A COMPUTER AIDED DESIGN FOR DIGITAL FILTER IMPLEMENTATION

by

P. K. M. Lai

A Thesis submitted for
the Degree of Doctor of Philosophy
in the Faculty of Engineering
University of London

Department of Electrical Engineering
Imperial College of Science and Technology
London

January, 1979.

To My Parents,
Brothers, Sisters,
and Paulina . . .

ACKNOWLEDGEMENT

I would like to express thanks first to my supervisor, Dr. A.G. Constantinides; my teacher in digital signal processing and a friend. Without his stimulations, and his critical but constructive comments, this research and thesis would not have come to fruition.

Thanks are also due to members of staff in the Communication Section; of these I would like to thank especially Professor E.C. Cherry and Dr. R. King for their encouragements and supports for the IEE Scholarship and travelling grant from the Ecole Polytechnique Federale De Lausanne (EPFL), Switzerland.

Dr. Bob Spence has constantly reinforced my interests in APL (A Programming Language) and its applications to computer-aided in design. Through him, I have had the firsthand experience of using Xerox APL and Sharp APL. This has greatly assisted me in understanding and knowing the capability and limitation of APL. These experience and knowledge are essential for the present research.

I would like to thank all my colleagues in the Communication Section. They have been generous with their assistance and friendship. Among them, I would like to especially mention Dr's Victor Lawrence, Henry Chasia, Abdullah Hashim, Michael Buckley, Majid Ahmadi, David Haigh and Hugh Koanantakool.

I am grateful to Mr. S. Thompson, Manager of Atkins Computing - On-line Systems, who provided me with unlimited APL facility prior to the advent of APLUM and APL.2 at the Imperial College Computer Centre. The APL support came at a very crucial moment of my research; the usefulness of APL in the "Top-down Design Approach" was confirmed. In addition SIMLOGS was formulated and extensively developed on the Xerox APL.

The experience of system design with emphasis on the "human aspects of computing" would not have been gained without the activities, as being a system analyst and a senior system consultant, concurrent to my research at Imperial College. During these activities, I have been able to apply the concepts and techniques formulated during my research. The former therefore has been aiding the progress of the latter. In this respect, I would like to acknowledge all my clients; in particular, Mr. Michael Linder (Manager of Telecommunication, Unilever)and Mr. Mike Scott (Manager of Financial and Resources Planning, Rank Xerox).

I am grateful to Professor Chu of Maryland University (USA) and Professor Brown of the Air Force University (USA) who respectively have provided me with a copy of the "Computer Design Language (CDL)" and "Digital Logic Simulator (DLS)". These simulators facilitated knowledge of the Users' attitudes towards using simulators for digital system design. These have formed the bases and prompted my efforts towards investigating computer-aided design systems.

My graditude should also be extended to Dr G. Lockhart of Leeds University, Dr. P. Rayner of Cambridge University and Dr. A. Davis of City University; for their readiness in exchanging ideas during conferences. Courses on Microprocessors and Microcomputers organized by Dr. A. Davis have also given me a better understanding of the capabilities of the technology.

The financial support of the one year IEE scholarship and the subsidy of travel for the presentation of a paper at the EPFL by Dr. A Lagos (of EPFL) are grateful acknowledged.

I am deeply indebted to my parents for their encouragements and financial assistance. The efforts of my youngest brother, Stephen, in preparing graphics for my publications and those of Miss Margaret Foo in typing a major part of this thesis are considerable.

Finally (but not the least), my fiancee, Dr. Paulina Chan, has been my permanent source of encouragement and assistance. The task of conducting responsible activities as a system analyst and a senior system consultant, and concurrently pursuing my research would not be possible without the incessant efforts of Paulina; for example, attaining information and material which are required for my research, and preparing drafts and final version of my publications till early in the morning to meet dead-lines. Her affection and encouragement have made my roles inside and outside the College both enjoyable and rewarding.

ABSTRACT

Together with the expanding applications of digital signal processing made possible by the increasing rapid changes in semiconductor technology and processing algorithms, the number of techniques available and possibilities to be explored are speedily growing. Hence, a large number of implementation approaches have grown up. Enormous efforts have been expanded in cultivating various design metaphors for the implementation of digital signal processors; the field of digital filtering is a particular example of this.

The fundamental principles behind hardware implementations have already been widely considered and reported by various authors. This author shares the opinion with other designers, that there is a great need for the provision of aids for systematically deciding among the variety of possible implementation **strategies** in existence; this forms the main objective of our research.

Our attempts in rationalizing the activities of a digital system designer have resulted the formulation of an "Interactive Model" which in turn has led to the development of a "Top-down Design Approach". The Approach, however, rests heavily on computer analyses and simulations; and the formulation, design and implementation of a good design software package(s) for this purpose is therefore of paramount importance. To this end, the criteria of designing suitable computer aids and the bases for the evaluation and comparison among computer programming languages have been treated at great length.

APL (A Programming Language) has been found to be the most suitable language for our "Top-down Design Approach". The advantages and disadvantages of choosing APL are discussed. This is followed by a review of available APL processors and their implications on the the design of computer aided in design (CAD) systems. In this connection, a CAD system based on a subset of APL is proposed and the implications of it are carefully considered. To ensure an economical use of the CAD system, we have discussed and proposed methods of interfacing APL and non-APL processors which may provide facilities lacking in APL. As a measure of sharing the cost of developing and maintaining a CAD system, we have proposed a method of a multi-user environment for the CAD system.

As an effort of providing subsystems for the "Top-down Design Approach", two extensive packages have been developed — DISPIC's and SIMLOGS. DISPIC's facilitate the design and simulation of programmable digital filters (and signal processors) at the functional and register transfer levels. While SIMLOGS provides the tools for analysis and simulation of logical circuits (or systems) at the gate/logic level. Possible extensions for these packages are indicated in the pertinent chapters.

Within the framework of the Top-down Design Approach, we considered the mathematical and algorithmic studies of the residue number system to implement digital filters. This number system has an inherent parallel structure which could lead to high speed real time signal processing. Hardware studies aided by computer tools developed (such as LOGMIN) are conducted. Significant progress in this area by other researchers is briefly reviewed.

ABSTRACT -- continued

Our efforts have demonstrated the feasibility and advantages of the proposed Top-down Design Approach. With the limitation of resources (human, material and financial) it has not been able to fully implement and thoroughly evaluate the Top-down Design Approach. However, we have made a starting point and a step towards the right direction for such a system. It is hoped that in its present form it will assist designers and the efforts of others will enhance the usefulness of the Design Approach.

The results of the present research have laid a foundation for extended studies which could proceed in several directions. The thesis concludes with a recommendation for possible research and related studies which have not been addressed or adequately considered during the present studies.

TABLE OF CONTENTS

		PAGE
ACKNOWLEDGEMENT		
ABSTRACT		
CHAPTER 1	INTRODUCTION	1
1.1	Background and Objectives	
1.2	Synopsis -- Structure of the Thesis	4
CHAPTER 2	A PROPOSED SYSTEMATIC APPROACH OF DESIGNING DIGITAL SIGNAL PROCESSORS	6
2.1	Consideration of the Activities of a Designer: to rationalize his activities	7
	2.1.1 Levels of Design	
	2.1.2 Conceptual Approach	9
2.2	The Design Process: An Interactive Model and A Systematic Approach	11
2.3	A Proposed Top-down Design Approach	15
	2.3.1 Description at various Levels	17
	2.3.2 Verification methods	21
	2.3.3 Heuristic approaches	24
	2.3.4 Epilogue	26
2.4	The Importance of Designing Suitable Computer Aids	29
	2.4.1 The Classifications of Computer Software Applications	30
	2.4.2 Design approaches to meet the requirement of softwares and user needs	33
2.5	The Choice of A Suitable Computer Language	39
	2.5.1 Bases for the Evaluation and Comparison among Computer Programming Languages	
	2.5.2 The Choice of APL (A Programming Language) for the Top-down Design Approach	45
	2.5.3 Review of available APL processors	53
	2.5.4 The Implications of Enhanced APL Processors on the Design of CAD systems	71
2.6	Implementation of the CAD and Environment for the Top-down Design Approach	76
	2.6.1 The Use of APL as a Hardware Design Language: Syntax & Semantics	77
	2.6.2 A Top-down Approach Illustrated by Examples	90
	2.6.3 A CAD Design System and the User Environment for the Top-down Design Approach	102

		PAGE
CHAPTER 3	THE DESIGN & SIMULATION OF A CLASS OF SIGNAL PROCESSORS — DISPIC's	121
3.1	Introduction	
	3.1.1 Scope of the Design	122
	3.1.2 System Level	123
	3.1.3 Functional & Register Transfer Levles	126
3.2	The Design of DISPIC's	126
	3.2.1 Objectives of DISPIC's	127
	3.2.2 Architecture of DISPIC's	
	3.2.3 Data formats: Instruction & Number	132
	3.2.4 Address modes	134
	3.2.5 Arithmetic — fixed and floating	135
	3.2.6 Index Register and Subroutines	136
	3.2.7 Input/Output Facility	142
	3.2.8 Instruction Cycle	
	3.2.9 Error-Messages	143
3.3	Implementation & Applications of DISPIC's	145
	3.3.1 Implementation of DISPIC's on a Host Computer	
	3.3.2 Simulation of an IIR filter with DISPIC	148
	3.3.3 Future Developments of DISPIC	
CHAPTER 4	MODELLING & SIMULATING LOGIC CIRCUITS AND SYSTEMS WITH SIMLOGS	155
4.1	Introduction	
	4.1.1 The Necessity of Employing Logic Simulation in Design Verification & Testing	
	4.1.2 SIMLOGS as a Subsystem for the Top-down Design Approach	156
4.2	The Design of SIMLOGS — Modelling & Implementation	
	4.2.1 Description/Specification of Circuits	157
	4.2.2 Physical Considerations of the Model	160
	4.2.3 Implementation & User Environment of	164
4.3	Applications & Future Developments of SIMLOGS	167
	4.3.1 Logic Design Aided by SIMLOGS	
	4.3.2 SIMLOGS as an Aid to Verify a Digital System Design	169
	4.3.3 Future Development of SIMLOGS	173

