1

MICROD

**National Library
of Canada**

**Bibliothèque nationale
du Canada**

Canadian Theses Service

Service des thèses canadiennes

**Ottawa, Canada
K1A 0N4**

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

**ANALYSIS OF AN ASIC DESIGN PACKAGE VIA
THE IMPLEMENTATION OF AN ECHO CANCELLER**

by

Kwing Kwan Li, B.Eng.

**This thesis is submitted to the Faculty of Graduate Studies and
Research in partial fulfilment of the requirements for
the degree of**

Master of Engineering

Ottawa-Carleton Institute for Electrical Engineering

Department of Electronics

Faculty of Engineering

Carleton University

Ottawa, Canada

Jan., 1990

© 1990, K. K. Li

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-59040-8

The undersigned recommend to the Faculty of Graduate Studies and Research the acceptance of the thesis :

**"ANALYSIS OF AN ASIC DESIGN PACKAGE VIA
THE IMPLEMENTATION OF AN ECHO CANCELLER"**

submitted by Kwing Kwan Li in partial fulfilment of the requirements
for the degree of Master of Engineering.

Prof. C. H. Chan
Thesis Supervisor

Prof. T. A. Kwasniewski
Thesis Supervisor

Prof. A. R. Boothroyd
Acting Chairman,
Department of Electronics

Carleton University

Jan., 1990

Abstract

The hardware design of an echo canceller for *ISDN* is reported in the thesis. The physical design of the echo canceller has been carried out using an ASIC design package - **SOLO 1200**.^{*} The echo canceller design serves as the evaluation vehicle for the software package.

The proposed echo canceller uses the sign-sign algorithm as the adaptive algorithm with the 2B1Q code assumed to be the line code used. The structure of the control path and the data path of the design have been carefully analyzed and optimized. A minimum complexity hardware architecture is thus achieved and the resulting gate count is only around 4,000. The design also incorporates a simple and efficient on chip test structure to ensure the testability of the design.

The results of the evaluation on the **SOLO 1200** design package are also summarized in this thesis. The design package is capable of handling a digital design with small to medium complexity. With the limited analog capability, the design package only allows a simple analog front end to be integrated together with the digital hardware.

^{*} **SOLO 1200** is a registered trademark of European Silicon Structures

Acknowledgement

The author would like to express his gratitude toward his project supervisors Dr. C. H. Chan and Dr. T. A. Kwasniewski for their guidance and the encouragement in the course of the work. The author would also like to express his thankfulness to Newbridge Microsystems and ES2, who have been providing technical support to the thesis.

The author is in debt to his good friend, Mr. T. M. Luk who provided invaluable opinions in refining the thesis.

Table of Contents

1	Introduction	1
1.1	Thesis Motivation and Objective	1
1.2	Thesis Format	2
2	ASIC Design and SOLO 1200	3
2.1	Introduction	3
2.2	Full Custom Versus Semi-Custom Design	3
2.2.1	<i>Standard Cell Approach</i>	4
2.2.2	<i>Gate Array Approach</i>	5
2.2.3	<i>Comparison of the Two Approaches</i>	5
2.3	Introduction to SOLO 1200	6
2.3.1	<i>Analog Capability of SOLO 1200</i>	6
2.3.2	<i>Hardware Aspects of SOLO 1200</i>	7
2.3.3	<i>Software Aspect of SOLO 1200</i>	9
2.4	Summary	21
3	System Consideration of an Echo Canceller	22
3.1	Introduction	22
3.2	ISDN and Echo Cancellation	22
3.3	Summary of the ANSI U-Interface Specifications	27
3.4	Systems Consideration of the Echo Canceller	28
3.4.1	<i>Filter Taps</i>	28
3.4.2	<i>Sampling Rate</i>	29
3.4.3	<i>Modification of the Algorithm for Efficient Implementation</i>	29
3.4.4	<i>Loop Gain Factor</i>	30
3.5	Functional Simulation of the Algorithm	31
3.5.1	<i>Data Generation program</i>	32
3.5.2	<i>Echo Cancellation Program</i>	34
3.5.3	<i>Simulation Results</i>	35
3.6	Summary	39
4	Hardware Design of the Echo Canceller	40
4.1	Introduction and Design Overview	40

4.2	Hardware Design Methodology	40
4.3	System Timing	42
4.4	Design of the Controller	45
4.4.1	<i>State Flow Diagram of the Controller</i>	48
4.5	Design of the Delay Line	52
4.6	Design of the Convolver and the Updater	55
4.6.1	<i>Design of the Convolver</i>	55
4.6.2	<i>The Updater</i>	58
4.7	Transfer Module	58
4.8	Design of the Rammod Module	60
4.9	Design For Testability	61
4.9.1	<i>Introduction</i>	61
4.9.2	<i>Scan Test Configuration</i>	64
4.9.3	<i>Testing of the Coefficient RAM</i>	67
4.9.4	<i>Observer Ports</i>	69
4.10	Simulation and Silicon Compilation	71
4.10.1	<i>Simulation for Design Correctness</i>	71
4.10.2	<i>Simulations for Testing</i>	71
4.10.3	<i>Silicon Compilation</i>	72
4.11	Summary	74
5	SOLO 1200 Design Features	75
5.1	Introduction	75
5.2	Schematic Entry and HDL	75
5.2.1	<i>Design Hierarchy</i>	76
5.3	Design Verification	78
5.3.1	<i>Logic Simulation</i>	79
5.3.2	<i>Timing Verification</i>	79
5.3.3	<i>Pre-layout Simulation</i>	80
5.3.4	<i>Simulation of the Analog Pads</i>	80
5.4	The Physical Layout System	81
5.4.1	<i>Exploration of the Placement Algorithm</i>	83
5.4.2	<i>Placement Control</i>	84
5.4.3	<i>Exploration of the Routing Software</i>	84
5.4.4	<i>Routing Control</i>	86
5.5	Comparison of Library	88

5.5.1	<i>Library Cells</i>	88
5.5.2	<i>Performance</i>	90
5.6	Notes on the SOLO Cells	90
5.7	Optimization Control	92
5.8	Tabulation of Parameters	95
5.9	Summary	96
6	Conclusion	98
6.1	Summary	98
6.2	Future Work	101
7	Appendices	103
	Appendix A	103
	Appendix B	124
	Appendix C	137
	Appendix D	140
	Appendix E	142
	Appendix F	143
	Appendix G	145
	Appendix H	146
	Appendix I	147
8	References	149

List of Figures

2.1	<i>The Construction of the Logic Gates Using PN Transistor Pair</i>	8
2.2	<i>The Floor Plan of the OA</i>	10
2.3	<i>The Pad Placement of the OA</i>	11
2.4	<i>The Power Supply Scheme of the Core Array</i>	11
2.5	<i>A Sample Net-list Description Using the MODEL Language</i>	14
2.6	<i>A Sample Position File Which Determines the Cell Placement</i>	18
2.7	<i>The Bonding Diagram Generated by Package</i>	20
2.8	<i>The List of Items Checked by SHIPDES</i>	21
3.1	<i>Full Duplex Transmission Using Echo Cancellation and Hybrid</i>	23
3.2	<i>The Filter Structure for the LMS Algorithm</i>	25
3.3	<i>The Structure of the Data Generation Program</i>	32
3.4	<i>The Hypothetical Impulse Response of the Echo Path</i>	33
3.5	<i>The Structure of the Echo Cancellation Program</i>	34
3.6	<i>The Residue Error after Echo Cancellation - 8b Adc</i>	36
3.7	<i>The Squared Residue Error after Echo Cancellation - 8b Adc</i>	37
4.1	<i>The Applied Design Method</i>	41
4.2	<i>The Operation Flow Diagram of the Echo Canceller</i>	42
4.3	<i>The Functional Block Diagram of the Echo Canceller</i>	43
4.4a	<i>The System Timing Diagram</i>	44
4.4b	<i>The Timing Relationship between etx2 and eact</i>	44
4.5a-b	<i>The Top Level Schematic Diagram of the Echo Canceller</i>	46-47
4.6	<i>The State Flow Diagram of the Echo Canceller</i>	49
4.7	<i>Legend of the State Flow Diagram</i>	50
4.8	<i>The Structure of the Fhflop</i>	50
4.9	<i>The Simulation results of the Fhflop</i>	51
4.10	<i>The Design of the Delay line</i>	53
4.11	<i>The Structure of the Convolver and the Updater</i>	56
4.12	<i>The Schematic Diagram of the Transfer Module</i>	59
4.13	<i>The Memory Mapping of the Coefficient RAM</i>	61
4.14	<i>The Structure of the Rammod Module</i>	62
4.15	<i>The Modified Scan Flip Flop</i>	65
4.16a	<i>Sequence in Changing from the Normal mode to the Scan Mode</i>	66
4.16b	<i>Sequence in Changing from the Scan Mode to the Normal Mode</i>	66
4.17	<i>The Cell Structure of the Mux8s</i>	70

4.18	<i>The Cell Structure of the Mux8o</i>	70
4.19	<i>The Layout of the Echo Canceller System</i>	73
5.1	<i>Design Tree of the Echo Canceller</i>	77
5.2a	<i>The Layout of the Register Bit Cell (No Intervention)</i>	82
5.2b	<i>The Layout of the Register Bit Cell (With Intervention)</i>	82
5.3	<i>The Folding of the Design Row Around the Core Array</i>	83
5.4	<i>The Wasting of Area Caused by Unequal Column Height</i>	85
5.5	<i>The Reduction of Loading by Row Flipping</i>	85
5.6	<i>The Routing of the 'Difficult Example' by the SOLO Router</i>	87
5.7	<i>The 1's and 0's Catching of the JK Flip Flop</i>	91
5.8	<i>Effect of the 'On' Signal on the A/D Converter Output</i>	92
5.9a-b	<i>The Class of Circuit Which Will Not Be Optimized</i>	93
5.9c-d	<i>The Class of Circuit Which Can Be Optimized</i>	94
A.1.1a	<i>Generation of the Controller State Variables</i>	106
A.1.1b-c	<i>Generation of the Controller Output Signals</i>	107-108
A.1.2	<i>The Converter Clock Generator - Div6ctr</i>	109
A.1.3	<i>The Simulation of the Converter Clock Generator</i>	109
A.1.4	<i>The Schematic Diagram of the Soc</i>	111
A.1.5	<i>The Simulation Result of the soco Signal Generation</i>	111
A.1.6	<i>The Simulation Result of the Controller State Transitions</i>	113
A.1.7	<i>The Simulation Result of the Controller Output Signals</i>	114
A.2.2.1a	<i>The Schematic Diagram of Compa</i>	115
A.2.2.1b	<i>The Schematic Diagram of Compcell</i>	115
A.2.2.2a	<i>The Schematic Diagram of Mux16b</i>	117
A.2.2.2b	<i>The Schematic Diagram of Muxcellb</i>	117
A.4.1	<i>The Simulation Result of the Ramctr</i>	121
A.4.2a	<i>The Schematic Diagram of Ctrcell</i>	122
A.4.2b	<i>The Schematic Diagram of Ctrcell2</i>	122
A.4.2c	<i>The Schematic Diagram of the Ramctr</i>	123
B.1.3.1	<i>The Design Flow Diagram of SOLO 1200</i>	128
B.4.1.1	<i>The Illustration of the 1-D Placement Algorithm</i>	133

List of Tables

3.1	<i>The Look Up Table for the 2B1Q Line Code</i>	27
3.2a	<i>The Simulations of the Modified Sign Sign Algorithm - 14b Adc</i>	38
3.2b	<i>The Simulations of the Modified Sign Sign Algorithm - 15b Adc</i>	38
4.1	<i>The Breakdown of the Multiplication Operations</i>	60
4.2	<i>Two Test Cases of the Transfer Module</i>	60
4.3	<i>The Code Table for the Memory Test Modes</i>	68
5.1	<i>The Comparison of the 1-D and 2-D Algorithms</i>	84
5.2	<i>The Performance of Various Router in Routing the 'Difficult Example'</i>	86
5.3	<i>Comparison of the Standard Cell Libraries</i>	89
5.4	<i>Comparison of the Gate Delay of Two Inverter</i>	90
5.5	<i>Various Parameters of SOLO 1200</i>	95
A.1a	<i>State Assignment Table</i>	104
A.1b-e	<i>Karnaugh Map of x_1, x_2, x_3, x_4</i>	104
A.2.3.1	<i>Eight Test Cases of the Convolver</i>	118
A.2.4.1	<i>Generation of the Updating Quantity</i>	119

List of Symbols and Abbreviations

1-D	: one dimensional
2-D	: two dimensional
A/D	: analog to digital
D/A	: digital to analog
ASIC	: application specific integrated circuit
ATPG	: automatic test pattern generator
CAD	: computer-aided design
CIF	: CALTECH intermediate form
CLA	: carry look-ahead adder
CMOS	: complementary metal oxide semiconductor
DFT	: design for testability
ES2	: European Silicon Structure
ESD	: electrostatic discharge
HDL	: hardware description language
IC	: integrated circuit
IDL	: intermediate hardware description language
ISDN	: Integrated Services Digital Network
LMS	: least mean square algorithm
LSB	: least significant bit
LSSD	: level sensitive scan design
NT	: network termination

OA	: optimized array
Op. Amp.	: operational amplifier
OS	: operating system
PLA	: programmable logic array
PN	: P-channel and N-channel
ROM	: read-only memory
TRIO	: Telecommunication Research Institute of Ontario
VLSI	: very large scale integration
VHDL	: very high speed integrated circuit hardware description language

Note :

SOLO 1200 is a registered trademark of ES2

SUN is a registered trademark of SUN Microsystems Inc.

APOLLO is a registered trademark of APOLLO Computer Inc.

VAX is a registered trademark of Digital Equipment Inc.

Chapter 1

Introduction

1.1 Thesis Motivation and Objective

During the past decade, the "Application Specific Integrated circuit" (ASIC) has gradually emerged as an acceptable alternative to standard and programmed devices in high performance electronic circuits and systems design. Literally, ASIC means an IC tailored to the specific requirements of a particular application. Foundries such as Texas Instruments and LSI Logic provide and support ASIC design facilities and services to the ASIC user community. The ASIC design approach adopted by each foundry varies. The amount of automation used in ASIC design also varies from foundry to foundry. However, there are similarities existing among foundries. The evolving common concept in this field involves : 1) the development and maintenance of a set of proprietary design cell libraries and 2) the integration of the design cell libraries and various IC CAD programs to form a complete design environment. Each foundry also formulates its own version of the design flow and defines the required design steps in the design flow. The structuring of the design flow is based on the foundries' 1) design experience and design philosophy and 2) the design tools which are available. At each design step, the required design checks or design documentations are also predefined.

The degree of automation in VLSI design has come to a point that a designer does not have to deal with the physical level design details. Automatic layout programs can be used to generate the layout information in several minutes for a complex design or in a few seconds for a simple logic circuit using a state-of-the-art engineering workstation - e.g. a SUN 3/160. With the combination of efficient design tools and a structured design methodology, a smooth, short turn-around time and high yield IC design process is envisioned.

In this project, an ASIC design package **SOLO 1200** which is available from a foundry known as European Silicon Structure (ES2) is being characterized. ES2 uses direct e-beam

write on wafer for pattern generation, which reduces the turn-around time. It also permits low volume production quantity (around 10 - 1000 chips) which is more cost effective. These factors are potentially appealing to low volume customers such as university-based researchers. The **SOLO 1200** design package also provides mixed analog/digital design capability. The capability to integrate the analog front-end and the digital logic together is an asset to most projects in digital signal processing or telecommunication applications.

An echo canceller is considered to serve as the evaluation vehicle for the **SOLO 1200** design package. The design package is to be characterized through the step by step implementation of the echo canceller from system block diagram to CMOS layout geometries. The design choice is based on the current research interest of the "Telecommunication Research Institute of Ontario"(TRIO). The Institute has recently been heavily involved in the hardware and software development for the *Integrated Services Digital Network (ISDN)*. The echo canceller is implemented using digital hardware which is dedicated to the execution of the echo cancellation algorithm. Simple analog front-end elements, including the A/D and D/A Converters, are used in the design to test the mixed-analog and digital capability of the package. The basic evaluation criteria set for **SOLO 1200** are : 1) the tool must be easy to learn, 2) the tool must be easy to use, 3) the use of the tool must be able to shorten the design time, 4) the layout must be correctly compiled, and 5) the compiled layout must be area efficient.

1.2 Thesis Format

This thesis consists of six chapters. Chapter 2 provides an overview on current ASIC design methodology and approaches. The hardware and software aspects of **SOLO 1200** are presented in this chapter. Chapter 3 introduces the principle of echo cancellation and its application in *ISDN*. Some properties of the cancellation algorithm which is being implemented in the design are examined. Finally, the system architecture, design considerations and the design methodology are outlined. Chapter 4 describes the design detail of the echo canceller system. Issues of design for testability considered in the system are also described. The implementation of the on chip test facilities is described in the last section of the chapter. In Chapter 5, the design package is described in details. The experience in using the design software is also summarized. Chapter 6 summarizes the previous chapters and provides concluding comments on the design package and the design architecture.

Chapter 2

ASIC Design and SOLO 1200

2.1 Introduction

This chapter is divided into two parts. The first part serves as a general introduction to ASIC design. The design approaches commonly used in the field are also discussed. The second part of this chapter outlines the software and hardware features of **SOLO 1200**. To describe **SOLO 1200**, the following conventions are used : the names of the design tools are in capital letters and bold face type. The keywords of **SOLO 1200** are italicized.

2.2 Full Custom Versus Semi-Custom Design

With the advancement in the integrated circuit (IC) technology, the scale of integration increases. Many board level systems can now be easily integrated into a single chip. The advantages of integrating a system into a single chip include the improvement in speed, and the reduction of area and power consumption.

For a complex circuit with a large scale of integration, many designer-months of effort are spent to realize the design. Actually, the scale of integration has already exceeded the limit which a human designer can comfortably handle. In order to ensure the correctness and the quality of a design within a short design cycle, CAD tools are used extensively in integrated circuit design. Attempts have also been made to automate the whole design process.

The term application specific IC (ASIC) design is a synonym to the custom design of an IC, which means the design of a micro-electronic circuit to meet a particular customer's requirements in contrast with the conventional off-the-shelf standard design. There are two different methods which are commonly applied in ASIC : the full custom design method and the semi-custom design method. In the full custom design method, the circuit components used in the design are hand-crafted and hand-packed at the device level with or without the use of CAD tools. On the other hand, in the semi-custom design method pre-characterized

and pre-laid out standard cells are used. Nowadays, most of the design tasks involved in the semi-custom design method are automated.

In the full custom design method, optimization is carried out manually at the device level. The design time and the design cost are relatively high. In the semi-custom design method, the use of standardized circuit components to meet design specifications of a wide range of applications causes the design to use more area, consume more power and run slower. The limited capability of the current automated design tools also imposes a limit on the performance of the semi-custom design method. Since the user does not need to layout and characterize the standard components, the design time used in the semi-custom design method is greatly reduced. Also the functionality of the standard components is guaranteed.

Based on the above considerations, circuits with mid-range performance requirements and small production quantities are commonly designed using the semi-custom approach. On the other hand, the full custom approach is more often used for high performance circuits which demand volume production.

The (two) design approaches which are commonly used in semi-custom IC design are :
1) the standard cell approach. 2) the gate array approach.

2.2.1 Standard Cell Approach

In the standard cell approach, one or more standard cell libraries with a set of pre-designed standard cells are available to the designer. Usually, the standard cells are being treated as black boxes. The only information available to the user about the cells is :

- 1) the functional specification of the cells.
- 2) the I/O interface of the cells.
- 3) the grid size and the area of the cells.

In general, the internal structure of the standard cells is not important to the user. However, the user can still instantiate the cells and view the structure of the cells through the layout system. The standard cells are usually designed at the gate level (one level above the transistor level). Functional macros at a higher level may also be included in the library.

To design using the standard cell approach, the user needs only to describe his circuit at the gate level or any functional levels which are supported by the cell libraries. The

circuit components are then placed and routed according to the circuit description.

2.2.2 Gate Array Approach

In the gate array approach, pre-fabricated wafers with circuit components arranged in an array structure are available to the designer. All the physical layers of the wafers, except the interconnection layer, are pre-processed. The designer can build his design by mapping the design entities onto the array components. This is the same as placing the standard cells. The interconnection patterns between the array components are then specified. There is no standard on the type of components which should be available as the building blocks of the array. It can be as complex as a counter, as simple as several transistors, or the array can be divided into several partitions with each partition being dedicated to a particular type of component.

2.2.3 Comparison of the Two Approaches

In both approaches, the design process can either be manually carried out or automated. If the design process is automated, the designer needs only to describe the circuit topology at the appropriate abstract level. The realization of the circuit will be left to the automated design tools. The final output of the tools is the physical layout information which will be used to produce the mask set for fabrication.

In the standard cell approach, all the mask layers are required. In the gate array approach, all but one of the device layers are pre-fabricated. Only the mask layer for interconnection is required to be specified. This results in a shorter production cycle time. Since the gate array is designed for general applications, the total number of components in the array is likely to be larger than the total number of components required by a particular application. The unused components are the main source of area inefficiency in the gate array approach. In the standard cell approach, since only the required cells are instantiated, the utilization of area is more efficient. Also, because there is no pre-fabricated wafer, channel routing can be performed with greater flexibility. In gate array, the interconnections have to route through the available channels between the array components. This constraint is removed in the channelless gate array. In the channelless gate array, the interconnections are routed over the unused array components.

2.3 Introduction to SOLO 1200

In the industrial environment, the semi-custom design approach provides to the designer an easy way to build and test a new system architecture. This thesis proposes a system architecture for an echo canceller. Through the implementation of the system, the ASIC design package **SOLO 1200** which is provided by the European Silicon Structure (ES2) is characterized.

SOLO 1200 is a complete ASIC design package which consists of several standard cell libraries and IC CAD programs. The package is embedded with the semi-custom design methodology advocated by ES2, and is organized to provide a streamlined design flow. The output data file of one program can be fed directly as input data to the next program.

The software under evaluation is **SOLO 1200** version 2.2.0. This package can be installed on SUN, APOLLO or VAX workstations. There is also a PC version available. The version acquired by Carleton University is currently installed on SUN 3/160 workstations. In this section, special features of the package will be highlighted, and its hardware and software aspects examined.

2.3.1 Analog Capability of SOLO 1200

Traditionally, the semi-custom design approach was limited primarily to digital circuits. The main reasons are :

- 1) analog circuits are often specific to applications; thus it is hard to standardize the analog circuit components.
- 2) the strong process dependence nature of the analog components.
- 3) the incapability of design verification tools for mixed mode circuits.

Nowadays, it is felt that the semi-custom approach should also be applied to the analog circuits, i.e. the integration of the analog and the digital circuits in one ASIC chip. The aims are to reduce the design time, design cost and to increase the utilization of the analog circuit components. Based on these considerations, many foundries try to improve their standing by including an analog cell library in their design libraries. Typically, for a particular analog device such as an Op. Amp., the cell library will include several versions of the same device

having different performance specifications. The designer will then select the version which has the specifications closest to his requirements. Reduction in the circuit performance is inevitable when sub-optimal analog devices are being used. In this analog standard cell approach, the design of the analog cells strongly depends on the technology. The problem of library modification, induced by process upgrade, is more complicated in the analog cells than in the digital cells. Modification of the analog devices in the libraries also takes more time.

Another approach involves the design of an intelligent compiler which can accept the specification of a particular analog device. The compiler either uses some optimization algorithm or data-base look-up table to generate the optimal device architecture and transistor sizes of the analog device under the constraints of area, power and performance specifications. Modification to the look-up table or the optimization algorithm may be required if the technology changes.

SOLO 1200 is a design package which allows the design of mixed analog and digital circuitry. An analog standard cell library is included in the package, and it provides common analog devices such as Op. Amp., Voltage Controlled Oscillator, A/D and D/A Converters. There is no inclusion of capacitors and resistors as standard cells in the library. Obviously, the design of this analog cell library is not intended to provide full blown analog design capability. For example, one cannot use the analog cells provided in the library to build a self-calibrating A/D Converter, which may require the use of ratioed capacitors. However, one can use these standard cells to design an analog front-end for a digital circuit. One can also use the Oscillator provided in the standard cell library to generate the clocking signal for the digital circuit.

2.3.2 Hardware Aspects of SOLO 1200

The underlying technology of **SOLO 1200** is a $2\ \mu\text{m}$ double layer metal N-well process. The $2\ \mu\text{m}$ refers to the drawn gate length. The process consists of 12 mask layers. an N-guard ring and P-guard ring are used, but not shown by the physical layout system. **SOLO 1200** uses a cell-based design method which is known as the *optimized array (OA)*. OA looks like a mixture of the gate array and the standard cell approach. Similar to the standard cell approach, **SOLO 1200** includes several standard cell libraries. All mask layers must be specified for fabrication. Although OA does not have a pre-fabricated wafer,

it does resemble the gate array. First of all, the layout of the cell based design has a highly regular array structure. Similar to the gate array, the OA has a design primitive (or array component) which is one PN transistor pair. Using ES2's terminology, one PN transistor pair constitutes one *stage* of the array. All the digital standard cells, with the exception of the handcrafted I/O cells, are designed by configuring one or more PN transistor pairs together.

Figure 2.1 shows how the logic gates are constructed out of the design primitives. Note that there is a one-half *stage* spacing at both sides of the gate. As a result, the total inter-gate spacing is one *stage*. This is the reason why an inverter is said to be a two *stage* device while it consists of only one PN pair, as stated in the manual. Similarly, the two input NAND gate consists of three *stages*.

Fig. 2.1 The construction of the logic gates using PN transistor pairs

The floor plan of the OA is shown in Fig. 2.2. The PN transistor pairs are placed side by side to form a row. Rows are stacked on top of each other to form a column. The rows and columns thus form an array with routing highways¹ running between the rows and the columns. The number of *stages* per row, the number of rows per column and the number of

¹ ES2 uses 'routing highway' for routing channel, and 'channel' for track

columns per array are user definable. The OA can be considered as a grid free array. The inter-column spacing and the inter-row spacing are flexible. However, vertical grid lines do exist within each column. The grid spacing within each column is equivalent to the width of one *stage*.

Channels in the row and the column highways are routed in the first layer metal. The second layer metal acts as a cross-over to bring the signals from the highways to the gates. The user can suppress the use of the second layer metal for interconnection. In this case, polysilicon wire will act as a cross-under to replace the second layer metal. The width of the routing highway between two rows or between two columns depends on the number of required channels.

In Fig. 2.2, the core array comprises only the digital circuitry. The analog cells are placed on the peripheral of the core array, i.e. the four sides of the array will accommodate both the analog cells, the power supply pads and the digital pads. Hence, the analog cells are also known as *analog pads* in contrast to the I/O pads. To improve the noise immunity, *analog pads* and the digital pads are powered separately.

