specified range and attenuates all frequencies above and below this range.

When the filter process is restricted to a single input, single output, causal and linear shift-invariant it can be shown that the output $y(m)$ at any time instance m can be found by [Rabiner and Gold 1975; Antoniou 1979]:

$$y(m) = \sum_{k=0}^{m} h(k)u(m-k)$$

where

$u(k)$ - is the input sequence and

$h(k)$ - is the unit sample or impulse response of the filter.

If the input is a unit sample

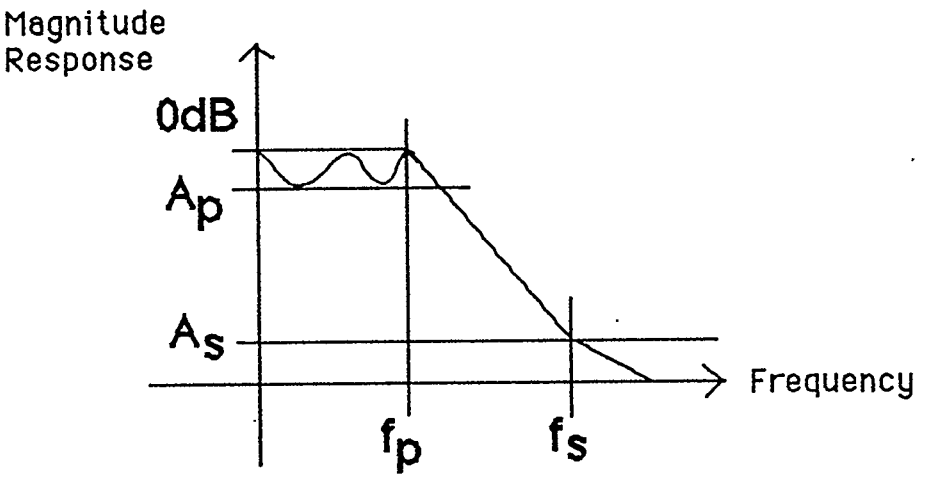


Figure 5-2 Magnitude Response for a Low Pass Filter

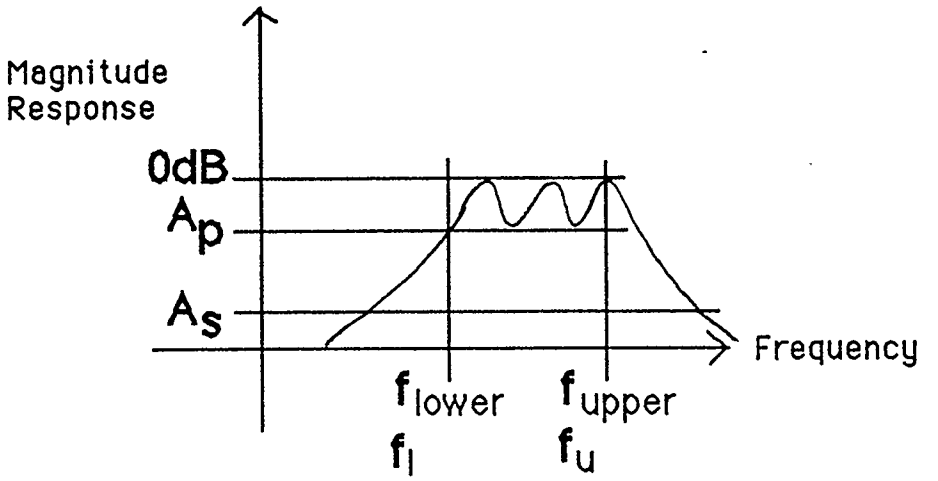


Figure 5-3 Magnitude Response for a Band Pass Filter

$$u(m-k) = \begin{cases} 1 & \text{if } m-k = 0 \\ 0 & \text{if } m-k \neq 0 \end{cases}$$

then $y(m) = h(m)$ which implies that $h(m)$ completely characterizes the digital filter [Rabiner and Gold 1975].

If an linear shift-invariant filter is defined with $h(k)$ as an infinite sequence then it is called an Infinite Impulse Response (IIR), or recursive filter. If $h(k)$ is finite a linear shift-invariant filter is called a Finite Impulse Response (FIR) filter, or non-recursive filter. The type of filter corresponds directly to the filter structures that can be implemented to realize the filter in practice [Rabiner and Gold 1975].

5.4. The z-transform

The z-transform is a useful way of representing and manipulating sequences. Some of the desirable properties of the z-transform are linearity, delays, and simplification of convolution of sequences [Rabiner and Gold

1975]. Linearity implies that the z-transform for:

$$ax_1(n) + bx_2(n)$$

is

$$aX_1(z) + bX_2(z)$$

for all real a and b. The delay property is useful in converting from the difference equation representation of a linear shift-invariant system to its z-transform representation. For example, the difference equation [Rabiner and Gold 1975]:

$$y(n) = x(n) - b_1 y(n-1) - b_2 y(n-2)$$

has a z-transform representation

$$Y(z) = X(z) - b_1 z^{-1} Y(z) - b_2 y z^{-2} Y(z)$$

An important property of the z-transform and of the inverse z-transform (which is the conversion back to a difference equation) is the ability to determine $y(n)$ of an linear shift-invariant system without having to perform a convolution. Since in a linear shift-invariant system with $x(n)$ as input with an impulse response $h(n)$ and $y(n)$ as the output

$$Y(z) = X(z)H(z)$$

where $X(z)$, $H(z)$, and $Y(z)$ are the z-transforms of $x(n)$, $h(n)$, and $y(n)$, the convolution is converted to a multiplication of z-transforms [Rabiner and Gold 1975].

The z-transform can also be used to evaluate the magnitude, frequency, and phase response of an linear shift-invariant filter. Since the z-transform for $h(k)$ is

$$H(z) = \frac{M\sum_{i=0}^{m}(z - z_i)}{\sum_{i=0}^{n}(z - p_i)} = \frac{Y(z)}{U(z)}$$

where p_i are called the poles and z_i the zeros of the filter [Rabiner and Gold 1975] then, for example, the steady state sinusoidal magnitude response is

$$\left|H(e^{jwT})\right| = \left|H(z)\right|_{z=e^{jwT}}$$

where,

w is the discrete frequency variable

e is the base of the natural logarithm and

j is the $\sqrt{-1}$

T is the sampling interval.

With some of the underlying concepts defined, it is now possible to focus on the design of a linear shift-invariant filter.

5.5. A Band Pass Filter

Assuming a frequency response specification is given, defining a digital filter involves the following stages [Rabiner and Gold 1975; Antoniou 1979]:

- Obtain the z-transform transfer function of the filter from the filter specification;
- Select a discrete time filter structure and calculate the filter parameters;
- Realize the discrete time filter using digital techniques. The calculated filter parameter values must be quantized at this stage;
- Analyze the magnitude response due to parameter quantization; and
- Analyze the nonlinear effects due to signal quantization in the digital filter.

Stages 2 to 5 are to be repeated if the quantized filter does not meet the design specification. To highlight this process the development of the specification for a band pass filter follows.

The band pass filter will be a second order filter with a transfer function defined by [Rabiner and Gold 1975; Moore 1978]:

$$y(k) = x(k) + Ay(k-1) + By(k-2) \qquad \{3.2\}$$

Also needed at this point are A_p and f_p (passband attenuation and frequency) and A_s and f_s (stopband attenuation and frequency). With these basics (i.e. filter order, A_p, f_p, A_s and f_s) a digital filter can be completely specified.

Using the fact that $z^{-i}Y(z)$ is the inverse z-transform of $y(n-i)$, equation 3.2 can be rewritten as

$$X(z) = Y(z) - Az^{-1}Y(z) - Bz^{-2}Y(z)$$

which can then be expressed as

$$H(z) = \frac{Y(z)}{X(z)} = \frac{1}{1-Az^{-1}-Bz^{-2}} = \frac{z^2}{z^2-Az-B}$$

A structure is now selected to implement the transfer function of the filter. There are a number of criteria for choosing a filter structure such as reducing the number of coefficient bits, system word length, actual number of multipliers or adders required, and hardware speed. The structure can be obtained from equation 3.2 and is shown graphically in Figure 5-4. This structure is a direct form and was chosen for simplicity, however other structures are possible by rewriting Equation 3.3. A detailed explanation and examples of some of the possibilities can be found in [Rabiner and Gold 1975].

The coefficients can be determined once a structure has been chosen. Two common approaches are bilinear transformation and direct design [Rabiner and Gold 1975]. Bilinear transformations map existing analog filter descriptions which are defined in the s plane to the z plane.

The direct form filter is implemented using the transfer function coefficients directly as filter coefficients. Programs, such as Ladder [Liu 1982], exist to aid the designer in determining these coefficients for particular filter structures. For example, Ladder accepts the filter order, passband frequency, and passband ripple of a digital lossless discrete integrator (LDI) ladder filter structure and returns the multiplier coefficients.

The physical implementation of a filter requires an analysis of the possible architectures available.

5.6. Architectures For Realizing a Digital Filter

Once a filter has been defined by its transfer function it can be implemented in hardware or software. The main factor in the choice of implementation is the sampling rate, the rate at which the filter must process the input signal.

General-purpose programmable computers are one option for digital filter implementation. They can be used for a wide variety of filters and are

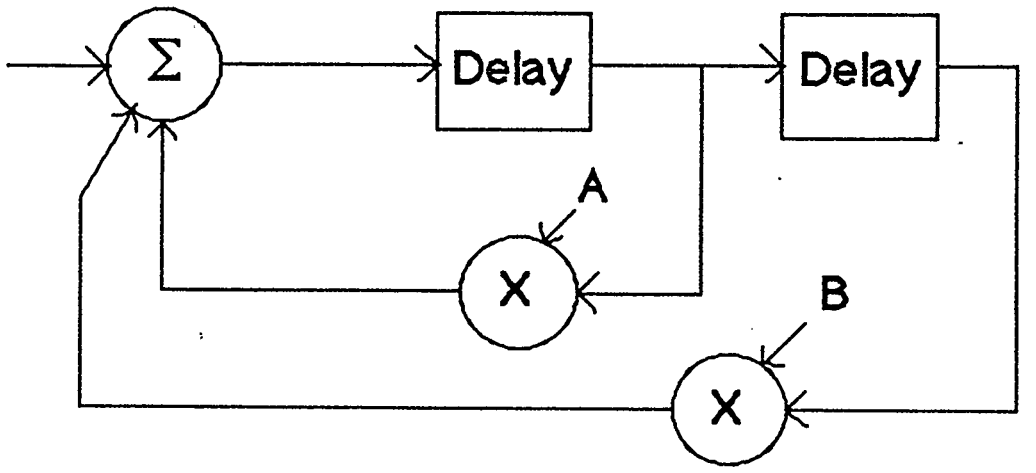


Figure 5-4 Direct Form Structure for Band Pass Digital Filter

relatively inexpensive. The limiting factor are their speed of operation, which is governed by their architectural design and actual hardware implementation (i.e. TTL, Nmos, ECL, etc). Some of these problems are being addressed by special programmable filter chips (eg. the Texas Instruments TMS320) which are aimed specifically at signal processing. The architectures of these chips eliminate some of the overhead of general purpose computers by offering a reduced instruction set and special hardware components designed for signal processing (eg. fast multipliers).

Another choice for digital filter implementation is specialized hardware for a specific filter transfer function. The filter is implemented either using discrete components or a custom designed chip, allowing the hardware to be fine-tuned to the application. Usually one of two architectures is employed: parallel [Norihiko, Tadakatsu and Masanobu 1979] or bit-serial [Freeny *et al.* 1971]. Parallel solutions are the most common and can handle high bandwidth (5-10 Mhz) applications. Bit-serial solutions are less common and can handle medium bandwidth (1-5 MHz) applications. The control strategy and components used in bit-serial solutions tend to be simpler than comparable parallel solutions, often allowing the sharing of operators (eg. multipliers) which in turn reduces the component count or area usage on a chip.

The drawback to hardware solutions is the cost of design and implementation. To offset these costs silicon compilers are being used to map the transfer functions onto a hardware architecture [Bergmann 1983]. The next section outlines the bit-serial architecture in more detail.

5.7. Bit-Serial Architecture

A bit-serial system performs arithmetic operations on a stream of binary digits, each digit represented by a 1 or a 0. Often a fixed point binary number is used where a m+p+1 binary string is represented by

$$B_2 = b_m\, b_{m-1} \cdots b_0 . b_{-1} \cdots b_{-p}$$

with the binary point between b_0 and b_{-1} [Rabiner and Gold 1975; Liu 1982]. The most significant bit (MSB) usually represents the sign of a binary number. Signed fixed point binary numbers commonly represent one of three forms: "sign and magnitude", "one's complement", and "two's complement".

The use of two's complement format can simplify the implementation of bit-serial operators, such as multipliers [Lyon 1976(b)], and the following discussion only deals with the two's complement fixed point binary number representation.

One of the main problems with bit-serial architecture is the variation in operator delay. For example, a circuit consisting of a bit-serial adder and bit-serial multiplier will be bounded by the time it takes to perform a multiplication. The performance of such a system can be improved by pipelining.