		PAGE
CHAPTER 5	STUDIES OF DIGITAL FILTER IMPLEMENTATION USING RESIDUE NUMBER SYSTEMS	176
5.1	Introduction to Residue Number Systems (RNS)	178
	5.1.1 The weighted and the mixed-radix number systems	
	5.1.2 The residue number system	179
	5.1.3 Characteristics of the number systems considered	180
5.2	Basic Arithmetic in the RNS	182
	5.2.1 Addition and Subtraction in the RNS	
	5.2.2 Multiplication in the RNS	183
	5.2.3 Division in the RNS	
5.3	General Comments on Rational Number Operations in in the RNS	185
	5.3.1 Sign Representation	
	5.3.2 Scaling	186
5.4	Fractional Multiplication	192
	5.4.1 Rational Number Representation	
	5.4.2 Systematic error correction & rounding	193
	5.4.3 Fractional Multiplication using Mixed-Radix Conversion	195
	5.4.4 Fractional Multiplication using Macro-coefficient Extraction	199
5.5	Implementation of Digital Filters using the RNS	206
CHAPTER 6	CONCLUSION	209
6.1	Summary & Concluding Remarks	
6.2	A Recommendation for Future Studies	210
	6.2.1 Future Developments of the CAD System	
	6.2.2 Subsystems for the Design Approach	211
	6.2.3 Interface with Non-APL Processors and Application Packages	
	6.2.4 Evaluation of the Top-down Design Approach	212
	6.2.5 Application of the RNS to Implement Digital Signal Processors	
REFERENCES		214
APPENDICES :		
Appendix 1	Summary of APL/360 Features	223
Appendix 2	LOGMIN : A Logic Minimization Package Implemented in FORTRAN IV	227
Appendix 3	A Note on the Backus-Nauer Form (BNF) Equation for Syntax Description	238

PAGE

Appendix 4	A Collection of APL Functions for Digital Filter Analysis and Simulation	242
Appendix 5	APL Plotting Routines	248
Appendix 6	A Collection of APL Functions for Signal Analysis	256
Appendix 7	A List of APL Functions for Binary-arithmetic ("rounding")	259
Appendix 8	APL Functions for the Algorithmic Representation of Computer Arithmetic	264
Appendix 9	Simulation of a Simple 8-bit Programmable Machine — BCOMP8 using APL	270
Appendix 10	Compiling Compound Expressions with APL	279
Appendix 11	Architectural Description & Simulation — use of APL at the Register-Transfer Level	283
Appendix 12	Description & Simulation of Machine Architecture using APL	288
Appendix 13	Source Code Listing of DISPIC's	291
Appendix 14	Source Code Lisiting of SIMLOGS	303
Appendix 15	Fundamentals & Algorithms for Residue Number Systems	310
Appendix 16	A Study on the Arithmetic Logic for Residue Addition and Multiplication	334
Appendix 17	The Application of Read-Only-Memory (ROM) to Implement Digital Filters	343

LIST OF FIGURES

		PAGE
Fig. 1	A Digital Signal Processor System (Digital Filter)	8
Fig. 2	An "Interactive Model" for the Top-down Approach to Designing Digital Signal Processors	14
Fig. 3	A Proposed Top-Down Design Approach	16
Fig. 4	Example of an "automatic generation of the Implementation"	22
Fig. 5	An example of a Man-Machine Dialogue illustrating the occassional-user & expert modes	36
Fig. 6	Workspace management under APL/360 environments	58
Fig. 7	Classification of the APL System Commands	59
Fig. 8	Overlaying multi-workspaces using □PCOPY or □COPY	63
Fig. 9	Overlaying multi-workspaces and data file(s) for a large amount of data storage	63
Fig. 10	Example of an Access Matrix	67
Fig. 11	The Schedule Table — an example of a communication using Shared Variables	70
Fig. 12	Functional arrangement of a CAD system	72
Fig. 13	A Schematic of a Concurrent Process — segment of a microprogram	88
Fig. 14	A Schematic Diagram of a Carry-Save Multiplier	96
Fig. 15	A Register-Transfer Description & Simulation of a Carry-Save Multiplier	97
Fig. 16	Illustration of the Carry-Save Principle	98
Fig. 17	Logic Diagram of a Divide-by-five (Modulo-5) Up/Down Counter	100
Fig. 18	Description & Simulation of the Modulo-5 Counter with DLS (Digital Logic Simulator)	101
Fig. 19	A CAD Design System based on a subset of APL (i.e. APL/HDL)	103
Fig. 20	Interface with Non-APL Environments	116
Fig. 21	A Multi-user Environment for the Top-down Design Approach	118
Fig. 22	Structure of a multi-user environment dialogue	120
Fig. 23	A Second order Biquadratic Section	123
Fig. 24	An APL Description of an Nth order IIR Digital Filter	124
Fig. 25	Impulse Response of a 2nd order IIR Digital Filter	125
Fig. 26	Frequency Response of the 2nd order IIR Filter	125
Fig. 27	Architecture of DISPIC's	128

PAGE

Fig. 28 Instruction & Number Formats of DISPIC 139
Fig. 29 A Cube root subroutine
Fig. 30 Flow Chart to compute $X = 2 + 3A^{1/3} + B^{1/3}$ 140
Fig. 31 A DISPIC program for $X = 2 + 3A^{1/3} + B^{1/3}$ 141
Fig. 32 Fetch-Execute Cycle 141
Fig. 33 Simulation of an APL "scan" operator on DISPIC2 150
Fig. 34 Definint System Parameters for DISPIC3 152
Fig. 35 A 2nd order IIR FILTER (IIR2) Programmed on DISPIC (Ver.1) 153
Fig. 36 Memory map of DISPIC after IIR2 is loaded 154
Fig. 37 Block Structure, Schematic Diagram & Symbolic Description of a Circuit 158
Fig. 38 General Interpretation of Z ID X 159
Fig. 39 Delay Parameters 160
Fig. 40 Specification of SN5472 using SIMLOGS 162
Fig. 41 The Environment of SIMLOGS 166
Fig. 42 A Digital System Design using Logic Simulator 168
Fig. 43 Logic Design Verification using SIMLOGS 169
Fig. 44 Description and Simulation of a Gated JK flip-flop 170
Fig. 45 Description and Simulation of a Synchronous Decade Counter 174

LIST OF TABLES

Table 1 A list of APL Implementation on Various Computers 52
Table 2 APL Services at U.K. Time-Sharing Bureaux 56
Table 3 Access Codes (permission) & Corresponding File System Functions 67
Table 4 A Selection of Computer Aids for the Top-down Design Approach 92
Table 5 Instruction set of DISPIC (version 1) 130
Table 6 System Commands of DISPIC (version 1) 131
Table 7 Primitives of SIMLOGS 158
Table 8 SIMLOGS Commands 161

Chapter 1 : INTRODUCTION

1.1 BACKGROUND & OBJECTIVES

Digital signal processing emerged in the mid 1960s and has since then become a major discipline. In digital signal processing (1-5), we are concerned with signals and systems that are the discrete-time counterpart of the more familiar continuous-time systems. The field may be subdivided into two inter-related areas : digital filtering and spectral analysis. Digital filters can perform the same function as that of the analog filters: while the analog approach, in some cases, may be difficult or unfeasible to implement practically. Digital filters offer important engineering advantages; such as perfect reproducibility and a guaranteed level of performance, the increased ease in changing the filter characteristics, and the possibility of time sharing the same hardware system among a multiplicity of filtering functions. These advantages alone would, in many cases, make digital filtering an attractive alternative to analog processing. There is a further important advantage: the possible modularized hardware for customized large scale integration(LSI). Digital spectral analysis has been given much impetus by the introduction of the Fast Fourier Transforms(FFT). These computationally efficient algorithms have gained widespread use in many diverse scientific disciplines, making possible accuracies and resolution that could not even be contemplated before with an analog approach.

The proliferation of the use of digital signal processing can be witnessed by its appearance in a variety of areas of scientific endeavour such as biomedical engineering, seismic and geophysical research, image processing and pattern recognition, radar and sonar detection,acoustics and speech research, and telecommunication(6-9). In many of these applications, there is a real need for the signal processing to operate at a sufficiently high speed as to permit real time processing. At the same time, these processors must be economically competitive to be within the reach of a large community.

Technology often dominates machine designs. The steadily increasing capabilities and decreasing cost of digital integrated circuits and large scale integration techniques (e.g. read-only-memory and microprocessors) have provided enormous design flexibility in digital hardware design. These developments have had a profound impact on almost all scientific disciplines, and the field of digital signal processing benefitted greately from these developments. As a result, a number of systems have been proposed and constructed for the solution of real-time problems (10-13) in preference to employing the more conventional techniques involving general-purpose computers, the speed of which often necessitate off-line analysis. At present, systems of hardware for digital signal processing may be categorized into hard-wired (or special purpose) circuitry and programmable machines. The former may operate at high speed, and is usually dedicated to the implementation of a single algorithm; although limited programmable capability is sometimes incorperated. The other type usually operates at medium speed, is flexible and can implement a variety of complicated algorithms. With the advent of microprogramming techniques[1] and the increasing power of semi-conductor technology, hard-wired and programmable machines will be less distinguishable in the future.

Together with the expanding applications of digital signal processing made possible by the increasing rapid changes in semiconductor technology and processing algorithms, the number of techniques available and possibilities to be explored are speedily growing. Hence, a large number of implementation approaches have grown up. Enormous efforts have been expanded in cultivating various design metaphors for the implementation of digital processors; the field of digital filtering is a particular example of this.

The fundamental principles behind hardware implementations have already been widely considered and reported (5,7,10,13). This author shares the opinion with other designers, that there is a great need for the provision of aids for systematically deciding among the variety of possible implementation strategies in existence. A preliminary study (14,15) has been conducted

---

[1] "Firmware" has been used to reference this type of machines design approach.

by this author with a view to formulating a framework to aid the designer. It is one of the main objectives of this research to extend this work and device some precepts by which both experienced and less experienced designers may be assisted.