The *analog pads* and the digital pads can be placed in a variety of ways. An easy way to place the pads is shown in Fig. 2.3. The *analog pads* and the digital pads are separated into two sections. The analog power supply pads and the analog ground pads must be placed at the two ends of the analog section, since these two pads are used to break and separate the *analog pad ring* from the digital pad ring. The digital power supply pads can be placed at any location in the digital section. There are some restrictions on pad placement which can be found in the reference manual vol. 2 [2].

The digital core array is powered by the digital pad ring at the top right and the bottom left corners. The floor plan of power distribution is shown in Fig. 2.4.

2.3.3 Software Aspect of SOLO 1200

A total of 17 CAD programs and 6 standard cell libraries are included in the design package. In this section, the standard cell libraries are described first. A brief introduction is then given to the design programs.

2.3.3.1 Design Library

Each of the 6 design libraries consists of the following information :

through the transistor area. However, by specifying a sequence of commands in the manual routing file, the *.rmf* file, the user is allowed to force a signal to route through the vertical highway instead of the transistor area. The command has the general format : *type, column, row, side*. The following command sequence in an *.rmf* file forces net 33 to route from the bottom highway to the first minor highway through the vertical highway at the left side of the column. The following abbreviations are used : B = bottom highway, V = minor highway, L = vertical highway at the left side of the column.

Example :

```
33
B,1,L
V,1,1,L
```

The layout information generated by **ROUTE** is stored in an *.rte* file. The program **DRAW** reads the *.rte* file and generates the CALTECH Intermediate Format (CIF) of the design. **DRAW** also extracts the parasitic capacitance loading at each net and stores it in an *.lda* file. This file is used by **FANOUT** under the *load* option. With the loading information available, **EXERT** can be re-run with the *load* option to verify the timing correctness of the design.

The design artwork produced by **DRAW** can be viewed by executing the program **ARTVIEW**. **ARTVIEW** allows the user to view the chip layout only. To prevent the user from violating the design rules, **ARTVIEW** is not equipped with editing capability. To alter the layout, the user can either alter the schematic diagram or re-run the layout programs with different parameters.

The program **PACKAGE** may be executed any time after running **DRAW**. **PACKAGE** helps the user to package his design. It also allows the user to choose the type of package to be used. It checks if all the bonding pads are adequately powered and if the bonding rules are observed. A bonding diagram as shown in Fig. 2.7 is generated if **PACKAGE** successfully packages the design.

2.3.3.5 Design Validation

After the layout of the design has been generated and the timing of the design has been verified by **EXERT**, the design can be validated for fabrication. To complete the design

Fig. 2.2 The Floor Plan of OA

Fig. 2.3 The Pad Placement of the OA

Fig. 2.4 The Power Supply Scheme of the Core Array

- 1) the symbol of the standard cells for schematic capture.
- 2) the structural description of the standard cells.

Among these six libraries, the **BASELIB**, the **LOGICLIB**, the **ES2CELL** and the **74LIB** constitute the digital logic cell library. **BASELIB** contains simple logic cells such as AND, OR gates, Latches and Flip Flops. The other three logic cell libraries contain more complicated cells. The **74LIB** is of particular interest. It contains the macro cells which are the CMOS equivalent of the 74 family.

BASELIB is the most fundamental library of all the logic libraries. The logic cells in the other logic cell libraries are built upon the logic cells in the **BASELIB**. The cell definitions in other logic libraries contain calls to the **BASELIB**. The **BASELIB** cells are wired directly by the **GATE** program which has the most compact layout.

The two pad libraries, **PADLIB2** and **PADLIB3**, provide digital supply pads and I/O buffers to the designer. I/O buffers with TTL compatibility are also provided in these two libraries. The basic difference between the two libraries is the size of the pads. The digital pads in **PADLIB2** are tall and narrow while **PADLIB3** contains pads which are wider in width. **PADLIB2** should be used for I/O intensive design. On the other hand, **PADLIB3** should be used for core limited design, i.e. the core array is relatively large and the number of input/output signals is limited.

The **ANALOGLIB** consists of *analog pads* such as Op. Amp., Voltage Comparator, Voltage Reference, Oscillator, A/D and D/A Converter, analog multiplexer and analog power supply pads. **ANALOGLIB** also includes several digital pads. These digital pads should be used in conjunction with the designs which include *analog pads*. Digital pads in **PADLIB2**, **PADLIB3** should not be used whenever *analog pads* are used. The **BLOCKLIB** contains two static RAM modules, 64×8 bits and 256×8 bits. In the future, PLA and ROM will be included in the **BLOCKLIB** by the foundry. The *analog pads* and the RAM modules are full custom designed. Only the outline and the second metal layer of these cells can be viewed through the layout system. Other details are hidden and will be filled in by ES2 in the factory.

Although static RAM is an analog device, it is being treated as a digital circuit and placed in the core array. The RAM module is considered to be 1 row high and 146 stages wide by the design programs. However, in mapping a design onto the OA, the designer has

to be aware that the height of the memory module actually spans more than 1 row.

2.3.3.2 Schematic Entry

The very first step in circuit design is to describe the circuit topology at the appropriate level (functional level, register transfer level, gate level or transistor level). The description of the hardware serves two purposes :

- 1) as the documentation of the design.
- 2) allowing the design tools to manipulate the design.

The designs are usually described in graphical format or in text format. The design description can either be manipulated by the design tools directly, or be compiled into a simpler binary format for further processing. In **SOLO 1200**, there are two ways of describing the circuit topology

- 1) coding the net-list manually using the **MODEL** language.
- 2) using the schematic editor, **DRAFT**

MODEL is a high level hardware description language (HDL). It describes the circuit topology at or above the gate level. The design description in **MODEL** is stored in the design file with *.mod* extension. **MODEL**'s language structure is similar to those high level programming languages such as PASCAL or C. A sample segment of the **MODEL** code is shown in Fig. 2.5. *Part* is the basic design unit of **MODEL**. It is an entity similar to a procedure in the programming languages. A *part* must be declared before it can be used or instanced. The declaration of a *part* includes :

- 1) *part* name.
- 2) the I/O interface declaration of the *part*. This corresponds to the formal parameter declaration of the procedure.
- 3) the internal structure of the *part* which consists of elementary building blocks (*parts*).

Just like the procedure in PASCAL, *part* in **MODEL** supports both modular and hierarchical design. To describe the hardware, the user has the freedom to use the top-down method in which he derives the functional block diagram from the specifications. The functional block diagram is then decomposed level by level, and maps to the *parts*

```
{ Model file created on Sun Mar 12 1989 at 16:23:31 }
{ Design generated from design file echocan }
```

```
Part soc [rset, soci, sysclk, adclk] -> soco
```

```
Signal slc2x1, slc3x1, slc2x3, slc4x2, slc4x1, slb4x1, slc3x3
```

```
{ Output from sheet s1 }
```

```
{ Instance name is $b4 }
not [rset] -> slb4x1
{ Instance name is $c2 }
bdfcet [adclk,Vdd,soco] -> slc2x3,--
  { output qbar missing}
{ Instance name is $c5 }
and [slc4x2,slb4x1] -> slc3x3
{ Instance name is $c6 }
bdfcet [soci,Vdd,slc3x3] -> soco,--
  { output qbar missing}
{ Instance name is $c4 }
bdfcet [sysclk,Vdd,slc3x1] -> slc4x1,--
  { output qbar missing}
{ Instance name is $d4 }
not [slc4x1] -> slc4x2
{ Instance name is $c2y2 }
not [slc2x3] -> slc2x1
{ Instance name is $c3 }
not [slc2x1] -> slc3x1
```

```
End
```

```
External Signal rset, soci, sysclk, adclk
```

```
External Signal soco
```

```
soc [rset, soci, sysclk, adclk] -> soco
```

```
End Of File
```

Fig. 2.5 A Sample Netlist Description Using MODEL Language

which are available in the standard cell libraries. The designer can also choose to use the bottom-up or the middle-out design method. *Parts* are instanced by assigning actual I/O signals to the formal I/O parameters. The instancing of a *part* is equivalent to a call of the *part* to the design libraries. The basic statement in the language to instance a *part* has the form :

part name [input] $\rightarrow$ output

Pad is another important entity in **MODEL**. It refers to the digital pads or *analog pads* placed at the peripheral of the chip. **MODEL** allows the user to declare a *pad*, instance the *pad* and determine its position at the four sides of the chip. Language constructs such as *For* loop, *If .. Then .. Else* statements are also available to increase the power of the language.

DRAFT is a menu driven schematic editor. Unlike **MODEL** which uses a textual description, **DRAFT** presents the circuit topology as a symbolic layout. The *parts* or the *pads* are displayed as symbols with wires connected between them. The graphic information is stored in a file with a *.dft* extension. Based on the symbolic information, **DRAFT** is capable of generating the **MODEL** code. The *.mod* file, either hand-coded or generated by **DRAFT**, is compiled by the **MODEL** compiler.

The **MODEL** compiler is capable of :

- 1) performing the syntactic and semantic check on the *.mod* file.
- 2) generate the net number for each net in the circuit. The net number is useful for simulating the circuit and for debugging the circuit.
- 3) assign an identification number to each *part* instance in the design.
- 4) translate the **MODEL** code into the IDL format. IDL stands for the intermediate design language. The translated code is stored in an *.idl* file which is a binary file. The file contains details of the transistors and their interconnection.

Since the *.idl* file is not a text file, the net number and the instance number generated during the compilation process cannot be obtained directly. Program **EXPAND** must be executed to reverse the compilation process. **EXPAND** reads the *.idl* file and generates an *.exp* file. The *.exp* file is a text file. It is a flattened version of the *.mod* file, with each instance assigned an instance number and each net assigned a net number.

2.3.3.3 Simulation

Simulator **EXERT** is included in **SOLO 1200** to verify the logic and timing correctness of the design. **EXERT** simulates the digital portion of the design at the switch level, while the analog section is simulated at the behavioral level. Usually before the simulation is carried out, the program **FANOUT** will be executed to examine the net loading. **FANOUT** calculates the *relative fanout* at each net.

Relative fanout in **SOLO 1200** can be defined as :

$$\text{Relative fanout} = \frac{\text{Absolute fanout}}{\text{Drive Strength}} \quad (2.1)$$

In **SOLO 1200**, an *absolute fanout* of one (or one unit load) is defined as the input gate capacitance of one PN transistor pair. An inverting buffer with five PN transistor pairs connected in parallel represents an *absolute fanout* of five. Each logic gate also has a drive strength parameter associated with it. It represents the capability of the gate to pull-up or pull-down the logic level of the net which is being driven by the logic gate, e.g. an inverter has a drive strength of one.

If the *relative fanout* of a net is greater than a certain value (this value is not fixed, but usually equals twelve), an error message will be generated to notify the designer that the loading on the net is too large. The circuit may not function properly. **FANOUT** can be executed either before or after the physical layout is generated. If **FANOUT** is executed after the physical layout has been generated, one can include the parasitic loading on the signal wires by using the *load* option. Under this option, **FANOUT** will convert the parasitic loading of a net to an equivalent number of unit loads, and add the number to the absolute fanout of the net.

If the loading condition on each net is satisfactory, one can proceed to simulate the circuit. To simulate the circuit properly, the user is required to specify whether the minimum, the maximum or the typical values of the rise time, the fall time and the propagation delay of the logic gates should be used in the simulation. The user should also specify the temperature range and power supply range in which the circuit will operate. Three different set of values can be specified :

- 1) 0 to 70 degree Celsius, 4.5 - 5.5 volts (Commercial).

- 2) -40 to +85 degree Celsius, 4.5 - 5.5 volts (Industrial).
- 3) -55 to 125 degree Celsius, 4.5 - 5.5 volts (Military).

After simulation, **EXERT** stores the simulated waveforms which are specified by the user into a *.trc* file. To view the signal waveforms, one can execute the **WAVE** program. **WAVE** displays the signal waveform on a graphic terminal. It can also plot the signal waveforms using a HP7380 plotter.

2.3.3.4 Physical Layout Generation

There are four programs (**PLACE**, **GATE**, **ROUTE** and **DRAW**) involved in generating the physical layout.

The **PLACE** program places the circuit part onto the OA and tries to minimize the net-length and the area. **PLACE** uses two placement algorithms, the 1-D algorithm and the 2-D algorithm. Upon invocation, **PLACE** will estimate the required OA dimensions, i.e. the number of columns, the number of rows/column and the number of *stages*/row. By default, all columns will have the same number of rows, and all rows will have the same number of *stages*. To increase the flexibility, **PLACE** allows the designer to use different number of rows/column and different number of *stages*/row for different columns.

If requested by the user, **PLACE** can generate a *.pos* file, which specifies the locations of the logic gates within the array. An example of the *.pos* file is shown in Fig. 2.6. To affect the placement, the user can modify the position of the logic gates in the file, and re-run **PLACE** with the *replace* option.

The **GATE** program takes the placement data produced by the **PLACE** program and generates the layout information which specifies the internal wiring pattern of each gate in the design. The layout information is stored in a *.gat* file. The wiring is created by selecting an appropriate connection pattern to combine a group of *stages* into the required standard cell. **GATE** runs without user intervention.

The **ROUTE** program performs highway routing and pad assignment based on the layout information stored in *.gat* file. **ROUTE** allows the user to specify some signals as *critical* in a *.crt* file. The *critical* signals have higher priorities and are routed before any other signals are routed. **ROUTE** allows the user to affect the routing process. Under automatic operation, the connections between adjacent minor highways are normally routed

```
{   Position file for MODEL file ctrcell2.idl
{-----
```

OPTIMISED

Metal_Layers=2

PROCESS_FILE_PRIVATE=/usr/kirk/solo12/tech/ECDM20A.ppf

PROCESS_FILE_PUBLIC=/usr/kirk/solo12/tech/ECDM20A.prc

MODEL version 5.6 Tue Mar 7 1989 20:05:53

PLACE version 5.6 Tue Mar 7 1989 20:06:28

Total number of parts: 24
 Total number of stages: 83
 Largest wire number used: 69
 Largest external wire: 22
 Maximum number of parameters: 25
 Total number of parameters: 131
 Maximum wire use: 13
 Total number of pads: 0
 Number of standard cell types: 0
 Number of block types: 0
 Number of columns: 1
 Maximum cells per column: 72
 Number of cells in column 1: 72
 Maximum rows per column: 2
 Number of rows in column 1: 2

Column 1 Row 1 L -> R

AND.2	15 17 0 21	{ Pos 1 Size 4 Out 21
NOR.4	21 25	{ Pos 5 Size 2 Out 25
AND.5	27 13 21 0 29	{ Pos 7 Size 5 Out 29
OR.6	30 29 0 35	{ Pos 12 Size 4 Out 35
AND.7	28 13 25 0 30	{ Pos 16 Size 5 Out 30
BLANK		{ Pos 21
OR.8	34 35 0 33	{ Pos 22 Size 4 Out 33
AND.9	16 18 0 22	{ Pos 26 Size 4 Out 22
NOR.11	22 26	{ Pos 30 Size 2 Out 26
AND.12	27 14 22 0 32	{ Pos 32 Size 5 Out 32
OR.13	31 32 0 34	{ Pos 37 Size 4 Out 34
BLANK		{ Pos 41

End { of Column 1 Row 1

Fig. 2.6 A Sample Position File Which Determines the Cell Placement

through the transistor area. However, by specifying a sequence of commands in the manual routing file, the *.rmf* file, the user is allowed to force a signal to route through the vertical highway instead of the transistor area. The command has the general format : *type, column, row, side*. The following command sequence in an *.rmf* file forces net 33 to route from the bottom highway to the first minor highway through the vertical highway at the left side of the column. The following abbreviations are used : B = bottom highway, V = minor highway, L = vertical highway at the left side of the column.

Example :

```
33
B,1,L
V,1,1,L
```

The layout information generated by **ROUTE** is stored in an *.rte* file. The program **DRAW** reads the *.rte* file and generates the CALTECH Intermediate Format (CIF) of the design. **DRAW** also extracts the parasitic capacitance loading at each net and stores it in an *.lda* file. This file is used by **FANOUT** under the *load* option. With the loading information available, **EXERT** can be re-run with the *load* option to verify the timing correctness of the design.

The design artwork produced by **DRAW** can be viewed by executing the program **ARTVIEW**. **ARTVIEW** allows the user to view the chip layout only. To prevent the user from violating the design rules, **ARTVIEW** is not equipped with editing capability. To alter the layout, the user can either alter the schematic diagram or re-run the layout programs with different parameters.

The program **PACKAGE** may be executed any time after running **DRAW**. **PACKAGE** helps the user to package his design. It also allows the user to choose the type of package to be used. It checks if all the bonding pads are adequately powered and if the bonding rules are observed. A bonding diagram as shown in Fig. 2.7 is generated if **PACKAGE** successfully packages the design.

2.3.3.5 Design Validation

After the layout of the design has been generated and the timing of the design has been verified by **EXERT**, the design can be validated for fabrication. To complete the design

Fig. 2.7 The Bonding Diagram Generated by PACKAGE

validation phase, the program **TIS** and **SHIPDES** must be executed.

The foundry provides on site functional testing using static test vectors for any validated design. The test signals used by testing equipment to drive the circuit are generated from the simulation output .trc file. To make sure that the tester can test the design successfully, **TIS** performs several checks on the simulation output file. Some of the checks performed by **TIS** are :

- 1) the simulation is performed at a vector time of 1000ns for pure digital circuit, or a vector time of 10000ns if analog circuit is included.
- 2) The output signals are assumed to be stable and are sampled at a time point corresponding to 90 % of vector repetition period, i.e. 900ns for pure digital circuit and 9000ns if analog circuit is used.
- 3) All the input/output signals to the chip have been monitored in the simulation.
- 4) All the digital pads have toggled at least once.

Note that to verify the performance of the design, the simulations must be performed at the operational speed (using the operational vector time). To allow the factory tester to test the design, the final simulation run should use a testing vector time of 1000ns or 10000ns.

The final step in validating the design is to run the **SHIPDES** program. **SHIPDES** ensures all the required design steps have been carried out before the design is shipped to the foundry. After the design has been validated by **SHIPDES**, the design is copied to a tape for shipment. The remaining task of the designer is to fill in the customer sign-off form which will be sent to the foundry together with the tape. The check list used by **SHIPDES** is shown in Fig. 2.8.

2.4 Summary

The semi-custom design method is used in ASIC more often than the full custom design method. Two approaches commonly used in the semi-custom method have been presented in this chapter. The recent trend in ASIC design is a shift towards the integration of the analog design capability into the automated design package. **SOLO 1200** includes analog capability. The design tools and the design data base of **SOLO 1200** have also been reviewed in this chapter.

Design Checks
stages per row < 120 existence of all necessary design files correct order of creation of design files software checks final simulations performed with -load and minimum, maximum delay option. track lengths < 10000 um consistent simulation results testability checks passed fanout checks passed only one pad file has been included simulation conforms to timing rules percentage of nodes toggled check for critical signals check for correct pad gap

Fig. 2.8 The List of Items Checked by SHIPDES

Chapter 3

System Consideration of an Echo Canceller

3.1 Introduction

To evaluate the **SOLO 1200** design package, a subscriber loop line interface **Echo canceller** was chosen to be an evaluation vehicle. The **Echo canceller** is a representative design of a telecommunication application. The design requires the use of a memory module and analog elements such as A/D Converter. It thus represents also an adequate evaluation vehicle for the **SOLO 1200** design package. The architecture and the performance of the **Echo canceller** depend on the design of other functional blocks in the transceiver. Different types of design trade-offs need to be considered, e.g. the simplification of the **Echo canceller** may impose stringent requirements on the timing recovery circuit or the line equalizer. As a result, the design of the **Echo canceller** itself represents a complex task.

In this chapter, the basic principles of echo cancellation and the system level aspects of the design are described. The following conventions are used : all signal names in the design are italicized. The part names or the module names in the design are in bold face with the first letter in upper case. Labels in the schematic diagrams do not follow the same convention, but the difference between the signal name and the part name is self evident in the schematic diagram. For digital signals, the suffix *stb* is used to denote strobe signals. Signal *t0b* is the inverted version of *t0*, i.e. $t0b = (t0)'$. The words 'zero', 'low', 'ground' and 'Vss' are synonyms. Also, the words 'one', 'high', 'power' and 'Vdd' have the same meaning. Whether the words 'zero' and 'one' refer to the logic states or the numeric values will depend on the context.

3.2 ISDN and Echo Cancellation

In a modern society with an intense information flow, an increase in the usage of non-voice services such as facsimile, video, and data communication can be observed. Instead

of using a separate network to provide different types of communication services, the integration of various services into an existing communication network using advanced digital technology would be advantageous. The design of the interface to the network is an important issue for providing the integrated services. The proposed *ISDN* U(user) - interface [3] will provide the interfacing function between the network and the network termination at the customer's premises. A pair of twisted copper wires is used as the transmission medium for the digital subscriber loops. There are two competing methods for providing the two way transmission capability at the U-interface : 1) the method of time compression multiplexing, or the Ping Pong model and 2), the method of echo cancellation. Comparing the two methods, the method of echo cancellation is capable of covering a larger transmission range. In North America, the echo cancellation method is selected as the scheme to be implemented in the *ISDN* U-interface transceiver, Fig. 3.1.

In the echo cancellation method, a hybrid transformer is used to convert the four wire circuit at the transceiver side to the two wire circuit at the network side. The hybrid decouples the transmitted data stream and the received data stream. Because of the mismatch between the impedance of the hybrid and the terminating impedance, an attenuated version of the transmitted data from the local transmitter will leak through the hybrid and couple into the received data stream from the far-end. This unwanted leakage signal is termed as the near end echo.

Fig. 3.1 Full Duplex Transmission Using Echo Cancellation and Hybrid

Typically, the hybrid loss of the near-end echo is around 10 dB, while the line loss of the received far-end data is around 50 dB. This implies that the signal to echo ratio at the local receiver is around -40 dB. Thus, the echo swamps the received data and degrades the performance of the local receiver. The echo can be removed by an Echo canceller. To maintain an adequate performance of the local receiver decision point, a minimum of 20 dB signal to echo ratio is required. As a result, the Echo canceller should be capable of suppressing the echo by at least 60 dB.

The Echo canceller is commonly implemented as self-adapting filter which tries to match the impulse response of the filter with the impulse response of the echo path. If the impulse response of the echo path is known and time invariant, one can use a filter with a fixed impulse response to remove the echo. However, in practice, the line parameters are slowly varying with time. This dictates the use of self-adapting filter to track the varying line conditions. There are in existence several different adaptive algorithms [4,5] which can be used to implement an adaptive filter. They include the following algorithms : Least Mean Square (LMS), Least Square (LS), Recursive Least Square (RLS) and the Kalman Filtering. Each of the algorithms results in a different filter structure.

In the following paragraph, the principle of echo cancellation using the LMS algorithm [4] is described. The LMS algorithm is based on the steepest descent method which tries to minimize the mean squared value of the residual error. The filter structure for the LMS algorithm is shown in Fig. 3.2. In this structure, a delay line is used to store the current and the past transmitted data. The stored data bits are tapped from the delay line. Each tap has a coefficient $C_{i,n}$ associated with it. The set of coefficients $\{C_{i,n}\}$ constitutes an estimate of the echo path impulse response in discrete time. For ease of explanation, the following conventions will be used in the tapped delay line description. The taps of the delay line are numbered from left to right. The leftmost tap (tap 0) corresponds to a zero data delay. The rightmost tap (tap $N-1$) corresponds to a delay of $N-1$, where N is the total number of taps. Tap 0 stores the data transmitted in the current baud period¹, while tap $N-1$ stores the data transmitted at $N-1$ baud periods ago. In every baud period, the stored data in the delay line is shifted one position to the right to make room for the storage of the current transmitted data.

¹ one baud period is equivalent to one symbol period

Fig. 3.2 The Filter Structure For the LMS Algorithm

By performing a discrete time convolution between the data stored in the delay line with the filter coefficients, one can generate an estimate, $\hat{E}_n$, of the echo

$$\hat{E}_n = \sum_{i=0}^{N-1} C_{i,n-1} D_{n-i} , \quad (3.1)$$

where

$\hat{E}_n$	denotes	the echo estimate in baud n ,
N	denotes	the total number of filter taps,
n	denotes	the baud index,
$C_{i,n-1}$	denotes	the i -th filter coefficient at $(n-1)$ -th baud period. $i = 0 \dots N-1$,
D_{n-i}	denotes	the data transmitted at $(n-i)$ -th baud period.

To cancel the echo, the echo estimate is subtracted from the received data sample. The resulting residual error e_n at baud period n is therefore given by

$$e_n = RS_n - \hat{E}_n , \quad (3.2)$$

where

RS_n	denotes	the received data sample at baud n ,
e_n	denotes	the residual error at baud n .

Since the echo estimate is only an approximated quantity, the removal of the echo is not exact. Either too much or too little of the echo is subtracted from the received data sample. A certain amount of the residual error remains in the received data sample after cancelling the echo. The residual error value, which may include the far-end data, is used to update the filter coefficients (modifying the filter impulse response) so that the echo estimate can be calculated more accurately in the next baud period. The updating of the coefficients is where the adaptation comes into play. In the LMS algorithm, the i -th tap coefficient value at baud period n , $C_{i,n}$, is updated according to

$$C_{i,n} = C_{i,n-1} + \gamma e_n D_{n-i}, \quad (3.3)$$

where γ denotes the loop gain.

By modifying eq. 3.3, one can obtain three other variants [6] of the LMS algorithm. They are listed in eq. 3.4 - 3.6.

sign algorithm :

$$C_{i,n} = C_{i,n-1} + \gamma \text{sign}(e_n) D_{n-i} \quad (3.4)$$

sign regressor algorithm :

$$C_{i,n} = C_{i,n-1} + \gamma e_n \text{sign}(D_{n-i}) \quad (3.5)$$

sign sign algorithm :

$$C_{i,n} = C_{i,n-1} + \gamma \text{sign}(e_n) \text{sign}(D_{n-i}) \quad (3.6)$$

where $i = 0 \dots N-1$

Note that in eq. 3.4 - 3.6

$$\text{sign}(x) = \begin{cases} \frac{x}{|x|}, & x \neq 0; \\ +1, & x = 0. \end{cases} \quad (3.7)$$

The differences between the three variants and the LMS algorithm are the clipping of either the magnitude of the transmitted data, the magnitude of the error or the magnitude of both the error and the data in the updating equation. The equations for echo estimation

and echo cancellation are the same for the four algorithms. All the three variants are relatively easier to implement than the LMS algorithm. In implementing the updating equation 3.3, full word length must be used for the transmitted data and the residual error. On the other hand, in the implementation of the updating equations 3.4 and 3.5, the word length of the transmitted data or the residual error is reduced to one bit by clipping their corresponding magnitudes. This results in a saving in the hardware. The saving in the hardware is greatest in the implementation of equation 3.6. By clipping the magnitude of the transmitted data and the magnitude of the residual error at the same time, the updating quantity simplifies to $\pm\gamma$. As a result, the sign-sign algorithm has been chosen because of its simplicity in implementation.