Pipelining divides up an operation, such as multiplication, allowing many partly completed tasks to be in progress at the same time. Although the time to complete any one operation is limited by the sum of the times for the partial tasks, the rate at which operations progress through the pipeline is only limited by the time for an individual task [Ibbett 1982; Gosling 1980].

In a bit-serial pipelined system the words arrive least significant bit (LSB) first, with an associated timing signal indicating the LSB for reset purposes. This simple timing scheme can be expanded to deal with timing at various levels within a circuit (Figure 5-5), eliminating the need for a central control unit [Lyon 1981].

Initial studies into bit-serial architecture were prompted by the high cost of early computer hardware. The objective was to develop a low cost general purpose computer such as the Sabrac [Lehman, Eshed and Netter 1963]. However, the introduction of low cost IC components in the 1960's made parallel architectures more affordable and attractive. Bit-serial techniques were not completely abandoned and have been used in areas such as digital signal processing [Lyon 1983] and general-purpose array processors [Fountain 1983].

The work done by Jackson, Kaiser, and McDonald [Jackson, Kaiser and McDonald 1968] laid the groundwork for applying bit-serial architectures to signal processing. Four key features were identified [Jackson, Kaiser and

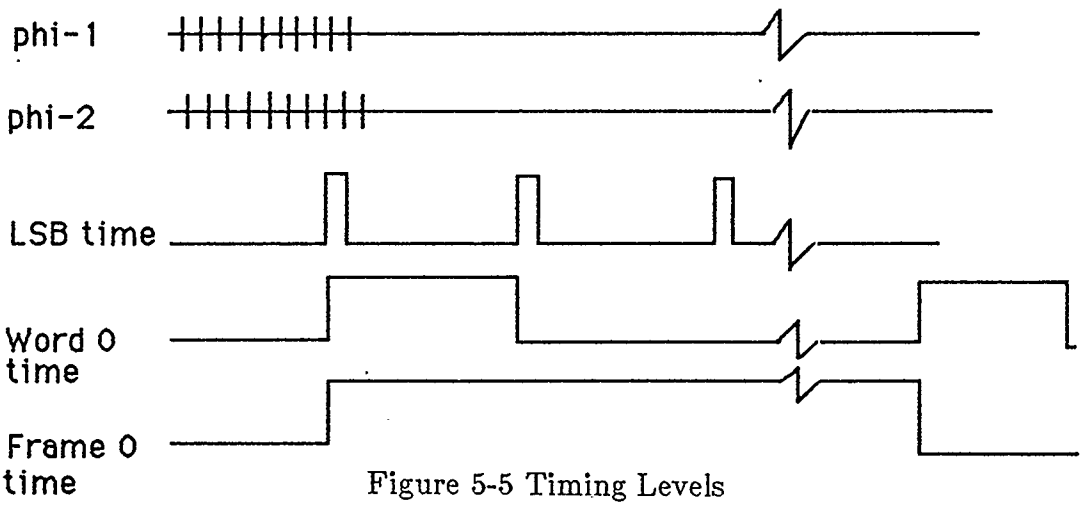


Figure 5-5 Timing Levels

McDonald 1968]:

- The filters are constructed from a small set of relatively simple digital circuits, primarily shift registers and adders.
- The configuration of the digital circuits is highly modular in form and thus well-suited to LSI construction.
- The configuration of the digital circuits has the flexibility to realize a wide range of filter forms, coefficient accuracies, and round-off noise levels (i.e. data accuracies).
- The digital filter may be easily multiplexed to process multiple data inputs or to effect multiple, but different, filters with the same digital circuits, thus providing for efficient hardware utilization.

As noted in [Lyon 1981; Denyer and Renshaw 1985] these features are still applicable today using VLSI technology.

In summary, digital filters are implementations of a discrete time filtering process. Filters can be implemented effectively both in software and hardware. A bit-serial architecture offers the advantages of a simplified control scheme, small operators and exploitation of concurrency through pipelining.

The next chapter discusses the details of the design and implementation of Electric's filter design organizer, an environment which supports a silicon compiler for digital filters.

CHAPTER 6

The System Design and Implementation of EFIDO

This chapter outlines Electric's FIlter Design Organizer (EFIDO). EFIDO helps the digital filter designer to move from a structural specification to a mask level layout easily and quickly. This smooth transition is achieved by providing an integrated set of tools that work from a common representation. Central to this strategy is the bit-serial silicon compiler which automates the tedious and error prone mask layout phase of the design. Of course a silicon compiler is of little use unless a design environment exists that provides the designer with easy access to tools, such as simulators, to do the job properly.

The first section of this chapter outlines a number of implementation considerations. Section Two covers the system design issues of user interface, Digicap simulation, bit-serial simulation, and bit-serial compilation. Original work is contained in:

- overall system architecture,
- definition of appropriate data flow primitives,
- implementation of a bit-serial silicon compiler,
- design and implementation of a number of basic cells,
- design and implementation of the macro cells used by the silicon compiler,
- implementation of a bit-serial simulator,
- automatic generation of a Digicap input file from a data flow diagram,
- and the automatic generation of a bit-serial description from the data flow diagram.

6.1. Design Considerations

6.1.1. A Silicon Compiler For Digital Filters

As outlined in Chapter 3, silicon compilation can be applied to a application domain which has a solid theoretical foundation and has a strong relationship between the specification and implementation architecture. Digital filters, as outlined in Chapter 5 have a theoretical foundation. Also, as noted by [Jackson, Kaiser and McDonald 1968], there is a strong relationship between a filter specification and a bit-serial implementation.

Digital filter design is also restricted in the sense that there are a number of filter structures which can be used to realize a filter, all of which are based on a general description as expressed by [Rabiner and Gold 1975; Moore 1978]

$$y(n) = \sum_{i=0}^{M} b_i\, x(n-i) - \sum_{i=1}^{N} a_i\, y(n-i)$$

where

$x(n)$ - is an input signal or sequence

$y(n)$ - is the output signal

b_i - are the set of M coefficients describing how $y(n)$ depends on the current input sample and the previous M input samples, and

a_i - are a set of N coefficients describing how $y(n)$ depends on the previous N output samples.

Most of these filter structures can then be realized using a relatively small set of components (adders, multipliers and delay elements) and can be represented graphically as a data flow graph.

6.1.2. Bit-Serial Architectural Considerations

As outlined in Chapter 5 there are a number of possible architectures that may be used when implementing a digital filter (i.e. bit-serial, parallel etc.). Bit-serial was chosen for its simplicity in both filter implementation and bit-serial operator design.

Each bit-serial operator can be viewed at a high level as in Figure 6-1 [Lyon 1981]. Even though the bit-serial architecture is simpler, some restrictions, such as a fixed word length of 24 bits, and pipelining, are required. Fixing the system word length limits input values to certain components to a range less than the total number of bits. For example, for each multiplier the multiplicand is in the range $[-2^{21},2^{21})$ and the coefficient must be in the range $[-2^{2k-1},2^{2k-1})$, where k is the number of actual hardware stages [Lyon 1981]. This restricts the coefficient size to a maximum of 12 bits which helps to prevent overflow of the multiplication result.

A pipelined architecture can be defined as " a system which has an operation period less than its operation delay, without parallel processing "

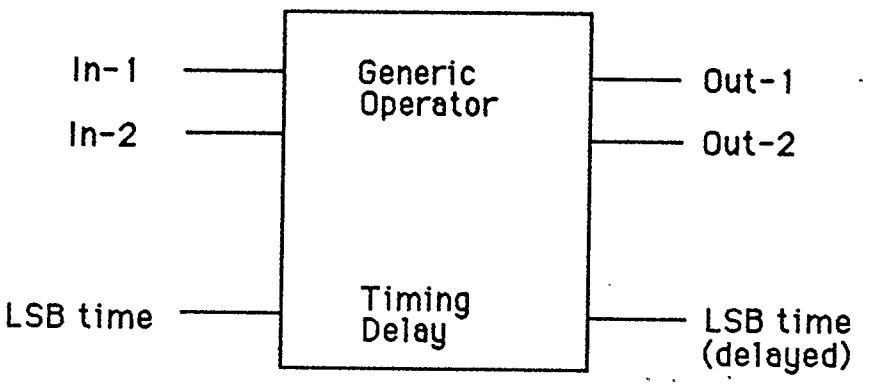


Figure 6-1 Generic Bit-Serial Operator

[Lyon 1976(b)]. As explained in Chapter 5 pipelining helps to divide the processing task so that propagation delays through components are not a major constraint. Also, separating operations, such as multiplication [Lyon 1976(b); Denyer and Renshaw 1985], into stages allows the design of basic sections which are identical for each stage. This turns the configuring of such operators into the simple task of placing and interconnecting the correct number of sections.

Multiplier coefficients are to be supplied off chip. This allows greater flexibility and simplifies the bit-serial compiler's task, but restricts the range of filters that can be implemented, due to the fixed pin count in packaging the chip. With a problem domain and suitable hardware architecture defined, silicon compiler implementation issues can now be addressed.

6.1.3. Other Design Considerations

To allow incremental improvements, a silicon compiler must be modular. This implies that clear boundaries should be drawn between distinct phases of the compilation. By doing this individual areas can be improved upon with minimal impact on other areas. For example, placement and routing can be handled independently, though there may be some interaction between the two phases when attempting to obtain an optimal layout.

The development of a random logic compiler (RLC) [Kroeker, Birtwistle and Esau 1984] also highlighted the need for an effective user interface. The original prototype of RLC required the user to write a piece of code in Simula, and though the code was descriptive, users were not receptive to this form of circuit specification. However, with the implementation of a graphical interface users were less intimidated by the tool and could quickly express

their logic diagrams graphically with minimal training. This influenced the decision to implement a graphical interface for the digital filter compiler from the outset.

RLC had been implemented using a local variant of LAP [Locanthi 1978], a Simula package for integrated circuit layout. LAP can be easily extended which allowed RLC to be developed quickly. Electric is also extendable and provides a much richer mask level circuit design environment. Also, the growing popularity of Electric in Canadian universities and the benefits of a larger community of users made Electric the eventual choice as the development environment for EFIDO.

Electric is a supported environment with a growing community of users. This will allow the use of tools developed by other institutions, as well as provide a vehicle for distribution of EFIDO and other tools developed at the University of Calgary. Also, a number of cell libraries are being developed which can be used to expand the capabilities of the silicon compiler as the libraries become available. More importantly, Electric provides an environment that is easily extendable and has a consistent user interface to all tools. These capabilities have allowed more than the development of a silicon compiler. Adding interfaces to tools used by digital filter designers allowed the development of an integrated design environment to support the silicon compiler.

6.1.4. Bit-Serial Cell Library Design

The bit-serial components library was developed with fixed grid cell design methodology [Schediwy 1986] and a generic two layer metal CMOS process which is scalable down to 0.6 of a micron process [Fry and Lewicki

1985].

The cell design methodology is based on a composition grid spacing of eight lambda to one composition grid unit, where lambda is one half the minimum transistor channel length in a given technology. For example, a two layer metal CMOS process with a minimum channel length of two microns the value of lambda is one micron. Cells designed using this methodology must follow a number of design rules which are summarized as follows:

- Cell height is 11 composition grid units, width is variable
- Ports are centered on composition grid units at the boundaries
- Port 10 is reserved for Ground and is no less than 4 lambda wide
- Port 9 is reserved for Power and is no less than 4 lambda wide
- d-well height is 34 lambda
- metal-1 to cell border spacing must be 2 lambda
- polysilicon to border spacing must be 1 lambda
- active to border spacing must be 2 lambda

These rules are designed to allow primitive cells to be composed into composition cells without design rule violations between cells. A set of rules for the use of composition cells has also been defined and a complete set of these rules can be found in Appendix A.

In addition to the composition rules, port naming conventions were adopted. The port naming conventions allow analysis aids to find port information quickly and effectively. A port name is of the form *name.layer.face.number* where:

- *name* is a description of the port (eg. IN, OUT, etc.),
- *layer* is the type of layer that this port can connect to (eg. m is for metal-1, p is for polysilicon etc.),
- *face* is n, s, e, and w for the north, south, east, and west faces respectively,
- and *number* is the port's composition grid position in a cell where the numbering is increasing to the right and up the faces of the cell.

For example, *GND.m-1.w.10* is the name of the ground port in metal-1 on the

west face at composition grid position 10.

The bit-serial building block components, called macro cells, were composed from a set of primitive cells such as adders, latches, and inverters. In general macro cells are defined as in Figure 6-1 [Lyon 1981].

The adder for EFIDO is composed of a full adder, three registers and a multiplexor as shown in Figure 6-2. The time delay element is composed of two registers as in Figure 6-3. The multiplier module was implemented as outlined in [Lyon 1976(a)] using Booth's algorithm recoding scheme to allow all the input data to be treated alike. The multiplier contains nine latches, one full adder and four 2-input multiplexors as shown in Figure 6-4. All of the macro cells were designed using Electric's layout editor and can now be used by the compiler to build filter chips.