During the investigation, as reported in chapter 5 of reference 14, it was observed that filtering speeds had often been increased (as required for real time processing) at the expense of complex hardware architectures and pushing technologies to their limits. It has been the contention of this author that these difficulties are the consequences of adopting the "positional" number system (e.g. decimal, binary) to realise the computational algorithms required in digital filtering. The residue number system (RNS) (15,16) is a "non-positional" (or polydic) number system. As such, true parallel processing is inherent in the number system. In addition, it is possible to increase the word-length of the filter, without excessively decreasing the computational speed.[1] The drawback of RNS, however, is that it is an integer number system; therefore, the notion of "radix point" has no meaning. This resulted the difficulties of operations such as general division (where the dividend is not a multiple of the divisor), scaling, sign or magnitude determination and overflow detection. Despite these difficulties, which reduce the attractiveness of the RNS for general purpose machine, yet the RNS could still offer alternatives or advantages over conventional realizations of digital signal processing. Therefore, the other main objective of this research has been in considering the application of RNS (as an integer number system) on digital filtering. Mathematical algorithms for rational number operations, which are generally required in digital filtering, has also been considered.

Chronologically, the application of the residue number system to digital filtering was the main concern. However, the application is ultimately hardware oriented. This necessitated efforts being spent in devising methods to verify the correctness of any hardware proposed or designed. Consequently, the emphasis of this research has gradually been shifted towards the development of a systematic approach to designing digital signal processors.

---

1 see chapter 5 of this thesis

This has resulted the proposal & implementation of a computer-aided-design (CAD) approach to implement digital filters. Within this frame of reference, the studies of the application of RNS on digital signal processing is an exposition of the CAD approach.

1.2 SYNOPSIS -- STRUCTURE OF THE THESIS

The area of our research (section 1.1) is large -- embracing digital filtering, residue number systems, computer-aided design systems, software & hardware design, and computer languages; in particular, the use of APL. To avoid the danger of overshadowing the objectives of the research and giving a disjoint appearance, the thesis is organized into six chapters making reference to seventeen appendices. These appendices contain concepts, techniques, and results that are not possible to be adequately addressed in the main text.

Chapter one explains the background and research area. Objectives of the research are indicated, while the structure of the thesis is outlined.

Chapter two presents the attempts of providing an aid for digital system (in particular digital signal) designers. To this end, the activities of a designer are rationalized and this has resulted the development of an "Interactive Model". The Model in turn, has led to the proposal of a "Top-down Design Approach", the success of which rests heavily on computer analyses and simulations. The design and implementation of computer programs & packages for this purpose is therefore of paramount importance. In the light of this, the importance of devising suitable computer aids and the choice of a suitable computer language are treated at depth. APL (A Programming Language) has been found to be most suitable for the "Top-down Design Approach". The advantages and disadvantages of the language are considered. This is followed by a review of available APL processors and a discussion of their implications on the design of CAD systems. This chapter ends with a presentation of the CAD and the environment for the "Top-down Design Approach".

In providing subsystems and thereby enhancing the Top-down Design Approach two extensive packages have been developed — DISPIC's and SIMLOGS, which are respectively presented in chapters three and four. DISPIC's facilitate the design and simulation of programmable digital filters (and signal processors) at the functional and register-transfer levels. While SIMLOGS provides the tools for analysis and simulation of logical circuits or systems at the gate/logic level. Further possible developments for these packages are indicated in the pertinent chapters.

Within the framework of the Top-down Design Approach, the mathematical and algorithmic studies of the residue number system (RNS) are considered for implementing digital filters (chapter five). Hardware studies of such implementation, aided by computer tools (such as LOGMIN, appendix 2) developed, are considered. Results of these are presented in appendices 16 & 17. Significant progress in this area (implementation of RNS digital filters) by other researchers is briefly reviewed.

The results of the present research have laid a foundation for extended studies which could proceed in several directions. Chapter six gives a summary and concluding remarks on the present investigation. The chapter ends and concludes the thesis with a recommendation for possible research and related studies.

Chapter 2 A PROPOSED SYSTEMATIC APPROACH OF DESIGNING DIGITAL SIGNAL PROCESSORS [1]

Summary

Along with the growing interest in digital signal processing, there is an increasing demand for the provision of aids for systematically deciding among the variety of hardware design strategies. The chapter reports on an investigation conducted by this author. The objective of this investigation is an attempt in devising some precepts by which both experienced and less experienced designers may be assisted in this respect.

A digital system (e.g. digital signal processor) designer's activies are rationalized by an "Interactive Model" which in turn leads to the development of a "Top-down Design Approach". The success of this Approach, however, rests heavily on computer analysis and simulation; and the formulation, design & implementation of a good design software package(s) for this purpose is therefore of paramount importanee. To this end, the criteria of designing suitable computer aids and the bases for the evaluation and comparison among computer programming languages are treated at great length.

Among some popular and widely used general purpose and problem-oriented computer languages, APL (A Programming Language) has been found to be ideally suited for the "Top-down Design Approach". The advantages and disadvantages of this choice are discussed. This is followed by a review of available APL processors and a discussion of their implications on the design of Computer-aided design (CAD) systems. The chapter concludes with a presentation of the CAD and the environment for the "Top-down Design Approach".

---

[1] This chapter is based on this author's publication (17) and seminars given during his activities as a senior system consultant to a number of commercial organizations.

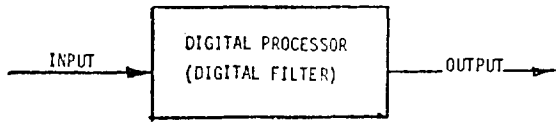
2.1 CONSIDERATION OF THE ACTIVITIES OF A DESIGNER : to rationalize his activities

The activities of a designer can be regarded as posessing two main aspects, namely the design level at which the designer is working and the conceptual processes he applies in solving his problems. The latter involves specifying the tasks to be achieved and the ways in which this determined-goal is to be attained. The design level on the other hand traces the increasingly detailed steps involved in executing the specified task(s) and reaching the intended goal. By subdividing the activities of the designer in this way, the design itself may then be subdivided into a series of smaller and more easily handled problems. These two design aspects will be discussed in turn and the way in which they interact with each other will be indicated. The discussion applies generally to the design of all digital processors or systems. However, to make our discussion more specific, we shall direct our attention to a class of digital signal processors; the linear, time-invariant digital filters.

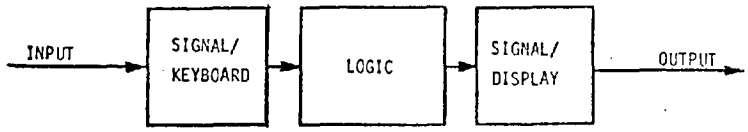
2.1.1 Levels of Design (15)

Five major levels of design can be classfied as : System, Functional, Register-Transfer, Gate/Logic and Circuit (including layout of modules/components) Levels. The meaning and associated activities at each level are best illustrated by an example; the design of a programmable filter (fig.1).

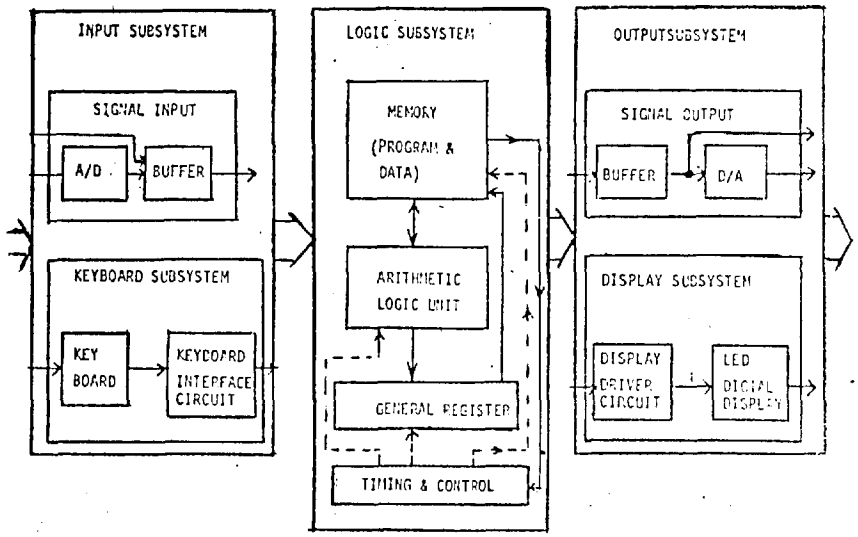
The complete system level description may be a black box as shown in figure 1a; while a more descriptive aspect of it (still at the system level) is that shown in figure 1b. In the example, the input consists of two parts; one of which is for signal input while the other is for manual entry of information via a keyboard. Likewise the output has one part for signal output and the other part for a visual and digital display. At this level, a designer is concerned with the overall flow of information in and out of the system and the amount of processing tasks he needs to incorperate into the system. The processing tasks (of the digital filter) will become clearer when we discuss the conceptual approach to design and its interaction with the levels of design.