3.3 Summary of the ANSI U-Interface Specifications

Although the performance of the **Echo canceller** is not optimized in this thesis, the design must have a reasonable and practical starting point. The U-interface specification of the American National Standard Institute (ANSI) [7] provides a good starting point. Before the design details are described, this section will summarize relevant part of the ANSI layer 1 specification.

in the current specification, the data is transmitted at a rate of 160 kb/s through the network. The line code being used is the 2B1Q code. 2B1Q stands for 2 binary 1 quaternary. The line code uses one of four analogue levels to represent 2 bits of data. This results in a transmission rate of 80 kbaud/s with a baud period of 12.5 μ s.

The 2B1Q line code is a non-redundant code with the following code table.

bit pair	logical symbol	example voltage levels (volts)
00	-3	-2.5
01	-1	-5/6
11	+1	+5/6
10	+3	+2.5

Table 3.1 The Look Up Table for the 2B1Q Line Code

In the specification, the transmitted data is scrambled by the scramblers which are characterized by the following 23rd order polynomials :

network - network termination scrambling

$$1 \oplus x^{-5} \oplus x^{-23} \quad (3.8)$$

network termination - network scrambling

$$1 \oplus x^{-18} \oplus x^{-23} \quad (3.9)$$

The binary data stream should be recovered in the local receiver by applying the same polynomial to the scrambled data as used in the remote transmitter. The purpose of the scramblers is to randomize the data. The randomization of the data makes the transmitted data stream and the received data stream uncorrelated. The uncorrelation of the two data stream reduces the effect of the received far-end data on the residual error of the **Echo Cancellor**.

3.4 Systems Consideration of the Echo Cancellor

3.4.1 Filter Taps

In the design, the **Echo canceller** is implemented as a digital transversal filter using the sign sign algorithm to update the filter coefficients. Since 2B1Q line code is a non-redundant code and the transmission system eliminates the DC signal component, a long echo tail is expected [8]. To cancel the long echo tail, the **Echo canceller** is required to have a long delay line with more than a hundred taps. However, in [9], a suggestion has been made to precede the **Echo canceller** with a high pass filter which can shorten the echo tail to about 20-40 baud periods. Using this scheme, an **Echo canceller** with 20 40 taps should be adequate.

A RAM module which is supplied by ES2 and will be used to store the filter coefficients has an organization of 64×8 bits. The 64 memory locations are more than enough to implement the **Echo canceller** as a 40 tap filter with 8-bit coefficients. Thus, a decision

has been made to implement the **Echo canceller** as a 32 tap filter. This arrangement makes full use of all the memory locations and allows each coefficient to be 16-bits long. Also, the number of filter taps is within the specified range of 20 – 40 taps.

3.4.2 Sampling Rate

In general, for the purpose of data recovery or echo cancellation, the use of baud period sampling (one sample per baud period) will be adequate. However, for the purpose of timing recovery, $\frac{T}{n}$ sampling with $n \geq 2$ (n samples per baud period) may be required. The echo cancellation scheme in $\frac{T}{n}$ sampling [3] would use about n times the amount of hardware required by baud period sampling. As baud rate sampling is possible, it is used to simplify the hardware in this design.

3.4.3 Modification of the Algorithm for Efficient Implementation

A logical way to implement the sign sign algorithm for echo cancellation is simply to follow the sequence of the operations :

- 1) estimate the echo (eq. 3.1),
- 2) subtract the echo estimate from the line signal to obtain the residual error (eq. 3.2),
- 3) use the residual error to update the filter coefficients (eq. 3.6).

However, the hardware implementation of the execution sequence mentioned above will be very inefficient. It requires the echo canceller to access every memory location three times within one baud period, i.e. one read operation from each memory location for estimating the echo, one read operation from each memory location and one write operation to each memory location for updating the coefficient.

The implementation would be more efficient if one could read the coefficient from the memory, performing the accumulation of the echo estimate and updating the coefficient at the same time. The updated coefficient is then written back to the memory. This results in only one read and one write operation per memory location per baud period. The above idea effectively exchanges the execution sequence of eq. 3.2, 3.6 and computes eq. 3.1, 3.6 concurrently.

The modification of the execution sequence seems possible and should lead to the convergence of the **Echo canceller**. However, one must exercise some caution when exchanging the execution sequence. Notice that the updating action in the modified execution sequence is performed before the residual error is computed. As a result, the coefficients are updated by the *sign* of the residual error which is computed in the previous baud period. The residual error computed in the previous baud period does not take into account the effect of the data transmitted in the current baud period. This may lead to the divergence and overflow of the **Echo canceller**. When coded in PASCAL and simulated, the sign sign algorithm with the modified execution sequence does indeed diverge.

After several trials, a solution was found which could at the same time make the algorithm converge and reverse the execution sequence of eq. 3.2 and 3.6. The solution involves the use of a delay line with 33 taps and 32 coefficients. Delayed data at tap 0 to tap 31 are used to calculate the echo estimate. The delayed data at tap 1 to tap 32 is used to update the coefficients. Since the delayed data at tap 1 to tap 32 in the current baud period is equal to the delayed data at tap 0 to tap 31 in the previous baud period and the sign of the residual error for updating the coefficients in the current baud period is calculated in the previous baud period, the coefficients will be correctly updated. This solution involves the use of one more delay element.

3.4.4 Loop Gain Factor

The loop gain factor γ is one of the factors which determines the mean squared residual error and the filter convergence time. In some designs, this value is made variable. A large loop gain is initially used to achieve fast convergence. After the filter has converged, the loop gain is reduced to a smaller value to improve the mean squared residual error. In this design, the loop gain is a constant value. To simplify the hardware further, the loop gain is moved from eq. 3.6 to eq. 3.1. Moving the loop gain away from eq. 3.6 reduces the updating quantity to ± 1 . The loop gain is also chosen to be $\frac{1}{256}$. The division of the convolution sum by 256 can be implemented easily by discarding the low byte of the convolution sum. The modified sign sign algorithm which is implemented in this design is shown in eq. 3.10 - 3.12. Note that the time index in eq. 3.11 is slightly different from the one in eq. 3.6.

The modification of the execution sequence seems possible and should lead to the convergence of the **Echo canceller**. However, one must exercise some caution when exchanging the execution sequence. Notice that the updating action in the modified execution sequence is performed before the residual error is computed. As a result, the coefficients are updated by the *sign* of the residual error which is computed in the previous baud period. The residual error computed in the previous baud period does not take into account the effect of the data transmitted in the current baud period. This may lead to the divergence and overflow of the **Echo canceller**. After coded in PASCAL and simulated, the sign sign algorithm with the modified execution sequence does indeed diverge.

After several trials, a solution was found which could at the same time make the algorithm converge and reverse the execution sequence of eq. 3.2 and 3.6. The solution involves the use of a delay line with 33 taps and 32 coefficients. Delayed data at tap 0 to tap 31 are used to calculate the echo estimate. The delayed data at tap 1 to tap 32 is used to update the coefficients. Since the delayed data at tap 1 to tap 32 in the current baud period is equal to the delayed data at tap 0 to tap 31 in the previous baud period and the sign of the residual error for updating the coefficients in the current baud period is calculated in the previous baud period, the coefficients will be correctly updated. This solution involves the use of one more delay element.

3.4.4 Loop Gain Factor

The loop gain factor γ is one of the factors which determines the mean squared residual error and the filter convergence time. In some designs, this value is made variable. A large loop gain is initially used to achieve fast convergence. After the filter has converged, the loop gain is reduced to a smaller value to improve the mean squared residual error. In this design, the loop gain is a constant value. To simplify the hardware further, the loop gain is moved from eq. 3.6 to eq. 3.1. Moving the loop gain away from eq. 3.6 reduces the updating quantity to ± 1 . The loop gain is also chosen to be $\frac{1}{256}$. The division of the convolution sum by 256 can be implemented easily by discarding the low byte of the convolution sum. The modified sign sign algorithm which is implemented in this design is shown in eq. 3.10 - 3.12. Note that the time index in eq. 3.11 is slightly different from the one in eq. 3.6.

echo estimation :

$$\hat{E}_n = \gamma \sum_{i=0}^{N-1} C_{i,n-1} D_{n-i} \quad (3.10)$$

where γ equals $\frac{1}{256}$

coefficient updating :

$$C_{i,n} = C_{i,n-1} + \text{sign}(e_{n-1}) \text{sign}(D_{n-i-1}) \quad (3.11)$$

where $i = 0 \dots N-1$

echo cancellation :

$$e_n = RS_n - \hat{E}_n \quad (3.12)$$

3.5 Functional Simulation of the Algorithm

The convergence behavior of the sign algorithm is well studied. The algorithm is known to converge. The conditions for its convergence are derived in [10]. It has been shown in [11] by graphical means that the sign regressor algorithm does not always lead to convergence. The condition for the convergence of the sign regressor and the sign sign algorithm is still an unsolved problem [6]. The analysis of the sign regressor algorithm and the sign sign algorithm is difficult because the magnitude of the data vector components is clipped. In the case of a two tap adaptive filter using 2B1Q code with the code words uniformly distributed, one can show that the sign sign algorithm always leads to filter convergence by the use of the Quiver diagram introduced in [6]. Whether this conclusion can be extended to N tap case is unknown. While the sign algorithm is known to converge, it is interesting to note that the sign sign algorithm is the same as the sign algorithm if a bipolar code is used. Also the 2B1Q code can be considered as two bipolar codes with different amplitudes interleaving with each other.

In this design, not only the sign sign algorithm is used, but the algorithm is also modified. In order to have a high confidence level that the modified algorithm will converge with the use of 2B1Q code, the modified algorithm is simulated at the functional level before the physical design phase has started. Two main PASCAL programs are written to perform the simulations. The first main program is a data generator which generates a

Fig. 3.3 The Structure of the Data Generation Program

sequence of transmit data and the echo. The second program takes the output from the data generator to drive the modified sign sign algorithm and computes the residual echo.

3.5.1 Data Generation program

The structure of the data generator program is shown in Fig. 3.3.

The data required by the simulation includes the transmitted data and the echo. To generate the echo, a hypothetical discrete time impulse response of the echo path (Fig. 3.4) is used. The hypothetical impulse response attenuates the mean power of the transmitted data by 0.099, i.e the echo power is being attenuated by approximately 10 dB. Although the echo path response is a hypothetical one, it resembles a true echo path response in the way that most of the echo power is concentrated in the current baud period.

Fig. 3.4 The Hypothetical Impulse Response of the Echo Path

A random number generator is used to generate a sequence of random numbers which are limited to the range of 0, 1, 2, 3. Next the random numbers are decomposed into data bits. The data bits are then scrambled by a polynomial to further randomize the bit stream. By using the look up table (Table 3.1), the outcoming bit stream from the scrambler is grouped in pairs and converted to 2B1Q logical symbols and line voltages. A discrete time convolution between the line voltages and the hypothetical echo path impulse response is performed to generate the echo. The echo is then quantized and converted to 2's complement code by an x-bit quantizer. Both the 2B1Q logical symbols and the quantized echo are written to a file which will be used by the other simulation programs. A total of 400,000 symbols are generated for each experiment. The data generator can also write the transmitted bit pairs and the quantized echo to another file in the EXERT .drv file format. The data generator program is listed in Appendix B.

3.5.2 Echo Cancellation Program

The structure of the simulation program (Appendix C) is shown in Fig. 3.5.

Fig. 3.5 The Structure of the Echo Cancellation Program

The program takes the logical symbols of the transmitted data and the filter coefficients to compute the echo estimate. The computed residual error and the logical symbols are

used to update the filter coefficients. The residual error is computed by subtracting the echo estimate from the quantized echo. The residual errors and the squared residual errors are written to a file by the program.

3.5.3 Simulation Results

Since the A/D Converter in the **SOLO 1200 ANALOGLIB** has a word length of 8-bits, the simulations are performed under the assumption that an 8-bit quantizer is used. Also, no far-end signal or random noise is included in the simulation. As will be implemented in the design, the loop gain is set to $\frac{1}{256}$ in the simulations. Under the above conditions, several runs of the simulation using different data sequence are performed. The modified algorithm converges in all the simulations. The echo suppression is around 31 dB. The residual error and the squared residual error for one of the simulations are shown in Fig. 3.6 and Fig. 3.7. The simulations also show that the residual error keeps oscillating around zero after the filter has converged. It can never stay at zero steadily. This is because the coefficients are constantly updated by ± 1 in the modified sign sign algorithm. Even if the coefficients align themselves perfectly with the sampled echo path impulse response in the current baud period, they will be knocked out of the perfect alignment in the next baud period due to the updating action. After the filter has converged, the updating action can be regarded as a noise source.

To be complete, simulations which assume the use of a 14-bit quantizer and a 15-bit quantizer have also been performed. For these two quantizers, the following conditions are simulated.

- 1) the line signal symbols consist of near-end echo only.
- 2) the line signal samples consist of both near-end echo and far-end data.
The mean power of the far-end data is being attenuated by 50 dB.
- 3) the line signal samples consist of both near-end echo and far-end data.
The mean power of the far-end data is being attenuated by 40 dB.

The far-end data stream is generated in a similar way as the near-end data. The differences are the use of the network - NT scrambling polynomial and the additional use of two hypothetical impulse responses for the channel. Like the echo path impulse response, the two channel impulse responses also span 32 baud periods. All of the above simulations

Fig. 3.6 The Residue Error after Echo Cancellation - 8b ADC

Fig. 3.7 The Squared Residue Error after Echo Cancellation - 8b ADC

assume noise free condition. The hypothetical echo path and channel impulse response are assumed to be time invariant. The simulated results are tabulated in Table. 3.2.

	$\frac{1}{\gamma}$	echo suppressed (in dB)	convergence time (in number of bauds)
No far end signal	128	68	128K*
	64	65	64K
Far end signal -50 dB	128	57	130K
	64	54	66K
Far end signal -40 dB	128	49	139K
	64	46	70K

Table 3.2a Simulation results of the Modified Sign Sign Algorithm - 14b ADC

	$\frac{1}{\gamma}$	echo suppressed (in dB)	convergence time (in number of bauds)
No far end signal	128	74	259K
	64	71	128K
Far end signal -50 dB	128	60	261K
	64	57	131K
Far end signal -40 dB	128	53	288K
	64	50	135.6K

Table 3.2b Simulation results of the Modified Sign Sign Algorithm - 15b ADC

In Table 3.2, the echo suppression is taken to be the mean power of the residual echo in the last 100K simulated baud period in terms of dB. The convergence time is the baud period at which the envelope of the convergence curve first dropped below the calculated echo suppression. Table 3.2 shows that increasing the word length of the quantizer by one bit reduces the residual error and increases the convergence time. Although the far-end data and the near-end data are uncorrelated by the use of different scrambling polynomials, the presence of the far-end data increases the squared residual error. This occurs when the signal strength of the residual error is reduced to the far-end signal level, whereupon the

* 1K denotes 1000X

coefficients will be updated by the sign of the random far-end data, rather than by the sign of the residual error. As a result, an increase in the squared residual error is expected as the mean power of the far-end data is increased. The presence of the far-end data also increases the convergence time. The effect of the far-end data can be suppressed by the use of the decision directed far-end signal estimation and compensation method [12]. In this method, the amount of the far-end signal in the residual error is estimated and then subtracted from the residual error. Removing the far-end signal from the residual error increases the amount of echo suppression.

3.6 Summary

In this chapter, the LMS algorithm and its variants were presented. The modifications to the sign sign algorithm, which is chosen to be the cancellation algorithm, is described. The design methodology, system timing and the design details of the functional blocks are described in the next chapter.

Chapter 4

Hardware Design of the Echo Canceller

4.1 Introduction and Design Overview

In this chapter, the major design details of the **Echo canceller** will be highlighted. Supplementary notes on the fine details of the design can be found in Appendix A. The next seven sections are devoted to describe the design methodology, system timing and the five functional blocks of the system. Section 4.9 describes how design for testability (DFT) is incorporated into the design. The simulation and the design compilation process are described in the last section.

The **Echo canceller** uses both 8-bit and 16-bit word sizes. Bit numbering of the data registers and the memory locations starts from right to left. Bit 0 refers to the least significant bit while bit-7 or bit-15 refers to the most significant bit. The notation $a(b : c)$ is used to describe the bus signal. a stands for the bus name. b and c are the bit numbering of the bus with $b < c$. The lower bound of b is 0 and the upper bound of c is 15. For a 16-bit bus, the term 'high byte' is an alias to bits (8:15) and the term 'low byte' refers to bits (0:7). The internal logic of the **Echo canceller** uses fixed point 2's complement arithmetic while the conversion code used by the A/D Converter, **ADC**, and the D/A Converter, **DAC** is Offset binary. The conversion from 2's complement to Offset binary or vice versa can be done simply by inverting the most significant bit.

4.2 Hardware Design Methodology

The design method used in this project is based on the top down design methodology suggested in [13]. Fig. 4.1 shows the applied design method.

The required operations to be carried out by the **Echo canceller** are defined by laying out the operation flow diagram based on the functional requirements. A functional block diagram is derived by partitioning the system into different functional units. The input

Fig. 4.1 The Applied Design Method

signals, output signals and the system timing are also defined at the same time. Using step-wise refinement, the details of the functional units, the operation flow diagram and the system timing diagram are defined. The refinement of the functional units results in a detailed schematic diagram of the circuit, while the expansion of the operation flow diagram results in the state flow diagram of the echo canceller's controller.

The operation flow diagram of the **Echo canceller** is shown in Fig. 4.2. It shows the operations to be carried out within one baud period.

The functional block diagram of the **Echo canceller** is shown in Fig. 4.3. It consists of a RAM module to store the tap coefficients. The **Delay line** is used to store the transmitted data. In Fig. 3.2 of Chapter 3, the filter taps are processed in parallel. In this design, the filter taps are processed serially in order to reduce the hardware complexity. To compute the convolution sum, the **Convolver** multiplies the i -th coefficient and the stored data at i -th tap together. The products are accumulated to form the echo estimate. The i -th tap coefficient is updated by the **Updater** concurrently. An 8-bit **A/D converter** is used to sample the line data which in practical situations will include the far-end signal and

Fig. 4.2 The Operation Flow Diagram of the Echo Cancellation

the echo. The computed echo estimate is subtracted from the sampled line signal in the Transfer module. The sign of the residue error is fed back to the **Updater** to update the coefficients. The **D/A converter** reconverts the residue error back to the analog domain so that the **Echo canceller's** behavior can be observed during testing.

4.3 System Timing

The **Echo canceller** system has ten external input output signals. They include : the system clock of the **Echo canceller** *csysclk* and its inverted version *csysclkb*, the

Fig. 4.3 The Functional Block Diagram of the Echo Canceller

transmitter clock *etxclk*, the reset input *erset*, the activation control input *eact*, A/D converter input signal *lin(0 : 7)*, D/A converter output signal *out(0 : 7)*, and the three input signals to the Delay line, *etx0*, *etx1*, *etx2*. Fig. 4.4 shows the timing relationship of the input signals.

The **Echo canceller** is designed to run at a system clock period of 83.33 ns (or a clock frequency of 12.0 MHz). The clock frequency is 150 times the transmitter clock frequency. Among these 150 clock cycles, 18 are idle cycles. The idle cycles occur at the end of baud period. In these cycles, the **Echo canceller** does not perform any operation and just waits for a new baud period to come. The inclusion of the idle cycles is to provide room for changes when design modification is required during the design process. The extra cycles may also be used for the purpose of coordinating the actions between the functional blocks of the transceiver. The idle cycles can always be removed by running the system at a lower clock frequency. The functionality of the system is verified by simulating the system at a system clock period of 80 ns (12.5 MHz) and a transmitter clock period of 12 μ s. A new baud period begins with the rising edge of *etxclk* and ends with the next rising edge of *etxclk*. The rising and falling edges of *etxclk* always occur at the rising edge of *esysclk*.

To simplify the system timing requirement, it is required in the design that the changes of all the digital input signals are synchronized with the *esysclk*. Because the system will

Fig. 4.4a The System Timing Diagram

Fig. 4.4b The Timing Relationship between *etx2* and *eact*

not respond to the defined changes after one or two clock cycles, there should be enough time for the signals to settle down. This removes the requirements on the set up time and the hold time of the external signals.

Signal *erset* is used to initialize the **Echo canceller** system. This signal is asserted (active) high. Both the assertion and the disassertion of *erset* are required to occur at the falling edge of *esysclk*. The duration of the reset pulse should last for at least one system clock cycle. The assertion of *erset* causes the registers to be cleared and forces the controller to enter the reset state. The initialization of the coefficient RAM starts

after *erset* is disasserted. The initialization process requires a total of 127 clock cycles. To complete the initialization within one baud period, the assertion of *erset* should occur within 22 system clock cycles after the rise of *etxclk*. The latest time at which *erset* should be disasserted is 22.5 system clock cycles after the rise of *etxclk*.

The delay line has three inputs. The 2B1Q bit pair applies to inputs *etx0*, *etx1*, while the transmission status of the near-end transmitter applies to *etx2*. The changes of these three inputs are synchronized with the rising edge of *etxclk*. The current and delayed versions of the signal *etx2* are used to control the computation of the echo estimate.

When the near-end transmitter is not transmitting, there will be no near-end signal leak through the hybrid transformer. After the echo tail dies away, the echo will eventually disappear. Under this situation, the **Echo canceller** should be deactivated. The disassertion of the activation signal *eact* inhibits the **Echo canceller** from computing the modified sign sign algorithm. The assertion and the disassertion of the activation signal *eact* must occur at the rising edge of *etxclk*. There is a timing relationship between *eact* and *etx2*. The *eact* signal should be disasserted only after *etx2* is held at high level continuously for 32 baud periods. The assertion of *eact* should occur at the same time as the high to low transition of *etx2*. The above timing relationships will be explained in section 4.5

Figure 4.5a-b shows the top level schematic diagram layout using **DRAFT**. It maps directly to the functional block diagram in Fig. 4.3. If not specified, signals which are described in this chapter are asserted (active) high by default. In the naming convention, signal *esigid* is the same as *sigid*. Finally, to avoid confusion it must be stressed that the meaning of *sign(x)* mentioned in Chapter 3 refers to the normalization of *x* by $|x|$. When *sign* is mentioned in this chapter, it has a different context. It refers to the sign bit of a number in 2's complement representation.

4.4 Design of the Controller

The **Echo canceller** is implemented as a state machine with a synchronous **Controller**. The **Controller** is the main core of the **Echo canceller**. Its design thus requires careful consideration. The design must be raceless and hazard free. The number of **Controller** states directly affects the clocking speed of the design. For a fixed length FIR filter, the more states it takes to run through one iteration of the computations, the higher the

Fig. 4.5a The Top Level Schematic Diagram

am of the Echo Canceller

Fig. 4.5b The Top Level Schematic Diagram

Diagram of the Echo Canceller

clocking speed that is required. On the other hand, the number of **Controller** states depends on the system architecture, and is inversely proportional to the amount of parallelism or pipelining embedded in the system.

4.4.1 State Flow Diagram of the Controller

The state flow diagram of the **Controller** is shown in Fig. 4.6. The state assignments are also shown in the diagram. The legend used in Fig. 4.6 is shown in Fig. 4.7. In Fig. 4.6, signal *rset*, *trclk*, *act*, *cv* and *tr* are used to control the state machine. The output signals of the **Controller** generated at each state are associated with the operations which are carried out in that state. The output signals of the **Controller** (Fig. 4.5b) can be divided into five groups :

- 1) signals to control the **Rainmod** module *rdwrb*, *ramclr*, *ceinc*, *up*.
- 2) signals to control the **Convolver and Updater** – *clrest*, *selb*, *shiestb*, *scla*, *hstb*, *lstb*, *trcostb*.
- 3) signals to control the **Delay line** – *smode*, *shiestb*.
- 4) signals to control the **Transfer** – *erstb*.
- 5) signals to control the **ADC** – *on*, *adclk*, *soco*.

The **Echo canceller** is initialized in state 1 and state 2. Iterations through these two states clear the memory. The registers are also cleared during the initialization phase. After the initialization, the controller enters state 3 to wait for a new baud period.

A low to high transition on *trclk* initiates a new baud period. The **Controller** enters state 4 in the next clock cycle after detecting the transition. In state 4, the **Controller** turns on the A/D Converter, **ADC**, to sample the line signal and clears the accumulator which stores the echo estimate. It also examines the activation control signal *act*. The signal is disasserted when the near end transmitter has ceased to transmit data for 32 consecutive baud periods. If it is asserted, states 7 to 12 will be stepped through to execute the modified sign sign algorithm. Otherwise, the **Echo canceller** is deactivated. The computation of the echo estimate and the updating actions are not required. The **Echo canceller** is only required to transfer the sampled line signal directly from the A/D Converter data register to the D/A Converter data register.

Fig. 4.6 The State Flow Diagram of the Echo Canceller

- = signal asserted high, asserted throughout the cycle
 □ = signal asserted low, asserted throughout the cycle
 □ = signal asserted high, asserted only in the first half of the cycle

Fig. 4.7 Legend of the State Flow Diagram

Fig. 4.8 The Structure of the Fhflipb

In the design, operations occur at both the falling edge and the rising edge of *sysclk*. State change occurs at the rising clock edge, while the data is strobed at the falling clock edge. Between the two clock edges, the state variables and the data are allowed to propagate and settle. As the **Controller** is designed in the form of a Moore finite state machine, there are no edge transitions on the **Controller** output signals at the falling edge of *sysclk*. Since all the storage registers used in the **Echo canceller** are falling edge triggered, a circuit to produce a falling edge on the strobe signals at the falling edge of *sysclk* is required. One method is to gate the **Controller** output signal with *sysclk*. However, this method is totally delay dependent, asynchronous and is likely to cause glitches on the strobe signals. Another method which is used in the design involves the use of a one shot device. The one shot device, **Fhflop** is shown in Fig 4.8. In this method, the **Controller** output signal applies to the input, *ctl*, of the one shot device. A rising edge on *ctl* triggers **Fhflop** and causes the output signal *fh* to go high. When *sysclk* goes low, it resets **Fhflop**. This produces a falling edge on signal *fh*.

Fig. 4.9 The Simulation Results of Fhflop

Owing to the delay from the rising edge of the *sysclk* to the assertion of *ctl*, the pulse width of *fh* can be greatly reduced. In the worst case, the pulse width can be reduced from

the nominal value of 40 ns to 28 ns. To allow a bigger margin for logic propagation, and also to meet the memory timing specification, the pulse width of fh must be lengthened. The pulse width is extended by pushing the falling edge of fh further to the right. By adjusting the amount of delay in the clock path, one can tune the one shot pulse to the required width. The use of the delay element in the clock path makes the method delay dependent. However, the method will not cause glitches on the strobe signal. Fig. 4.9 shows the simulation result of the **Fhflip**.