6.2. System Implementation

The EFIDO environment was developed to support a bit-serial compiler for digital filters. To coordinate the development of such an environment a

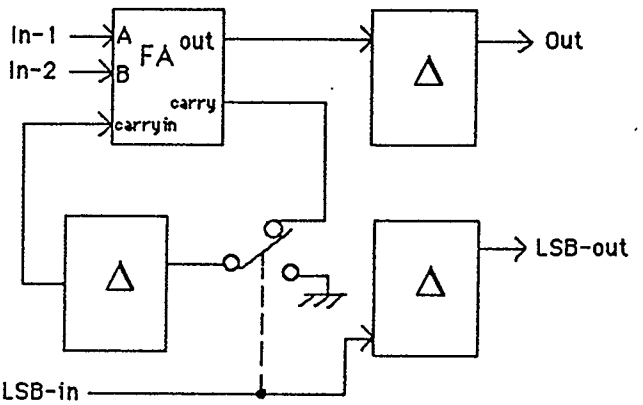


Figure 6-2 Adder Macro Cell

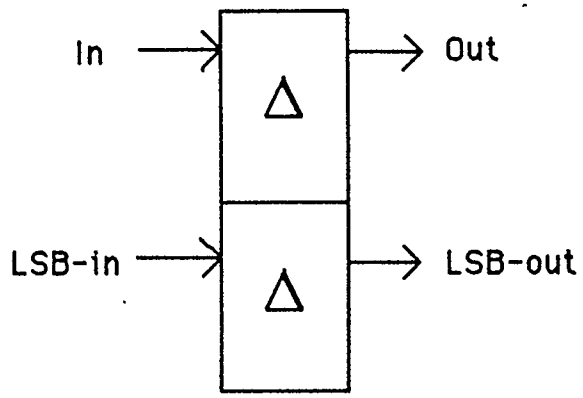


Figure 6-3 Time Delay Macro Cell

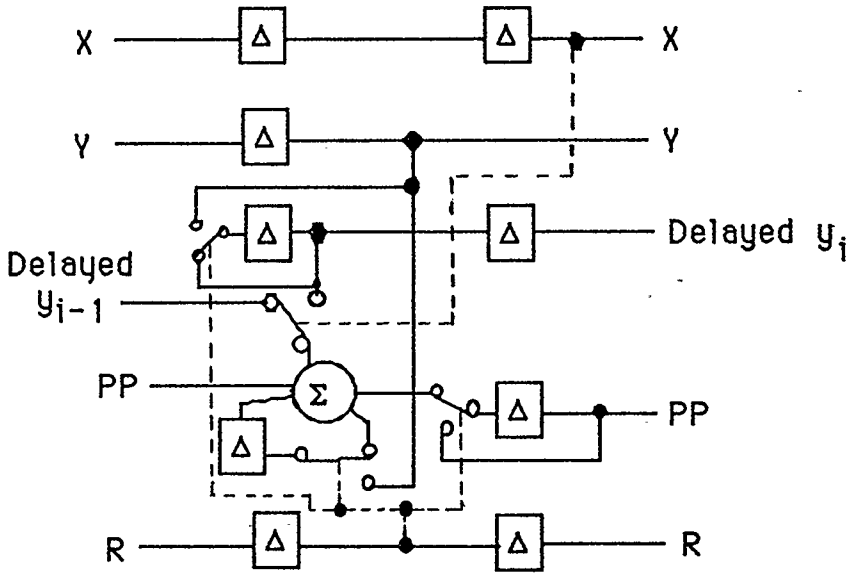


Figure 6-4 Multiplier Macro Cell (after Lyon)

digital filter designers' perspective was imposed. As shown in Figure 6-5 the digital filter design cycle has a number of loops, and large amounts of time are often spent within each loop or proceeding from one step of the design cycle to the next. The reason for the large amount of time spent is that each tool is individual with a unique data format, user interface and possibly a

unique machine. The designer is continually translating from one format to the next and finally must correctly transform what has been formally specified into a working circuit.

Silicon compilation imposes a structure on this perspective by having only one description of the filter which is used for both the functional and hardware analysis. Also, a silicon compiler focuses this perspective to a particular architecture, in EFIDO's case bit-serial.

The central specification used in EFIDO is a structural description of a digital filter. This structural specification is represented by the data flow description of the filter. The data flow graph gives the filter designer a high-level perspective of a filter. The filter structure is representative of the form the filter is to take, that is, its order and its recursive or non-recursive nature.

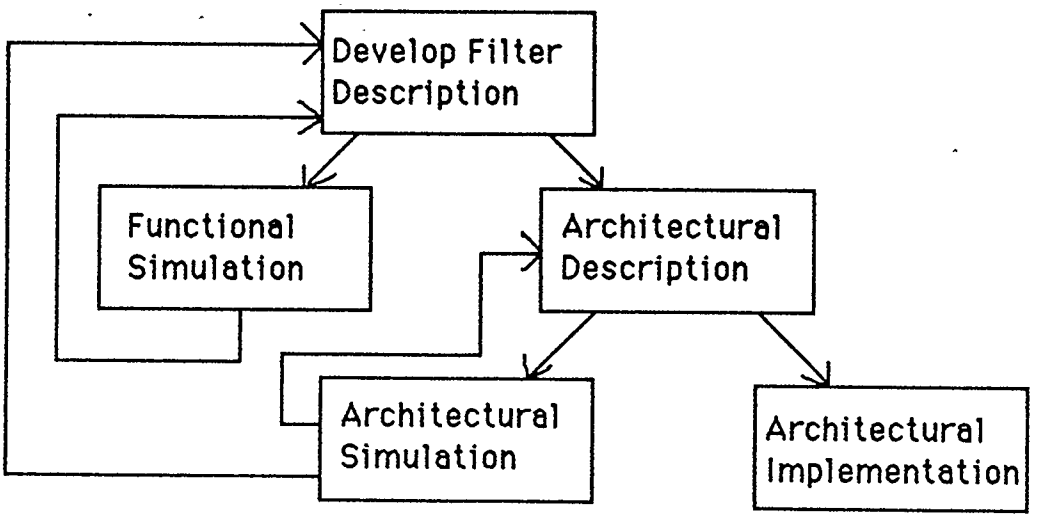


Figure 6-5 Filter Designers' Perspective

Figure 6-6 shows the tool organization and how they relate to the designer's perspective, within the EFIDO environment. At the top level EFIDO provides a direct mapping from the structural specification into a flow graph format which is used for a functional simulation of the filter. Once satisfied with the functional simulation a bit-serial representation can be extracted from the functional description and be used for bit-serial simulation or the specification of the mask level bit-serial hardware implementation via silicon compilation. All these tools exist as part of Electric or are interfaced via extraction programs which convert from Electric's format to the tool's input format.

6.2.1. A Graphical Filter Description

The first step was defining a basic set of primitives that could be used to describe a wide variety of digital filters at the structural level. The digital

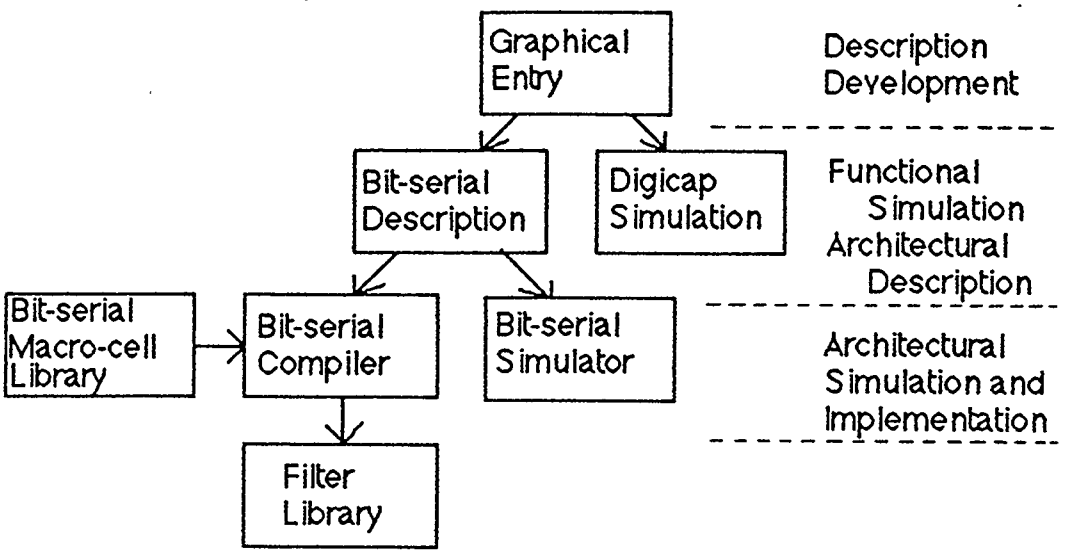


Figure 6-6 EFIDO Tools

filter technology provides five basic filter primitives: Input, Output, Adder, Delay, and Multiplier, as shown in Figure 6-7. These primitives were selected on a functional basis and need not map directly into an implemented component (eg. a subtractor could be modelled as an adder with a negated input). They are implemented in Electric as a new technology allowing the use of the graphical editing capabilities to define data flow graphs of the filter structure.

Electric, also allows variables to be associated with graphics primitives. This feature is used in EFIDO to store multiplier coefficients and coefficient bit length. Once a filter is specified graphically using the EFIDO technology and multiplier coefficients are supplied, the specification is ready to be used by the Digicap extraction program.

6.2.2. The Digicap Interface

Digicap simulation provides a functional simulation of a filter structure which verifies the magnitude, frequency, and phase response. The interface to the Digicap simulator is handled by generating an input file from a filter's structural specification.

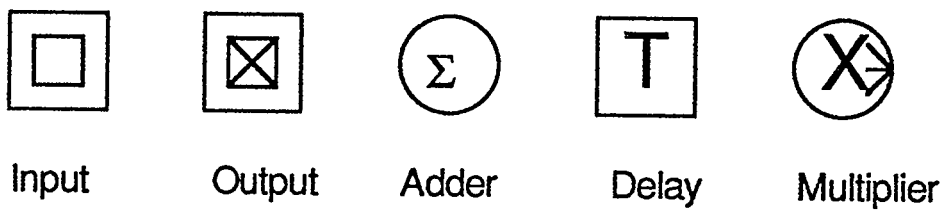


Figure 6-7 Basic Filter Primitives

A simulation analysis aid exists in Electric which supports other simulators such as SPICE [Nagel 1975]. The Digicap simulator is added as another option within the simulator analysis aid and with routines to handle the extraction and generation of the Digicap input file from a filter's structural description.

A filter such as that in Figure 6-8 is represented internally in Electric by a partially directed graph. This graph is partially directed since it is only possible to determine the direction of arcs at major nodes such as adders, time delays and multipliers, while intermediate connection nodes are bi-directional. Thus the generation of the Digicap code requires the depth-first search of a partially directed graph.

The depth first search for undirected graphs was implemented using an approach outlined in [Aho, Hopcroft and Ullman 1983], but tailored to the task of extracting a directed graph of the filter (see Figure 6-9). In this graph

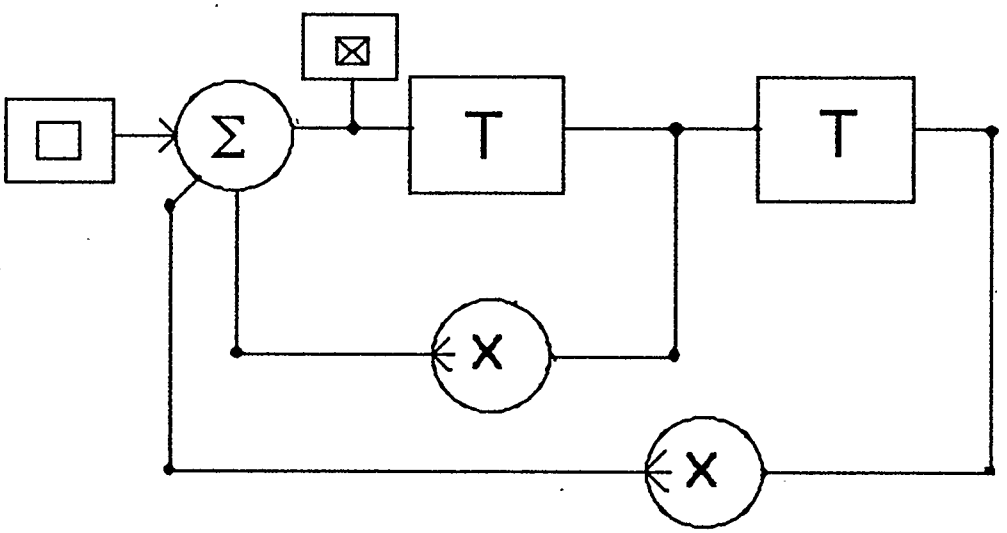


Figure 6-8 A Second Order Direct Form Filter Section

representation the arcs represent time delays, multiplications, or inputs, while the nodes represent addition or intermediate delay points.

The extraction from Electric's internal representation starts with a call to the recursive routine shown in Figure 6-10, with the input node of the filter structure passed as the starting parameter. There are major nodes (i.e. adders, multipliers, and delays) which are treated specially and minor nodes which act as intermediaries. The minor nodes accept the value passed to them and then pass this value on to their successors. On return the minor nodes return the last value passed back to them from their successors.