(1a.) The Highest Level of the System

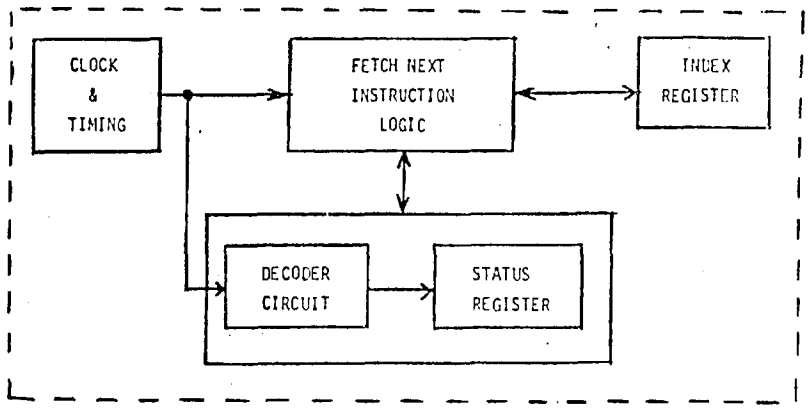


(1b.) System Level : A More Detailed Aspect of the System

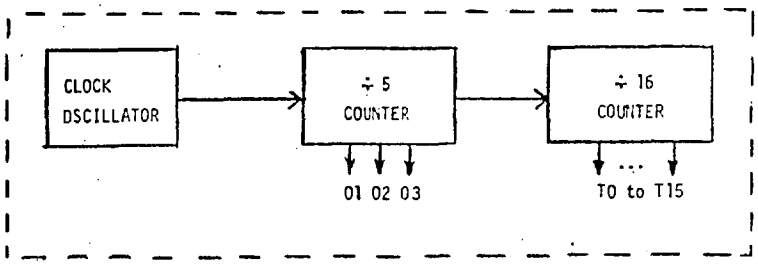


(1c.) Functional Level : Subsystems Decomposed into Units

Fig. 1 : A Digital Signal Processor System (Digital Filter)



(1d.) Register-Transfer Level : Division of Units(Timing & Control) into Subunits



(1e.) Gate/Logic Level : Components of a Sub-Unit (Clock & Timing)

At the next level, functional level, the system or its sub-systems are subdivided into several units (fig. 1c) -- the structure of the machine is beginning to emerge. At this stage, the sequence of operations within a subsystem and the flow of data between units and subsystems is determined. Further subdivision of a unit (fig. 1c) brings us to the register-transfer level (fig. 1d). Detailed internal operations of the sub-unit and the flow of data between sub-units and registers are now designed and described. Figure (1e) illustrates the further segmentation of a sub-unit that leads to the gate/logic level of design. Components (e.g. a divide-by-5 counter) of each sub-unit are realized by logical gates and flip-flops. Logic synthesis, minimization, fault analysis are conducted. At the next level, circuit level, a designer then considers electrical aspects of the gates and flip-flops; e.g., fan-in/out, layout considerations, etc..

2.1.2 Conceptual Approach

A digital filter (or system) may be formulated conceptually by considering : Mathematical, Algorithmic, Architectural and Technological aspects. In this section we shall enumerate the design parameters involved with each concept and in the next section (section 2.2), we shall discuss how these concepts interact with the levels of design.

A) Mathematical - The mathematical aspects involve the modelling of the system in terms of appropriate parameters (1-8,18); for example,

   1) Desired system (or filter) response :
      transfer functions or set of differenee equations;
      finite impulse response (FIR), infinite impulse response (IIR) and fast-fourier transform (FFT)

   2) Form of filter :
      recursive or non-recursive; direct, cascade, parallel, or wave digital filter

   3) Tolerance (qunatization effects) :
      coeffieient & data accuracy and roundoff errors, reduction of quantization by permuting sections for the cascade form

   4) Sensitivity/Stability :
      Limit cycle (small-scale) and overflow (large-scale)

B) Algorithmic - The algorithmic aspects are concerned with ways of executing the computations involved in realizing the system; for example,

   1) Procedure for execution of computations involved in realizing system response and filter structure

2) Methods of dealing with quantization effects due to finite word length

   a) coefficient accuracy
   b) data at input/output and within the machine
   c) roundoff error due to truncation or rounding
   d) overflow - detection and correction

3) Consideration of number properties
   a) number system - polyadic (positional), mixed radix, negative radix or non-polyadic (e.g. residue number system)
   b) digit representation - decimal, binary, binary-coded-decimal (BCD)
   c) value representation - sign and magnitude, radix complement, radix-minus-one complement or negative radix
   d) format - fixed point, floating points or block-floating point

4) Type of arithmetic : arithmetical operations associated with the consideration of a number system, etc. (as discussed in item 3 above)

5) Sampling rate - aliasing effects

6) Mode of operations - real-time or off-line

C) Architectural - The architectural aspects account for the hardware/software structure of the system; for example,

1) Type of machine operations - synchronous or asynchronous

2) System partitioning - modular or distributive, multiplexing, pipeling

3) Type or information or data transfer - serial, parallel, or pipelining

4) Interfacing - data word format, interconnection buses, operation speed mismatch, A/D and D/A conversion techniques and speed of operation

5) Processing & speed - structure and operation of an arithmetic unit, size and cycle time of memories

6) Programming facilities - design of assembly codes, construction and formulation of higher level of programming language

7) Control units - hardware of firmware (i.e. micro-programming)

D) Technological - The technological aspects focus on the physical realization of the system

1) Custom Approach - design of tailor-made chips to perform a specific application

2) Standard parts - choice of commercially available integrated circuit (IC), medium-scale-integration (MSI), large-scale-integration (LSI), or microprocessors

3) Choice of logic family - bipolar, unipolar; e.g. transistor-transistor logic (TTL), emitter-couple logic (ECL), integrated injection logic (I^2L); metal-oxide-semiconductor (MOS), complementary MOS (CMOS)

4) Use of subsystems - programmed logic arrays (PLA), micro-processors, microcomputer, or minicomputer

5) Types of test - the availability of test points to checkout a fabricated design, functional and component tests and test procedures

6) Constructional techniques - amount of modular card file packaging (IC packages, printed circuit board, subrack or card file, subrack-group, rack or cabinet, etc.), cabling technique (soldered or wire-wrapped), placement of components, generation of wire lists, etc.

7) Power-delay product (roughly constant) consideration - for a given family of logic, there is a trading between power dissipation and operating speed of circuit. Semiconductor manufacturers are continuing to produce new families of logic to give a lower power-delay product; e.g. low power Schottky TTL, ECL, and I^2L. It is equally important to consider power consumption of circuitry and the cooling required.

8) Maintenance - easy of maintenance by operating personnel, possibility of future extension

9) Reliability - estimation of failure (system or components), minimization of probability of failure, dectection or/and correction of failure

10) Cost effectiveness - the trade-off between speed and complexity, techniques of production and ease (including cost) of maintenance, marketing requirements, etc.

It should be observed that there exists no distinct demarcation between activities discussed above. However, the objective of considering the activities of the designer is an attempt to rationalise his activities with the view of devicing some aids to assist him. This will be considered in subsequent sections.

2.2 THE DESIGN PROCESS : AN INTERACTIVE MODEL AND SYSTEMATIC APPROACH

The interactions between the two aspects of design (sections 2.1.1 & 2.1.2) are best illustrated by the digital filter shown in figure 1 as an example. This filter may be conceived mathematically : to have some arbitrary FIR or IIR system response, to be realized in cascade form (consisting of second-order sections) and to have some desired tolerance (e.g. 1db in the passband, and 3 db in the stopband). To achieve these specifications, the designer begins the first level of design - the system level. The

transfer function associated with the desired FIR or IIR system responses may be obtained by several techniques (1-8). Without any immediate commitment to conceptual approaches (e.g. technology) or design details, the transfer functions may be evaluated and modified to satisfy the desired specification.[1] Algorithmic aspects of the design are next to be considered; particularly, the effects of finite word-length limitation, results of quantization effects and methods of overcoming or coping with these (3,5,7,) must be employed. The choice of a number system, type of arithmetic, sampling rate and mode of operations are also to be determined. A clear overall view of the system to be designed thus emerges.[2]

The next phase is to consider the architectural aspects of the system. The filter shown in figure 1, for example has been chosen to be programmable with its memory being partititoned into program and data stores. The overall information flow is serial, the system is modular and is hardwired controlled. Furthermore, although the information flow is serial between subsystems, yet it may be serial or parallel within each of these subsystems. At the input, the provisions for signal interface (which can be either digital or analogue) are specified and at the output, a digital/analogue signal interface is similarly chosen. The filter coefficient program is entered via a keyboard, with each program specifying a particular system response for the filter. A digital display provides an indication of the states of the registers or memory locations within the filter, the keyboard providing the selection mechanism. With this initial architectural specification, the filter may be functionally designed; and the result is the structure shown in figure 1c. The arithmetic unit may be arbitrarily specified; i.e. it may perform addition only or addition/multiplication. The detailed structure needs not at this

---

[1] This phase of the activity is refered to in the literature, as "digital filter design".

[2] This may be considered as the "realization of digital filters".

stage be precisely specified. When the functional design meets the architectural specification, one can enter a more detailed design of the functional blocks (units in figure 1c). This is the register transfer (RT) level of design. Very often, the architectural specification cannot be met and the original (i.e. architectural) concept then has to be modified and a further functional level of design (followed by another RT level design) be conducted. This process is repeated until architectural specifications are satisfied both at the functional and RT levels. At this stage, the design may still be technologically independent.

By introducing the technological concepts, detailed design of the system can then be conducted. This leads to logic synthesis and minimization of each part of the system (i.e. the component or sub-unit such as Figure 1e) and the analysis of logic hazards and faults are conducted at the gate/logic level design. If conventionally available modules are employed, the amount of design at the logic level is reduced. If on the other hand, a custom built approach is chosen, a circuit level design is essential; so that the circuit must be thoroughly analyzed before integrated circuits and chip desgins are committed to fabrication. In using MSI, LSI, programmed logic arrays (PLA) or microprocessors, a return to the RT level of design and analysis is sometimes necessary. This may even trigger off a repetition of the series of design processes discussed so far.[1]

The above discussion indicates a number of points :

1) Conceptual aspects are themselves inter-related;
2) The conceptual approach interacts with the level of design;
3) The design process is interactive;
4) The degree of details and amount of interactions increases as one progresses through the various levels and concepts;

---

[1] The "process" commencing from the completion of "digital filter realization" to attaining a digital filter is often referred to as "digital filter implementation".

5) It can be seen that at any particular level of the design, only certain concepts need to concern the designer, e.g. at the system level, only mathematical and algorithmic concepts need to be considered, architectural considerations being left till the next design level

A further comment (as a direct consequence of point 5) of the above discussion is that some form of aid is desirable for analysis, design and verification (via simulation, which will be further elaborated in section 2.3) at the various levels. The aid also enables a designer to enlist the experience and established methodologies of other experienced practioners. These considerations suggest the 'interactive model' (17) for a top-down[1] design approach (fig. 2), and the model now forms a framework within which an interactive and systematic approach to the design of hardware digital signal processors can be formulated.