4.5 Design of the Delay Line

Shift registers are used to implement the **Delay line** which stores the current and past transmitted data. The **Delay line** consists of 3 shift registers (Fig. 4.10). Shift registers **Sra** and **Srb** consist of 33 storage elements each. They are used to store the 2B1Q bit pairs. **Sra** stores bit-0 while **Srb** stores bit-1 (or the sign bit). **Src** is used to store signal $tx2$. This shift register has 32 storage elements. **Src** operates in the same way as the other two shift registers. The reason for introducing $tx2$ is because 2B1Q code is a non-redundant code. All the possible code words are completely used for symbol representation. To indicate the non-transmitting state of the near end transmitter, a third bit must be introduced. Signal $tx2$ provides this extra information and stores in the shift register **Src**. D-type Flip Flops, **Bdffct**, with an asynchronous reset input are used to implement the shift registers.

All 3 shift registers shift the data from left to right. The furthest left Flip Flops of the three shift registers store the current transmitted data, while the furthest right Flip Flops of the three shift registers store the data transmitted 31 (**Src**) or 32 (**Sra**, **Srb**) baud periods ago. The furthest left Flip Flops corresponds to tap 0, while the furthest right Flip Flops correspond to tap 31 or tap 32.

The following discussion is based on **Sra** and **Srb**, but it applies to **Src** as well. Because the filter taps are processed in sequence, it is not necessary to bring out all the stored data from the **Delay line** at the same time. Since the stored data at tap 0 to tap 31 is used to compute the convolution sum, and stored data at tap 1 to tap 32 is used to update the coefficients, only the output of Flip Flop 31 and 32 is required to be brought out of the **Delay line** module. The output of Flip Flop 31 is used to control the **Convolver** and the output of Flip Flop 32 is used to control the **Updater**. Within each baud period, data

stored in each Flip Flop will be shifted one by one to Flip Flop 31 and 32 for processing. This results in a processing sequence which starts from the previous symbols back to the current symbol.

Two shift modes are required: the right shift mode and the rotate mode. A simple right shift is used to shift the current transmitted data into the **Delay line** early in the baud period. The shift mode changes to the rotate mode immediately after the current transmitted data has been shifted into the **Delay line**. In the rotate mode, the output of Flip Flop 31 is fed back to the input of Flip Flop 0. After 32 rotations, the data stored in Flip Flops 0-31 will return to the original status before the rotations were started. The original data in Flip Flop 32 will be lost. This does not create any problems, since the current content of this Flip Flop will not be used in any further computations.

The signal $tx2$ is an on-off signal which indicates the transmitting status of the near end transmitter. When it is disasserted, the near end transmitter is transmitting data to the far end in the current baud period. On the other hand, if it is asserted, the near end transmitter is not transmitting in the current baud period. **Src** plays its role during the period in which the near end transmitter changes from a transmitting to non-transmitting state. When the near end transmitter stops transmitting, the **Echo canceller** can be turned off. In the presence of the long echo tail, an immediate turning off of the **Echo canceller** will lead to a sudden increase of the residue echo and will corrupt the received data. To prevent this from occurring, the cancellation operation should not be stopped until the echo tail dies away. If the cancellation of the echo is to be continued after the near end transmitter has ceased transmission, product terms $C_{i,n}D_{n-i}$, corresponding to the non-transmitting baud periods, should be masked out from the convolution sum because these terms have no contribution to the echo. If these terms are not masked out, the convolution sum will not be closed to the true echo content presented in the signal samples; and this results in a large residue error. The masking action can be performed by bringing the output of **Src**'s furthest right Flip Flop to the reset input of the data register which holds the product term. A logic one at Flip Flop 31 of **Src** will cause the product term to clear. Based on the above discussion, the timing scenario presented in section 4.3 has been developed. Whenever the near end transmitter ceases to transmit, signal $tx2$ is set to logic one. The act signal will remain asserted for an additional 32 baud periods. After the echo tail dies out, the **Echo canceller** is deactivated by disasserting the signal act . It is assumed that the echo tail

does not last longer than 32 baud periods.

The 3 shift registers have a total of 98 D-type Flip Flops. These 98 Flip Flops present a large loading to the strobe signal and the reset signal. Several buffers for each signal are used to distribute the loading.

4.6 Design of the Convolver and the Updater

The schematic diagram of the **Convolver and Updater** module is shown in Fig. 4.11. The **Convolver** includes **Mux16a,b**, **Reg16a,b,c**, **Add2** and **Compa**. The **Updater** includes **Add1** and **Mux8**. Other random logic in the figure is used to generate signals for controlling the **Convolver** and the **Updater**. The two modules work in parallel.

4.6.1 Design of the Convolver

The **Convolver** computes the discrete time convolution of the filter impulse response and the transmitted data stream. According to equation 3.2, the computation involves two operations : 1) compute the product term $C_{i,n}D_{n-i}$ and 2) accumulate the product terms.

4.6.1.1 Computation of the Product Term

Although the coefficients and the transmitted data are both the multiplicands of the product term, the term ‘multiplicand’ whenever mentioned in this section, refers to the transmitted data. The operations required to implement the multiplication depend on the value of the multiplicand. Since the multiplicand is either ± 1 or ± 3 , the multiplication operation involves only a maximum of one 2’s complement operation, one left shift and one addition operation. The design of the **Convolver** is optimized to the above computation.

Multiplying the coefficient by -3 requires both a left shift and complement operations. One may wonder whether the left shift should follow the 2’s complement operation or vice versa, i.e. which order will minimize the hardware. The following analysis reveals that it is more efficient to do the complement operation first. Let the coefficient be C and recall that the 2’s complement of a number is equal to the 1’s complement of the number plus one. Suppose the left shift precedes the complement operation, thus

Fig. 4.11 The Schematic Diagram

ram of the Convolver and Updater

$$\begin{aligned}
& -3 \times C \\
& = -2 \times C + -1 \times C \\
& = -1 \times S(C) + -1 \times C \\
& = F2(S(C)) + F2(C) \\
& = F1(S(C)) + F1(C) + 2
\end{aligned} \tag{4.1}$$

where S denotes single left shift operation
 $F2$ denotes 2's complement operation
 $F1$ denotes 1's complement operation

If the complement operation precedes left shift the equation becomes :

$$\begin{aligned}
& -3 \times C \\
& = 3 \times (F1(C) + 1) \\
& = 3 \times F1(C) + 3 \\
& = S(F1(C)) + F1(C) + 3
\end{aligned} \tag{4.2}$$

Although eq. 4.2 requires the addition of an extra 3, it uses only one complementer instead of two complementers as in eq. 4.1. Equation 4.2 is implemented in the design. Following the above discussion, the operations required to carry out the multiplication are summarized in Table 4.1.

multiplicand	operations on coefficients
+3	left shift and add, strobe
+1	strobe
-1	1's complement, add 1, strobe
-3	1's complement, left shift and add, add 3, strobe

Table 4.1 The Breakdown of the Multiplication Operations into Shift, Add and Complement.

4.6.1.2 Accumulation of the Convolution Sum

After the product term is computed, it is added to the old partial convolution sum in the accumulator **Reg16b** to form the new partial convolution sum. After the convolution

sum has been fully accumulated, it is subtracted from the received data sample. Although the convolution sum is 16-bit wide, only the high byte of the sum, $est(8 : 15)$, is used in the echo subtraction (the received data sample only has 8-bit resolution). To carry out the subtraction of the echo in the **Transfer** module, $est(8 : 15)$ is brought out of the **Convolver and Updater** module.

4.6.2 The Updater

The coefficients are updated by the quantity ± 1 . The updating quantities are determined by the logic equation :

$$tm1b \oplus ersign \quad (4.3)$$

where $tm1b$ = sign bit of the transmit data from Flip Flop 32 of **Srb**.
 $ersign$ = sign bit of the error

The Exclusive-or operation, $\oplus$, multiplies the two sign bits together and generates the sign bit of the data-error product. The updating quantity is then derived from the sign bit of the data-error product.

The two adders in the **Convolver and Updater** module are 16-bit adders. The worst case propagation delay of an adder occurs when the two input of the adder are -1 and 0 respectively, with the carry input to the LSB being equal to 1 . If a ripple adder is used, the required propagation delay is estimated to be about 70 ns (assuming a 2.5 unit load at each gate output). Since the adder must be settled down within 83 ns, the 13 ns margin may not be enough if delays on the rest of the data path are considered. A decision is thus made to use a carry look ahead adder (CLA). Similar analysis shows that the worst case delay of the CLA is about 40 ns while the simulation indicates that the worst case delay of the CLA is about 35 ns.

4.7 Transfer Module

The **Transfer** module is shown in Fig. 4.12. In this module, the data register **ADCreg** holds the received data sampled by the A/D Converter. The strobe signal to the **ADCreg**, $adstb$, is a delayed version of the end of conversion signal generated by the A/D converter. When the **Echo canceller** is in active state, the main function of this module is to compute

Fig. 4.12 The Scheme

eq. 3.11. In this case, the multiplexer **Mux8** will select the residue error from the adder **Add3** and present it to the D/A data register **DACreg** which holds the digital data to be converted back to the analog signal by the D/A Converter. Signal *erstb*, which is pulsed in state 12, is the strobe signal to **DACreg**. The sign bit of the **DACreg** is brought out of the **Transfer** module to the **Convolver and Updater** module for the purpose of updating the coefficients. The sign bit is also inverted to change the 2's complement code to the Offset binary code which is used by the D/A Converter. If the **Echo canceller** is deactivated, **Mux8** will transfer the received data sample from the **ADCreg** directly to the **DACreg**.

In eq. 3.10, the convolution sum is multiplied by a loop gain of $\frac{1}{256}$ to generate the echo estimate. This results in a fractional number. One can use either truncation or rounding to convert the fraction back to an integer. Truncation is implemented in the **Transfer** module.

For a positive 16-bit 2's complement number, the operations of dropping the low byte and right shift of the high byte 8 bits are the same as dividing the number by 256 and truncating the fractional part. However, if the 2's complement number is a negative number, the same operation will not give the correct truncated result. This is because the 2's complement number system is asymmetrical about zero. To implement the truncation correctly, an additional 1 must be added to the truncated number. The efficient implementation of the truncation with compensation for off by one error, together with the subtraction operation, is analyzed in Appendix A.3. Two test cases of the implementation are shown in Table 4.2. Again the numbers are in base 10.

RS_n	$\hat{E}_{16,n}$	$\hat{E}_{8,n}$	ϵ_n
0	+1023	3	-3
0	-1023	-4	+3

Table 4.2 Two test cases of the **Transfer** module

4.8 Design of the Rammod Module

The memory organization of the 16-bit coefficients in a 8 bit/byte static RAM is shown

Fig. 4.13 The Memory Mapping of the Coefficient RAM

in Fig. 4.13.

The module **Rammod**, where the coefficient RAM resides, is shown in Fig. 4.14. The control signals to the memory include chip enable *ceinc*, read write control *rdwr tb*, and the 6-bit address bus *ad(0 : 5)*. Memory access must be qualified with a high on *ceinc*. A low on *rdwr tb* initiates write operation, while a high on *rdwr tb* initiates a read operation. The memory address is supplied by a 6-bit updown counter **Ramctr**. The **Ramctr** is clocked by *ceinc*. Depending on the logic state of the control signal *up*, the memory counter increments or decrements at the falling edge of *ceinc*. The **Ramctr** is initialized to *3F_H* during system reset. In each baud period, the **Ramctr** decrements from *3F_H* toward the low end of the address and wraps around to *3F_H* in state 10 of the last iteration.

The memory uses two data buses. The read bus *b8(0 : 7)* is tri-statable and carries data read from the memory. The write bus *din(0 : 7)* carries the data to be written to the memory. **Mux8s** selects either the test patterns or the updated coefficients (on bus *b9(0 : 7)*) as the data to the memory. The test patterns are generated in **Mux8s** in response to the control signals. **Mux8o** brings several signals out of the chip for observation.

4.9 Design For Testability

4.9.1 Introduction

There are two testing strategies which are commonly used in testing : 1) the functional

Fig. 4.14 The Schematic Diagram

Diagram of the Rammod Module

testing strategy. In this strategy, the chips are being tested for their functionality under the operating environment. 2) the strategy of direct monitoring/control of internal nodes. This strategy is embedded with the idea of design for testability. Design for testability (DFT) [14] is an important consideration that an ASIC designer must keep in mind throughout the design process. A malfunctioning chip which does not incorporate testability provisions may not provide any useful information for the designer to carry out the re-design work. However, it must also be stressed that trying to test a circuit completely is not at all practical, since the vast amount of the circuit nodes to be tested makes such a test extremely expensive and infeasible.

The difficulty in LSI or VLSI circuit testing lies in the fact that there is only a limited access to the internal nodes of the integrated circuit. On-chip test devices must be included to facilitate access to the critical internal nodes. The inclusion of a testing facility generally serves two purposes : 1) to be able to observe the internal node of the circuit. 2) to be able to control the circuit by setting the logic value of the nodes. The access of the internal nodes from the outside world is either through the functional pins or the unused pins of the chip. Means must be provided to switch the test pins and the test devices from either the normal operation mode or the idle mode to the testing mode. The difficulty in testing increases with the scale of integration. This is due to the reduction of the number of I/O pin to the number of on-chip devices ratio. Another difficulty towards testing is the generation of test vectors which are to be applied by the test bed to the chip under test. The number of required test vectors depends on the number of on chip devices. In general, for a few thousand gate designs, it is not uncommon to have more than ten thousand test vectors. The generation of so many test vectors is a time consuming task. Not only does the number of test vectors increase in relation to the number of on chip devices, but also the amount of simulator or tester usage time increases exponentially. The total testing time is proportional to N^x [15], where N is the number of on chip devices and $x > 2$. To reduce the testing time, the method of divide and conquer is commonly used. In this method, the circuit is divided into several partitions with each partition being tested individually. If the whole chip is divided into M partitions, the testing time for each partition is proportional to $(N/M)^x$ and the testing time of the whole chip is proportional to $M(N/M)^x$. Usually, the partitioning of the circuit for testing will come naturally and will follow the functional partitioning of the circuit.

In general, the testing scheme used will affect the design details of the circuit. As a result, DFT must be considered right from the beginning of the design. Otherwise, the circuit will either have poor testability or the circuit will have to be redesigned to accommodate the desired testing scheme. The incorporation of an extra on chip testing facility will generally increase the area of the design and slow down the circuit. This presents a trade off between the degree of testability, performance and efficiency of the design.

4.9.2 Scan Test Configuration

Due to the feedback nature of the sequential logic, it is more difficult to test a sequential circuit than a combinational circuit. The test vectors for sequential logic are also more difficult to generate. To test a sequential circuit, the scan method is introduced [16], [17], [18]. The idea is to link all the sequential circuit elements together to form a long shift register which is known as the scan path. The scan path spans the whole circuit and partitions the whole circuit into different sections. By setting the circuit into the scan mode, test data can be shifted (or scanned) into the shift register. After the test data has been shifted to the designated partition, the circuit is changed back to the normal operating mode. The circuit is then clocked for one or more periods in normal operating mode. Afterwards, the circuit is changed back to the scan mode and the resulting data is shifted out. The behavior of the circuit partition under examination can be deduced from the scanned out data.

The formation of the scan path essentially breaks the feedback loop existing in the sequential circuits. This allows the sequential circuit to be tested like a combinational circuit. Compared with signature analysis[14], a scan test allows an additional degree of freedom in testing. Since one can generate subsequent scan test patterns according to the behavior of the circuit partitions in the previous test, one can zoom in to obtain the fine details of the circuit behavior. In signature analysis, many errors can hash into the same signature. The resolution is limited. As a result, signature analysis is commonly used in production testing while a scan test is commonly used in prototyping. The scan test does not impose too much overhead on the I/O requirement because of its serial nature. In general, to scan a design four additional test pins are required: one for mode control, one for scan input, one for scan output and one for scan clock (if a separate clock is needed).

The scan test is relatively easy to incorporate into the design. The main disadvantage of the scan test is its time consuming nature. For each scan test, one scan in and one scan out operation is required.

In the **BASELIB** of **SOLO 1200**, a D-type Scan Flip Flop is available. The Scan Flip Flop can be preset but cannot be cleared. In order to build a scannable data register which can be cleared, slight modifications are required. The modified Scan Flip Flop is shown in Fig. 4.15. It has been used as a building block for all the data registers in this design. As it can be noticed from Fig. 4.15, that the Scan Flip Flop has only a single clock input which serves as the strobe input of the data register. Since the strobe signal is only pulsed occasionally, the scan action would be too slow if it were to be used as the scan clock. It also complicates the design of the **Controller**. In order to speed up the scan operation, the system clock *sysclk* and the inverted system clock *sysclkb* are used as the scan clock.

Fig. 4.15 The Modified Scan Flip Flop.

Since both the data registers and the state variable Flip Flops are incorporated in the same scan path and the data registers are negative edge triggered and the state variable Flip

Flops are positive edge triggered, the state variable Flip Flops are scanned by *sysclk* and the data registers are scanned by *sysclkb*. While the scanning of the state variables does not involve clock multiplexing, *sysclkb* and the various strobe signals to the data registers will be multiplexed in this design. In general, clocking signals should not be multiplexed. By imposing a simple timing sequence, the changeover from the normal operating mode to the scan mode can be carried out safely. The proposed sequence is shown in Fig. 4.16

clock edges at which the first scan action takes place

Fig. 4.16a The Sequence in Changing from the Normal Mode to the Scan Mode. (note $a = b = \text{one half cycle.}$)

clock edges at which the last scan action takes place

Fig. 4.16b The Sequence in Changing from the Scan Mode to the Normal Mode. (note $a = b = \text{one half cycle.}$)

As shown on Fig. 4.16, it is required that both the *sysclk* and the *sysclkb* can be stopped independently. Also, because *sysclkb* is used only as scan clock, it must be turned off during the normal operating mode to reduce the switching related power dissipation. As

two states. The only difference with memory read test is signal *rdwr_{tb}* which must be held high by additional hardware. The testing of the memory is controlled by two external control signals *c1* and *c2* (Fig. 4.5a). The code table for the memory test modes is shown in Table 4.3.

<i>c1</i>	<i>c2</i>	test
0	0	normal operation
0	1	write test pattern 01010101
1	1	write test pattern 10101010
1	0	read back

• Table 4.3 The Code Table for the Memory Test Modes and the Mode Control Signals *c1* and *c2*

The **Rammod** module shown in Fig. 4.14 includes the memory test devices. The test devices include **Mux8o**, **Mux8s** and the six logic gates. The six logic gates are used to generate signal *rdwr_{tb}* and signal *ramtst*. Signals *mode* and *c2* in Fig. 4.17 are the same as the test control signals *c1* and *c2* respectively in the top level schematic diagram (Fig. 4.5a).

The read/write control signal *rdwr_b* from the controller is always set to low in state 2 during normal operating mode. In order to use the state for memory read test, *rdwr_b* is gated with *c1* and *c2* to generate signal *rdwr_{tb}* which serves as the read/write control signal to the memory. Signals *c1* and *c2* will hold *rdwr_{tb}* at logic high level in state 2 during the read test.

Mux8s is a special two-to-one 8-bit multiplexer which selects either the coefficient (on bus *b9(0 : 7)*) or the two test patterns. The selected data will be put onto the memory write bus *din(0 : 9)*. The selection is decided by the logic value of *ramtst*. In the normal operating mode, bus *b9(0 : 7)* is selected by disasserting *ramtst*. In the memory write test mode, the checker board patterns are selected by asserting *ramtst*. The cell structure (**Mcells**) of **Mux8s** is shown in Fig. 4.17. The two test patterns are generated directly from signal *mode*. In Fig. 4.17, the inverting and non-inverting versions of signal *mode* applies to the input *b* of the eight **Mcells** in alternation. When *mode* is set to one, the test pattern 10101010 will be applied to input *b* of **Mux8s**. Otherwise, the pattern 01010101

is applied. In **Mcells**, each bit of $b9(0 : 7)$ is masked by the inverted version of $ramclr$ (Fig. 4.17), so that an all zero bit pattern can appear on the memory write bus $din(0 : 7)$ to clear the memory during system initialization phase.

4.9.4 Observer Ports

The 8-bit multiplexer **Mux8o** in the **Rammod** module is used to multiplex several internal signals out of the chip for external verification. The internal structure of the **Mux8o** and the **Muxcello** is shown in Fig. 4.18. During the write cycles, the 6-bit memory address bus $ad(0 : 5)$ from the **Ramctr**, the converter clock $adclk$ and the controller control signal trf are selected by **Mux8o**. Both trf and $adclk$ are generated by the **Controller**.

The memory read bus $b8(0 : 7)$ is selected by **Mux8o** when the memory is not being written. This configuration allows the test patterns stored in the memory to be examined during the memory read test, and the filter coefficients to be observed on the fly during the normal read cycles.

Notice in Fig. 4.18 that each bit of the multiplexer output has a D-type Flip Flop. It is necessary to locate the Flip Flop here since, when the memory is not being written, the memory read bus is always selected. If the memory is not being accessed, the memory read bus is either floating or at high impedance. If there is no Flip Flop or latch to hold the pad driver input at a definite level, this may cause a short from Vdd to Ground in the pad drivers of the observer port OP0-OP7 (Fig. 4.5a). Since the memory chip enable signal $ceinc$ is pulsed whenever the memory is accessed, the Flip Flops are clocked by $ceinc$.

The test procedure of the memory can be summarized as follows :

- 1) set $c1, c2$ to values 0, and 1 respectively (write test pattern 01010101).
- 2) reset the **Controller** and force it into state 1.
- 3) clock the system as usual to write the test patterns to the memory.
- 4) set $c1, c2$ to values 1, and 0 respectively (read back).
- 5) reset the **Controller** and force it into state 1.
- 6) clock the system to read the test patterns back from the memory.
- 7) set $c1, c2$ to values 1, and 1 respectively (write test pattern 10101010).
- 8) repeat procedure (2) to (5).

Fig. 4.17 The Cell Structure of the Mux8s.

Fig. 4.18 The Cell Structure of the Mux8o.

4.10 Simulation and Silicon Compilation

The design of the **Echo canceller** has been fully simulated with the functionality and the timing verified. The physical layout of the design in CIF code was generated using the auto-place and route program of **SOLO 1200**. Several iterations through the simulation and the silicon compilation process were required. All the design work was done on a SUN 3/160 work-station with the CPU running at 16 MHz

4.10.1 Simulation for Design Correctness

The functionality of the circuit components has been verified by simulating each component individually. To verify the functional and timing correctness of the overall circuit, the data generated by the data generator used in the behavioral simulation has been used (Appendix C). A total of 40 baud periods has been (or an equivalent of 6,000 simulation vector) simulated. It took about 25 CPU hours (or 59 machine hours) to complete the simulation on a SUN 3/160 work station.

4.10.2 Simulations for Testing

In these simulations, the overall circuit was simulated as if it was a black box, with the test patterns and the test results scanned in and scanned out of the circuit. The simulation vectors are designed to verify the functionality of the circuit. No attempt was made to locate the stuck-at faults. Also the fault coverage of the simulation vector set is not estimated. The simulation results were observed through the scan out pins and the 8 observer ports. The data input files used in these simulations could be converted to a test vector set to drive the tester. The testing results logged by the tester could be compared with the .trc file produced by the simulator to verify whether the fabricated circuit behaved as expected.

Two data input files are used. Input file one (Appendix E) relates to the **Convolver and Updater**, the **Transfer** and the **Rammod** modules. In this simulation, after the system has been reset, the test pattern 1100 is scanned into the 3 scan paths simultaneously and repetitively to verify the shifting capability of the scan paths. The simulation of the four memory tests using the procedures mentioned in section 4.6.4 is carried out after the shift test.

After the four memory tests, test patterns are scanned in to verify the functionality of the **Echo canceller's** computation units. The order of scanning in the test patterns is the same as the order specified below. The specific test patterns are shown in Appendix 1.

- 1) *eight scan* in patterns to verify the 16-bit adder, **Add2**, of the **Convolver**.
- 2) *sixteen scan* in patterns to verify the **Convolver's** multiplication properties.
- 3) *eleven scan* in patterns to verify the 16-bit adder, **Add3**, of the **Transfer** module.
- 4) *thirty – seven scan* in patterns to verify the 16-bit adder, **Add1**, of the **Updater**.

Note that in the testing of **Add1**, the test results for the scan in patterns are written to the memory first. The memory read test is then used to read back the test results.

Input file two (Appendix F) tries to verify that the **Controller** can exercise all possible state transitions. Some of the state transitions are verified during the simulation using input file one. The remaining state transitions are verified in the input file two.

4.10.3 Silicon Compilation

The design uses a total of 20,810 transistors, or an equivalent of 4,162 2-input NAND gates. The placement of the design has been done with some minor manual intervention. By modifying the .pos file, the position of several gates are swapped to reduce the net loading. The routing of the design is not intervened at all. In the component placement, the *vary* option has been used due to the presence of the memory module. In most designs, the column width is seldom larger than 120 stages (owing to the loading consideration). The memory module has a width of 146 stages. The use of the *vary* option allows the user to specify the column width of the column which contains the memory module to be 146 stages wide. The column width of other columns are much less than 120 stages. The array dimensions are specified with care so that the wasting of area is not excessive. In the final version of the layout, the core array has the following dimensions :

Number of columns	= 4
Number of rows in column 1	= 38
Number of stages in column 1	= 88
Number of rows in column 2	= 28
Number of stages in column 2	= 88
Number of rows in column 3	= 19

Number of stages in column 3	= 146
Number of rows in column 4	= 34
Number of stages in column 4	= 92

The final layout of the core array has an area of 23.48mm^2 . Including the pad area, the total chip area is about 38.97mm^2 . The layout of the design is shown in Fig. 4.19 (Only the second layer metal is shown). The added area for testability is estimated to be 4.41mm^2 which is approximately 11.3 % of the total chip area.

Fig. 4.19 The Layout of the Echo Canceller System (only the second layer metal are shown)

4.11 Summary

In this chapter, the implementation of the various functional blocks in the **Echo canceller** is described. The design incorporates testability features. The test scheme described in section 4.9 enhances the controllability and the observability of the design. The experience in using the **SOLO 1200** design package will be summarized in Chapter 5.

Chapter 5

SOLO 1200 Design Features

5.1 Introduction

In the industry, a common method to bench-mark a new automatic placement and routing package is by allowing the package to work on a design with a few thousand gates. Another previously evaluated package can also be used to work on the same design. The performance indicators are the area of the compiled chip and the time required to carry out the compilation. The CAD design features among packages can also be compared. Because a comparable design package from other foundries was not available, performance and feature comparison have not been carried out in this project. Also in the evaluation, the source code of **SOLO 1200** is not available; the evaluation can only be based on the user's point of view, i.e. evaluate the software from the external. The above two considerations limit the scope of this evaluation.

Sections 5.2 - 5.4 discuss the features of the design entry system, design verification system and the physical design system respectively. Section 5.5 compares **SOLO 1200** cell libraries with a similar standard cell library from Texas Instruments. Notes on the **SOLO** design cells are presented in section 5.6. Section 5.7 comments on the optimization control of the package. The execution speed of the various design programs is tabulated in section 5.8. Supplementary notes on the package can be found in Appendix B.