An adder increments the node count by one and is assigned the new value. Then, this value is passed down to any output nodes and passed back to the calling node. A time delay or multiplier node outputs a Digicap statement with the from node being the value passed and the to node being the value returned.

Using the high-level filter description the bit-serial representation of the filter is extracted using an approach similar to Digicap extraction. Bit-serial

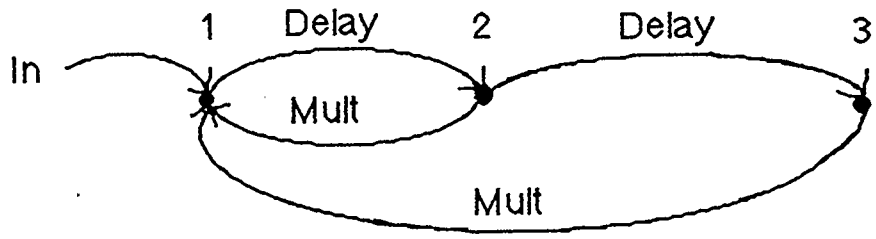


Figure 6-9 Directed Graph of Second Order Direct Form Filter

```
procedure sim_dfs (node,passed_val)
begin
   if < Node has been visited already> then
      return(node.node_number);

   case node of
   ADDER:
     begin
        node.node_number ← nodes + 1;
        <Visit node at this nodes output>
        sim_dfs(node.out_node,node.node_number)
        return (node.node_number);
     end
   DELAY:
     begin
        node.node_number ← nodes + 1;
        <Visit node at this nodes output>
        ret_val ← sim_dfs(node.out_node,node.node_number)
        < write out a digicap statement >
        return (passed_val);
     end
   MULTIPLIER:
     begin
        node.node_number ← passed_val;
        <Visit node at this nodes output>
        ret_val ← sim_dfs(node.out_node,node.node_number)
        < write out a digicap statement >
        return (passed_val);
     end
   OUTPUT:
     begin
        node.node_number ← passed_val;
        return (passed_val);
     end
   otherwise:
     begin
        < Visit all nodes from this intermediate node>
        return(< last value passed back >);
     end
end
```

Figure 6-10 Digicap Extraction Routine

extraction requires that all components have the allowable number of inputs (eg. two bit adders), multiplier lengths (as determined by the coefficients and

word length) and time delay (specified by the filter designer). For example the second order section is represented at the bit level as shown in Figure 6-11

The bit-serial representation can now be used for bit-serial simulations or bit-serial circuit layouts which are the topics of the next two sections.

6.2.3. The Bit-Serial Simulator

Bit-serial simulation provides verification of system timing. The simulation helps in detecting errors created by improper delay lengths resulting in the incorrect arrival time of the least significant bit. The EFIDO bit-serial simulator was written using an object-oriented event-driven simulation package called Demos [Birtwistle 1979], which is written in Simula.

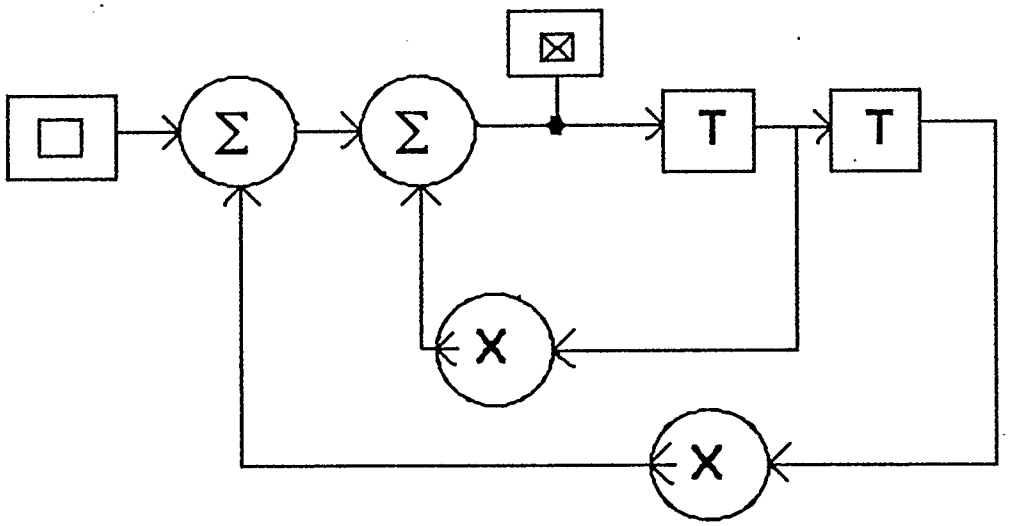


Figure 6-11 Bit-Serial Description of a Second Order Section

The bit-serial simulator can be implemented at either the bit level or the word level. If implemented at the bit level the detection of timing errors is simplified since control signals will correspond directly to the least significant bit. However, simulation at the bit level is costly in compute time for systems with large word lengths or with a large number of components. On the other hand simulation at the word level improves performance at the expense of not detecting some bit level timing errors. The approach taken for EFIDO's bit-serial simulator was to combine the two levels to take advantage of the performance of word simulation and the error detection at the bit level.

The EFIDO bit-serial simulator represents the control at the bit level and data at the word level. That is, all signal and operator delays are represented as bit delays while the data is stored and manipulated at the word level, with each word being an object which is passed between operator objects.

A word object as defined in Figure 6-12 plays a passive role in the simulation and must only record the arrival time in a queue and wait to be employed by an operator (adder, multiplier, or time delay). When a word object has been taken from a wait queue it is up to the operator which removed it from the queue to use the word and eventually assign it to another queue at the appropriate simulation time. The operators which take control of the input words, use the contents of these words to ensure the arrival times and control signals coincide and to perform operator operations, such as addition and multiplication. Once a word is assigned to another queue it is scheduled for execution at the current simulation time. When finally activated the word records the current simulation time and performs a wait on the new queue.

The bit-serial operators (eg. adders, multipliers and time delays) are also defined as objects and have the following execution steps:

- wait for a least significant bit signal (LSB) to arrive,
- take over control of the input word or words,
- check that the LSB signal and input words arrived at the same times and report any errors,
- hold for the operators delay time,
- and supply output words and delayed LSB signal.

An example of an adder operator definition is shown in Figure 6-13. All of the operator objects are designed to reflect the generic bit-serial description of the operator as was outlined in Section 6.1.4 This model simplifies control distribution since it is built into each operator.

The control signals, which represent the least significant bit (LSB), are implied in the operators, however a master control object is needed to generate an LSB signal for each input word. This signal is then distributed throughout the system via the operators. The master control object shown in

```
entity class word(init); value init; real init;
begin
  real toa;  ! Time of Arrival ;
  ref(waitq) COOPTQ;

  procedure word_coopt(INQ); ref(waitq) INQ;
  begin
    COOPTQ :- INQ;
  end;

  loop:
    toa := time;
    COOPTQ.wait;
  repeat;
end *** word ***;
```

Figure 6-12 The Word Object

```
entity class adder(SIG,INQ1,INQ2,OUTQ,adder_delay,words_out,OUT_POINT);
  ref(waitq) INQ1, INQ2, OUTQ;
  ref(bin) SIG; real adder_delay; integer words_out; boolean OUT_POINT;
begin
  ref(word) new_word,word1,word2; real sig_time;

  real procedure add(x,y); real x,y;
    add := x + y;

  if (OUT_POINT = TRUE) then words_out := words_out + 1;
  loop:
   SIG.take(1);
   sig_time := time;
   word1 :- INQ1.coopt;
   word2 :- INQ2.coopt;
   if (word1.toa NE word2.toa) then
     < WARNING Adder inputs did not arrive at the same time>
   if (word1.toa NE sig_time) then
     < WARNING Adder Signal did not arrive at the same time as WORD1>
   if (word2.toa NE sig_time) then
     < WARNING Adder Signal did not arrive at the same time as WORD2>
    ! hold for propagation delay ;
   hold(adder_delay);
     ! give adder output ;
     < output word(s) >
 repeat;
end *** adder ***;
```

Figure 6-13 Adder Object

Figure 6-14 holds for the word length of the system, checks to make sure all of the signals it has generated are used, then supplies the signals for the next cycle. If all the signals are not used during a cycle, an error is reported since some operator object is not processing its word fast enough.

Input to the filter is supplied from an input object. Figure 6-15 gives an outline for an input object which provides an impulse to the filter for one word unit, allowing analysis of the filters impulse response. Figure 6-15 also shows the outline for an output object, which monitors an output queue, removes one word from that queue when it becomes available and prints its

```
entity class word_len_cntrl(cntl_bin,comp_count,wdelay);
  ref(bin) cntl_bin; integer comp_count; real wdelay;
begin
   ! The Control Signals need to have priority to insure that the signals
     are available when the components execute
   ;
  priority := 10;
 loop:
  hold(wdelay);
  cntl_bin.give(comp_count);
  repeat;
end *** word_len_cntrl ***;
```

Figure 6-14 Master Control Object

arrival time and word value.

With all the basic objects defined, it is now possible to define a digital filter for simulation. The bit-serial representation of the second order section in Figure 6-11 can be used to generate an input file similar to that used for Digicap. This description is then used by the bit-serial simulator to define the filter objects and their interconnection.

Once the designer is satisfied with the filter performance determined using simulation, a mask level description of the filter can be generated using the bit-serial compiler.

6.2.4. The Bit-Serial Compiler

The bit-serial compiler can be broken into two phases (often called passes in software compilation): a virtual phase and a physical phase. The virtual phase deals with the relative positioning of the filter components and the connections between these components. The physical phase actually generates the filter by placing and wiring actual components.

```
entity class input_to_filter(OUTQ); ref(waitq) OUTQ;
begin
  ref(word) new_word;
  integer word_count;

 word_count := 0;
 loop:
    ! hold for input rate ;
   hold(input_rate);
   if < Time to generate spike > then
         new_word :- new word("word",10000.0)
     else  new_word :- new word("word",0.0)
   new_word.word_coopt(OUTQ);
   new_word.schedule(0.0);
 repeat;
end *** input_to_filter ***;

entity class output_from_filter(INQ);
  ref(waitq) INQ;
begin
 ref(word) word_out;
 loop
   out.take(1);  ! Wait for available output word ;
   word_out :- INQ.coopt
   < Print arrival time and word value
   outfix(word_out.toa,3,10);
   outfix(word_out.word_value,6,15); outimage;
 repeat;
end *** output_from_filter ***;
```

Figure 6-15 Input and Output Objects

The compiler has a fixed floorplan, shown in Figure 6-16, that has a wiring channel with components on each side. Power is distributed on each side of the channel in metal-1, while the ground is run in metal-1 on the outer edges of the components. The two phase clock is distributed in metal-2 on top of the power and ground lines with phi-1 distributed above the power line and phi-2 distributed above the ground line. With this floorplan in mind the problem of mapping the filter from the bit-serial description onto the floorplan may now be tackled.

The virtual phase of the compilation process is technology independent and is represented in Figure 6-17 by the routines to configure the bit-serial components (adders, multipliers, and time delays), place the configured components, and assign routing channels. This phase of the compilation must deal with some technology information, however the information used is similar for all technologies, such as component size and port names. This only requires that cells conform to the naming rules as outlined in section 6.1.4 when building a bit-serial component library.

Component configuration assigns a prototype to each of the components in the filter's bit-serial description. For example, the bit-serial description of a second order recursive section as in Figure 6-11 requires the assignment of a prototype of an adder to both adders and the configuration of prototypes for the multipliers and time delays. The adder prototypes exist as cells in the bit-serial components library and can be used directly.