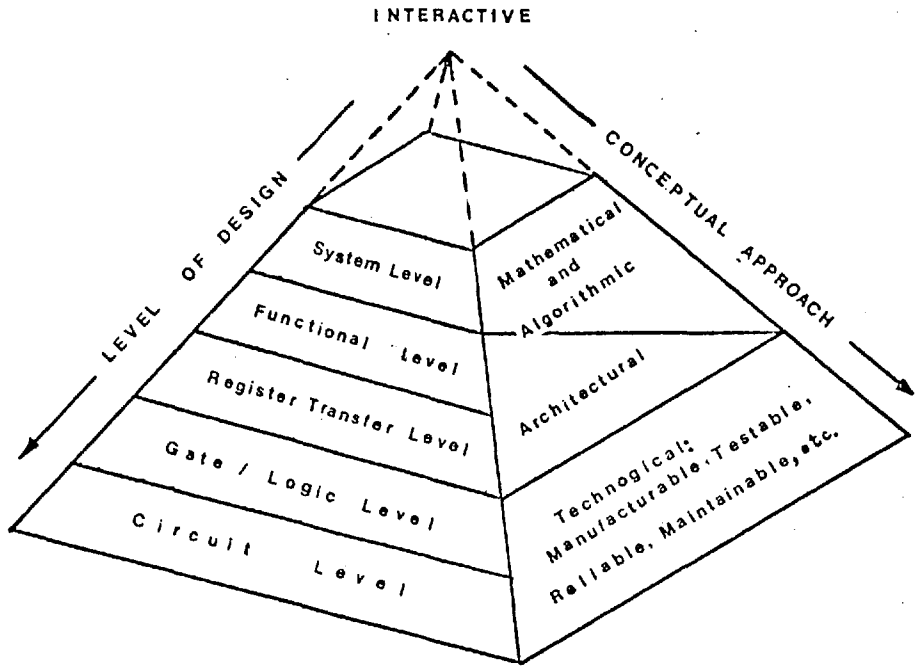


Fig. 2 : An "Interactive Model" for the "Top-down Design Approach" to Designing Digital Signal Processors

---

[1] "Top-down" is a term borrowed from the methodology of designing computer software. The design of digital signal processors is analogous to that of computer software; in which, one identifies the major functions to be accomplished, and then proceeds from there to an identification of the lesser functions that are derived from the major ones. Other synonymous terms are "systematic programming", "hierachical program design", and "design by explosion".

2.3 A PROPOSED TOP-DOWN DESIGN APPROACH

While figure 2 depicts the activities and their hierachical order (with increasing complexity of design), it however does not show the iterative process of the design. Figure 3 presents the nature of this process. Within this scheme, a design objective is first determined conceptually (section 2.1.1) which are shown in figure 2.

1) At each level, details (increasing in complexity) of the partial system and its associated hardware are then formulated, verified and evaluated (with computer-aids, as discussed in section 2.3.1)

2) Verification of the system refers to the activities of synthesis, analysis minimization and checking for mistakes in concepts and design details (see further comments in section 2.3.2). During each stage, a number of implementation possibilities may arise.

3) Evaluation refers to the examination of a current version of the design at a particular level against the design criteria.

4) During the design process the number of alternatives grows rapidly and optimal solutions are not generally unique. Within this section,[1] we shall discuss that this situation necessitates a decision on the part of the designer. This decision will undoubtedly draw heavily on past experience. Hence the additional concept is indicated in figure 3 (enclosed by dotted lines).

It should be noted that figure 3 is not in itself a flow-chart of a computer program or package[1]. However, the success of the proposed "Top-down Design Approach" relies heavily on the design and implementation of computer software packages. In the following subsections we shall discuss the implications of figure 3 and in subsequent sections the requirements of these computer packages.

---

[1] A package of this type implies an automatic design which is possible in principle. This will be discussed in section 2.3.3.

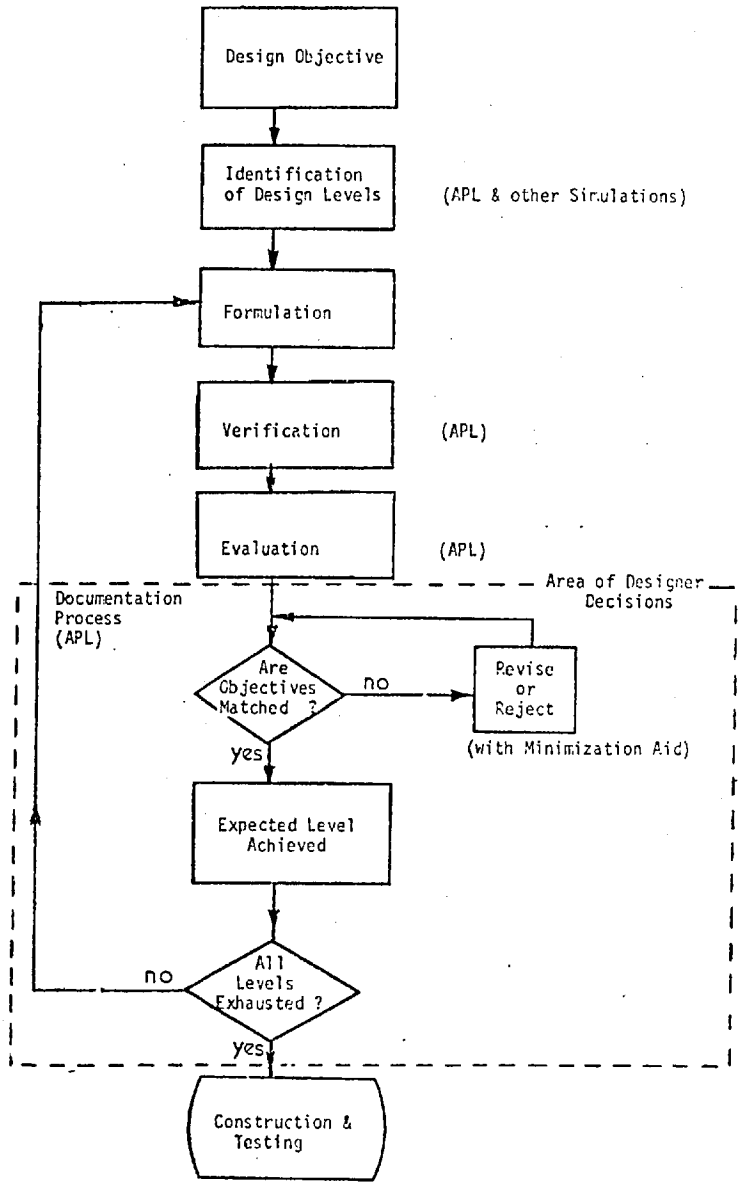


Fig. 3 : A Proposed "Top-down Design Approach"
(Interactive & Systematic Design Approach)

2.3.1 Description at various Levels : Difficulties and some unresolved areas

In order to formulate, verify and evaluate a digital system at vartious levels, it is necessary to describe the system at these levels. Numerous techniques (19-23) have been reported for this purpose; these methods may be generally classified into three basic areas, which are as follows.

1) Functional descriptive programming language - such as register-transfer languages (RTL), simulation languages (e.g. GPSS, SIMSCRIPT), and some general-purpose high level languages such as FORTRAN, ALGOL 60, PL/1 and APL.
2) Finite-state Machine (FSM) techniques - such as state tables, regular expression, flow-charts, etc.
3) Graph-theoretic methods - employing transition graphs, Petri nets, occurrence graphs, state diagrams, etc.

Apart from the objective of describing a complex digital systems, many of these methods have been constructed to develop an automatic design system (which are often computer oriented) the primary concerned of which is to reduce the cost & development time normally required in engineering complex digital systems. However, in general all the methods described to date fall short of the ideal methodology which will be discussed in section 2.5.1.

For instance, problem-oriented programming languages (such as simulation languages) often suffer from the inherent disadvantage that they have no formal mathematical structures. Consequently, system behaviour must be interpreted indirectly from a program performance whilst operating on certain specified data types (e.g. link list, tree & ring structures). Simulation languages in particular suffer from this disadvantages. A simulator language usually describes a logic system in terms of simulated components (that is gates, bistables, etc.) and their interconnections. In order to evaluate or analyze a logic network represented in this manner it is necessary to perform an exhaustive, sequential examination of all possible logic conditions; that is, the circuit model must be driven (by providing specified input conditions) and then "exercised" step by step to determine the resultant outputs. It will be obvious that this is a time-consuming process and that large amounts of storage will be required to represent the circuit model. Moreover, since a logical

system is described as a topological circuit network, rather than by specifying system functions, the simulation description is difficult to follow and of little use for documentation and communication purpose.

A useful comparison between formal and simulation methods may be drawn by constrasting the functional description of a binary counter circuit, using a state table showing the present and next-state transitions, as against a simulation-language description detailing bistables, logic gates, etc. and their interconnections. An important difference is that the formal specification describes a logical algorithm which could be implemented in either software (e.g. programming a microprocessor to implement logical operations) or hardware form. Problem-oriented design languages, because they are constrained to operate on well-defined data types, are normally restricted to hardware representation.[1] Note that procedure (or general purpose programming languages (e.g. FORTRAN, ALGOL, PL/1, APL) will still be employed even with formal methodologies, since the notational system must be represented in some way in order to input a description to the computer; in particular, when a design automation or a computer aided in design (CAD) is employed. The difference is, however, that the programming language is simply acting as a vehicle to describe the variables (represented as data structures) and to control the execution of the associated function operations; not as an end in itself —— for example, in a general-purpose programming language used to represent graphical structures and their manipulations.

The major problem to be overcome in any specification language for a CAD is how complex systems containing a very large number of binary variables may be represented and analyzed. In the general case the system variables will be both primary inputs and internal state conditions. Finite-state theory, though theoretically capable of representing any digital system (which must of course in its engineered form be deterministic) is not viable in practic owing to the considerable difficulties involved in expressing the problem and the inordinate amount of computation required for manipulating the resultant structures.