5.2 Schematic Entry and HDL

Comparing the two methods of design description which are available in **SOLO 1200** reveals that the schematic editor **DRAFT** has the advantage of allowing the designer to view the circuit components and the interconnections concurrently. Reading the **MODEL** code is a sequential action, and does not provide the designer with a broader view of the

design. In addition, placing a part symbol from the cell library on the design sheet is considered to be an easier task than coding and typing. Automatically generating the signal names is also a very convenient feature of **DRAFT**. However, coding the hardware description using **MODEL** allows the designer to incorporate more features into the design. The **MODEL** code generated by **DRAFT** does not contain all the available control statements which can affect the compilation process.

MODEL is a simple hardware description language. The language syntax does not conform to the industrial standard of VHDL and is used mainly for structural description. Except for the *constraint* statement, the language does not include any features for functional or behavioral modeling. Since the language construct is very much like a high level programming language, **MODEL** is not very difficult to learn. The concept of using a high level language construct for hardware description can also be found in [19], [20]. The important language constructs include the loop statement (*For*), the decision branch statement (*If*), the constraint statement (*constraint*), the placement and routing control keywords.

The lack of behavioral modeling capability in **MODEL** is a drawback of the language. Even though the language allows the user to employ either the top-down, middle-out or the bottom-up approaches to describe the design, without the behavioral modeling capability the user is forced to use the bottom up approach to simulate his circuit. The use of high level language for behavioral modeling is possible, but not as convenient.

Aside from the fact that **DRAFT** does not have a very good user interface, it is an easy tool to use. The bugs in the program are mainly due to its interactive nature. To prevent the unwanted loss of data in any single editing session, the user is advised to save his design file frequently. To make the best use of the available resources, one can use **DRAFT** to generate the net-list description in **MODEL** code. A text editor can subsequently be used to include the desirable language constructs.

5.2.1 Design Hierarchy

No matter what design approach the user chooses to describe the hardware, the design in its final form will have a tree-like structure. Figure 5.1 shows the design tree of the **Echo canceller** with **Cancelsys** as the root node. Both the schematic diagram and the

2

OF/DE

2

MICRO

Fig. 5.1 Design Tree of the Echo Canceller (First and Second Levels)

MODEL code are organized in this structure. In the interactive mode, the simulator also presents the net-list in this structure. Each *part* is represented by a node in the design tree. Any tree node is a child of its parent node and has one or more child nodes. Each node is associated with the following information : 1) the input/output signals of the node and 2) the interconnection pattern of the child nodes. The hierarchy of the tree follows the design partitioning devised by the designer in the schematic capture phase.

5.3 Design Verification

In general, to verify the correctness of a design before fabrication, three types of design verification are required. They are : 1) the verification of logic correctness, 2) the verification of timing correctness, and 3) the verification of physical design correctness. In ASIC design, the physical layout is generated by the design tools directly from the net-list description. Assuming that the software which generates the layout is error free, then the correctness of the physical design is guaranteed. The task of physical design verification is thus removed from the designer.

To verify the logic and timing correctness of a design, a simulator is often used. Although a simulator can be used to verify the timing of a design, it is considered to be slow, incomplete and test case dependent. This leads to the development of the timing verifier [21, 22] which can be used to uncover the timing problem of a design in a much more expedient manner. In **SOLO 1200**, a timing verifier is not included. The simulator **EXERT** is used both for the purposes of logic verification and timing verification. To assess the amount of faults which can be covered by the simulation vectors, some designers may wish to perform fault simulation. Unfortunately, **EXERT** does not have a fault simulation capability.

EXERT uses the event driven simulation with selective traces approach. Conceptually, **EXERT** can be considered to be a mixed mode simulator [23, 24, 25]. The *analog pads* and the memory modules are simulated at the behavioral level. The digital cells are simulated at the switch level. The simulation levels cannot be changed dynamically, i.e. the digital circuits are always simulated at the switch level while the analog circuits and the memories are always simulated at the behavioral level.

A switch level simulator [26, 27, 28] stands somewhere in between the timing simulator and the gate level simulator in the simulator hierarchy. Similar to the gate level simulator,

it ignores the low level electrical aspects of the transistors. Unlike the gate level simulator, it attempts to capture the logic behavior of the MOS transistor instead of capturing the logic behavior of a logic gate.

5.3.1 Logic Simulation

EXERT uses 3 logic levels. They are {1, 0, X} with X = unknown. The high impedance state is also represented by the X value. The number of drive strengths associated with each logic level is not specified by the foundry. **EXERT** has a *charge decay* modeling capability to facilitate the logic simulation. When the decay model is turned off, the net which is not being driven will hold the logic level indefinitely. When the decay model is turned on, the net which is not being driven will go to unknown state after a certain time delay. For a transmission gate, the duration is measured to be 0.93 ns after the transmission gate is turned off. This duration does not seem to increase with the capacitive loading on the signal. It can be concluded that **SOLO 1200** uses a worst case approach to model the *charge decay* process.

5.3.2 Timing Verification

To qualify the simulator as a timing verifier, **EXERT** is capable of checking whether the set up time, the hold time and the minimum pulse width of certain signals are satisfied during the simulation. It will monitor the memory timing if any memory element is included. The checking is performed in parallel with the simulation. Any timing constraint violation will cause **EXERT** to generate an appropriate error message immediately. Most of the timing violations can be detected by **EXERT**. However, it is possible that timing errors are still embedded in the design while **EXERT** does not report any violation of the timing constraint, i.e. the minimum pulse width, the set-up time and the hold time of the signals are all satisfied. Most of the embedded timing error shows up as logic error. This situation usually occurs when the path delay is excessively large. Such a situation has been encountered during the development of the **Convolver and Updater** module. The accumulator of the module strobed in the data before the accumulator input data lines settled. Timing error mentioned above usually can be detected by a timing analysis program.

5.3.3 Pre-layout Simulation

The simulations which are performed before the physical layout is generated take into account the loading effect of the fanout gate, but ignore the parasitic loading of the metal one and metal two traces. However, to be more efficient, and to reduce the number of iterations through the design steps, one can estimate the wire loadings and use them in the simulation. It can be done by manually creating an *.lda* file and using the *load* option when invoking the simulator. The *.lda* file created by DRAW is divided into two portions. The first portion reports the net number and the corresponding parasitic loading on the net in pF. The second portion reports the length of metal one, metal two and poly wire used by each net. It seems that EXERT uses only the first portion of the information during simulation. As a result, one can create an incomplete *.lda* file without specifying the length of the wires used by each net. This incomplete form of the *.lda* file is not rejected by EXERT.

5.3.4 Simulation of the Analog Pads

Although SOLO 1200 incorporates an analog cell library, the software is not structured to provide full blown analog design capability. The ANALOGLIB provides a set of well-characterized *analog pads*; the designer will pick the analog device from the library and enter it into his design. This approach is similar to what one used to do in board level design using off-the-shelf components. Since the user may not design his own analog devices, the simulator EXERT does not provide circuit or timing simulation capability.

In EXERT, an analog signal is represented by an 8-bit quantity or has a total of 256 discrete analog levels. The designer is required either to use 'hand-quantization' or write a program to quantize the analog input signal before one can proceed to simulation. The analog Vdd should always map to level 255 and the analog Vss should always map to level 0.

There are two disadvantages¹ associated with this scheme of analog signal representation. 1) the two reference voltages to the A/D Converter are quantized, this increases the

¹ the disadvantages occur when the reference voltages of the data converter are not the same as the power supply rails (Appendix B.3.3)

quantization error at the A/D converter output and 2) the 'hand-quantized' analog signal is further quantized by the A/D Converter which further increases the quantization error.

Simulator EXERT does not include Resistor (R) and Capacitor (C) as the simulation models. This implies that one can instantiate several uncommitted Op. Amps. which can be configured into the desired filter structure after the chip is fabricated, but the designer cannot use EXERT to model the behavior of the filter. In the ANALOGLIB, three Op. Amps. are internally configured as Voltage Followers. They are modeled by EXERT as Voltage Followers, other Op. Amps. are modeled as ideal Op. Amps..

In a mixed mode simulator, one main concern is how to interface the analog and digital portions of the design in an efficient manner. In EXERT, since the *analog pads* are simulated at the functional level, the analog and the digital circuit can be interfaced easily. The outputs of the RAM module, Comparator and Oscillator are digital in nature; these devices do not have interfacing problems. The Voltage References output usually applies only to the input of the Comparators, Op. Amps. or data converters. They also do not have interfacing problems. Output signals from other analog devices will require the use of data converters before entering or leaving the digital core array.

5.4 The Physical Layout System

The physical layout system of SOLO 1200 generates the design layout from the compiled net-list in a step-wise manner. Except in the case of component placement, the layout generation process can be considered to be completely automatic. For the component placement, the user is required to guide the placement program by specifying the array dimensions.

The layout of the core array can be divided into the device area and the routing area. The size of the device area relies both on the logic implementation developed during the schematic entry phase and on the placement scheme. Once the details of the schematic diagram are defined and the design is being placed, the area occupied by the devices is fixed. The area occupied by the routing channels depends not only on the logic implementation and the placement scheme, but also on the routing scheme. Figure 5.2a-b depicts how component placement can affect the routing area.

Fig. 5.2a The Layout of the Register Bit Cell (No Intervention)

Fig. 5.2b The Layout of the Register Bit Cell (With Intervention)

The two layouts represent the same register bit cell. The layout in Fig. 5.2a is generated without manual intervention, while the layout in Fig. 5.2b is generated by modifying the gate placement specified in the .pos file (swapping the positions of two inverters). Comparing the two layouts, Fig. 5.2a uses one more horizontal channel, two more via contacts and an extra metal two over-pass.

5.4.1 Exploration of the Placement Algorithm

The component placement is performed by the **PLACE** program. As mentioned before, **PLACE** has two placement algorithms, the *1-D* and the *2-D* algorithms [29]. The *1-D* algorithm considers the logic gates of a design as initially lined up in a single row. The line has an orientation which starts from left to right. The algorithm then swaps the logic gates in the row and tries to minimize the total net length of the design. After swapping, the algorithm then folds the row around the array (Fig. 5.3). The *2-D* algorithm considers the logic gates of a design as initially placed on a plane with an array dimensions as specified by the user. Again the gates on the array plane are swapped to minimize the total net length of the design.

Fig. 5.3 The Folding of the Design Row Around The Core Array

Although the placement schemes generated by the two algorithms are not identical, they are similar in the sense that if two logic gates are placed close together using one algorithm, they would not be placed far apart by the other algorithm. The basic criterion for the two logic gates to be placed close to each other is that interconnection exists between them. Table 5.1 compares the performance of the two algorithms. There is no significant difference between the two algorithms in terms of area efficiency. In terms of execution time, the *2-D* algorithm always takes much more time than the *1-D* algorithm. Also, the *2-D* algorithm usually requires the user to specify larger array dimensions, which results in

CIRCUIT	AREA (mm ²)		CPU (second)	
	1-D	2-D	1-D	2-D
Convolver	10.44	10.11	107.3	749.0
Delayline	3.50	3.54	13.0	152.8
Transfer	1.66	1.60	15.8	82.7
Controller	2.46	2.54	22.6	67.8

Table 5.1 Comparison of the 1-D and 2-D Algorithms

the presence of more blank or unused stages in the final layout.

5.4.2 Placement Control

Various placement control options are provided to the designer for controlling the placement scheme. They are described in Appendix B.4.2.

Since the placement algorithm uses a particular approach to optimize the placement of the whole design, it may not be effective in intervening in the placement scheme to reduce the total area of the design. The placement control can be used for the purpose of loading redistribution. All the placement control options mentioned above allow the designer to place certain parts in close proximity to reduce the loading. The use of the .pos file for load redistribution is particularly simple and useful. Figure 5.5 illustrates an example in which the loading of a net is reduced drastically simply by flipping the row.

5.4.3 Exploration of the Routing Software

Conceptually, circuit routing can be divided into the global routing phase and the highway routing phase. In global routing phase, the router assigns the signal nets to the routing highway. In highway routing phase, the signal nets in each routing highway are assigned to one or more channels (detail channel assignment). In practice, the router may not perform the routing in two separate and distinct phases. In terms of area efficiency, the highway router is one of the prime factors which determines the compactness of the design. Common routing algorithms such as the Dog-leg algorithm[30], Greedy algorithm[31] and Yoshimura's efficient routing algorithm [32] use different approaches to minimize the channel width, the wire length or the occurrence of via contacts.

Fig. 5.4 The Wasting of Area Caused by Unequal Column Height

Fig 5.5 The Reduction of Loading by Row Flipping

In a fixed gate array, because of the limited highway width and the presence of a constraint loop, the highway routing may not achieve 100 % completion. The problem is usually solved by re-placing and re-routing the design. SOLO 1200 uses cell based design style, there is no limitation on the width of the routing highways. The router, **ROUTE**, uses the second layer metal as “dog-legs” to break the constraint loop. As a result, the router can complete the routing most of the time. For all the designs and trial circuits used in this project, **ROUTE** can always achieve 100 % completion.

The routing performance of **ROUTE** is evaluated using Deutsch’s “difficult example”[30]. The “difficult example” is a circuit which consists of 301 terminals, and 72 nets. The optimal highway density of this circuit is known to have a lower bound of 19 channels. The routing pattern of the “difficult example” produced by **ROUTE** is shown in Fig. 5.6. **ROUTE** uses 16.6 sec to complete the routing. A total of 23 channels is used. Compared with other routers, the performance of **ROUTE** is adequate. The performance of other routers on the “difficult example” [30, 31, 32] is summarized as follows:

Routers	Channels
Dog-leg	21
Efficient	20
Greedy	20

Table 5.2 The Performance of Various Routers in Routing the ‘Difficult Example’

5.4.4 Routing Control

As discussed in Chapter 2, **ROUTE** has a *manual* option which allows the user to affect the global routing and reduce the end to end delay of some signals. The user can use the *manual* option to force a signal to route through the vertical routing highway instead of routing through the transistor area.

As an example, **SOLO 1200** allows the user to specify whether the polysilicon cross-under or the second layer metal cross-over is to be used in the signal routing. If polysilicon cross-under is used, the end to end signal delay can be reduced by forcing a signal to route through the metal wire in the vertical highway instead of the polysilicon feedthrough in the

Fig. 5.6 The Routing of the "Difficult Example" By the SOLO Router

transistor area. The use of this control option is usually not required if the second layer metal cross-over is used.

Owing to the fact that **ROUTE** does not provide any option for the designer to affect the assignment of channels, the designer has very limited control over the router. The user can neither inhibit a signal nor force it to route through a particular channel or a particular minor highway. This implies that the designer can not manually compact the width of the routing highways or reduce the channel length of a signal. Notice that the *.rte* file generated by the router is a text file. This file contains the routing information. However, the data structure used in this file is not supplied by the foundry.

The *critical* option is another routing control option available from **ROUTE**. The option directs the router to give routing priority to the signals which are specified as *critical* in the **MODEL** program or in the *.crt* file. This option does not always work in favor of the user. In some cases, the loading on a net is reduced after the net has been declared *critical*, but the reduction is very small. In some cases, the loading is in fact increased after the net has been declared *critical*. Observed from the layout, a *critical* net is always routed in the same channel of the corresponding highway, while a non-*critical* signal may be split and routed in different channels. As a result, a *critical* net does not use as many via contacts and metal wires as the non-*critical* net. In principle, the usage of this option should reduce the net loading in most cases. Since a *critical* net has to route in the same channel, if a *critical* net collides with another *critical* net which has already occupied a portion of the channel, the router would fail without any remedy. This is the reason why the router can only handle two to three critical signals. For more than three *critical* signals, the probability of collision is very high.

5.5 Comparison of Library

In this section, the **SOLO 1200** cell libraries [2] will be compared with a 2 μm standard cell library from Texas Instruments (TI) [33]. Table 5.3 shows the comparison between the two libraries.

5.5.1 Library Cells

The TI library provides far more digital standard cells and cell types than the **SOLO**

	SOLO 1200	TI
1) Combinational Logic	✓	✓
2) I/O Pads	✓	✓
3) High Level Macro	✓	✓
4) Sequential Elements	✓	✓
5) 74 Family	✓	✓
6) RAM	✓	✓
7) ROM	X	✓
8) PLA	X	✓
9) Analog Cells	✓	✓
10) Bit Slice Processors	X	✓

Table 5.3 Comparison of the Standard Cell Libraries

1200 libraries. For each of the basic combinational logic cell such as the AND, OR gate, the **TI** library has at least two different performance levels for the user to choose from, while the equivalent cells in the **BASELIB** of **SOLO 1200** has only one performance level. The transmission gate is included in both libraries as a standard cell. Both libraries have an adequate supply of I/O buffers. All I/O buffers from the two libraries have ESD and latch-up protection. The buffers from **TI** can by-pass and dissipate up to a 4 kv voltage spike, and the immunity against latch-up current is 400 mA. The maximum voltage that the **SOLO 1200** I/O buffers can withstand is not specified, while the immunity against latch-up is specified to be about 100 mA. The 74 family is not fully supported by both foundries. Each foundry selects a proper subset of the family for implementation. The **ANALOGLIB** of **SOLO 1200** has a total of 26 analog cells. It supports far more analog functions than the **TI** library. The most prominent example will be the availability of the data converters in the **ANALOGLIB**. A total of 9 Op. Amps. are included in the **ANALOGLIB**. The 9 Op. Amps. are divided into 3 performance series: high speed series (OP1X); high drive series (OP2X) and the low power series (OP3X). Op. Amps. in the same series have the same specifications. The **TI** library has a total of 6 analog cells. Only one Op. Amp. is available⁵.

⁵ **TI**'s analog library is still under development; more analog cells are expected by 1990.

5.5.2 Performance

The performance of the digital standard cells in SOLO 1200 libraries is comparable to the digital standard cells in the TI library. Table 5.4 compares the gate delay of two inverters from the two libraries. Notice that the TI inverter being compared has the best performance level within the industrial specification (-40 to +85 degree Celsius, 4.5 - 5.5 volts).

	SOLO			TI		
	MIN	TYP	MAX	MIN	TYP	MAX
t_{PLH}	0.4ns	0.5ns	0.8ns	0.3ns	0.4ns	0.7ns
t_{PHL}	0.3ns	0.4ns	0.5ns	0.1ns	0.3ns	0.7ns

Table 5.4 Comparison of the Gate Delay of Two Inverters

Based on the above discussion, one can conclude that the SOLO 1200 design libraries are functionally complete in the sense that they supply most of the design cells offered by other foundries and a designer can quite comfortably realize a design using the cells provided in the library. In general, the cell libraries of SOLO 1200 are not as sophisticated as the TI cell library. The availability of the analog cells enhances the design capability of SOLO 1200.

5.6 Notes on the SOLO Cells

This section summarizes the author's experience in using some of the standard cells in the SOLO 1200 libraries.

- 1) To begin with, most of the digital cells in BASELIB have a drive strength of one. However, some of the cells have a very weak drive. Examples would include the 3 input NAND gate and the D-type Scan Flip Flop. Both cells have a drive strength of 0.33. Applying three unit loads to the cells will result in a relative fanout of 9. These cells can thus be easily overloaded. It remains an open question why the foundry does not increase the drive strength of these cells, particularly the D-type Scan Flip Flop which will be used frequently. When a weak drive cell is to be used, the user is advised to

- build a user defined macro which contains the weak drive cells and the buffer(s). The buffer(s) are used to boost up the drive strength of the cell. The encapsulation of the cell and the buffer(s) ensures that they will not be placed far apart by **PLACE** (see Appendix B.4.1). The placement control keywords can be used in the macro if necessary.
- 2) The design of the clocked JK Flip Flops (Fig 5.7) in the **LOGICLIB** cell library has the notorious one's and zero's catching problem (spurious pulse(s) with a duration longer than 4 ns on the J or K input, which occurs when the clock is inactive, will cause the Flip Flop output switching to the corresponding state at the next active clock edge). The user should use them with care. In particular, they should not be used with an asynchronous input.

Fig. 5.7 The 1's and 0's Catching of the JK Flip Flip

- 3) Finally, the functional behavior of the A/D Converter under simulation does not correspond exactly to the specification in the library manual. The manual states that when the **on** signal is disasserted (standby mode), all the digital output of the A/D Converter will be pulled up high. However, in the simulation (Fig. 5.8) the logic state of the signal **on** does not affect the digital outputs of the Converter, while the assertion of **soc** clears the digital outputs. Also, the end of conversion signal **eof** is asserted at

about the same time as the digital output becomes valid. As a result, if *eoc* is used as the strobe signal directly, one could strobe into the A/D data register an invalid data when the delay on the data path is large enough (the invalid data will be approximately $\pm 1\text{LSB}$ away from the valid data).

Fig. 5.8 Effect of the 'On' Signal on the A/D Converter Output

5.7 Optimization Control

The design package is claimed to be capable of optimizing the design and removing the redundant logic. This section discusses the optimization strategy used by SOLO 1200.

By experimenting with the software, it has been found that the optimization is carried out by the MODEL compiler. Also the software package can only optimize a certain class of redundant logic. The package does not optimize any redundant logic which is related to the internal and external signals, i.e. the package will not perform logic reduction for the designer. However, it will remove any redundant logic which is related to power or ground. Figures 5.9a-d are used to clarify the above point. This optimization scheme is a reasonable one, since the scheme will not remove any redundant logic cells which are introduced deliberately to add extra delay to the circuit or to remove logic hazards.

In some applications, the library macros or user defined cells may have more output ports than is required. The logic cells related to the unused ports are thus redundant. To verify whether the software can remove the redundant logic associated with the unused port, two user defined parts are used. One contains dual two input NAND gates and the other one contains dual JK Flip Flops. Only one NAND gate and one JK Flip Flop from each part is used. The remaining NAND gate and the JK Flip Flop represent the redundant logic. The **MODEL** compiler removes the unused NAND gate, but retains the unused JK Flip Flop. Some other trials reveal that the **MODEL** compiler removes the redundant signal paths related to the unused output port only if the signal paths contain pure combinational logic. If any sequential element or transmission gate is in the signal path, the compiler will only remove the combinational logic gates between the unused port and the first sequential element or the first transmission gate in the signal path.

Fig. 5.9a-b The class of circuit that will not be optimized.

Fig. 5.9c The class of circuit that can be optimized.

Fig. 5.9d The class of circuit that can be optimized.

5.8 Tabulation of Parameters

Various performance parameters of SOLO 1200 are measured. The measurements are listed in this section. All of them are related to the execution speed. To obtain an average value of these parameters, two other designs are captured, simulated and compiled. The first one is a digital phase lock loop (PLL) with an equivalent gate count of approximately 570. The other one is an address generator (ADGEN) of a waveform recorder system. The equivalent gate count of ADGEN is approximately 680. The two systems are designed by former Master's students of the university and are fully described in [35], [36]. Since the execution speed of each program depends on the specific option which is being used, the parameters are measured under the same option. These measurements are only valid for the SUN 3/160 workstation running at 16 MHz. The measurements are tabulated in Table 5.5 (Note : EC = Echo canceller, PLL = Phase lock loop, ADGEN = Address generator).

SoftWare	EC	PLL	ADGEN	AVG.
MODEL (statements/sec)	13.9	27.2	22.1	21.1
PLACE (stages/sec)	66.6	86.7	91.2	81.5
GATE (stages/sec)	116.9	118	124.3	119.7
ROUTE (channels/sec)	3.8	7.2	9.2	6.7
DRAW (CIF statements/sec)	291.4	297.8	296.2	295.1
EXERT (changes/sec)	142.3	179.0	128.7	150

Table 5.5 Various Parameters of SOLO 1200

5.9 Summary

The main objective of the thesis is to evaluate the ASIC design package **SOLO 1200**. The evaluation method used is to perform a complete design using the design package. The capability of the software is explored through the use of the software. In this chapter, the properties of the main design programs in **SOLO 1200** are explored. Fine details are described in Appendix B. The methods to estimate the area occupancy and power consumption of a **SOLO 1200** design are also derived in Appendix B. Interested readers may refer to it.

The design package basically eliminates the need for an expert VLSI designer. A good logic designer with basic knowledge in VLSI can easily compile a design into layout information. Most of the design phases such as schematic capture, simulation, placement and routing in **SOLO 1200** are also needed in a full custom design. While using **SOLO 1200**, the physical design phase is no longer a bottle-neck in the entire design process. Instead, system level design and simulations consume most of the design time. The compilation of the design into layout information is just a matter of 10 to 15 minutes in each iteration, with a circuit complexity (about 4,000 gates) like the design used in the thesis. The time spent in the physical design phase is reduced from several man months in a full custom design down to a couple of minutes by using **SOLO 1200**.

With an integrated data base and tools, many features can be added on and automated. The designer is not required to prepare the source deck for the simulator. The net-list is generated automatically. The designer no longer needs to code the characteristic of each logic cell he uses. The simulator will go directly to the data base for the required cell characteristics. Depending on the chosen fabrication process and the simulation condition, the simulator will choose only the relevant set of parameters from the data base.

After one has become familiar with the design tools and the design concepts in ASIC, one should be able to implement a design in a relatively short time span. The time allocation, which is based on the experience obtained in this project, is estimated to be : about 2 - 4 months in defining the system architecture, timing requirements, on chip testing scheme and the logic implementations. Another 1/2 - 1 month is allocated to set up the test cases, code the input waveform and verify the logic and timing correctness of the design. The

turn-around time of the chip is about 1 – 2 months. As a result, the duration of the design from the starting time to the chip return time is approximately four to seven months.

SOLO 1200 is a down to earth rather than a "Novel" design package. It provides a basic set of ASIC design tools with a structured design framework. The set of libraries is complete enough for the user to carry out any random logic design. The availability of memory modules and analog cells are further advantages of the package. Advanced features such as the timing verifier, ATPG, are not supported. The circuit check and the constraint check capability of **SOLO 1200** offers great help to the designer. DFT can be incorporated into the design using the circuit elements provided in the **BASELIB**. The package can enhance its support for DFT in the future by including the LSSD and pad scan devices as standard cells. The performance of the physical design system can be rated as about average, which is quantified by the routing performance of the highway router. There are still some bugs existing in the package, bugs which are related to the user interface.

Although claimed to be a mixed analog-digital design package, the tools in **SOLO 1200** mostly support digital logic design. The support for analog design is limited. Uncommitted Op. Amps., which are the basic building blocks of the continuous time or the sampled analog filters, can be instanced on chip. However, the filters must be designed as if they are discrete components. Facilities to model the behavior of the filters are not supported.

In certain speech processing and telecommunication applications, high signal to quantization noise ratio is required. In Chapter 3, it has been shown that a 14b – 15b quantizer is required in order to have more than 60 dB echo cancellation if the modified sign sign algorithm is used. With only an 8 bit resolution, the A/D Converter in **ANALOGLIB** cannot be used in applications which require high resolution. This limits the applicability of **SOLO 1200** in high performance mixed analog-digital design. Despite the above fact, the **ANALOGLIB** does allow the integration of simple analog front-end into the design.