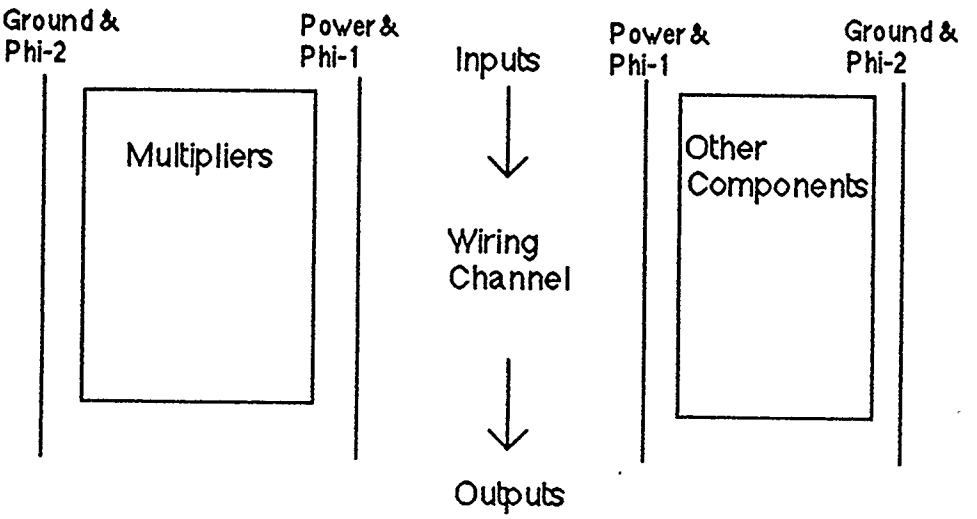


Figure 6-16 Floorplan of the Bit-Serial Compiler

```
dfo_COMPILE(cell,filter_cell)
  NODEPROTO *cell;
  NODEPROTO *filter_cell;
{ /* begin COMPILE */

        /* Configure Components */
       dfo_set_gate_protos();
          /* Place Filter Components */
       dfo_placement();
        /* Virtual Signal Routing Determined */
        dfo_PACK(dfile);
        /* Prepare for a CMOS Layout */
        dfoCMOS_SETUP();
        /* Assign a channel for each wire */
        dfoINITIALIZE_WIRES();
        /* Physically Place all the Cells */
        dfoDRAW_CELLS(cell,fil_cell);
        /* Physically Place all Interconnect Wires */
        dfoMTCMOS_WIRES();

} /* end ** COMPILE ***/
```

Figure 6-17 Main Routine of the Bit-Serial Compiler

The multipliers and time delays are variable length components. The number of stages in a multiplier depends directly on the number of bits used in the coefficient, up to a maximum of twelve bits. If this value is not assigned at the time the filter is specified, the coefficient length will default to twelve (as explained in section 6.1.2) which is the maximum multiplier length in the current system. The lengths of the delay elements are determined during the mapping from the structural description to the bit-serial description and are dependent on the word length, multiplier lengths and other components in the delay path.

Using the basic building blocks defined in section 6.1.4 a time delay component is created by placing and connecting the sections as shown in Figure 6-18. Note that the output signal is buffered to avoid fanout problems.

Fanout problems occur when one gate output is required to drive many gate inputs causing improper switching due to a weak signal. Since the design of the latch components requires minimum size transistors to save area, unbuffered devices are only capable of driving two to four inputs depending on the load of the inputs. The multiplier is handled in a similar manner using the multiplier components defined in section 6.1.4.

With the prototypes assigned the components must be placed on appropriate sides of the channel. The compiler originally grouped components with multipliers occupying alternate sides of the channel. Other components (adders and time delays), which were directly connected to a multiplier, were then clustered around the multiplier as shown in Figure 6-19. This style of floorplan had to be abandoned because of the irregularity of the filter perimeter leads to an excessive amount of space wastage. An alternate plan was adopted where the multipliers occupy one side of the channel and the delays and adders occupy the other. Figure 6-20 shows this placement style for the second order section. This strategy places elements of similar width

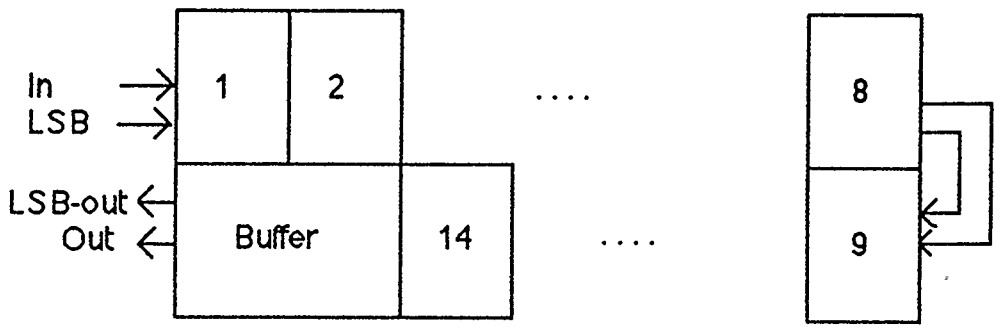


Figure 6-18 Fourteen Bit Time Delay Component

on each side of the channel making the perimeter more regular.

The procedure dfo_PACK assigns routing channels to the interconnection, input, and output signals. First the components (adders, multipliers, and time delays) on each side of the routing channels are numbered from 1 to n, where n is the number of gates on a channel side. Next the virtual position of the signals is defined using the gate locations and the input and output ports on the gates. Each signal has a top and a bottom endpoint with the values ranging from 0 for input points to 99999 for output points. The signals are now sorted in ascending order of top endpoint as

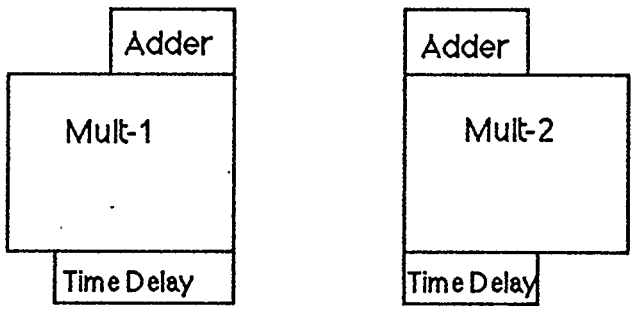


Figure 6-19 Prototype Component Placement

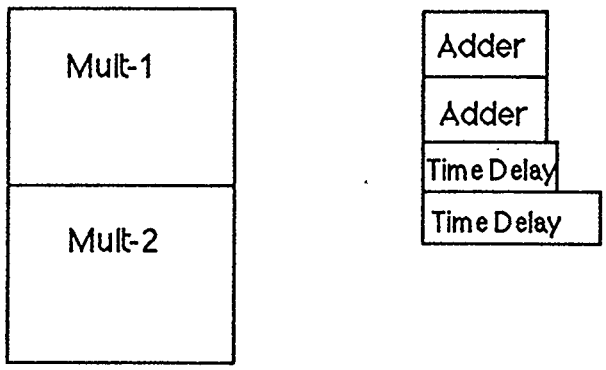


Figure 6-20 EFIDO's Final Component Placement

shown in Figure 6-21. The sorted signals are then packed using a recursive routine. It is simplest to explain the pack routine with an example. Using the sorted signals in Figure 6-21 the calls to the routine (PACK_SIG) are outlined in Figure 6-22, with the packed result shown in Figure 6-23. The PACK_SIG procedure executes as follows:

- Find a signal with a top endpoint that is greater than the value passed to the PACK_SIG routine.
- If a signal can be found then call PACK_SIG again using the value of the signal's bottom endpoint as the parameter which is passed.
- Else simply return.

The PACK_SIG procedure is called repeatedly from the top level with a value of -1 to ensure that at least one signal is chosen each time. With each return to the top level the channel number is incremented. This continues until there are no signals left in the sorted list.

The first step of the physical compilation phase initializes the global variables such as actual channel width. Once these variables are set up the routine dfo_Initialize_wires assigns a physical wire object to each virtual signal. The wire objects are modified during cell placement to reflect the

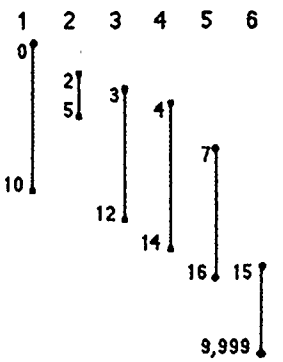


Figure 6-21 EFIDO's Sorted Signals

Channel = 1	Channel = 2	Channel = 3	Channel = 4
PACK_SIG(-1); Use signal 1	PACK_SIG(-1); Use signal 2	PACK_SIG(-1); Use signal 3	PACK_SIG(-1); Use signal 4
PACK_SIG(10); Use signal 6	PACK_SIG(5); Use signal 5	PACK_SIG(12); return;	return;
PACK_SIG(9999); return;	PACK_SIG(16); return;		

Figure 6-22 EFIDO's Trace of Signal Pack

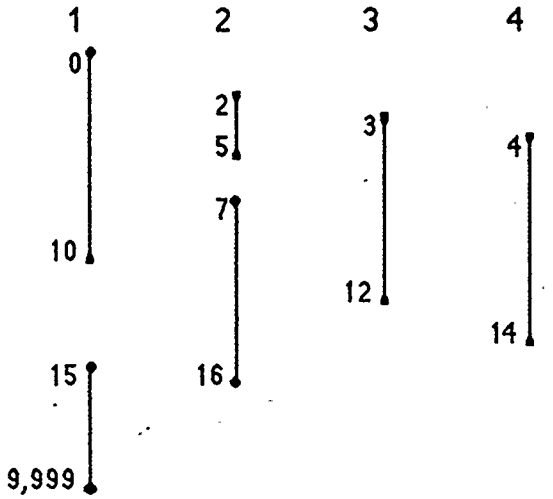


Figure 6-23 EFIDO's Packed Signals

physical position of the signal endpoints and the associated cell ports.

The dfo_Draw_cells routine now creates the layout of the filter and generates instances of each component's prototype. These instances are then placed on their assigned channel side in the correct location. The location is determined using the channel width, cell height, and position on a channel side. Once placed the input, output, power, and phi-1 connection wires for each component are extended into the channel and the ground and phi-2 connection wires are placed to the outer edges as shown in Figure 6-24.

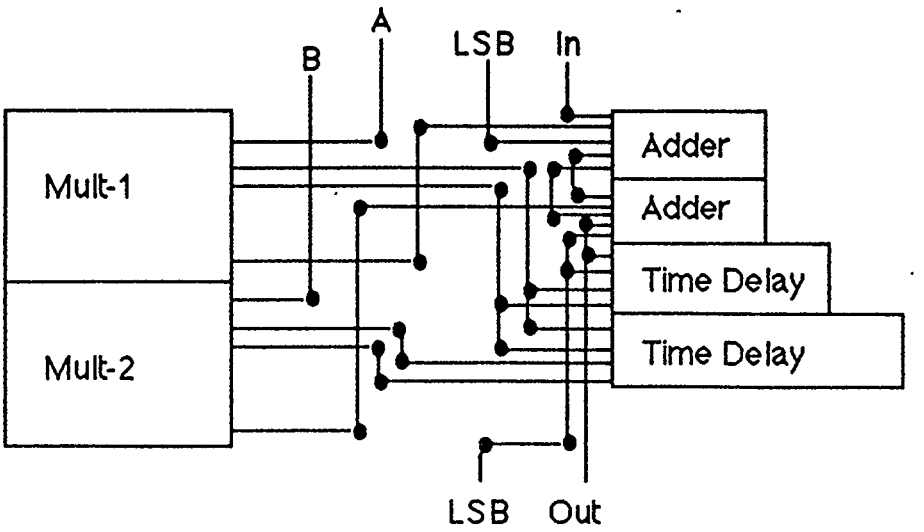


Figure 6-24 Physical Placement and Partial Route

The last step of the Physical compilation phase places all the wires. Each of the signal endpoints will now contain a physical description which is used to generate a metal wire between the endpoints. The power wires are placed in metal-1 on each side of the channel in tracks that have been left vacant for them and the ground wires are placed in metal-1 along the outer edges of the components. The clock wires are placed in metal-2 above the power and ground wires to complete the routing of the filter. The circuit can now be placed within a design context [Mead and Conway 1980; Coates and Kendall 1984; Schediwy 1985] which provides input, output, power and ground pads, and on chip generation of a 2-phase clock to supply the phi-1 and phi-2 clocking signals.

6.3. Summary

This chapter described the design and implementation of EFIDO, an integrated environment which supports a bit-serial silicon compiler for digital filters.

A silicon compiler directs the implementation of its associated environment by requiring a formal specification of the problem and a target hardware architecture. However, when implementing the integrated environment the perspective of the designer must be considered to allow a smooth transition from specification to implementation.

The next chapter presents a scenario of use of the EFIDO system. The scenario involves the design and implementation of a Low Pass Filter using a direct form structure. To demonstrate the flexibility of EFIDO, a Lossless Discrete Integrator [Turner 1983] structure is used to implement this type of filter.

CHAPTER 7

Example of Use

This chapter presents a scenario on the use of EFIDO. EFIDO exists in Electric's tool environment, which provides the common user interface to all the tools.

In Chapter 5, a specification for a Low Pass digital filter was defined by the difference equation:

$$y(k) = Cx(k) + Ay(k-1) + By(k-2) \qquad \{4.1\}$$

whose z-transform was found to be:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{Cz^2}{z^2 - Az - B} = \frac{C}{1 - Az^{-1} - Bz^{-2}}$$

The design of the digital filter may now be completed by using the tools provided by EFIDO.

The first step is entering the data flow diagram of the filter structure. The data flow diagram represents the discrete time filter structure graphically and is entered into the system using the graphical editing capabilities provided in Electric. The data flow diagram in Figure 7-1 is a direct form filter structure representing the filter as defined by Equation 4.1. To enter such a diagram the user selects the desired components from a palette of filter components (Figure 7-2), places them where appropriate on the screen, then connects the components using the wiring mode provided.