---

[1] However, in some cases (see chapter 4), it is often essential to use an implemented hardware version of the circuit; an example of this is the generation of test sequences for logic circuits.

There are also some more fundamental disadvantages of the FSM concept(23). In general the FSM accepts a serial input (or inputs) and progresses from state to state producing an output sequence in the process. Owing to its finite memory capacity (i.e. the number of internal states) the FSM is best suited to describing systems where the amount of memory required to record past events (that is the effect of earlier inputs) is small and finite, for example, serial adder-substractor circuits and pattern detectors where the computation can proceed as a step-by-step operation on the input and the amount of information required to be "carried" or "remembered" is very small (1 bit in the case of the adder-substractor).

Serial multiplication, however, is very different. In this case it is necessary to have all the input data available before the computation can proceed; moreover, large amounts of information need to be stored during the course of the multiplication (for instance, to accumulate partial sums). It follows logically from this constraint that it is impossible to specify a finite-state machine that can multiply together arbitrarily large pairs of numbers. Note also that the FSM does not have the ability to refer back to earlier inputs unless the entire input sequences is initially stored; this implies that the input sequence of interest is of known finite length.

These limitations can of course be overcome by using an infinite-machine model[1] where the amount of available storage is unlimited.[2] Using this type of model the designer is unconstrained in his thinking, allowing general system concepts to be specified without reference to a particular implementation. It is important to remember, however, that all machine models must eventually be implemented as a finite memory machine. Unfortunately the transformation from a conceptual infinite model to a

---

[1] derived from the "Turing machine" (23)

[2] Some of the more promising formal methods of specification, such as Petri nets, ect., are in essence infinite model.

practical realization as a finite-state machine can, and does, present serious difficulties.

Another inherent difficulty is encountered in the analysis of large systems. It would appear inevitable that, if a detailed analysis of a logic algorithm is required, there is no other choice but to examine all possible alternatives in an iterative manner. In general, particularly if an infinite-machine model is adopted, it is necessary to determine the system operation by specifying a restricted set of input and state conditions. This technique, which corresponds to examining particular paths through the algorithmic structure (for example, a particular state sequence in a state table) results in a loss of information and can affect the accuracy of the analysis. However, there seems no alternative but to adopt this method for complex systems.

In order to design complex systems, as we have seen in section 2.1, it is necessary to partition the system into suitable subsystem functions. This fact has been extensively used to alleviate some of the problems of implementing large systems. In particular the approach of dividing the system into a control function and a data structure has been widely adopted. Using this technique a data structure may be specified in terms of registers, processing units, etc. and their interconnections; the control algorithms, specifying the required processing operations and their timing, are then defined separately. Note that the data structure also includes processing operations and is not restricted to data storage. Thus, the complexity of the system is reduced by partitioning the principal components into comprehensible subsystems — the usual human heuristic for handling any complex situation (see section 2.3.3).

Despite much efforts have been directed to the area of system specification and many techniques have been proposed for this purpose, the problems of describing systems at various levels are not completely resolved. In fact, many of the techniques enumerate in the above are still in the development stages and that a viable system (an automatic design system, or CAD) meeting the requirements (which are outlined in section 2.5.1) remains to be developed. However, in the remainder of this chapter, we shall discuss our proposed design approach as an attempt to meet the requirements; with specific reference to the implementation of digital

filters. One of the purposes of specifying a digital system is to verify & evaluate it (an implication of figure 3); this is to be examined in the next two subsections (2.3.2 & 2.3.3).

2.3.2 Verification methods[1]

Verification establishes the equivalence between the implementation and the specification of the design. Since a design may be specified, there exist many methods of verification. These methods may be divided into static and dynamic methods (24).

1) Static Verification : Static verification is performed independent of the operands (i.e. inputs & outputs) of the system. This verification may be obtained by automatic translation from the problem statement to its solution, or by algebraic verification.

A) Automatic Generation of the Implementation - With an automatic generation of the implementation, the burden of proof is shifted from the correctness of the solution to the correctness of the translation program. This method is satisfactory between implementation levels, for instance, in transforming a general implementation description into a specific one (fig. 4).[2] However, the transition from an architecture (i.e. specification of a design) to an implementation usually requires some imagination and invention of a designer. Hence, in general, an automatic translation between these two design domains is not possible.

---

[1] Verification has much influence on the "specification" of digital systems. This topic was touched upon when formal and simulation methodologies were discussed in section 2.3.1

[2] APL (A Programming Language) is used for the implementation description; this will be explained in section 2.5

```
     ∇ RIPPLE;J
[1]    ⍝ TRANSMISSION AND GENERATION
[2]    T←A≠B
[3]    G←A∧B
[4]    ⍝ CARRY PROPAGATION
[5]    J←⍴T
[6]    C←(J+1)⍴0
[7]    C[J]←CIN
[8]   CONT: J←J-1
[9]    C[J]←G[J]∨T[J]∧C[J+1]
[10]   →(J≠0)/CONT
[11]   ⍝ CARRY ASSIMILATION
[12]   S←T≠1↓C
[13]   COUT←1↑C
     ∇
```

(a) A general implementation description : a ripple adder

```
     ∇ RIPPLE4
[1]    ⍝ TRANSMISSION AND GENERATION
[2]    T←A≠B
[3]    G←A∧B
[4]    ⍝ CARRY PROPAGATION
[5]    C←5⍴0
[6]    C[4]←CIN
[7]    C[3]←G[3]∨T[3]∧C[4]
[8]    C[2]←G[2]∨T[2]∧C[3]
[9]    C[1]←G[1]∨T[1]∧C[2]
[10]   C[0]←G[0]∨T[0]∧C[1]
[11]   ⍝ CARRY ASSIMILATION
[12]   S←T≠1↓C
[13]   COUT←1↑C
     ∇
```

(b) A specific description : 4-bit ripple adder

Fig. 4 : Example of an "automatic generation of the implementation" — between levels of implementation

B) Algebraic Verification - Algebraic verification of the equivalence of some expressions is possible in principle (e.g. APL permits classical algebraic manipulations - ref. 25). This has the advantage that usually the equivalence is established via another route than was used in the original design. In practice, however, the designer may be inclined to distrust his algebraic manipulations as much as he distrusts his design. This objection also holds for the verifications of the equivalence of programs by such methods as the assertion method of Floyd (26).

2) Dynamic Verification : Dynamic verification uses the operands (i.e. inputs & outputs) of the system. It usually simulates the design for a set of test cases. Where static verification, when applicable, is complete, dynamic verification normally is not.

A) Simulation - An algorithmic implementation description is a major help in verifying the correctness of a design. Verification may be exhaustive or selective.

i) Exhaustive verification : For small parts of a design, it may even be possible to check the system (or circuit) exhaustively for all possible input values.

ii) Selective verification : For larger parts of a design the exhaustive test, as a rule, is too time-consuming. Several well-chosen test cases, however, may provide much assurance; even though absolute certainty is not achieved.

iii) Selection of Test Cases : The designer can prepare several test series by constructing the corresponding matrices.[1] It is worthwhile to develop a proper set of tests, since these tests may also be used in later stages of the design; even in the physical test of a prototype.

---

[1] Resembles the "decision table" which are used to depict the function of a software program/package in computer programming (27)

Candidates for inclusion in a test set are :

(a) Extreme cases, such as all 0s and all 1s operands.

(b) Critical cases, which test the critical path.

(c) Branch cases, which test both sides of a branch condition.

(d) Boundary cases, which test both sides of the boundary of the operating domain, such as between overflow and no overflow.

(e) Circuit cases, which test the functional operation of individual components, assuming that their enviroment is faultless.

(f) Diagnostic cases, which give easily recognizable intermediate and final results, thus helping in finding the cause of a malfunction.

Several criteria can often be combined in one case. The order of the tests should be such that some diagnostic cases are used first and a rough overall verification is quickly established.

From the above discussion, we observe that static verification when applicable is more complete than dynamic verification. The lack of suitable formal theory (e.g. switch theory) and the complexity of most digital system have, in general, necessitate the use of simulation either in software or hardware (the construction of a prototype). This reinforces the concept of our proposed "Top-down Design Approach". In the next section, we shall discuss the heuristic aspect of this approach.

2.3.3 Heuristic approaches

Figure 2 summarises the activities (and their interactions) of a design progress; namely the conceptual approach and the levels of design. While figure 3 depicts the interactive and iterative top-down design process. The design of a system and its implementation may be considered as a translation process (section 2.3.2) consisting of the application of

transformation and expansion steps. The specification of a design, as for example, the description of an architecture, stipulates the functions to be performed by the system. This leads to the implementation description (after successive applications of transformation and expansion steps) which indicates how the system may be constructed. Formal languages (e.g. procedure languages) may be used to describe the system at each of the levels depicted in figure 2.

As remarked in section 2.3.2, general formal methods have not been developed for the transformations and expansion algorithms which could be implemented as part of a computer aided design system (i.e. automatic generation of implementation). Therefore, some authors[1] deny the possibility of an automatic generation of implementations (24), while other authors claim that their design systems include a module that performs exactly the expansion of a behavioral into a structural description;[2] thus giving rise to a seeming contradition. A closer look at these latter systems (28), however, reveals that such an expansion is performed on the basis of a small number of standard solutions which provide very little variability for optimization. Technical constraints given by the available logic systems are often difficult to be considered and, as a result, such an automated design system may yield a solution of some sort but need not be the desired optimized solution. The problem is further complicated by the fact that the current state of digital technology does not provide a uniform level for the detailed structural design. Certain standard components (e.g. Arithmetic Logic Units, multiplexers, etc.) are given as MSI devices. Other, more specialized networks are not available as commercially available (off-the-shelf)

---

[1] Typically those who design systems for the industrial environment

[2] In the literature of design automation, "behavioral (information flow) description" treats a system as an interconnection of functional modules specified by their input-output (transfer function) characteristics. While "structural (implementation) representation" describes in detail the actual gates, bistables, MSI chips, software procedures, etc., used to realize the subsystem function physically. Behavioral corresponds to our "system level" while "structural" corresponds to the "register-transfer", "gate" and "circuit" levels.

components and thus must still be designed at the gate level (e.g. LOGMIN - Appendix 2).