Chapter 6

Conclusion

6.1 Summary

In this thesis, a user oriented evaluation of a computer aided design package for VLSI - SOLO 1200 has been performed. A complete Echo canceller with a simplified structure is used as an evaluation vehicle. Through the implementation of the Echo canceller, the design package is shown to have several advantages. To begin with, the package has mixed digital and analog capability; and it surpasses many "digital only" design tools. In considering the five basic evaluation criteria set out in Chapter 1, the design package is easy to learn. A new user under the guidance of an experienced user should be able to capture a simple design (e.g. 3 to 8 decoder), simulate the design and compile the layout within 1 or 2 days. If the user learns on his own by referring to the reference manual, he can familiarize himself with the tools in 2 or 3 weeks. However, to understand the software package and use the tools efficiently and fluently, it requires at least 1 month.

The software package is self-contained and easy to use. The "ease of use" feature manifests itself in several places. As an example, the package normalizes the loading in terms of unit load. It thus presents a higher level abstraction of a low level electrical property; the capacitance. The low level concept of current drive capability is also transformed into the concept of unit drive strength. The advantage of higher level abstraction comes from the integration of the data-base and the design tools. The low level details are hidden inside the data-base while the design tools access those low level details automatically.

The full custom approach is the traditional methodology used by the university. Using the full custom approach to design the Echo canceller requires at least 3 - 5 months to do the physical design manually. The exact time depends on how much effort the designer put into compacting the area, increasing the speed and reducing the power consumption of the design. In view of this, the use of SOLO 1200 indeed shortened the physical design time by many orders of magnitude. A complete design from system definition, pre-layout

simulation, physical design to post-layout simulation requires a time span of only 4 - 7 months. The designer is allowed to spend more time in system verification and is able to try out alternative system architectures.

The software itself was found to be not absolutely bug free; however, it does not seem to have major flaws. In most cases, the schematic editor **DRAFT** can generate the netlist correctly. In some cases, the net-list generated by **DRAFT** indicates that the input to a logic gate is floating, while the schematic diagram indicates that the input is connected to a signal net. This happens when the logic gate has been moving around in the schematic diagram. The layout for some simple sub-blocks of the **Echo canceller** design have been examined, the interconnection between the logic gates and the internal connections of the logic gates seems to be generated correctly. Note that the examination is not exhaustive.

Considering the criterion of area efficiency of the compiled layout, the area efficiency is mostly dependent on routing efficiency. As discussed in Chapter 5, there are other channel routers which can out-perform the **SOLO 1200** router in routing the "difficult example". However, it is worthwhile to mention that the routing performance of the router also depends on the topology of the circuit which is being routed.

As mentioned earlier, an **Echo canceller** has been used as an evaluation vehicle. The design has a complexity of about 4,000 gates. Because of the stability consideration, the **Echo canceller** was implemented using the digital finite impulse response filter structure (FIR) with variable coefficients which are updated by the sign-sign algorithm. The sign-sign algorithm is shown to be stable in simulations under ideal situations. The line code used is chosen to be the 2B1Q line code. The use of the sign-sign algorithm together with the 2B1Q line code allows the **Echo canceller** to be designed with minimal hardware and achieves an optimal architecture. By clipping the magnitude of the data and error product in the sign-sign algorithm, the updating quantity is reduced to ± 1 . A single adder can thus be used to implement the coefficient updating action. The use of the 2B1Q line code reduces the multiplication operation required by the convolving equation to, at most, one left shift operation, one addition and one 2's complement operation. Note that if the LMS algorithm is implemented, a true multiplier is required by both the updating and convolving equation. A parallel multiplier would require a very large silicon area.

In the design, the computation is sped up by the parallel execution of the updating and

convolving equations. This is made possible by the addition of an extra delay element to the delay line. The whole system is controlled by a finite state machine which is designed to be hazard free. The hardware is further reduced by compressing the number of machine states, sharing of the control signals and simplifying the structure of the data path. The details of the hardware optimization are discussed in Appendix A. In the current implementation, the delay line of the Echo canceller is implemented using static D-type Flip Flop. A further reduction of hardware is possible by implementing the delay line in the form of a dynamic shift register. This structure is not implemented in the design because of the following reasons : 1) the dynamic shift register is more or less like an asynchronous circuit and in general ASIC foundries are reluctant to implement an asynchronous circuit, 2) the storage capability of the dynamic shift register depends on the charge retention capability of the delay elements, 3) the structure requires the use of two phase clocks. 4) the charge retention properties and the use of two phase clocks make the structure difficult to test.

Being an important concept in ASIC, the issue of design for testability has been carefully considered and incorporated into the Echo canceller design. A test sequence has also been developed for the design. They are covered in Chapter 4. A fairly testable design was built using only the available resources provided in the package. The controllability of the design has been enhanced by using the scan test methodology, and the observability of the design has been increased by adding observer ports to the design. A simple on chip checker board pattern generation circuit was used to generate the testing patterns of the memory. Although the issue of fault coverage has not been addressed in the design, the set of simulation or test vectors has been designed with care. The set of test vectors can achieve a 99 % toggling not including X state, or 100 % toggling including X state. The power consumption of the Echo canceller has been estimated (Appendix B) by using a SOLO 1200 feature of reporting number of toggling of the signal net by the EXERT simulator. The "report on node toggling" is thus an important feature of the design package.

The Echo canceller design has been fully simulated and shown to be functional. Every necessary step in the design flow has been followed through. The tools and the structured design methodology, together with the undertaken design precautions, maintain a high confidence level of the design correctness.

6.2 Future Work

Aside from characterizing the design package, the thesis also looks into the implementation of a signal processing algorithm (the sign-sign algorithm) using the hard-wired digital technique. The use of the sign-sign algorithm in *ISDN U-interface Echo canceller* seems possible. Table 3.2b indicates a possible implementation involving the use of a 15-bit quantizer together with a loop gain of $\frac{1}{24}$. Under this scheme, an echo suppression of 57 dB is achievable. By suppressing the effect of the far end signal [11] and including the effect of noise and cross talk on the signal, an echo suppression of 60 dB probably still can be maintained. However, in Table 3.2b imperfections such as converter non-linearity and the effect of timing jitter on the algorithm have not been looked at. Also, Table 3.2b is derived under a hypothetical situation. Studying the algorithm under a practical situation and taking into account of the above mentioned imperfections would be a good future topic.

In the design, great effort has been expended to reduce the amount of hardware. Reduction of hardware not only decreases the amount of device area, but also improves the operating speed of the system. In a semi-custom design, the intrinsic gate delay and the signal delay due to wire loading are the speed limiting factor of a design. For a digital system whose operations involve only simple data transfer, the two factors in general do not impose restrictions on the design. However, for a high performance system like the *Echo canceller*, which carries out intense computation in a predefined timing window, the two factors become important design consideration. Although extending the design using a slightly modified architecture to $\frac{T}{2}$ sampling can still be done in $12.5 \mu s$, to increase the word length and the number of taps more parallel or pipelined structure must be added on. Thus, it is worthwhile to consider implementing the *Echo canceller* in the form of a systolic array. A systolic array implementation greatly relaxes the speed constraint imposed on the design. However, the amount of hardware involved in a systolic architecture will greatly increase the amount of required hardware. Investigation into the feasibility of implementing the *Echo canceller* using systolic architecture would be another good topic for the future.

Appendix A

A.1 Supplementary Notes on the Controller

The following description refers to the state flow diagram in Fig. 4.2. To initialize the **Echo canceller**, the reset signal, *rset*, is asserted to force the **Controller** to state 1. The assertion of *rset* clears all the internal registers and keeps the **Controller** in state 1. After *rset* is disasserted, the **Controller** enters state 2 in the next clock cycle. In state 2, signal *ramclr* is generated by the **Controller**. The signal selects a zero pattern to be written to the coefficient RAM which resides in the **Rammod** module. The **Controller** iterates between state 1 and 2 to clear the memory byte by byte.

After all the 64 bytes of the memory are cleared, signal *cv* from the **Rammod** module is asserted and causes the **Controller** to exit from state 2. The **Controller** enters state 3 to wait for a new baud period. A low to high transition on *txclk* commences a new baud period. The **Controller** enters state 4 in the next clock cycle. In state 4, the **Controller** turns on the A/D Converter **Adc** to sample the line signal and clears the accumulator which stores the echo estimate. It also examines the activation control signal *act*. The signal is disasserted when the near end transmitter has ceased to transmit data for 32 consecutive baud periods. If it is asserted, the **Controller** enters state 7 to initiate the required computations for cancelling the echo.

One tap of the filter is being processed during one iteration through state 7 to state 10. A total of 32 iterations is required for all the 32 taps. In state 7, the low byte of the coefficient is read from the memory and strobed at the falling edge of *lstb*. The falling edge of *shiestb* causes the current transmitted data to be shifted into the **Delay line**. *Shiestb* is also used to strobe the partial convolution sum into the accumulator. Note that in the first iteration through state 7, *shiestb* does not have any effect on the accumulator. The accumulator remains cleared. In state 8, the high byte of the coefficient is read from the memory and strobed at the falling edge of *hstb*. In state 9, the low byte of the updated coefficient is written back to the memory. In this state, the product term $C_{i,n-1}D_{n-i}$ is computed and strobed at the falling edge of signal *txcostb*. In state 10, the high byte of

the updated coefficient is written back to the memory. In the second half cycle of state 10, the strobed product term $C_{i,n-1}D_{n-i}$ is added to the old partial convolution sum. The new partial convolution sum is not strobed into the accumulator until state 7 in the next iteration.

In the final iteration, signal *cv* from the **Rammod** is asserted and causes the **Controller** to enter state 11. In state 11, the **Controller** strobes the complete convolution sum of the echo estimate and subtracts it from the received data sample. The residue error is strobed into the D/A Converter data register at state 12. The **Controller** then returns to state 3 at the next clock cycle to wait for a new baud period.

If signal *act* is found to be disasserted in state 4, the **Echo canceller** is deactivated. The computation of the echo estimate and the updating action are not required. The **Echo canceller** is only required to transfer the sampled line signal directly from the A/D Converter data register to the D/A Converter data register. When the **Echo canceller** is activated, it takes a total of $(32 \times 4 + 1) = 129$ clock cycles to compute the echo estimate. To make the behavior of the **Echo canceller** consistent and transparent in both the activation and the deactivation phase, the **Echo canceller** will wait in state 5 for 129 clock cycles before it carries out the transfer. Next, the **Controller** goes back to state 3 and waits for a new baud period.

The state table and the Karnaugh maps of the state machine, derived from the state flow diagram, are shown in Table A.1a-e. The state machine has a total of 12 states, and is implemented using 4 state variables. It can be seen from Table A.1a, that in the normal system operation, all the state transitions of the **Controller** occurs only between adjacent states. The occurrence of non-adjacent transition is possible only when the **Controller** is being reset. Glitches may occur on the control signals under this situation. However, these glitches can be ignored. One can remove the possibility of having any glitches on the control signals during system reset by restricting the *rset* signal to be asserted only in state 4. The transition from state 4 to the reset state is an adjacent transition.

Effort has been made to optimize the logic of the controller globally, however, the controller may not be optimal.

The schematic diagram of the **Controller** is shown in Fig. A.1.1a-c. Fig. A.1.1a shows the generation of the four state variables. Part Count129 in this figure is the binary

		<i>x1x2</i>			
		00	01	11	10
<i>x3x4</i>	00	x	8	7	12
	01	x	9	10	11
	11	x	x	1	2
	10	6	5	4	3

Table A.1a State Assignment Table

		<i>x1x2</i>			
		00	01	11	10
<i>x3x4</i>	00	x	0	0	1
	01	x	1	1	1
	11	x	x	1	1
	10	1	0	act	1

$$x1p = x2b + x4 + x1x3act$$

Table A.1b Karnaugh Map of *x1*

		<i>x1x2</i>			
		00	01	11	10
<i>x3x4</i>	00	x	1	1	0
	01	x	1	cvb	0
	11	x	x	0	cvb
	10	0	svb	1	txclk

$$x2p = x1bx3b + x1x2x4b + x2x4btvb + x2x3bx4b + x1x3x4btclk + x2x3bcvb + x2bx3x4cvb$$

Table A.1c Karnaugh Map of *x2*

		<i>x1x2</i>			
		00	01	11	10
<i>x3x4</i>	00	x	0	0	1
	01	x	0	0	0
	11	x	x	1	1
	10	1	1	actb	1

$$x3p = x2bx4b + x3x4 + x1bx3 + x3actb$$

Table A.1d Karnaugh Map of *x3*

		<i>x1x2</i>			
		00	01	11	10
<i>x3x4</i>	00	x	1	0	0
	01	x	1	rcv	0
	11	x	x	1	rcvb
	10	0	0	0	0

$$x4p = x1bx3b + x2x3x4 + x3x4cvb + x2x4cv$$

Table A.1e Karnaugh Map of *x4*

counter which causes signal *tv* to assert after counting for 129 clock cycles in state 5. The assertion of signal *tv* causes the Controller to leave state 5 and transfers the received data sample to the other modules of the transceiver (transfers to the D/A data register in this design). Fig. A.1.1b-c shows the generation of the Controller output signals. All the one shot devices are shown in Fig. A.1.1c. In this figure, various types of Fhflop are shown. All of them are similar in structure, the differences are in the amount of delay which is put into the clock path and whether *fh*, *fhb* or both is brought out of Fhflop

In Fig. A.1.1c, it can be seen that some one shot signals are logic combinations of other one shot signals. The sharing of one shot signals results in the saving of hardware. For example, instead of having four Fhflops to generate four one shot pulses for signal *ceinc* from state 7 to state 10, the design combines four one shot pulses generated for other purposes to form *ceinc*.

The following paragraph will describe several special output signals shown in Fig. A.1.1c. The first one is the Converter clock *adclk*. The minimum clocking period of the A/D Converter in the analog library is specified to be 400 ns with a minimum pulse width of 100 ns. To meet the above specification, a 3 bit Johnson Counter (Fig. A.1.2) is used to generate the *adclk*. Since 3 bits are used, the resulting Converter clock period is 500 ns, with a pulse width of 250 ns. The data Converter can complete the conversion in 14 clock cycles. The total conversion time for each sample in this implementation is thus 7.00 μ s.

The second signal to be described is the start of conversion signal *soco*. As defined in the ES2 library specification, this signal is required to stay high at the falling edge of *adclk*, and to return to zero shortly afterwards. Note that in the manual, the hold time of *soco* with respect to the falling edge of *adclk* is not specified. In this design, the hold time is set equal to one half of the system clock cycle. There is no timing constraint violation reported on this hold time by the simulator.

Part Soc is designed to generate the signal *soco* which satisfies the timing requirements. The structure of Soc is shown in Fig. A.1.4. Assertion of *rset* causes Dflop1 to clear. A low on *soci* will also hold both the Dflop2 and Dflop3 in a clear state. After *rset* is disasserted, Dflop1 is allowed to respond to the signal *soci*

To initiate the conversion process, the one shot signal *soci* is pulsed in state 4. The falling edge of *soci* triggers *soco* to go high, which releases Dflop3 from the clear state and

Fig. A.1.1a Generation of the Controller State Variables

Fig. A.1.1c Generation of the Controller Output Signals

Fig. A.1.2 The Converter Clock Generator - Div8ctr

Fig. A.1.3 The Simulation of the Converter Clock Generator

allows it to respond to *adclk*.

The falling edge of *adclk* releases **Dflop2** from the clear state and allows it to respond to *sysclk*. At one half of the system clock cycle later, the falling edge of *sysclk* triggers **Dflop2** to clear *soco*. A low on *soco* clears **Dflop3** and **Dflop2** in sequence. This in turn releases **Dflop1** from the clear state and allows it to respond to *soci* again.

The three inverters at the output of **Dflop2** and **Dflop3** in Fig. A.1.4 add extra delay to the signal path. The delay is added to remove the race condition. When **Dflop1** is cleared, the feedback of *soco* to clear **Dflop3** and **Dflop2** occurs rapidly. A race condition between the clock and reset input of **Dflop2** is created. Strictly speaking, this race condition can be ignored, since the reset input should override the clock input according to the specification of the Flip Flop. The addition of these three inverters is to assure that the simulator is not reporting a violation of timing constraint. The simulation result of this circuit is shown in Fig. A.1.5.

The third signal, *on*, when asserted is used to turn on the comparator of the A/D Converter, when disasserted causes the comparator of the A/D Converter to enter the standby mode. The assertion of *on* is triggered by the one shot signal *soci*. The signal is cleared by the strobe signal, *erstb*, which strobes the residue error into the D/A Converter data register.

The last two signals which will be discussed in this section are the signals *smode* and *clrest*. The logic level of these two signals are always complementary to each other. A single RS latch is used to implement both signals. In every baud period, before any computations can begin, the accumulator which holds the convolution sum must be cleared. Signal *clrest* when asserted high clears the accumulator. On the other hand, signal *smode* is used to control the shift mode of the Delay line. As will be described in the following section, the Delay line has 2 shift modes. They are the right shift mode and the rotate mode. In state 6 of the Controller, the one shot signal *soci* is used to set the RS latch. This selects the right shift mode for the Delay line and clears the accumulator. In the first iteration of the computation, the one shot signal *hstb* resets the RS latch in state 8. This releases the accumulator from the clear state, and change the Delay line to rotate mode. The Delay line remains in rotate mode throughout the rest of the computation cycles. The RS latch will not be set again until the next baud period.

To end this section, the simulation result of the state transitions is shown in Fig. A.1.6. All possible state transitions are exercised by setting the input appropriately. The associated output signal of the Controller is shown in Fig. A.1.7.

A.2 Supplementary Notes on the Convolver and Updater

A.2.1 Notes on the Convolver

The following discussion refers to Fig. 4.7. Computation is performed by the module in iterations through state 7 - state 10. In each iteration of the computations, the low byte of a tap coefficient is read from the memory in state 7, while the high byte of the same coefficient is read in state 8 (Fig. 4.2). In these two states, multiplexer **Mux16a** multiplexes the two bytes of the coefficient on the memory read bus $b8(0 : 7)$ to **Reg16a**. By pulsing $lstb$ in state 7, the low byte of the coefficient is strobed into the low byte of **Reg16a**. The high byte of the coefficient is strobed into the high byte of **Reg16a** by pulsing the strobe signal $hstb$ in state 8. Both the convolver and the updater share the use of **Reg16a**.

A.2.2 Computation of the Product Term

In Fig. 4.7, the output of **Reg16a** feeds into the 1's complementer, **Compa**. When the multiplicand of the product term has a negative value, the control signal $comp$ will be asserted and causes the complementer to perform 1's complementation on the coefficient. Otherwise, the coefficient will pass through **Compa** without any alteration. The internal structure of **Compa** is shown in Fig. A.2.2.1a-b. Because the product term is computed in state 8 and strobed in state 9, the logic equation for $comp$ should be written as :

$$comp = (in\ state\ 8\ or\ 9) \cdot (negative\ multiplicand)$$

or

$$comp = (scla)' \cdot (t1b) \quad (A.2.2.1)$$

where $scla$ = select signal of **Mux16b**

$t1b$ = sign bit¹ of the multiplicand from Flip Flop 31 of **Srb**.

The output bus of the complementer $b3c(0 : 15)$, which represents the unshifted version of the complemented or uncomplemented coefficient, feeds directly into the input b of **Add2**.

¹ The sign convention of 2B1Q code and 2's complement are opposite to each other.

Fig. A.1.6 The Simulation Result of the Controller State Transitions

Fig. A.1.7 The Simulation Result of the Controller Output Signals

Fig. A.2.2.1a The Schematic Diagram of Compa.

Fig. A.2.2.1b The Schematic Diagram of Compcell.

Since the left shift operation always shifts by one bit, the operation can be hard-wired. It is done by connecting $b3c(0 : 14)$ to bit (1:15) of Mux16b's input a . The remaining bit 0 of input a is connected to the signal $comp$. As a result, if the multiplicand is positive, a 0 is shifted into bit 0. Otherwise a 1 is shifted into bit 0.

When the multiplicand is ± 1 , the left shift operation is not required. The left shifted version of the coefficient can be zeroed by asserting the control signal $zero$ to Mux16b in state 8 and state 9. Since bit 0 of ± 1 in 2B1Q code is always a 1 and bit 0 of ± 3 is always a 0, the logic equation of $zero$ is written as :

$$zero = (\text{in state 8 or 9}) \cdot (\text{multiplicand equal } \pm 1)$$

or

$$zero = (sela)' \cdot (t0) \quad (A.2.2.2)$$

where $t0$ = bit 0 of the multiplicand from Flip Flop 31 of Sra.

The internal structure of Mux16b is shown in Fig. A.2.2.2a-b.

For the multiplicand -1 , a 1 must be added to convert the 1's complemented coefficient to the 2's complement number. In this case, by setting the cin input of Add2, this extra 1 can be added to the product term. For the multiplicand -3 , the number 3 must be added to obtain the correct product term. Shifting a 1 into bit 0 of Mux16b's input a has the effect of adding 1. By setting the cin input of Add2 in state 8 and state 9, an additional 1 is added to the product term via cin . In state 10 (of the current iteration) and state 7 (of the next iteration), the remaining 1 is added to the echo estimate via cin . (Note : in the final iteration, since the Controller will not loop back to state 7. State 11 is the next state of state 10).

Based on the above discussion, the logic equation for cin can be written as :

$$cin = (\text{in states 8 or 9}) \cdot (\text{negative multiplicand}) + (\text{multiplicand is } -3)$$

or

$$cin = (sela)' \cdot (t1b) + (t0)' \cdot (t1b) \quad (A.2.2.3)$$

where $t1b$ = sign bit of the multiplicand.

$t0$ = bit 0 of the multiplicand.

Fig. A.2.2.2a The Schematic Diagram of Mux16b

Fig. A.2.2.2b The Schematic Diagram of Muxcellb

A.2.3 Accumulation of the Convolution Sum

In Fig. 4.7, **Reg16b** serves as the accumulator for the convolution sum. After the product term is strobed into **Reg16c** at state 9, the new partial convolution sum is computed in state 10 and strobed in state 7 of the next iteration. In state 10, the following events occur :

- 1) signal *selb* is asserted to select the old partial convolution sum on bus *est(0 : 15)* to pass through **Mux16a**
- 2) The old partial convolution sum is strobed into **Reg16a** at the falling edge of *lstb* and *hstb*. Since there is no complementation involved in the accumulation, the old partial convolution sum stored in **Reg16a** will be applied directly to the input *b* of **Add2**.
- 3) signal *sela* is asserted to select the product term on bus *txco(0 : 15)* to pass through **Mux16b**, whose output is connected to the input *a* of **Add2**. After the signals are stabilized, the output of **Add2** will carry the new partial convolution sum.
- 4) *sela* is maintained high in state 7 of the next iteration (or state 11 after the last iteration) to allow the strobing of the new partial convolution sum into the accumulator at the falling edge of *shiestb*.

multiplicand	coefficient	product term	old partial sum	new partial sum
-3	-256	+767	0	+768
-1	-256	+256	0	+256
+3	-256	-768	0	-768
+1	-256	-256	0	-256
-3	+255	-766	0	-765
-1	+255	-255	0	-255
+3	+255	+765	0	+765
+1	+255	+255	0	+255

Table. A.2.3.1 Eight test cases of the Convolver

Eight test cases quoted from the simulation of the **Convolver** are shown in Table

A.2.3.1 The numbers shown are in base 10.

A.2.4 Notes on the Generation of the Updating Quantity

The following discussion also refers to Fig. 4.7. To generate the corresponding updating quantity (+1 or -1), the inverted sign bit of the data-error product applies to the input a of **Add1**. The input a is internally extended to 16 bits; one more inversion is performed before they are applied to the 16 full adders in **Add1**. The inverted sign bit of the data-error product also applies to the carry input of **Add1**.

sign of error	sign of data	sign of product	carry in (cin)	16 bit input to the full adders
0 (+)	0 (+)	0 (+)	1	0000000000000000 = 0
0 (+)	1 (-)	1 (-)	0	1111111111111111 = -1
1 (-)	0 (+)	1 (-)	0	1111111111111111 = -1
1 (-)	1 (-)	0 (+)	1	0000000000000000 = 0

Table A.2.4.1 Generation of the updating quantity which is equal to the sum of column 4 and column 5

The updated coefficient at the output of **Add1** is written back to the memory in state 9 (low byte) and state 10 (high byte). The 8 bit multiplexer **Mux8** selects either the high byte or the low byte to be written to the memory. Reading the low byte of the old coefficient first and writing the low byte of the updated coefficients back to the memory first allows a certain amount of operation overlap. When the high byte is being read and strobed in state 8, the adder logic is allowed to propagate through the low byte. When the updated low byte is being written back to the memory in state 9, the adder logic is allowed to propagate through the high byte. This increases the safety time margin for logic propagation.

A.3 Supplementary Notes on the Transfer Module

The following analysis shows how the division, truncation (eq. 3.10) and subtraction operation (eq. 3.12) can be implemented efficiently. Let $\hat{E}_{16,n}$ to be the 16 bit convolution sum before the division and $\hat{E}_{8,n}$ to be the high byte of the convolution sum.

Recall

$$\begin{aligned}
 e_n &= RS_n - \hat{E}_n \\
 &= RS_n - \hat{E}_{16,n} \text{ div } 256 \\
 &= RS_n - (\hat{E}_{8,n} + U_0(-\hat{E}_{8,n})) \\
 &= RS_n + F1(\hat{E}_{8,n}) + 1 - U_0(-\hat{E}_{8,n}) \\
 &= RS_n + F1(\hat{E}_{8,n}) + U_1(\hat{E}_{8,n})
 \end{aligned} \tag{A.3.1}$$

where $U_0(x)$ = *unit step function*, with $U_0(0) = 0$. $U_1(x)$ = *unit step function*, with $U_1(0) = 1$. Note that $U_1(\hat{E}_{8,n})$ = the value of the inverted sign bit of $\hat{E}_{8,n}$

A.4 Supplementary Notes on the Rammod Module

The simulation of the **Ramctr** is shown in Fig. A.4.1. The structure of **Ramctr** is shown in Fig. A.4.2a-c. **Ramctr** not only generate the 6 bit memory addresses, but also generates the control signal *cv*. Signal *cv* is asserted when the last memory location *00_H* is being accessed. It informs the **Controller** to exit from the computation loop or the memory initialization loop in the next clock cycle.