The data flow diagram is the central representation of the filter. Any changes that the designer performs must be performed on the data flow

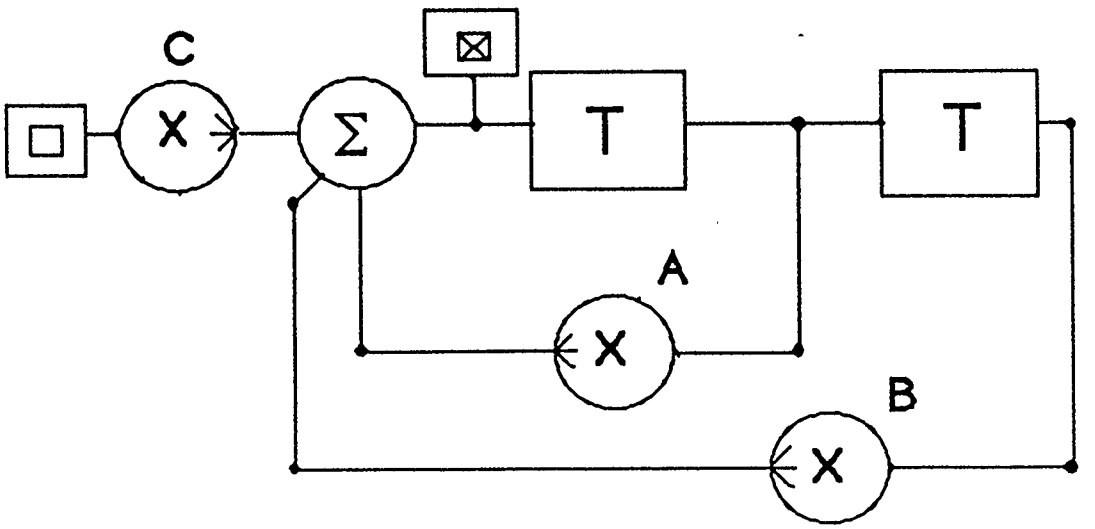


Figure 7-1 Data Flow Diagram

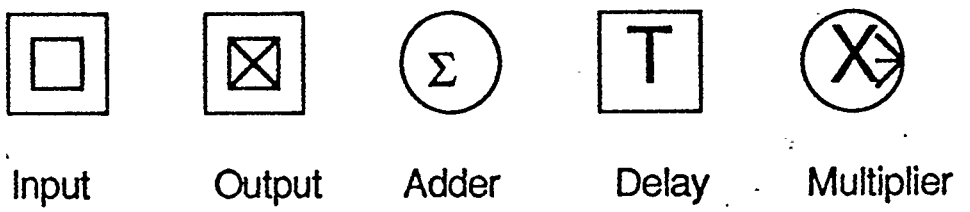


Figure 7-2 Palette of Components

description. This assures consistency throughout the design process and simplifies error detection and correction.

To enter the multiplier coefficients and delay lengths for use in the simulators, the user selects a multiplier component, then assigns the value using the variable assignment macro. The following will assign *-1.018* to an attribute/value pair on the currently highlighted node instance:

-var set ~ .coeff -1.018

Once set, these variables can be used by the analysis aids, such as the

simulator extractors. With the data flow entered and variables assigned the user is now ready to start the testing and debugging phase.

The first step in the refinement of the filter specification requires the functional simulation of the proposed filter structure. EFIDO provides an interface to one such simulator, Digicap. Digicap was developed by the Electrical Engineering department at the University of Calgary and is used to analyze at a functional level the magnitude and phase response of digital filters. By invoking the Digicap analysis aid the designer can have an input file for the Digicap simulator generated. The output in Figure 7-3 is the Digicap representation for the filter's data flow diagram shown in Figure 7-1.

A simulation can be run on this file by entering into a UNIX [Ritchie and Thompson 1974] command shell mode in Electric Digicap allows analysis of the magnitude and phase response as shown in Figure 7-4. After the simulation, if the filter needs modification the designer simply edits the data flow diagram appropriately and regenerates the Digicap file. This edit test loop is continued until the designer is satisfied that the filter is exhibiting the desired filter characteristics.

```
input 1 ac
mult 1 2 C
delay 2 3
delay 3 4
mult 3 2 A
mult 4 2 B
```

Figure 7-3 Digicap Input Deck

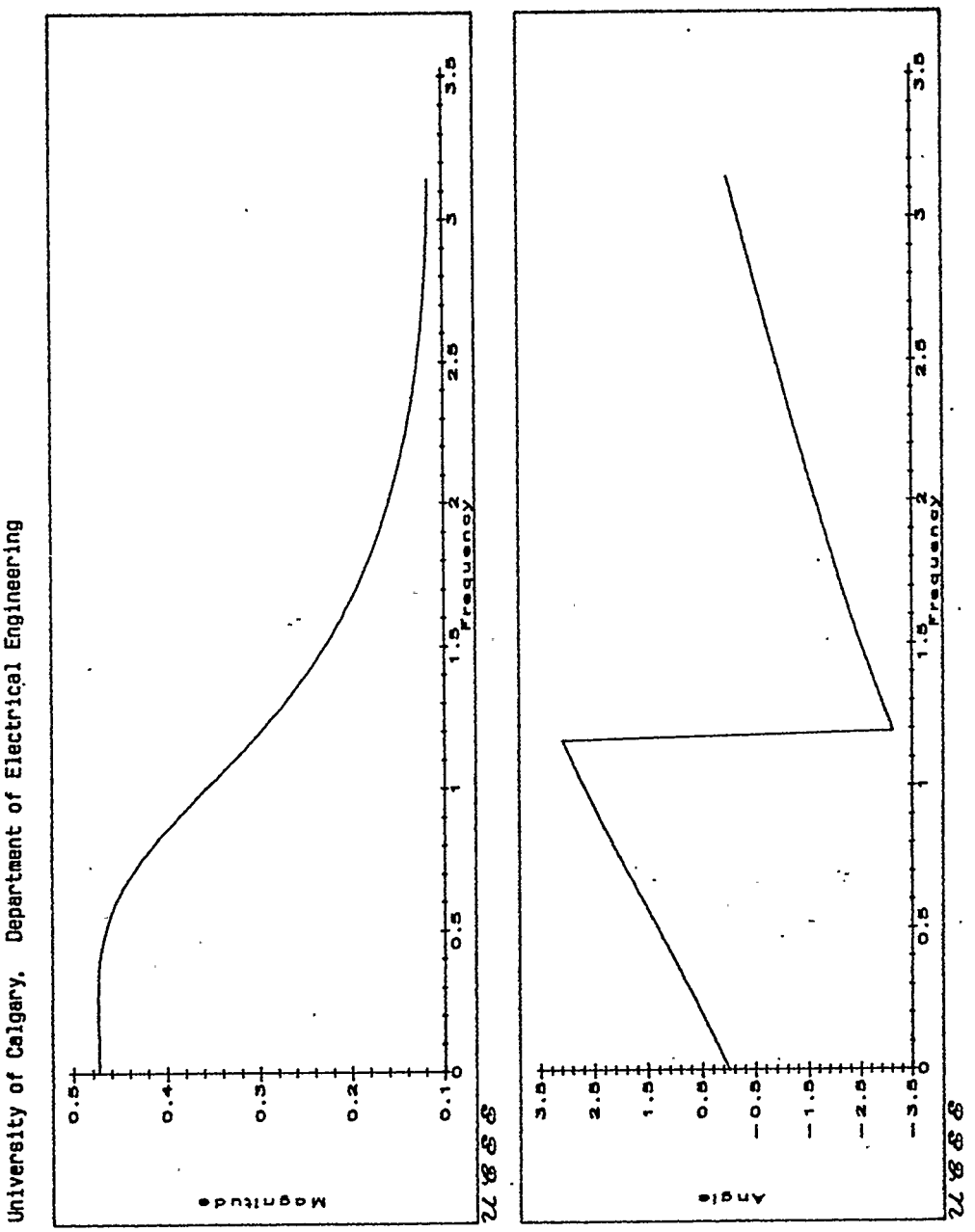


Figure 7-4 Magnitude and Phase Response

The filter is now ready for simulation and implementation using the provided bit-serial environment. The data flow graph representation in Figure 7-1 is used to generate an internal bit-serial representation as depicted in Figure 7-5.

This internal representation, which the designer need not view graphically, is then used to generate an input file for the bit-serial simulator. The bit-serial simulation highlights errors introduced by restrictions in the actual implementation, such as word length. Adjustments for word length restrictions can sometimes be achieved by refining a multiplier's coefficient length, which determines the actual number of sections required to realize a bit-serial multiplier [Lyon 1976(b)]. In addition, timing errors may be detected, such as negative delay lengths which require the Least Significant Bit (LSB) of a word to be available before it can physically arrive at the input to a component. Timing errors, however, must be corrected in the high-level filter specification, and a number of may be required iterations between the Digicap and Bit-serial simulations.

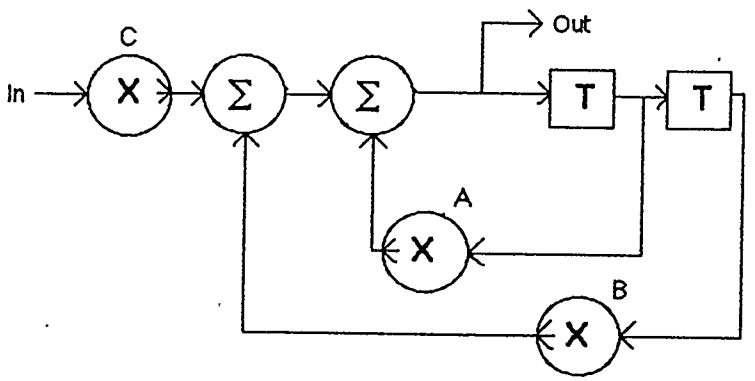


Figure 7-5 Internal Bit-Serial Representation of Filter

Once the bit-serial simulation proves satisfactory, the EFIDO can actually lay out the chip from the internal bit-serial specification. The components are configured (correct time delays lengths, etc.), placed and routed as shown in Figure 7-6. The design can now be placed inside a design context, which supplies input and output pads for fabrication.

The above scenario used a direct form structure realization of a digital filter, which has a high sensitivity to the multiplier coefficients [Rabiner and Gold 1975]. A Lossless Discrete Integrator (LDI) filter structure is one which is less sensitive to coefficient values. Figure 7-7 shows the data flow description for a second order LDI filter.

The extraction of the Digicap representation requires more information be present in the data flow graph because Digicap requires certain paths to be modelled by multiplications with coefficient values of 1 or -1. This problem is overcome by specifying specific *arcs* in the data flow graph as negative or positive. The Digicap extraction program is then adjusted to check the *arcs*

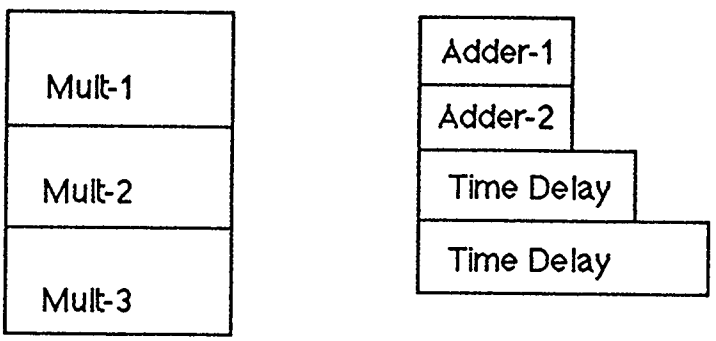


Figure 7-6 Chip layout

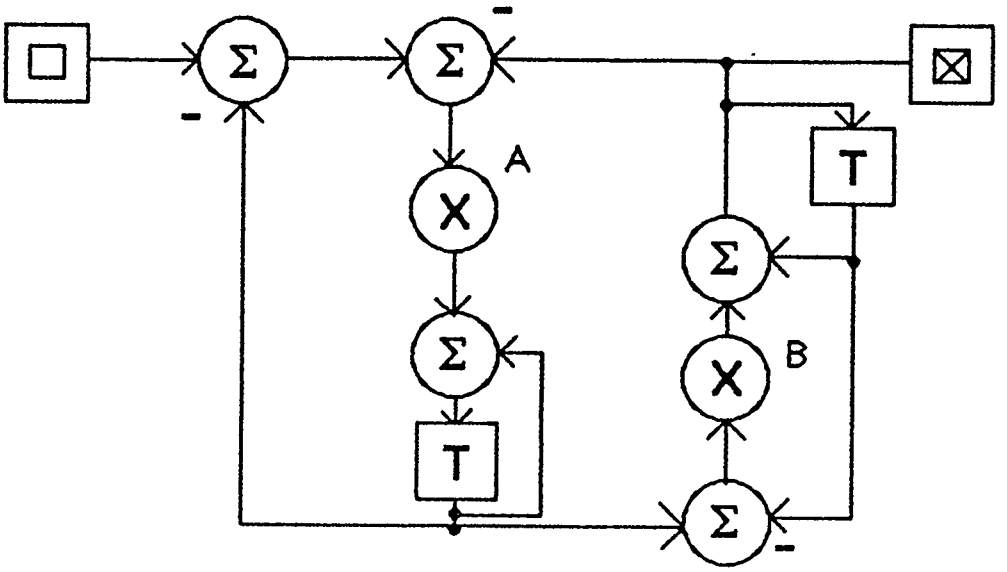


Figure 7-7 LDI Filter Structure

and modify the input deck appropriately. The Digicap deck for the LDI structure above is shown in Figure 7-8.

```
input 1 ac
mult 1 2 1
mult 2 3 A
delay 3 4
mult 4 3 1
mult 4 1 -1
mult 4 5 1
mult 5 6 B
mult 6 7 1
delay 7 8
mult 8 6 1
mult 8 5 -1
mult 7 2 -1
```

Figure 7-8 LDI Digicap Input Deck

The bit-serial compilation proceeds as it would for a direct form structure with the resulting LDI layout in Figure 7-9.