Our "Top-down Design Approach" is an attempt to resolve the above problems. The interactive design system allows the user to apply an iterative top-down procedure of step-wise refinements, capitalizing on his cognitive faculties, decision-making abilities, and design experience.[1] The task of the program modules of such an interactive design system is then to provide the user, in each stage of the design process, with the tools required for testing and verifying the solution decided upon. As the major tool for such an interactive design procedure is simulation, a practical system must provide the possibility of performing simulations at each level of the top-down process (these will be discussed in subsequent chapters of this thesis). To this end, the hardware description language (HDL) on which such a design system is based should allow at any given level the behavioral or structural description to be used as the source program for a simulation -- either by direct execution or, wherever this is feasible, by automatic translation into an executable simulation program (since automatic translation at the implementation levels is possible, section 2.3.2). The heuristic aspect of the design process is a feature of the "Top-down Design Approach"; this accounts for the dotted area shown in figure 3.

2.3.4 Epilogue

In this section (2.3), we have considered the activities of a designer with the view of formulating a framework for a systematic design approach. To this end, we have attempted to model (fig. 2) a designer's activities

---

[1] While the less experience designer may benefit from the results of the more experienced whom have contributed to the developments of the packages which form the "Top-down Design Approach"

and this model in turn has led to the conception of a proposed "Top-down Design Approach". The implications of it have been discussed at length. The approach relies heavily on computer simulations and the developments of computer software packages. It would appear that the problems of designing digital systems (e.g. digital signal processors) have been shifted towards those of software systems design; yet this is inevitable.

Existing design techniques are still inadequate to cope with the ever-increasing requirements of present day digital systems; reaching a stage where the sheer physical and intellectual complexity are encroaching upon barriers which impede further progress. The phase "complexity barrier" was coined by Glazer (29) as early as 1969 to describe this situation, which unfortunately is even more apparant at this time of writing. Currently, digital engineering is still in the main performed intuitively, with the system being designed, generally from the lowest level upwards, using MSI and LSI logic subsystems. Though a partitioning of the system can be achieved in this manner, a true hierarchical structure is not necessarily obtained. This demands that the partitioning is performed taking into consideration the specified relationships that are required between the subsystem components. This can only be achieved by designing from the top-down, that is from a system and functional specification to actual operational components. Note that there are a number of differing criteria (section 2.2) that may be used to partition the system, for example, logical operation, ease of manufacture (e.g. size of board, number of pins, etc.), diagnostic testing, use of existing subsystems, hardware-software implementation, etc. Ideally it should be possible to partition the system in a number of different ways attempting to optimize globally about certain system-design parameters, for instance trade-offs between development and tanufacturing costs and reliability and maintainability. Unfortunately, as repeatedly remarked in the above, at the present time the lack of suitable design aids normally means that an inspired guess as to the correct partitioning must be made at the initial design stages. Once the design has been commenced it is then far too time consuming and costly to attempt to re-configure the system. This "do or die" method of warking also applies to a limited extent at the lower levels of subsystem design.

Thus, we have a situation where complex systems are being developed without any fundamental basis of design, except that of hard-won experience. It is essential to partition the system to handle the complexity involved. However, the partitioning should be done in a logical manner so as to achieve a hierarchical structure capable of providing a viable engineering design; to date no technique or methodology exists which provides the digital engineer with the necessary tools to perform this function.

The use of CAD (as advocated in our approach) techniques can improve the situation by system-level simulation, automated designs (when applicable) and testing procedures, etc.; but until a suitable means of representing and evaluating system is developed,[1] no real improvements will be obtained. This is not to say that our proposed approach has adequately resolved all the problems; however, it does at least point towards a right direction. To conclude this section, it is appropriate to review the advantages of the proposed approach as follows.

1) The ability to clarify concepts and design specifications at an early stage without any real commitment to one particular concept or the detailed design.

2) The ease of testing new ideas and exploring other less conventional techniques, so that an optimum or near optimal design may be determined.

3) The provision of means for testing hardware either before or during the construction of a prototype and the possibility of extensive debugging of hardware prior to its construction.

4) The facility of continuous, concurrent documentation of the system and hardware resulting in any future modification or improvements being easily made.

5) The availability or more time to spend on the creative aspect of a design and the reduction of effort spent in the inevitable routine work such as documentation, constructing hardware, testing ideas, etc.

6) It provides a useful aid[2] for both experienced and less experienced designers; and may even serve as a learning tool for beginners.

---

[1] There is a sad lack of methematical formalism in this area.

[2] For example, a tool for exploration into less than conventional techniques (e.g. the use of residue number system for digital filtering).

In the remaining sections of this chapter, we shall consider the aspects of designing suitable computer aids (CAD) and the choice of a suitable computer language for our proposed design appraoch.

2.4 THE IMPORTANCE OF DESIGNING SUITABLE COMPUTER AIDS[1]

In concluding section 2.3 we have seen that the problems of designing digital systems are apparantly, but inevitably, shifted towards those of software system design. However, there are correspondences between hardware and software. A system is a collection of programs while a program is a formal statement of an algorithm or a process. Furthermore, programs that comprise a system may be implemted in various gradations of hardware & software. It is therefore not surprising to note that techniques (e.g. structured programming, programming debugging techniques, modular design) developed in one area, say software, may be applicable to the other. What distinguish computer programs designed for computer aids from those for "system softwares"[2] are the intents and the applications of the programs. Irrespective of these viewpoints, the criteria and methods of program design are often identical. On the other hand, the utilization of programs has much influence on the choice of a formal language and the implementaion of the programs. For example, the purpose of a CAD is to assist a designer; hence the aspect of human factor is of great importance. To provide some insight to the selection of design techniques and formal languages, we shall attempt to characterise and discuss computer software application areas (section 2.4.1). This is followed by the consideration of the design approaches to meet the requirements of these areas and users (section 2.4.2). These considerations preamble the discussion (section 2.5) of the choice of a suitable computer language for our design appraoch.

---

[1] Materials for section 2.4 & 2.5 are extracted from a seminar given by this author at Bell Telephone Laboratories, Holmdel, U.S.A.

[2] i.e. the software that comprise a system, as indicated above.

2.4.1 The Classifications of Computer Software Applications

The applications of computer software may be characterised as: Scientific, Data Processing, Text Processing, and Systems Programming. This classification, however, does not suggest that every application falls neatly into one of these four classes. In fact, many applications have elements that would put them into two or more of the four classes. Further, some applications may fall partly or totally outside all of these classes. Nevertheless, these four classes serve as a frame of reference for discussing programming applications, since they are reasonably definitive and comprehensive.

1) Scientific applications : Programs that fall into the scientific application area can be characterized into the following way. First, their predominant activity is the manipulation of numbers and arrays of numbers. These numerical methods encompass such problems as statistical tabulation, linear programming, regression analysis, curve-fitting, numerical differentiation and integration, and the solution of simultaneous linear equations. Second, the volume of data involved in such problems is often relatively small and usually has a simple structure, such as that of a one-or-two dimensional array. Thus, the data verification problem in these applications is usually insignificant. Third, the algorithms found in these applications are usually complex and strongly based in methematical foundations. Thus, programmers must be well aware of the underlying mathematics (notably, vector and matrix algebra as well as error analysis). They should anticipate in their programs those data combinations that will adversely affect the correctness of the solution scheme. This complexity usually implies the execution time of a program will be dominated by calculations rather than input-output operations. This characteristic of a program is often referred as "CPU-bound".

2) Data Processing Applications : Programs that fall within the data processing application area can be characterized in the following way. First, their predominant activities are the manipulation of files and the periodic generation of reports from these files. These activities typically

deal with the administrative support functions of an organization, such as payroll, accounting, billing, inventory control, production scheduling, and order entry. Second, the data which is processed by these programs is generally large in volume. It is strictly organized by files, records, and fields, and has a high level of content change within a short period of time. Third, the programs themselves are primarily concerned with the effective representation, checking, transfer, merging, extraction, sorting, and summarizing of this data on a prescheduled basis -- daily, weekly, or monthly. These data manipulations are often intricate and require careful program organization and coding. However, such programs are usually bound in their execution time by the speed with which a file can be read or written, rather than the speed with which the computations can be performed. Therefore, they are often referred to as "input-output-bound", rather than "CPU-bound".

Three additional factors have a significant influence on the activities of a data processing programmer. They are program maintenance, "one-shot" jobs, and data recording.

Data processing applications of any appreciable size (e.g. a payroll system) often require a worker-year or more for development and implementation. However, after such a system of programs is implemented, it is continually subject to modification because of unpredicatable outside events (e.g. a change in company policy on the overtime rate for certain employees). This is the essence of program maintenance.

One-shot jobs occur as a result of a special request by management for information which does not occur among the normal day-to-day reports produced by existing programs. For example, the personnel department of a company may require a listing of all employees who will have been continuously employed by the company for twenty five years or more as of the end of the current year. Such a request usually comes within a week or two of the date when the information is required. Thus, the normal "systems approach" to the task is bypassed in favour of a more expedient and probably less elegant program that gets the job done quickly.

Since a high volume of data is recorded continuously in a computer-readable form (e.g. punched cards or magnetic tape), the

programmer must design record formats for data processing applications in a way that makes recording as expedient and error-free as possible. Such a design often places extra burdens on the program which edits the data. The data definition and verification problem in data processing is of immeasurable importance.

3) Text Processing Applications : Programs which fall into the text processing application area deal predominantly with data in the form of free-running text. Usually that data is in English and is either completely unstructured or minimally structured, such as the entries in an annotated bibiography. Many of the programming techniques used to identify, check, transform, and print such data are distinct from those used by the scientific or data processing programmer. Yet, the use of sound text processing techniques has become increasingly important in computer applications -- so important that its treatment as a separate application area is warranted.