Fig. A.4.1 The Simulation Result of the Ramctr

Fig. A.4.2a The Schematic Diagram of Ctrcell

Fig. A.4.2b The Schematic Diagram of Ctrcell2

Fig. A.4.2c The Schematic Diagram of the Ramctr

Appendix B

B.1 Overview of SOLO 1200

B.1.1 File and Library Organization

SOLO 1200 is a set of design programs which are linked together to form a design pipe-line. The programs in the set can either perform internal data conversion or use compatible data formats. Users are not required to carry out data conversion manually. The amount of time and the number of errors involved in performing data conversion are reduced. As **SOLO 1200** does not have a centralized data-base, each designer will usually store his design data under his own directory. Since **SOLO 1200** does not have a data-base manager, if one design is splitted among two or more designers, the integrity of the design data cannot be guaranteed. The enforcement of the data integrity and the version control then relies on the consensus among designers.

Design data is stored on hard disk in the form of sequential formatted files. Two file formats are used by **SOLO 1200**. The text format is used to bridge the man-machine gap and the binary format is used for its compactness. The two basic design files, the *.mod* file and the *.dft* file from which all other design files are generated, are stored in the text format. This is an important feature since the design being done on the SUN workstation can be transported to the VAX or the APOLLO workstation. This allows the sharing of designs among design centers. The standard cell libraries from the foundry are installed in a single directory. The libraries are also stored in text format. Each library contains at least two files, the *.inc* file and the *.dft* file. The *.inc* file contains the **MODEL** code while the *.dft* file contains the schematic symbols of the library parts. For the **BLOCKLIB**, **ANALOGLIB**, **PADLIB2** and **PADLIB3**, the CIF¹ files are also available. The user is allowed to create and maintain his own macro libraries. To define a macro library, the user will use **DRAFT** to define the symbol for each macro. The symbols are stored in a *.dft* file. The structural information of the macros in terms of the **MODEL** code are written and

¹ For the **BLOCKLIB** and **ANALOGLIB**, the CIF files contain only the information of the second layer metal.

stored in an *.inc* file with the same name as the *.dft* file. The user defined library allows the sharing of commonly used *parts*, increases the utility of the *parts* and reduces the design effort.

B.1.2 User Interface

DRAFT, **EXERT** and **WAVE** are the three programs which require frequent man-machine interaction. Although the user interface of these three programs provides facilities for the user to perform his design tasks, it is not robust. The following examples are only some of the problems which were encountered.

- 1) **DRAFT** does not provide an undo option for the available functions.
- 2) In some designs, circuit components may be drawn on several pages. If a signal name appearing on one page is changed, **DRAFT** will not update the name of the same signal on the other pages automatically. This may create consistency problems when the design is very complex and the designer does not exercise adequate care.
- 3) **WAVE** provides a logic-analyzer-like waveform display. A line cursor is provided to high-light the signals. The cursor always positions itself on the signal transitions. The cursor moves by jumping from one transition to the next, or to the previous transition of the selected signal. When another signal is selected, the cursor always jumps back to the left end of the display. The above cursor movement makes the measurement of the timing interval between two signals extremely difficult.
- 4) **EXERT** does not have a good exception handling on write collision. Instead of looping and retry, the simulator crashes when it tries to access a busy file server. This often occurs in the middle or at the end of the simulation. This leads to the loss of a vast amount of simulation time.
- 5) All three design programs do not allow the user to close the current design, to retrieve and work on another design. To work on another design, the user is required to exit the program, and re-invoke the program again.

B.1.3 Design Process Control

The integration of the design programs provides an easy way to exercise design control. The design control process is automated in **SOLO 1200**. The automation of the design

control process ensures a high probability of design correctness when the circuit is fabricated. SOLO 1200 does not form a full blown design system and does not create a design environment of its own. Each program executes independently under the UNIX operating system. The package does not have a kernel or a centralized executive process which performs job initialization, resource scheduling, or data-base management. The task of a design manager is distributed across various programs, i.e. 'a distributed design manager'. For each individual design, an information file (*.inf* file) is created early in the design process. The programs² play the role of the design manager by updating the *.inf* file. Usually, the *.inf* file is created by the MODEL compiler. Subsequent programs will append their execution information to the file. However, each program³ is capable of creating the *.inf* file if it is not pre-existed. The data which is logged into the information file includes :

- 1) the identification of the design program and the time of execution.
- 2) the files or the libraries used by the design program.
- 3) the options specified when invoking the program.
- 4) data generated during the execution of the program.

The information in the *.inf* file is organized in the sequence of : MODEL, PLACE, GATE, ROUTE, DRAW, EXERT, PACKAGE, TIS, SHIPDES. The re-execution of one program will remove the information generated previously by the subsequent programs. For example, if PLACE is re-run, the information generated by GATE and ROUTE at a previous time will be lost. If the designer needs to review the information at a later stage, he should copy them to his own log file before he re-executes a certain program.

One important aspect of the design control process is the existence of a structured design flow (Fig. B.1.3.1). Although structurally simple, Fig. B.1.3.1 does bring out the important steps involved in ASIC design and shows their logical sequence. ES2 has specified the temporal order of performing each step in the design flow. In the initial design phase, the temporal order need not be observed. To validate the design for fabrication, the temporal order must be followed strictly during the final design run. The temporal

² except FANOUT and EXPAND. EXERT will update the *.inf* file only when it is run under -load option.

³ except FANOUT, EXPAND, EXERT and PACKAGE

order is the same as the entry sequence of the .inf file listed above. **SHIPDES**, the "ship design" program is the most important part of the distributed manager. It performs the miscellaneous check on the design process (chapter 2) and determines whether the design can be shipped for fabrication.

Under a rigid control process, a loss of design flexibility is inevitable. A design change as simple as adding a buffer forces the designer to go through every step right from the beginning of net-list description and simulation, down to packaging. In the final check, an error or warning message pointed out by the simulator, which may not affect the circuit's operation, will prevent **SHIPDES** from validating the design. The designer can either modify the design to remove the warning or prove to the foundry that his design is functional. On the other hand, the control process forces the designer to be more organized, helps to uncover the potential sources of error and increases the level of design correctness. The loss of design flexibility is not a bad price to pay.

B.1.4 Design Correctness and Support for Testability

The use of the automatic design tools removes several sources of human error. Simulation and design control are the main tools for ensuring the correctness of a design. To help the designer catches these minor but potentially disastrous design errors, **SOLO 1200** is equipped with a circuit check capability which is an important attribute of the package. Common design errors due to carelessness such as implied-wired-or (output of two devices connected together), dangling input (input connected to nowhere) and dangling output (output connected to nowhere) can be checked by **DRAFT** and the **MODEL** compiler. These two programs can also check for syntactic errors such as naming faults and uniqueness of identifier. Programs such as **PLACE** and **PACKAGE** also have error checking capability. **PLACE** can verify that the input/output signals of a chip do not connect to the core array without connecting to the I/O buffers. **PACKAGE** ensures that the power supply pads are included in the design. The third approach used by **SOLO 1200** to stress the importance of design correctness is by educating the user. A small section of the start up manual is devoted to introducing some safe design practices.

Design for testability (DFT) does not have a good support in **SOLO 1200**. Among all the digital cell libraries, only one D-type Scan Flip Flop is included in the **BASELIB**. The design of the Scan Flip Flop is a simple D-type Flip Flop with the input multiplexed

order is the same as the entry sequence of the *.inf* file listed above. **SHIPDES**, the "ship design" program is the most important part of the distributed manager. It performs the miscellaneous check on the design process (chapter 2) and determines whether the design can be shipped for fabrication.

Under a rigid control process, a loss of design flexibility is inevitable. A design change as simple as adding a buffer forces the designer to go through every step right from the beginning of net-list description and simulation, down to packaging. In the final check, an error or warning message pointed out by the simulator, which may not affect the circuit's operation, will prevent **SHIPDES** from validating the design. The designer can either modify the design to remove the warning or prove to the foundry that his design is functional. On the other hand, the control process forces the designer to be more organized, helps to uncover the potential sources of error and increases the level of design correctness. The loss of design flexibility is not a bad price to pay.

B.1.4 Design Correctness and Support for Testability

The use of the automatic design tools removes several sources of human error. Simulation and design control are the main tools for ensuring the correctness of a design. To help the designer to catch these minor but potentially disastrous design errors, **SOLO 1200** is equipped with a circuit check capability which is an important attribute of the package. Common design errors due to carelessness such as implied-wired-or (output of two devices connected together), dangling input (input connected to nowhere) and dangling output (output connected to nowhere) can be checked by **DRAFT** and the **MODEL** compiler. These two programs can also check for syntactic errors such as naming faults and uniqueness of identifier. Programs such as **PLACE** and **PACKAGE** also have error checking capability. **PLACE** can verify that the input/output signals of a chip do not connect to the core array without connecting to the I/O buffers. **PACKAGE** ensures that the power supply pads are included in the design. The third approach used by **SOLO 1200** to stress the importance of design correctness is by educating the user. A small section of the start up manual is devoted to introducing some safe design practices.

Design for testability (DFT) does not have a good support in **SOLO 1200**. Among all the digital cell libraries, only one D-type Scan Flip Flop is included in the **BASELIB**. The design of the Scan Flip Flop is a simple D-type Flip Flop with the input multiplexed

Fig. B.1.3.1 The Design Flow of SOLO 1200

between the scan data and the normal data. The Flip Flop has only one clock input. The Level Sensitive Scan Device (LSSD) with two phase clocks would be a possible addition to enhance the digital libraries. Besides, the I/O buffers in the standard cell libraries do not have scanning capability. To perform I/O scan, extra design effort is required. Also the scheme would not be very efficient, since the scan devices and the scan control circuitries cannot be placed adjacent to the I/O buffers (they have to stay in the core array area). Finally, two useful and important design tools for testing are not included in the package, the testability analysis program (TA) and the automatic test pattern generator (ATPG). TA helps the designer to evaluate the testability of his design while ATPG generates the test patterns for the designer.

B.2 Notes on the Schematic Entry System

The **Model** code produced by **DRAFT** contains only the *part* and *pad* declaration, *part* and *pad* instancing, internal and external signal declarations and interconnection specifications. The generated code maps directly to the symbols in the schematic diagram. The I/O ports of a symbol map to the interface and signal declarations of the *part*. Each symbol in the schematic diagram maps to one instance statement in the **MODEL** code. Other compilation control statements of the HDL are not included in the automatically generated **MODEL** code. Among the four important constructs mentioned in chapter 5, the loop statement enables the compact description of a regular circuit structure. The decision branch statement allows the user to do conditional compilation based on a boolean flag or perform error checks on a *part* instance during compilation. This is a very useful feature for building the user defined macros. The constraint statement defines the timing relationships among the signals which are declared in the user defined macros. All the Flip Flops in **BASELIB** use the *constraint* statement to define the set up time and hold time of the data with respect to the active clock edge. The placement and routing control statements allow for control over the physical layout system.

It is unfortunate that **DRAFT** does not have the equivalent entities for language constructs such as the loop statement. The code generated by **DRAFT** is thus slightly longer than the hand-written code. Also the code of the user defined macros cannot be parameterized and does not have the self checking capability.

B.3 Notes on the Simulator

B.3.1 Transistor Model for Simulation

The transistor model used by **EXERT** is characterized by four parameters :

- 1) the 'on' channel resistance of the device.
- 2) the required time to pull up a net with unit loading from Gnd to Vdd/2.
- 3) the required time to pull down a net with unit loading from Vdd to Vdd/2.
- 4) the time to turn on a transistor.

The timing information in **EXERT** is derived by calculating the RC time constant at each net. R is the 'on' channel resistance of the transistors which are driving the net and C is the total capacitive loading at the net.

B.3.2 Simulation modes and the Simulator Interface

EXERT has a total of 3 simulation modes : the driven mode, the interactive mode and the batch mode. The driven mode is executed without any man-machine interaction. The design is driven by a .drv file which contains the encoded waveforms of the input signals. The input signals which are being driven must be the input signals to the root node of the design. The interactive mode allows the user to communicate with the simulator in real time. In this mode, a graphic window is used to display the hierarchical net-list of the design. The logic value of the net is displayed along with the net name. The interactive mode provides a direct visual impact and an immediate response to the user.

The batch mode is very much like the interactive mode. All the keyboard commands used in the interactive mode can be used in the batch mode. The differences between the two modes are 1) there is no graphic display. 2) all the commands to the command interpreter are coming from a command file instead of coming from the keyboard. The batch mode appears to be redundant in the beginning. However, when a design has to be modified and re-simulated many times, the command sequences which are used to guide the simulator in each iteration are more or less the same. Batching the commands in a command file saves time and effort in typing.

In the interactive mode, the user is allowed to go to any level of the design tree, set the logic value of a net, instruct the simulator to simulate the circuit based on the new setting

for a user specified duration and verify the simulation result. With these facilities the interactive mode and the batch mode are very useful in debugging the logic of the design. Although the simulator will also check for timing constraint violations in this mode, it is not the best means for the purpose of timing verification. To verify the timing, the driven mode should be used. In this mode, the design will be driven at full speed and run through many simulation cycles. After simulation, the signal traces can be displayed by **WAVE**. The displayed waveforms can be cross-referenced with the timing error messages generated by the simulator. Note that the driven mode can also be entered from the interactive mode and the batch mode. However, the driven mode entered from these two modes is different from the driven mode entered directly. In the latter case, the *.drv* file must always drive the input signals of the design tree's root node. In the former case, the *.drv* file can be coded to drive the input signals of the design tree's non-root node.

Division of the simulator into 3 modes complicates the invocation sequence and the software design. The division of these 3 modes is not clean and clear cut: overlapping of functions exists between modes. The interactive mode overlaps with the batch mode while the driven mode overlaps with both the interactive mode and the batch mode. During a driven simulation in the interactive mode, the graphic display is not turned off. This would be considered an unpleasant feature. When the circuit is being driven, the displayed logic values are changing on the fly. The designer will not have the time to discern what is happening. The display is not useful, while refreshing the display slows down the simulation speed greatly. It is believed that the structure of the simulator and the user interface can be simplified further. Instead of differentiating the three modes, a uniform invocation sequence should be provided. After the command interpreter of the simulator is entered the user can have the choice of turning on or turning off the graphic display, read a *.drv* file or use a command file. The command interpreter should also be able to put the simulation at the background under the request from the user.

In a multi-tasking environment, when a job is running in batch it is commonly running as a background process. Although the batch mode and the driven mode do not involve man-machine interaction, they are designed to run in the fore-ground. To free up the computer resources during long simulation runs, the simulation can be put at the background by appending the invocation command with : > & *fileid* &. This applies to the **SOLO 1200** version which runs under the UNIX shell. The symbol > & means redirect the output of

the simulator to a file called *fileid*. The symbol & at the end instructs the OS to put the job at the background. Note that without the re-direction command, the simulation job will go into an infinite loop as soon as the user has logged off from the system.

B.3.3 Simulations of the Analog Pad

The representation of an analog signal by the use of 8 bit quantity is probably because 1) the data converters provided in **ANALOGLIB** has an 8 bit resolution. 2) All analog input signals will eventually pass through the A/D Converter before it can get into the digital core array or all the digital output signals must pass through the D/A Converter before it can get out of the core array. Under this scheme, if the two reference voltages of the A/D Converter have the same value as the analog power supply, the conversion process can be modeled easily by copying the 8 bit analog input directly to the 8 bit digital output of the A/D Converter. If the two reference voltages of the A/D Converter do not have the same value as the supply rail, the A/D Converter has to carry out further conversion. The 'hand-quantization' of the reference voltages and the extra level of data conversions increases the quantization error.

The lack of R and C model in **EXERT** also causes the RC Oscillator to be modeled in an artificial way. The RC Oscillator, which can be disabled by disasserting the enable signal, is implemented using a Schmitt Trigger. When disabled, the RC Oscillator can be used straight forwardly as a Schmitt Trigger. To simulate the RC Oscillator in the enable mode, the user is required to code the output waveform of the RC Oscillator and apply it to the Schmitt Trigger input. During simulation, the simulator will take the input waveform of the Schmitt Trigger, invert the waveform and copy to the output. If the waveform coded by the user has a frequency outside the specified frequency range of the RC Oscillator, the simulator will set the Oscillator output to the unknown state.

B.4 Physical Layout System

B.4.1 Exploration of the Placement Algorithm

The component placement using *1-D* algorithm is found to be affected by the design hierarchy. The designer is required to partition the design carefully. Two interconnected components at the same hierarchical level and under the same *part* will always be placed

close to each other. Two interconnected components at the same hierarchical level but under different *part* may not be placed closely. Finally, two interconnected components at different hierarchical level may not be placed closely. Fig. B.4.1.1 is used to illustrate the above points. Conceptually, the components **W**, **X**, **Y** in the figure will be placed closely since they are at the same hierarchical level and under the same *part B*. Similarly, the components **T**, **U**, **V**, will be placed closely. The placement of **T**, **U**, **V** will not be intermixed with the placement of **W**, **X**, **Y** because the two sets of components belong to two different *parts*. The inverter may not be placed close to component **V** because it is at a different hierarchical level. If it is critical that the inverter be placed right next to the component **V** in order to reduce the net loading, one should move the inverter into *part A*. The design hierarchy does not seem to have effect on the placement scheme generated by the 2-D algorithm.

Fig. B.4.1.1 The Illustration of the 1-D Placement Algorithm

B.4.2 Placement Control Options

In the **SOLO** design package, there are several means through which the user can affect the placement scheme either directly or indirectly.

The resulting placement generated by the two placement algorithms depends on how the gates are initially placed. It has been verified that the initial placement schemes used by the algorithms are derived from the textual order of the **MODEL** code or the entry order of the schematic diagram. The same design, with the same array dimension specification,

but with different textual order, will result in a different placement. The effect of the initial placement is more prominent when the design size gets larger. Small designs are relatively unaffected by the initial placement. Controlling the textual order or the schematic entry order provides the indirect method of placement control. The main motivation of this method is to provide the placement program with a good initial start.

Another indirect placement control method is to partition the design hierarchy carefully, as mentioned in the previous section. This method is used to ensure the loading on the interconnection between two parts is not excessive and can be used only in the 1-D placement algorithm.

The placement control keywords of the **MODEL** language provide a direct method to control the placement. Experiments show that these keywords have control over the 1-D algorithm only. They cannot affect the placement scheme generated by the 2-D algorithm. The available placement control keywords include : *Serial*, *Locally Serial*, *Reverseserial*, *Uninterrupted*, *Newrow* and *Nofeeds*. As an example, the keyword *Serial*, if it appears in a part declaration, instructs the placement software to place all the components of the current part using the textual order of the **MODEL** code. The usage of each keyword is explained in the reference manual.

Although the control keywords in the **MODEL** code do not apply to the 2-D algorithm, one can use the *manual* option to affect the placement scheme generated by the 2-D algorithm. This option specifies that the 2-D algorithm should consult the manual placement file (*.pmf* file) when performing the placement. This option allows the designer to add blank stages to the design or force a particular part to be placed at a specified location of the array. This option cannot be used in 1-D algorithm.

The final way to influence the placement is by modifying the *.pos* file. An example of the file is shown in Fig. 2.7. Since the location of a gate in this file is not specified in an absolute sense, one can change its position without difficulty. The change can be made by deleting the current line which specifying the gate and move it to the new position. One can also flip a row by changing the notation $L \rightarrow R^4$ to $R \rightarrow L$ or vice versa. After changing the *.pos* file, **PLACE** can be re-run with the *replace* option. Under this option,

⁴ the notation means the placement starts from the left hand side of the row to the right hand side of the row

the software will re-place the design following the specification in the .pos file.

B.5 Area and Power Estimation

Whether a design should be implemented in one chip or as a chip set depends on the area utilization, the power consumption and yield considerations. Area and power estimation help the designer to revise and partition his circuit. SOLO 1200 does not provide an empirical formula which enables the designer to estimate the area and power consumption⁶. Through the use of the software, methods to estimate the above two quantities are derived.

B.5.1 Area Estimation

The total area of the design can be written as :

$$\text{Total area} = \text{core array area} + \text{pads area} \quad (B.5.1.1)$$

Using the pad dimensions given in the library specifications, the area occupied by the input, output and analog pads can be calculated directly

To estimate the area of the core array, one can use the empirical formula :

$$\text{core array area} = (1984.5 \times \text{total no. of stages of the design}) (\mu\text{m})^2 \quad (B.5.1.2)$$

and

$$1984.5 = 3.5 \times \text{area of one stages} \quad (B.5.1.3)$$

The factor 3.5 can be decomposed into 1 + 2.5. The factor 1 refers to the contribution of the device area and the factor 2.5 refers to the estimation of the routing area in terms of the device area.

B.5.2 Power Estimation

The power consumption can only be estimated after the design has been completed. The total power consumption of the chip can be written as :

$$\text{Total power} = \text{digital power} + \text{analog power} \quad (B.5.2.1)$$

⁶ The DRAW program does compute the total chip area after the physical layout has been drawn

The power consumption of the analog components can be obtained directly from the data provided with the SOLO design package. It is common to assume that the static power caused by the leakage current is small and can be ignored. The digital power is basically the switching power of the digital devices. In [35], the switching power of net i with load capacitance C_i and toggles at a frequency f is derived as :

$$\text{avg. power of net } i = C_i \times V_{dd}^2 \times f \quad (B.5.2.2)$$

The total power consumption can then be computed as :

$$\text{avg. power of chip} = \sum_i \text{avg. power of net } i \quad (B.5.2.3)$$

The .lda file generated by the DRAW program describes the loading capacitance in pF at each net. The remaining problem becomes the estimation of the toggling frequency at each net. It is done by simulating the design under a normal operating mode with the *toggle* option enabled. The *toggle* option instructs the simulator to generate a .cng file which reports the number of transitions occurred at each net during the simulation period. The .cng file aims to signal the designer to revise his simulation vector, so that each net of the design will toggle at least once. However, one can also use it to estimate the toggling frequency at a net :

$$\text{toggling frequency of net } i = \frac{\text{total no. of changes at net } i}{2 \times \text{total simulation time}} \quad (B.5.2.4)$$

A PASCAL program has been written to read the .lda and .cng files and compute the switching power of the design. This program is shown in Appendix G.

APPENDIX C

DATA GENERATOR PROGRAM

```
{*****
Program : datagen
author: K.K. Li
```

This program generates the transmit data (bit pairs) and calculates the corresponding echo. A pseudo random number generating algorithm which generates 0, 1, 2, 3 randomly. The number are decomposed into bit pairs 00, 01, 11, 10. The bit pairs are scrambled by a 23 order scrambling polynomial. The output bit pairs from the scrambler are considered to be the bit pair that sent onto the line. The program uses a hypothetical echo path impulse response and use the generated bit stream to perform a discrete time convolution for calculating the echo in the corresponding baud interval. The channel impulse response is read from the file : INIT. The impulse response duration is assumed to be 32 baud period. The MSB of the bit pair is written to file : TX2F, the LSB of the bit pair is written to file : TX1F, the echo is written to file : LSYMf. The file TX1F, TX2F, LSYMf will be inserted into the EXERT drive file for simulation. The data and the echo is also written into the file : CO, which will be used by another Pascal program : canceller (Appendix B) to simulate the modified sign-sign algorithm)

```
(program datagen (init, tx2f, tx1f, lsymf, co);
)program datagen (init, co);
{program datagen (init, col);}
const
nooftaps = 32;
{maxnoofdata = 400000;}
{maxnoofdata = no of transmitted data + 32}
maxnoofdata = 400032;
```

```
type
delay = array [1..nooftaps] of integer;
scram = array [1..23] of integer;
code1 = array [0..3] of integer;
code2 = array [0..3] of real;
imp = array [1..nooftaps] of real;
alpha = array [0..1] of char;
```

```
var
init,
{tx1f, tx2f, lsymf,
col: text;
}co : text;
level: alpha;
codetable1: code1;
codetable2: code2;
scrambler: scram;
coef: delay;
line: imp;
impulse: imp;
linesymbol, number : real;
echo: integer;
resolution: integer;
estimate: integer;
seeds: integer;
```

```

a,b,i,j,k: integer;
count,
txd2, txd1,
otxd2, otxd1: integer;

function randomsymbol (var seeds : integer): integer;
const
modulus = 2147483647;
multiplier = 16807;
increment = 13849;
noofsymbol = 4;

begin
seeds := ((multiplier*seeds)+increment) mod modulus;
randomsymbol := abs(seeds) mod noofsymbol;
end; (randomsymbol)

{ initialization }
begin
reset (init);
(rewrite (tx2f);
rewrite (tx1f);
rewrite (lsymf);
)rewrite (co);
(rewrite (co1);)
linesymbol := 0;
txd2 := 0;
txd1 := 0;
otxd1 := 0;
otxd2 := 0;
count := 0;
level[0] := 'l';
level[1] := 'h';
readln (init, seeds);
readln (init, otxd2, otxd1);
for a:= 1 to 23 do
scrambler[a] := 0;
for a:= 1 to nooftaps do
readln (init, line[a], impulse[a]);
readln (init, resolution);
for a := 0 to 3 do
readln (init, codetable1[a]);
codetable2[0] := -2.5;
codetable2[1] := -(5/6);
codetable2[2] := 2.5;
codetable2[3] := (5/6);

{ scrambling and convolution }
for a := 0 to maxnoofdata do
begin
linesymbol :=0;

{ do the convolution of past transmitted symbol }
for i := nooftaps - 1 downto 1 do
begin
line[i+1] := line[i];
linesymbol := linesymbol + (line[i+1]*impulse[i+1])
end;

```

```

{generate random data and converts it to bit pair}
j := randomsymbol(seeds);
txd2 := j div 2;
txd1 := j mod 2;

{scrambling the bit pair}
otxd1 := (txd1 + scrambler[18] + scrambler[23]) mod 2;
otxd2 := (txd2 + scrambler[17] + scrambler[22]) mod 2;
{otxd1 := (txd1 + scrambler[5] + scrambler[23]) mod 2;
otxd2 := (txd2 + scrambler[4] + scrambler[22]) mod 2;
}for i := 21 downto 1 do
scrambler[i+2] := scrambler[i];
scrambler[2] := otxd1;
scrambler[1] := otxd2;

{check transmitting state}
if (a>(maxnoofdata-32)) then
*line[1] := 0
else
line[1] := codetable2[otxd2*2+otxd1];

{include the current transmitted bit pair into the convolution sum}
linesymbol := linesymbol + (line[1]*impulse[1]);

{quantization}
if (linesymbol > 2.5) then linesymbol := 2.5;
  if (linesymbol < -2.5) then linesymbol := -2.5;
  number := ((linesymbol +2.5)*2*resolution)/5;
if not odd(trunc(number)) then
echo := trunc(number) div 2
else echo := (trunc(number)+1) div 2;
if (echo > 16383) then echo := 16383;

{This section of code generates the vector file for ES2 tester it also generates the
file for functional simulations. Note that the transmitter stop transmitting at max-
noof data --32 symbol period}
{ write the data to vector file
write (tx2f, ' per2*2=', level[otxd2]);
write (tx1f, ' per2*2=', level[otxd1]);}
{the addition of 128 is to make the sampled line symbol to offset binary code
write (lsymf, ' per2*2=10_', echo:4);
}writeln (co, codetable1[otxd2*2+otxd1]:6, (echo-(resolution div 2)):10)
{writeln (co1, codetable1[otxd2*2+otxd1]:6, (echo-(resolution div 2)):10)
if count =4 then
begin
writeln (tx2f, '-');
writeln (tx1f, '-');
writeln (lsymf, '-')
end;
count := (count +1) mod 5)
end
end.