In summary, this chapter has presented a scenario of use of the EFIDO system. EFIDO was designed to handle the digital filter structures of the general form:

$$y(n) = \sum_{i=0}^{M} b_i\, x(n-i) - \sum_{i=1}^{N} a_i\, y(n-i)$$

This general form can be built upon to support other filter structures, such as an LDI filter implementation.

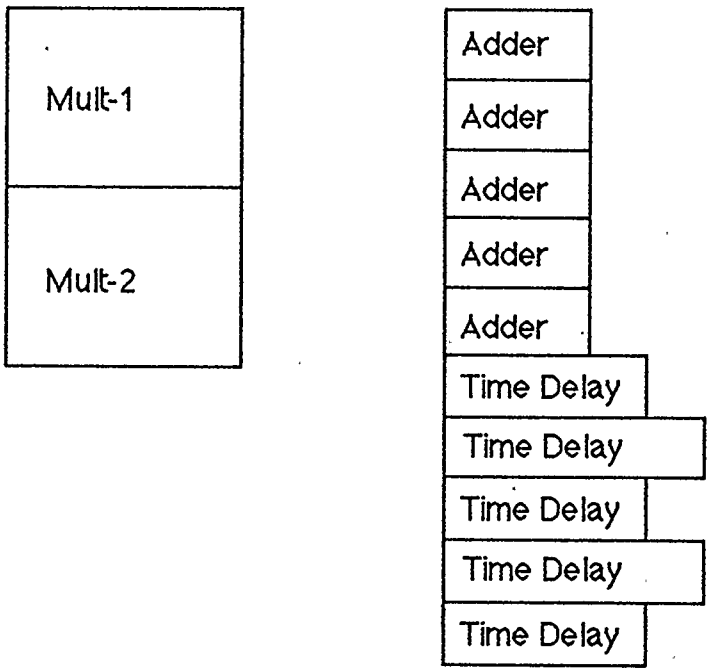


Figure 7-9 LDI Filter Layout

CHAPTER 8

Conclusion

8.1. Summary

Silicon compilation partially automates the design process by transforming a high-level specification into a mask level layout. However, the successful application of silicon compilation techniques must be supported by an integrated environment. Furthermore, silicon compilation is only feasible for a properly chosen application domain. These claims have been defended by showing an analogy between software and silicon compilation.

These ideas were also illustrated by the presentation of an integrated environment to support a silicon compiler for digital filters. This environment was developed as an extension to Electric, an established framework for VLSI software development.

Working from a central high-level specification and using silicon compilation the filter designer is able to move from a structural description to mask level implementation in a consistent manner. To represent the structural specification a graphical input capability for data flow diagrams was added to Electric. From the data flow diagram, extraction tools were used to generate an input file for a functional simulator and an intermediate bit-serial representation. A bit-serial simulator was implemented and an input file could be generated from EFIDO's internal bit-serial representation. Finally, a bit-serial silicon compiler used the bit-serial representation to generate a mask level layout of the digital filter, completing the transition

from specification to implementation.

By working from a single specification, consistency of the design is maintained and the designer is assured that the filter being simulated and laid out are identical. This frees filter designers from the task of extracting simulation information and actual layout specification, allowing them to concentrate on the digital filter's operation.

8.2. Observations

EFIDO is an integration of tools, some of which were available and some of which had to be written, making the choice of the development environment very important. By using a prototyping environment such as Electric, development time was minimized because such things as user interface were predefined and easily extended to meet development needs. Also, an extendable environment such as Electric allowed the use of existing tools (eg. simulators, compactors) and will allow the use of future tools (eg. test pattern generators).

A silicon compiler also covered a wide range of problems facing VLSI designers, such as floorplanning, placement and routing. EFIDO's silicon compiler was designed such that each of these problem areas could be tackled independently without any major effects on the other areas.

The bit-serial compiler originally assigned multipliers to alternate sides of the wiring channel with other components clustered around the multipliers. This style of floorplan had to be abandoned because of the irregularity of the filter perimeter leads to an excessive amount of space wastage. An alternate plan was adopted where the multipliers occupy one side of the channel and the delays and adders occupy the other. This strategy places elements of

similar width on each side of the channel making the perimeter more regular.

There are a number of restrictions within the current system, such as word length. The word length is restricted to 24 bits, which in turn restricts the multiplier coefficient length to a maximum of 12 bits. This may cause undesirable effects, such as limit cycles, in certain filter structures.

Currently, EFIDO has a bit-serial target architecture, though a simple modification would re-target it towards a bit-parallel architecture. This suggests yet another analogy between software and silicon compilation. A software compiler can be built to generate machine code for a number of different machines. However, the integrated environment which supports a software compiler need not change. Similarly, a silicon compiler can generate different architectural layouts without major changes to the environment. This implies that a strong relationship exists between the application domain and the environment.

While the tools in EFIDO have been successfully integrated, the EFIDO compiler has yet to produce a chip. The original cell library was implemented for a specific fabrication process. When the fabrication line was changed the cell library had to be redesigned. The introduction of a set of generic design rules will allow fabrication over a wider range of facilities. Currently the author and colleagues at the University of Calgary are implementing a new cell library and hope to submit filter designs for fabrication in April 1986.

8.3. Future Directions

Future work to be done on the existing EFIDO environment includes optimizing the bit-serial compiler. Using techniques such as simulated annealing to better utilize the wiring channel, and floorplanning tools such as

Topologizer [Kollaritsch and Weste 1984] could help minimize chip area. Other optimizations include the reduction of component count by sharing large operators (such as multipliers) between filter sections.

An interesting research area is the development of compiler hierarchies [Goldberg, Hirschhorn and Lieberherr 1985]. This approach requires the development of a basic structural silicon compiler and behavioural compilers which map a purely behavioural description onto a purely structural description. For example, a structural silicon compiler would map a structural description such as MODEL [Lattice Logic 1982] onto a hardware architecture and a behavioural *structural* compiler would map a behavioural description such as HOL [Gordon 1983] onto MODEL. The development of such a hierarchy of compilers will have a profound impact on design environments.

Silicon compilation is currently looked upon as just another tool to support the designer's environment. However, as integrated circuits become more complex, designers will have to move away from traditional design methods towards silicon compilation. This in turn will lead to developing the designer's environment so that it supports silicon compilation.

References

Aho, A.V., Hopcroft, J.E., and Ullman, J.D. (1983) *Data Structures and Algorithms.* Addison-Wesley Publishing company.

Antoniou, A. (1979) *Digital Filters: Analysis and Design.* McGraw Hill.

Ayers, R. (1979) "IC Specification Language" *16th Design Automation Conference Proceedings,* San Diego, California.

Bergmann, N.W. (1983) "A Case Study of the FIRST Silicon Compiler" *Second Caltech Conference on VLSI,* 413-430, Computer Science Press.

Birtwistle, G.M. (1979) *Discrete Event Modelling on Simula.* MacMillan Publishers Ltd.

Birtwistle, G.M., Hill, D., Kendall, J., Coates, W., Esau, R., Kroeker, W., Liblong, B., Liu, E., Melham, T., and Schediwy, R. (1984) "Edict - an Environment for Design using Integrated Circuit Tools" (84/155/13), Research Report, University of Calgary, Dept. of Computer Science, June.

Birtwistle, G.M., Joyce, J., Liblong, B., Melham, T., and Schediwy, R. (1986) "Specification in VLSI Design" *Proc. of the Edinburgh Workshop on Formal Models of Verification,* North Holland.

Breuer, M.A. and Carter, H.W. (1984) "VLSI Routing" in *Hardware and Software Concepts in VLSI,* edited by Guy Rabbat, pp Van Nostrand Reinhold.

Brown, H., Tong, C., and Foyster, G. (1983) "Palladio: An Exploratory Environment of Circuit Design" *IEEE Computer,* 41-56., December.

Bryant, R.E. (1981) "A Switch Level Simulation Model for Integrated Circuits" Technical Report-259, MIT Laboratory for Computer Science.

Buchanan, I. (1980) "Modelling and Verification in Structured Integrated Ciruit Design" Ph.D. Thesis, Department of Computer Science, University of Edinburgh.

Buchanan, I. (1982) "SCALE - A VLSI Design Language" Report CSR-117-82, Department of Computer Science, University of Edinburgh, May.

Coates, W. and Kendall, J. (1984) "Design of a Basic VLSI Project Context" *Canadian Conference on Very Large Scale Integration,* Edmonton, Alberta.

Denyer, P. and Renshaw, D. (1985) *VLSI Signal Processing: A Bit-Serial Approach.* Addison-Wesley Publishing Company.

Edif (1985.) "Electronic Design Interchange Format" *1.*

Sproull, R. F. and Lyon, R. F. (1980.) "The Caltech Intermediate Form for LSI Layout Description" in *Mead and Conway.* Section 4.5.

Fountain, T.J. (1983) *A Survey of Bit-Serial Array Processor Circuits, in Computing Structures for Image Processing.* Academic Press.

Freeny, S., Kieburtz, B., Mina, K., and Tewksbury, S. (1971) "Design of Digital Filters for an All Digital Frequency Division Multiplex-Time Division Multiplex Translator" *IEEE Transactions on Circuit Theory, ct-18* (6),

November.

Fry, A. and Lewicki, B. (1985) *Generic Cmos Design Rules.* The MOSIS Project, Information Sciences Institute, Marina del Rey, California 90292-6695.

Fung, H.S. and Hirschhorn, S. (1986) "An Automatic DFT System for the Silc Silicon Compiler" *IEEE Design and Test,* February.

Goldberg, A.V., Hirschhorn, S.S., and Lieberherr, K.J. (1985) "Approaches Toward Silicon Compilation" *IEEE Circuits and Devices,* May.

Gordon, M. (1983) "Proving a Computer Correct" (42), Technical Report, University of Edinburgh, March.

Gosling, J.B. (1980) *Design of Arithmetic Units for Digital Computers.* The Macmillan Press Ltd..

Hedges, T.S., Slater, K.H., Clow, G.W., and Whitney, T. (1982) "The Siclops Silicon Compiler" *IEEE Design Automation Conf..*

Ibbett, R.N. (1982) *The Architecture of High Performance Computers.* Macmillan Publishers Ltd..

Jackson, L.B., Kaiser, S.F., and McDonald, H.S. (1968) "An Approach to the Implementation of Digital Filters" *IEEE Trans. Audio and Electroacoust., AU-16,* 413-421, September.

Johannsen, D. (1979) "Bristle Blocks -- A Silicon Compiler" *Proceedings of the 16th Design Automation Conference,* 310-313.

Johannsen, D. (1981) "Silicon Compilation" Ph.D. Thesis, Department of Computer Science, California Institute of Technology.

Katz, R. (1985) "Configuration Management" Personal Communications.

Keller, R.M., Lindstrom, G., and Patil, S.S. (1980) "Data-Flow Concepts for Hardware Design" *COMPCON 80,* 105-111, February.

Kollaritsch, Paul and Weste, N. (1984) "Topologizer: An Expert System Translator of Transistor Connectivity to Symbolic Cell Layout" *Proceedings of ESSCIRC.*

Kroeker, W., Birtwistle, G. M., and Esau, R. (1984) "RLC a Random Logic Compiler" *Canadian Conference on Very Large Scale Integration,* Edmonton, Alberta.

Lattice (1982) "MODEL: A High-Level Structured Design Language" *Designing with Gate Arrays,* Lattice Logic Ltd.

Ledgard, H. and Marcotty, M. (1981) *The Programming Language Landscape.* Science Research Associates, Inc.

Lehman, M., Eshed, R., and Netter, Z. (1963) "Sabrac - A New Generation Serial Computer" *IEEE Trans. on Electronic Computers,* 618-625, December.

Liblong, B. (1984) "SHIFT -- A Structured Hierarchical Intermediate Form for VLSI Design Tools" MSc thesis, Department of Computer Science, University of Calgary, September.

Liu, E. (1982) "Design and Analysis of Recursive Digital Filters" MSc thesis, Department of Electrical Engineering, University of Calgary, December.

Locanthi, B. (1978) "LAP: A Simula Package for IC Layout" (1862), Caltech Technical Report Display File, July.

Lusky, S., Bossart, P., Kollaritsch, P., Lewis, R., and Thompson, C. (1985) *DROID - An Object Oriented Data Base for VLSI Design.* Texas Instruments, Dallas, Texas, Personal Communications.

Lyon, R.F. (1976(a)) *Patent for a Two's Complement Pipeline Multiplier.* US Patent 3,956,622.

Lyon, R.F. (1976(b)) "Two's Complement Pipeline Multipliers" *IEEE Trans. on Communications,* 418-424.

Lyon, R. (1980) *Video Course Notes.* VTI Video.