```

APPENDIX D

SIMULATION PROGRAM OF THE MODIFIED SIGN - SIGN ALGORITHM

```

program canceller (initial, dat, cerr, merr);
const
  nooftaps= 32;
  {maxnoofsample= 400000;}
  {maxnoofsample = no. of symbol + 32}
  {maxnoofsample= 64; for exert}
  maxnoofsample = 400032;
  noofupdatepert= 1;

type
  delay = array [1..33] of integer;
  code= array [0..3] of integer;
  imp= array [1..33] of real;

var
  cerr,
  merr,
  initial,
  dat: text;
  delayline: delay;
  coef: delay;
  line: delay;
  impulse: imp;
  linesymbol: integer;
  estimate: integer;
  error: integer;
  seed: integer;
  a, i, j, k: integer;
  {%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
procedure initialization;
begin
  reset (initial);
  rewrite (cerr);
  rewrite (merr);
  linesymbol := 0;
  error := 0;
  readln (initial, seed);
  delayline[33] := 0;

  for a := 1 to nooftaps do
    readln (initial, delayline[a], coef[a]);
  reset (dat)
end;
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
procedure getsymbol (var delayline : delay;
  var linesymbol : integer);
var
  i: integer;
begin
  delayline[33] := delayline[32];
  for i := nooftaps -1 downto 1 do
    delayline[i+1] := delayline[i];

```

```

readln (dat, delayline[1], linesymbol);
end; (Getsymbol)
{~~~~~}
procedure cancelandupdate (linesymbol : integer;
    delayline : delay;
    var coef : delay;
    var estimate,
        error : integer;
        a : integer);

var
    sign, i : integer;

begin
    estimate := 0;
    if (a <= (maxnoofsamples - 32)) then
        for i := 1 to nooftaps do
            estimate := estimate + delayline[i]*coef[i]
        else
            for i := (a-maxnoofsamples+32)+1 to nooftaps do
                estimate := estimate + delayline[i]*coef[i];
            for i := 1 to nooftaps do
                begin
                    if ((error >= 0) and (delayline[i+1] >= 0)) or
                        ((error < 0) and (delayline[i+1] < 0)) then
                        sign := 1
                    else sign := -1;
                    coef[i] := coef[i] + sign
                end;
                error := linesymbol - trunc(estimate/256);
                if (error > 127) then
                    error := error - 256;
                if (error < -128) then
                    error := 256 + error
                end;
            {~~~~~}
            {~~~~~: main program ~~~~~}

begin
    initialization;
    for a := 0 to maxnoofsamples do
        begin
            getsymbol (delayline, linesymbol);
            for j := 1 to noofupdatepert do
                begin
                    cancelandupdate (linesymbol, delayline, coef,
                        estimate, error, a);
                    {if true then begin}
                    if (a mod 20) = 0 then begin
                        {for i := 1 to nooftaps do
                            writeln (cerr, 'coef =', coef[i]:12);
                        writeln (cerr, a:7, ' error =', error:12);}
                        writeln (cerr, a:7, error:12);
                        writeln (merr, a:7, sqrt(error):12)
                    end
                end
            end
        end.

```

APPENDIX E **EXERT DRIVE FILE FOR DESIGN** **CORRECTNESS**

```

#per1= 40
#per2= 6000
etx2 = 1 per2*66=h
etx1 = 1 per2*4=1 per2*2=h per2*2=1 per2*2=1 per2*2=1-
      per2*2=1 per2*2=1 per2*2=h per2*2=1 per2*2=h-
      per2*2=h per2*2=1 per2*2=h per2*2=1 per2*2=h-
      per2*2=h per2*2=1 per2*2=h per2*2=1 per2*2=1-
      per2*2=h per2*2=1 per2*2=h per2*2=h per2*2=h-
      per2*2=h per2*2=1 per2*2=1 per2*2=h per2*2=h-
      per2*2=h per2*2=1 per2*2=1 per2*2=1 per2*2=h-
      per2*2=h per2*2=1 per2*2=1 per2*2=1 per2*2=h

etx0 = 1 per2*4=1 per2*2=h per2*2=1 per2*2=h per2*2=1-
      per2*2=h per2*2=1 per2*2=h per2*2=1 per2*2=h-
      per2*2=h per2*2=h per2*2=h per2*2=1 per2*2=h-
      per2*2=h per2*2=h per2*2=h per2*2=h per2*2=h-
      per2*2=h per2*2=h per2*2=h per2*2=1 per2*2=1-
      per2*2=1 per2*2=1 per2*2=h per2*2=h per2*2=h-
      per2*2=h per2*2=1 per2*2=h per2*2=h per2*2=h-
      per2*2=1 per2*2=1 per2*2=h per2*2=h per2*2=1

erset = h per1*3=h per1*4=1
escan = 1
etxcik = h [per2]*82
eactive = h per2*82
ec2 = 1
ec1 = 1
esysclk = h [per1]*12300
{
  esysclkb = 1
  lin(0:7) = 1 per2*4=10_38 per2*2=10_135 per2*2=10_36 per2*2=10_70 per2*2=10_16-
            per2*2=10_58 per2*2=10_10 per2*2=10_109 per2*2=10_16 per2*2=10_111-
            per2*2=10_37 per2*2=10_83 per2*2=10_137 per2*2=10_27 per2*2=10_126-
            per2*2=10_144 per2*2=10_86 per2*2=10_136 per2*2=10_86 per2*2=10_81-
            per2*2=10_135 per2*2=10_85 per2*2=10_135 per2*2=10_206 per2*2=10_225-
            per2*2=10_238 per2*2=10_67 per2*2=10_96 per2*2=10_169 per2*2=10_167-
            per2*2=10_163 per2*2=10_43 per2*2=10_83 per2*2=10_120 per2*2=10_124-
            per2*2=10_121 per2*2=10_122 per2*2=10_127 per2*2=10_129 per2*2=10_130
  {
    vm(0:7) = 1
    vrp(0:7) = h
    vrp1(0:7) = 10_204 {4v
    vm1(0:7) = 10_36 {0.7
    scaout
    scaout2
    scaout3
    eop7
    eop6
    eop5
    eop4
    enp3
  }
}

```

APPENDIX F

EXERT DRIVE FILE ONE

```
#perl= 40
etx2 = l perl*1050=h [perl*4]*43
etx1 = l perl*1050=l [perl*4]*43 perl*64=h [[perl*2]*3 perl*139]*7
    [perl*2]*3 perl*2=l
    [perl*141=h perl*2=l]*3 perl*4=h perl*69=l perl*68=h
    perl*2=l [perl*4=h perl*133=l perl*4=h perl*2=l]*3
    perl*73=h perl*64=l [perl*4=h perl*2=l perl*89=h perl*48=l]*3
    [perl*4=h perl*2=l]*2 perl*83=h perl*48=l
    perl*4=h [perl*2=l perl*4=h perl*85=l perl*52=h]*3
    perl*2=l perl*73=h perl*16=l perl*52=h perl*2=l
    perl*123=h [perl*2]*6 [perl*4]*2 perl*123=l [perl*2]*8
    perl*4=h perl*2=l perl*135=h perl*2=l perl*4=h perl*139=l
    perl*4=h perl*123=l [perl*2]*8 perl*4=h perl*2=l perl*121=h
    [perl*2]*6 [perl*4]*2 [perl*139=l perl*4=h]*2 perl*121=l
    [perl*2]*7 [perl*4]*2 perl*123=l [perl*2]*7 perl*2=l
    perl*4=h perl*121=l perl*16=h perl*2=l perl*4=h
    perl*71=l perl*2=h
    perl*207=l perl*4=h perl*2=l perl*143=h
    [perl*2]*2 perl*73=l perl*46=h perl*18=l perl*4=h perl*2=l
    perl*77=h perl*62=l perl*4=h perl*2=l perl*75=h perl*2=l
    perl*66=h perl*2=l perl*2=h perl*77=l perl*64=h perl*2=l
    perl*2=h perl*79=l perl*62=h perl*2=l perl*2=h perl*81=l
    perl*60=h perl*2=l perl*2=h perl*83=l perl*58=h perl*2=l
    perl*2=h perl*85=l perl*56=h perl*2=l perl*2=h perl*87=l
    perl*54=h perl*2=l perl*2=h perl*89=l perl*52=h perl*2=l
    perl*2=h perl*91=l perl*50=h perl*2=l perl*2=h perl*93=l
    perl*48=h perl*2=l perl*2=h perl*95=l perl*46=h perl*2=l
    perl*2=h perl*97=l perl*44=h perl*2=l perl*2=h perl*99=l
    perl*42=h perl*2=l perl*2=h perl*101=l perl*40=h perl*2=l
    perl*2=h perl*103=l perl*38=h perl*2=l perl*2=h perl*73=l
    perl*68=h perl*2=l perl*121=h perl*18=l perl*4=h perl*2=l
    perl*2=h perl*137=l perl*4=h perl*2=l perl*2=h perl*75=l
    perl*66=h perl*2=l perl*77=h perl*2=l perl*64=h perl*2=l
    perl*79=h perl*2=l perl*62=h perl*2=l perl*81=h perl*2=l
    perl*60=h perl*2=l perl*83=h perl*2=l perl*58=h perl*2=l
    perl*85=h perl*2=l perl*56=h perl*2=l perl*87=h perl*2=l
    perl*54=h perl*2=l perl*89=h perl*2=l perl*52=h perl*2=l
    perl*91=h perl*2=l perl*50=h perl*2=l perl*93=h perl*2=l
    perl*48=h perl*2=l perl*355=h perl*2=l perl*46=h perl*2=l
    perl*97=h perl*2=l perl*44=h perl*2=l perl*99=h perl*2=l
    perl*42=h perl*2=l perl*101=h perl*2=l perl*40=h perl*2=l
    perl*103=h perl*2=l perl*38=h perl*2=l perl*105=h perl*2=l
    perl*36=h perl*2=l perl*75=h @perl*12205=l
etxc = l perl*1050=h [perl*4]*43 perl*72=h perl*81=l [perl*2]*15
    perl*4=l [perl*2]*15
    perl*83=l [perl*2]*14 perl*4=h [perl*2]*14 perl*85=l
    [perl*2]*31 perl*81=l [perl*2]*32 perl*81=h perl*32=l
    perl*145=h perl*32=l perl*81=h perl*64=l perl*73=h
    [perl*143=l perl*143=h]*8 perl*58=l [perl*2]*6 perl*131=h
    [perl*2]*6 perl*272=l perl*431=h [perl*2]*6 perl*131=l
    [perl*2]*6 perl*131=h [perl*2]*6 perl*131=l [perl*2]*5
    perl*145=l perl*73=h @perl*12205=l
erset = h perl*5=l [perl*256=h perl*4=l]*3 perl*256=h perl*4=l perl*5338=h
    perl*2=l perl*4641=h perl*2=l perl*1128=h perl*2=l
```

```

escan = 1 per1*1046=h per1*176=h [per1*139 per1*6]*8 [per1*139 per1*4]*27 -
      per1*139 -
      per1*4=h [per1*139 per1*6]*31 per1*139 per1*266=h [per1*139 per1*6]*5 -
      per1*139
lin(0:7) = 1
vm(0:7) = 1
vrp(0:7) = h
vm1(0:7) = 10_36
vrp1(0:7) = 10_204
etxclk = 1
cactive = 1
ec2 = 1 per1=h [per1*260]*3
ec1 = 1 per1=l per1*260=h per1*780=l per1*9986=h per1*260=l per1*8/l=h -
      per1*50=l
esysclk = h per1=l [[per1]*258 per1*2=h]*4 [per1]*4 per1*2=h [per1]*176 -
      [[per1]*137 per1*2=h [per1]*3 per1*3]*8 -
      [[per1]*137 per1*2=h per1 per1*3=h]*27 [per1]*137 -
      per1*6=h [[per1]*137 per1*2=h [per1]*3 per1*3]*31 -
      [per1]*137 per1*2=h [per1]*3 per1*5=h [per1]*255 -
      per1*3=h [[per1]*137 per1*2=h [per1]*3 per1*3]*5 -
      [per1]*137 per1*2=h [per1]*3 per1*5=h [per1]*47

esysclkb = 1 per1=h [[per1]*258 per1*2=h]*4 [per1]*4 per1*2=l [per1]*176 -
      [[per1]*136 per1*6=h per1*3]*8 [[per1]*136 per1*4 per1*3]*27 -
      [per1]*136 per1*3=h per1*4=l -
      [[per1]*136 per1*6=h per1*3]*31 [per1]*136 per1*7=h -
      per1*262=l [[per1]*136 per1*6=h per1*3]*5 [per1]*136 -
      per1*7=h per1*60=l

scaout
scaout2
scaout3
eop7
eop6
eop5
eop4
eop3
eop2
eop1
eop0
out(0:7)

(Signals to be monitor entered here
up#144
cv#137
sela#157
shiestb#171
smode#170
tm2#84
tm1#161
t1#86
t2b#85
selb#156
lstb#159
hstb#158
zero#1422
on#181

```


APPENDIX G

EXERT DRIVE FILE TWO

```

(This drive file verifies the state transition of the controller
(The first 256 states are used to clear the memory
(the state being tested are state7->state8
{
    state6->state3
{
    state4->state5
{
    state4->state7
{
    state3->state3
{
    state3->state4
{
    state4->state5->state6
{reset    -> state7->state8 state9->state10->state11
{
{
    state7->state8 state1->state2-> state3
#per1= 40
#per2= 6000
ctx2 = h
ctx1 = 1  per1*260=l per1*69=h [per1*2]*3 per1*139=l per1*4=h per1*4=l
per1*135=l -
    per1*2=h per1*2=l per1*139=l per1*2=h per1*6=l per1*135=l -
        per1*2=h per1*6=l per1*135=l per1*2=h per1*2=l per1*2=h -
        per1*137=l per1*2=h per1*2=l per1*2=h per1*137=l per1*2=h -
        per1*6=l per1*72=l per1*254=l per1=h per1*139=l per1*6=l -
        per1*64=l per1*4=h per1*4=l per1*135=h per1*2=l per1*2=h -
        per1*2=l per1*139=l per1*4=l per1*4=h per1*135=h per1*8=l -
        per1*212=l

ctx0 = 1  per1*265=h
erset = h per1*4=l per1*256=h per1*4=l per1*1543=h per1*4=l
escan = 1 per1*265=h per1*139=l per1*6 [per1*139 per1*4]*6 per1*139 -
    per1*260=h per1*139=l per1*6=h per1*139=l per1*4=h
    per1*139=l per1*6=h per1*139=l per1*4=h per1*139=l
    per1*6=h per1*139=l
ctxclk = 1 per1*260=l per1*1002=h per1*6=l
cactive =l per1*260=l per1*710=h per1*6=l
cc1 = 1
cc2 = 1
csysclk = 1 [per1]*264 per1*2=h [per1]*137 per1*2=h [per1]*3
per1*3=h
    [[per1]*137 per1*2=h per1=l per1*3=h]*6 [per1]*137
    per1*2=h [per1]*257 per1*3=h [per1]*137 per1*8=h
    [[per1]*137 per1*2=h per1=l per1*3=h [per1]*137
    per1*2 [per1]*3 per1*3=h]*2
{
csysclkb = h [per1]*264 per1*2=l [per1]*136 per1*6=h per1*3=l
    [[per1]*136 per1*4 per1*3]*6 [per1]*136
    per1*256=l per1*4=h per1*3=l [per1]*136 per1*6=h per1*3=l -
    [[per1]*136 per1*4=h per1*3=l [per1]*136 per1*6
    per1*3]*2

ln(0:7) =l
vm(0:7) = 1
vtp(0:7) = h
vtp1(0:7) = 10_204 {4v
vm1(0:7) = 10_36 {0.7

```

APPENDIX H

POWER ESTIMATION PROGRAM

```

(This program estimates the power consumption of a design)
program power (input, output, lda, cng);
const
  skiplda = 11;
  skipcng = 1;
  pico    = 1e-12;
  nano    = 1e-9;
  vddsquare = 25;

var
  lda, cng: text;
  first, last,
  sim: integer;
  ldnode, cnode: integer;
  i: integer;
  avgpower,
  cngval,
  ldval: real;

begin
  avgpower := 0;
  reset (lda);
  reset (cng);
  readln (cng);
  for i := 1 to skiplda do
    readln (lda);

  writeln ('please enter the first node');
  readln (first);
  writeln ('please enter the last node');
  readln (last);
  writeln ('please enter the duration of the simulation in ns. ');
  readln (sim);

  repeat
    read ('lda, ldnode');
    read (cng, cnode);
    while (ldnode <> cnode) do
      begin
        readln (cng);
        read (cng, cnode)
      end;
    readln (lda, ldval);
    read (cng, cngval);
    readln (cng, cngval);
    avgpower := avgpower + ldval*vddsquare*(cngval/2)/sim
  until (ldnode = last);
  writeln ('The estimated power consumption in the core array is : ', avgpower, ' mW')
end.

```

APPENDIX I

TEST CASES IN DRIVE FILE ONE

The test cases in drive file one (APPENDIX D) are listed below.
 The testing sequence are the same as in the drive file.
 The part names and signal names can be found in Fig. 4.15.

Let a=Reg16b output, b=Reg16c output, cin=carry input of Add2
 Test cases for Add2 are

```
a=0000000000000000 b=0000000000000000 cin=0
a=0101010101010101 b=1010101010101010 cin=0
a=1010101010101010 b=0101010101010101 cin=0
a=0101010101010101 b=0101010101010101 cin=0
a=1010101010101010 b=1010101010101010 cin=1
a=0000000000000000 b=1111111111111111 cin=1
a=1111111111111111 b=0000000000000000 cin=1
a=1111111111111111 b=1111111111111111 cin=1
```

The following patterns are used to test the multiplier :
 Each of the four coefficients are multiplied by -1, -3, +1, +3.

```
coefficient=0000000000000000
coefficient=1111111111111111
coefficient=1111111000000000
coefficient=0000000011111111
```

Let a= est(8:15), b=Adcreg 8b output.
 The following test patterns are used to test Add3 :

```
a=01010101 b=10101010
a=10101010 b=01010101
a=11111111 b=00000000
a=00000000 b=11111111
a=00000000 b=01010101
a=00000000 b=10101010
a=10101010 b=11111111
a=01010101 b=11111111
a=10101010 b=10101010
a=11010101 b=01010101
a=01111111 b=00000000
```

Let a=16b internal data of Add1, b=coefficient in Reg16a.

(Note that a + Add2 carry input = updating quantity)

```
a=0000000000000000 b=1111111111111111
a=0000000000000000 b=0000000000000000
a=0000000000000000 b=1111111111111110
```

```

a=0000000000000000 b=0000000000000001
a=0000000000000000 b=0000000000000011
a=0000000000000000 b=0000000000000111
a=0000000000000000 b=0000000000001111
a=0000000000000000 b=0000000000011111
a=0000000000000000 b=0000000000111111
a=0000000000000000 b=0000000001111111
a=0000000000000000 b=0000000011111111
a=0000000000000000 b=0000000111111111
a=0000000000000000 b=0000001111111111
a=0000000000000000 b=0000011111111111
a=0000000000000000 b=0000111111111111
a=0000000000000000 b=0011111111111111
• a=0000000000000000 b=0111111111111111
a=0000000000000000 b=1111111111111111
a=1111111111111111 b=0000000000000000
a=1111111111111111 b=1111111111111111
a=1111111111111111 b=0000000000000001
a=1111111111111111 b=0000000000000010
a=1111111111111111 b=0000000000000100
a=1111111111111111 b=0000000000001000
a=1111111111111111 b=0000000000010000
a=1111111111111111 b=0000000000100000
a=1111111111111111 b=0000000001000000
a=1111111111111111 b=0000000010000000
a=1111111111111111 b=0000000100000000
a=1111111111111111 b=0000001000000000
a=1111111111111111 b=0000010000000000
a=1111111111111111 b=0000100000000000
a=1111111111111111 b=0001000000000000
a=1111111111111111 b=0010000000000000
a=1111111111111111 b=0100000000000000
a=1111111111111111 b=1000000000000000

```

State transitions verified in drive file one are as follows :

```

state 1 -> state 2
state 2 -> state 1
state 8 -> state 9
state 9 -> state 10
state 10 -> state 7
state 11 -> state 12

```

The other state transitions are all verified by drive file two.
They are listed in APPENDIX G.

Bibliography

- [1] ES2, *Getting Started Solo 1200*, release 2.2.0.
- [2] ES2, *SOLO 1200 Reference Manual*, release 2.2.0, Vol. 1 & 2.
- [3] D. G. Messerschmitt, "Design Issues in the ISDN U-interface Transceiver", *IEEE J. Selected Areas in Commun.*, Nov., 1986
- [4] S. Haykin, *Adaptive Filter Theory*, Englewood Cliffs, N.J., Prentice-Hall, 1986.
- [5] C. W. K. Gritton, "Echo Cancellation Algorithms", *IEEE ASSP MAGAZINE*, April, 1984.
- [6] C. R. Eplevitch, W. A. Sethares, G. J. Rey, C. R. Johnson, JR., "Quiver Diagrams and Signed Adaptive Filters", Feb. 1989.
- [7] ANSI, *T1D1 Specification T1D1.3/87-003R8*.
- [8] A. Kanemasa, A. Sugiyama, S. Koike, T. Koyama, "An ISDN Subscriber Loop Transmission System Based on Echo Cancellation Technique", *IEEE J. Selected Areas in Commun.*, Nov., 1986.
- [9] P. F. Adams, R. P. Colbeck, "Considerations in the VLSI Circuit Implementation of 2B1Q Transceivers", *IEEE ISCAS*, 1988.
- [10] V. J. Mathews, S. H. Cho, "Improved Convergence Analysis of Stochastic Gradient Adaptive Filters Using the Sign Algorithm", *IEEE Trans. Acoust., Speech, Signal Processing*, vol. ASSP-35, No. 4, April, 1987.
- [11] T. A. C. M. Claasen, W. F. G. Mecklenbrauker, "Comparison of the Convergence of Two Algorithms for Adaptive FIR Digital Filters", *IEEE Trans. Acoust., Speech, Signal Processing*, vol. ASSP-29, No. 3, June, 1981.
- [12] D. D. Falconer, "Adaptive Reference Echo Cancellation", *IEEE Trans. Commun.*, vol. COM-30 No. 9, Sept., 1984.
- [13] W. I. Fletcher, *An Engineering Approach to Digital Design*, Englewood Cliffs, N.J., Prentice-Hall, 1980
- [14] E. J. McCluskey, *Logic Design Principles*, Englewood Cliffs, N.J., Prentice-Hall, 1971.

- [15] P. S. Bottorff, R. E. France, N. H. Garges, E. J. Orosz, "Test Generation for Large Logic Networks", *Dig. 14th Des. Autom. Conf.*, New Orleans, La., June, 1977.
- [16] T. W. Williams, E. B. Eichelberger, "Random Patterns Within a Structured Sequential Logic Design", *Dig. 1977 IEEE Semiconductor Test Conf.*, Cherry Hill, N.J., Oct., 1977
- [17] E. B. Eichelberger, "Latch Design Using Level Sensitive Scan Design", *Dig. COMPCON Spring '83*, San Francisco, Feb., 1983.
- [18] E. B. Eichelberger, "A Logic Design Structure for LSI Testability", *Proc. 14th Des. Autom. Conf.*, New Orleans, La., June, 1977.
- [19] K. J. Lieberherr, S. E. Knudsen, "Zeus: A Hardware Description Language for VLSI", *Proc. 20th Des. Autom. Conf.*, Miami, Florida, June, 1983.
- [20] P. M. B. Veiga, M. J. A. Lanca, "HARPA: A Hierarchical Multi-Level Hardware Description Language", *Proc. 21st Des. Autom. Conf.*, Albuquerque, New Mexico, 1984.
- [21] R. B. Hitchcock, "Timing Verification and the Timing Analysis Program", *Proc. 19th Des. Autom. Conf.*, Las Vegas, Nevada, 1982.
- [22] L. C. Bening, T. A. Lane, C. R. Alexander, J. E. Smith, "Developments in Logic Network Path Delay Analysis", *Proc. 19th Des. Autom. Conf.*, Las Vegas, Nevada, 1982.
- [23] H. Tahawy, G. Mazare, B. Hennion, P. Senn, "A New Implementation Technique for the Simulation of Mixed (Digital-Analog) VLSI Circuits", *IEEE International Conference on Computer-Aided Design*, Nov., 1987.
- [24] A. R. Newton, "Techniques for the Simulation of Large Scale Integrated Circuits", *IEEE Trans. of Circuits and Systems*, Vol. CAS-22, pp. 744-749, Sept., 1979.
- [25] P. H. Reynaert, H. De Man, G. Arnout, J. Cornelissen, "DIANA: A Mixed Mode Simulator with a Hardware Description Language for Hierarchal Design in VLSI", *Proc IEEE Intl. Conference on Circuits and Computers*, Port Chester, N.Y., Oct., 1980.
- [26] R. E. Bryant, D. Beatty, K. Brace, K. Cho, C. Sheffler, "COSMOS: A Compiled Simulator for MOS Circuits", *Proc. 24th Des. Autom. Conf.*, Miami, Florida, June, 1987.
- [27] V. Ramachandran, "An Improved Switched Level Simulator for MOS Circuits", *Proc 20th Des. Autom. Conf.*, Miami, Florida, 1983.
- [28] D. Adler, "A Dynamically Directed Switch Model for MOS Logic Simulation", *Proc*

25th Des. Autom. Conf., Anaheim, June, 1988

- [29] S. L. Hurst, *Custom-Specific Integrated Circuits*, New York, Marcel Dekker Inc., 1985.
- [30] D. N. Deutsch, "A Dog-leg Channel Router", *Proc. 13-th Des. Autom. Conf.*, San Fransico, June 1976.
- [31] R. J. Lipton, J. D. Nash, "A Greedy Channel Router", *Proc. 19-th Des. Autom. Conf.*, Las Vegas, June 1982.
- [32] T. Yoshimura, "An Efficient Channel Router", *Proc. 21-st Des. Autom. Conf.*, Albuquerque, June, 1984.
- [33] T. L., *2 μ m CMOS Standard Cell Data Book*, 1986.
- [34] N. Weste, K. Eshraghian, *Principles of CMOS VLSI Design - A Systems Perspective*, Addison-Wesley, 1985.
- [35] T. M. Luk, *Implementation of a Timing Recovery Circuit for a Mobile Radio Receiver - Master Thesis*, Carleton University, May 1989.
- [36] R. E. Vallee, *Development of a CMOS VLSI Design Environment and the Implementation of a Multi-Mode Data Acquisition Control Unit - Master Thesis*, Carleton University, August, 1985.

END

1 9/0 2/9 1

FIN