Lyon, R.F. (1981) "A Bit-Serial VLSI Architectural Methodology for Signal Processing" *VLSI 81 International Conference,* Edinburgh, Scotland.

Lyon, R.F. (1983) "Filters: An Integrated Digital Filter Subsystem" Printed in *VLSI Signal Processing: A Bit-Serial Approach,* Denyer and Renshaw.

Mead, C. and Conway, L. (1980.) *Introduction to VLSI Systems.* Addison-Wesley.

Moon, D. and Weinreb, D. (1981) "Lisp Machine Manual" Technical Report, Massachusetts Institute of Technology Artificial Intelligence Laboratory.

Mosteller, R.C. (1981) "REST - A Leaf Cell Design System." MSc Thesis, Silicon Structures Project Technical Report 4317, California Institute of Technology, December.

Mostow, J. (1985) "Toward Better Models of the Design Process" *The AI Magazine,* Spring.

Nagel, L.W. (1975.) "SPICE2: A Computer Program to Simulate Semiconductor Circuits" ERL Memo ERL-M520, University of California, Berkeley, May.

Norihiko, O., Tadakatsu, K., and Masanobu, D. (1979) "LSI's for Digital Signal Processing" *IEEE Journal of Solid State Circuits, sc-14* (2), April.

Ousterhout, J. (1981) "Caesar: An Interactive Editor for VLSI Layouts" *VLSI Design, II* (4) 34-38.

Rabiner, L.R. and Gold, B. (1975) *Theory and Application of Digital Signal Processing.* Prentice-Hall, Inc., Englewood Cliffs, NJ.

Ritchie, D.M. and Thompson, K. (1974) "The UNIX Time-Sharing System" *Comm. ACM, 17* (7) 365-375, July.

Rose, C. (1985) "Dec's Latest Work Station Uses The VAX-ON-A-CHIP" *Electronics Week,* 68-70, May 27.

Rubin, S. (1983) "An Integrated Aid for Top-Down Electrical Design" *VLSI 83 International Conference,* 63-72, North Holland, Amsterdam, August.

Schediwy, R. (1985) "Pad Context" Personal Communications.

Schediwy, R. (1986) "Composition Cells" MSc thesis, Department of Computer Science, University of Calgary.

Siewiorek, D.P., Bell, C.G., and Newell, A. (1982) *Computer Structures: Principles and Examples.* McGraw-Hill Book Company.

Siskind, M.J., Southard, J.R., and Crouch, K.W. (1981) *Generating Custom High Performance VLSI Designs From Succinct Algorithmic Descriptions.* MIT Lincoln Laboratory, Lexington, MA., December.

Stallman, R.M. (1981) "EMACS — The extensible, customizable, self-documenting display editor" *SIGOA Newsletter (Proc. ACM Symposium on Text manipulation, Portland, Oregon), 2* (1/2) 147-156, Spring/Summer.

Teitelman, W. and Masinter, L. (1981) "The Interlisp Programming Environment" *IEEE Computer, 14* (4) 25-33, April.

Trimberger, S. (1981) "Combining Graphics and a Layout Language in a Single Interactive System" (4281), Technical Report, California Institute of Technology, May.

Turner, L.E. (1983) "Elimination of constant-input limit cycles in recursive digital filters using a generalised minimum norm" *IEEE Proc, 133* (3), June.

Turner, L. (1984) *Digicap Reference Manual.* Electrical Engineering Dept., University of Calgary.

Walker, R.A. and Thomas, D.E. (1985) "A Model of Design Representation and Synthesis" *Proc. of the 22nd Design Automation Conference,* 453-459.

Electronics Weekly (1985) "Development System Breaks Productivity Barrier" *Electronics Week,* July 8th.

Weste, N. (1981) "Mulga - An Interactive Symbolic Layout System for the Design of Integrated Circuits" *Bell Systems Technical Journal, 60* (6) 823-857, July-Aug.

Whitehead, A.N. (1911) *An Introduction to Mathematics.* Oxford University Press, Oxford, England.

Williams, T. (1983) "Design for Testability - A Survey" *Proc. IEEE, 71* (1).

Moore, Z. (1978) "An Introduction to the Mathematics of Digital Signal Processing" *Computer Music Journal, 2* (1 and 2).

Zippel, R. (1984) *Schema.* Personal Communications.

Zissos, D. (1979) *Problems and Solutions in Logic Design.* Oxford University Press.

APPENDIX A

Composition Cell Design Rule Summary

THE COMPOSITION GRID:

Rule No.	Rule Specification
0.0	C grid verticies are spaced by 8 lambda

DESIGN RULE DERIVED CONSTRAINTS:

Rule No.	Layer	Distance outward in relation to cell border
1.0 -	Metal1	2 lambda
1.1 -	Metal2	2 lambda
1.2 -	Polysilicon	1 lambda
1.3 -	Active (S or D type)	2 lambda (1.5 lambda for 5x5 contact areas)
1.4 -	Select	0 lambda (-0.5 lambda for 5x5 contact areas)
1.5 -	Well	-4 lambda (suggested rule: allow well area to be as large as possible with full 4 lambda outset from cell border)

STANDARD CELL SPECIFIC CONSTRAINTS:

Rule No.	Pertaining To	Rule Specification
2.0 -	Cell Height	11 C units (11 ports)
2.1 -	GND Port	Centered on Port 10 (10th from south border)
2.2 -	PWR PORT	Centered on Port 9 (9th from south border)
2.3 -	PWR/GND Track Width	5 lambda
2.4 -	D-Well	34 lambda high from south border

PORT RULES:

Rule No.	Rule Specification
3.0 -	All ports are pins 4x4 lambda in size
3.1 -	Ports are centered on C grid verticies

INTERCONNECT RULES:

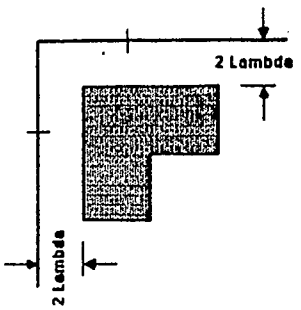
Rule No.	Rule Specification
4.0 -	Valid Arc Layers: - Metal 1 - Metal 2 - Polysilicon
4.1 -	Valid Contacts and Vias: - Metal 1/Metal 2 Via - Polysilicon/Metal 1 - Stacked contacts are dis-allowed
4.2 -	All arcs are centered on C grid
4.3 -	All contacts and vias are centered on C grid
4.4 -	Any arcs or contacts may be placed on the first C grid outside any standard or composition cell.

CELL ABUTMENT AND SPACING RULES:

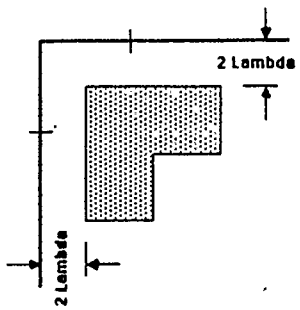
E	=	East std cell face
W	=	West std cell face
N	=	North std cell face
S	=	South std cell face
CC	=	Composition cell
FCC	=	Foreign composition cell

5.0 -	E or W	to	E or W	0 C units
5.1 -	E or W	to	N, S or FCC	1 C unit
5.2 -	N	to	N	0 C units
5.3 -	N	to	S, E, W or FCC	1 C unit
5.4 -	S	to	S	0 C units
5.5 -	S	to	S, E, W or FCC	1 C unit
5.6 -	CC	to	FCC	1 C unit

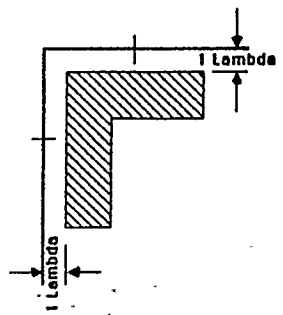
5.7 - Composition cell borders may be coincident with borders of cells contained within.



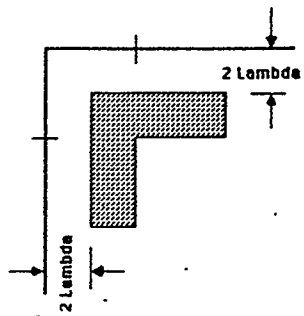
1.0 Metal1 to Border



1.1 Metal2 to Border

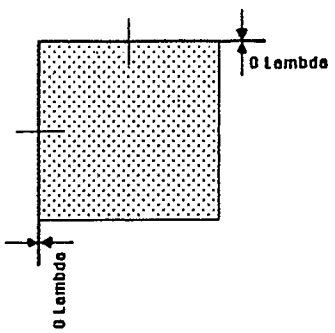


1.2 Poly to Border



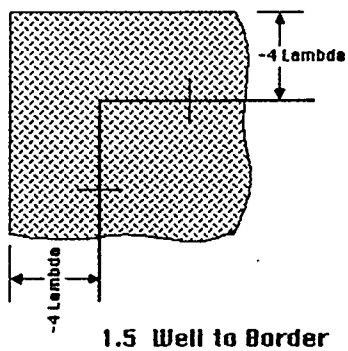
1.3 Active to Border

Note: 1.5 lambda is allowed for metal1/active contacts

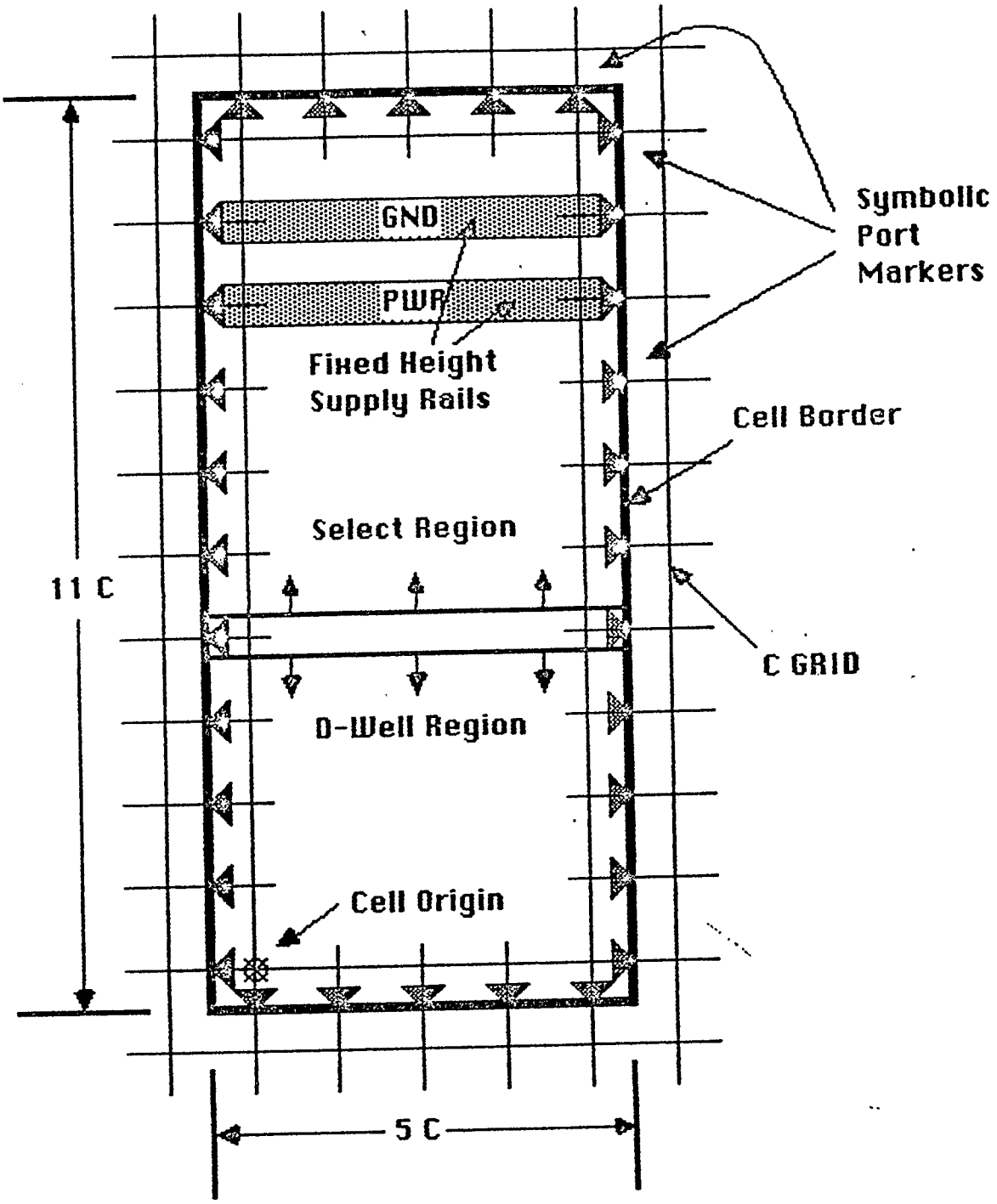


1.4 Select to Border

Note: -0.5 lambda is allowed for metal1/active contacts



1.5 Well to Border


GND
PWR
Fixed Height
Supply Rails
Symbolic
Port
Markers
Cell Border
Select Region
11 C
C GRID
D-Well Region
Cell Origin
5 C