4) System Programming Applications : Programs which fall into the systems programming area are those which are used to expedite the interface between the programmer and the computer. Typical system programs include the routines composing an operating system, the assemblers and compilers which translate programs into machine language, and various utility programs which perform common file-processing tasks. System programs are generally the most frequently executed programs in a computer installation. They must therefore be designed and implemented in such a way that their demands on system resources, notably time and space, are minimized. This rather stringent requirement, coupled with the greater efficiency of machine-level languages relative to high-level languages in computer resource utilization, has forced most system programming to be performed in machine-level languages.

2.4.2 Design approaches to meet the requirements of softwares and user needs (with special reference to CAD)

The design procedure of software[1] parallels that of digital systems design. The procedure consists of seven steps:

1) problem definition,
2) problem partitioning,
3) algorithm development for each partition,
4) writing the program for each partition,
5) debugging each program,
6) integrating the programs back into the system,
7) final system debug.

Numerous design approaches(30-38) have been proposed to achieve the above steps. Of these the most wellknown methods are: structured program design[2], decision table methods, top-down design approach, and modular design. All of these approaches have been developed with the view of improving program quality,[3] increasing program reliability, reducing the problems of program maintenance, facilitating a concise & precise communication medium among programming team-members (and users) and thereby speeding the process of program development. Hence the methods predominately emphasize on meeting the specifications or requirements of software.

---

[1] the term "software" assumes the general meaning as that indicated at the beginning of section 2.4; i.e. a collection(s) of programs that comprise a system.

[2] Examples of commercial design methods are the IBM's HIPO (36). Michael Jackson's Design Methodology (37,38).

[3] The goodness of a program is difficult to define; for this we appeal to an excellent study by G. Weinberg, entitled The Psychology of Computer Programming (31).

Software specifications, however, often reflect the importance and the need of accounting for the man-machine interface or man-machine dialogue of a computer system; CAD is a typical example of this. Experience to date[1] has shown that, unless CAD systems are easy to use, exploitation of the system by designers rapidly diminishes after the initial novelty value has passed and the difficulties involved in using the system become all too apparent. It is essential that any theoretical concepts employed in a CAD scheme should, if possible, be invisible to the user. The input-output transfer characteristic of the system must relate as far as possible to accepted design practice, and the operating procedure should be simple and efficient to use.

There is, however, an alternative school of thought which holds the view that, in order to provide the necessary design tools for complex logic systems, designers must inevitably learn new techniques. It is this author's opinion that any realistic solution must of course be a compromise; but it is prudent to realize that novel techniques are only acceptable if the advantages to be accrued far outweigh any difficulties encountered in accommodating the new disciplines.

Advances in the computer industry have enabled CAD systems to be implemented as interactive systems rather than those based entirely on batch-processing. This extended facility allows a digital system designer to rapidly modify the parameters of the design in accordance with a previous trial "computer run(s)". The object of designing an interactive CAD system is therefore to establish a direct communication between the man (the digital system designer) and the machine (a computer or computer-aided system) in such a way that their inherent incompatibilities are matched as efficiently as possible. The basic requirements of an interactive CAD system are as follows.

---

[1] A personal experience and observations of this author gained during his activities as a system analyst and a system consultant to a number of commercial organizations and the design & implementation of a number of computer systems. This view is shared by the CAD group of the Communication Section, Electrical Engineering Department at Imperial College; as much research is activity pursued into the man-machine dialogue of CAD systems.

1) A problem-orientated conversational language which is easy to use and understand.
2) Fast feedback of results and operational messages so that the designer's imaginative and creative abilities are not inhibited.
3) A satisfactory compromise must be made between the vastly differing volumes of data obtainable from man and machine in order to achieve a maximum mutual comprehension.
4) Input devices should require the minimal amount of mutual intervention, and output data must be suitably displayed and capable of being edited as required.
5) Automatic error detection and guidance through the operating procedures.
6) The technicalities of the software and design procedures should be "invisible" to the user.

These principles should always be borne in mind when designing the input (& output) procedure for a CAD package. It is interesting to observe that user-behaviours of scientific and technology users differ greatly from those of business and commercial users. For the latter users group, passwords are often requested by an application system when sensitive data is being accessed; while "idiot proof" procedures (such as long & short prompts and error recovery procedures) are necessarily be designed into the man-machine dialogue to guide inexperience or occasional users. However, the feature of an "expert mode" (e.g. abbreviated prompts and multiple-inputs, see figure 5) is incorporated to avoid boring experienced users. For the scientific applications, however, measures are taken to prevent inquisitive users from modifying the computer system so that erroneous or unintended system operations for subsequent users may be avoided. A common technique is to terminate a computer session when the application system detects a "user sabortage" and initiates appropriate measures before any fatal effect has taken place. Instruction from an application system is often comparatively simpler and most application packages are structured in a "skeletal form"[1] so that a personal package may be constructed even for a "one-off" application. Example of such a package is the MARTHA(39), an APL electric circuit analysis package.

---

[1] i.e. programs or collection of programs are loosely grouped into modules form which a user may construct his own application package.

```
NAME : JONES, M.
LOCATION : MAYFAIR
TITLE : D
  $$ VALID TITLES ARE -- MR,MS,MISS,DR,REV,SIR,PROF
TITLE : DR
JOB : MANAGER
          .... etc. ....
```

(A) Dialogue for an occassional user

```
NAME : JONES, M./ MAYFAIR/ D/ MANAGER
  $$ VALID TITLES ARE -- MR,MS,MISS,DR,REV,SIR,PROF
TITLE : DR/ MANAGER
          .... etc.  ....
```

(B) An Expert-mode dialogue (multiple-input facility)

NOTE : Observe the capability of data validation; when an erroneous answer is detected, all answers from the first erroneous position are ignored. The system issues an error report and re-issues an appropriate prompt.

Fig. 5 : An example of a Man-Machine Dialogue illustrating the occassional-user & expert modes

Even though engineering and scientific applications are predominantely "CPU-bound", yet care should still be exercised to ensure that the designer is not frustrated by input procedures which demand excessive typing or graphical maneuvers on a cathode-ray tube (CRT) display. Where possible input requirements should be kept to a minimum. This philosophy also applies to output data; facilities should be provided to allow the output (or irrelevant sections) to be suppressed when not required.

In addition to the use of problem-orientated languages (e.g. SIMSCRIPT, GPSS), it is also necessary to provide a conversational system command language (see chapters 3 & 4) to control the organizational and "house-keeping" functions associated with the CAD software packages. Various time-sharing commands must be provided (assuming multi-user operation); typical commands are as follows.

1) Logging in and out of the system;
2) Creating, deleting, and manipulating data files;
3) Accessing CAD packages (including the problem-oriented languages) and sequencing their operation;
4) Determining current system status;
5) Controlling input-output devices.

Many of these functions can be provided by standard operating systems, where they exist; otherwise it is necessary to implement them as part of the total CAD system.

The choice of input-output devices is also critical, since this can often be the determining factor in achieving an effective CAD system. The principal input devices used in CAD systems are the teletypewriter and CRT displays (40). In particular, interactive graphic displays, which can be used for both input and output operations, have great advantages when performing design processes requiring the monitoring and editing of topographical systems such as circuit layout, routing and placement, etc. Manual intervention can be provided by the use of light-pens, tracker-balls, and simple cursor-following devices in conjunction with a simple function keyboard, However, the cost of graphic terminals can be substantial, and it is a better practice to develop systems using teletype terminals (which are more freely available) unless the problem is such that graphic displays are essential. It is a bad practice to use graphic terminals to input basic data, such as logic circuit configuration. This

operation can best be done off-line by using a digitizing system based on a movable cursor which is manually tracked round a circuit diagram (drawn on a grid paper) generating digital XY coordinates (on a paper tape or cards) at operator-specified places. In many cases when a graphical output is required standard peripheral equipments, such as line-printers and incremental XY plotters etc., can be employed.

To summarize, the design of a CAD system involves the solution of three basic problems:

1) The selection and implementation of a suitable data structure for the representation of system models and design parameters.

2) The development of design algorithms which operate on the data structure, together with an appropriate design methodology.

3) The provision of a viable man-machine interface, with realistic interactive problem specification and editing facilities.

The choice of a suitable computer programming language facilitates the the solution of these problems and thereby attaining a satisfactory CAD system; this will be the topic for the next section.

2.5 THE CHOICE OF A SUITABLE COMPUTER PROGRAMMING LANGUAGE

2.5.1 Bases for the Evaluation and Comparison among Computer Programming Languages

The selection of a computer programming language is more an art than an exact science; very often the choice may be a personal and emotional one. Nevertheless there are still guidelines to assist us in the selection process. For example, if an application requires an appreciable amount of character manipulation, the language chosen must necessarily contain at least a minimum amount of string processing facilities. Secondly, an effective language choice may depend on the quality of its available compilers. If the applications for which a language will be used requires an appreciable amount of debugging and/or program maintenance activity, the language selector must consider the varying quality of support (for debugging and maintenance) that are provided by various compilers or interpreters. Thirdly, an effective language choice may depend on the relative efficiencies of different implementations of the language. By "efficiency" we mean to denote the following three elements:

Compile/Interpreter speed
Object program size
Execution speed

Different installations view these three elements of "efficiency" with varying levels of importance. For instance, a highly development-oriented installation may view compile speed more critically than the others, while a production-oriented installation may consider it as the least critical among the three. On the other hand, an installation that enjoys an abundance of excess computer time may consider all three elements of efficiency to be less important than the functional capabilities (of the language) in support of the programming, debugging, and maintenance activities.

Hence the problem of language evaluation cannot be solved simply by measuring its efficiency. Indeed, the value of a language depends at least as much upon how well its programming features support the writing,

debugging and maintenance of programs. It should be noted that the above three elements of efficiency can be quantified, while the effectiveness of a language cannot. That is, we can quantitatively measure the compilation speed, object program size, and execution speed. However, we cannot assign a numerical value to the quality with which the features of a language support, for example, character manipulation. Secondly, the effectiveness of a language is a relative one. It is relative to the particular application area and installation where the language is used. It is also relative to the particular compiler or interpreter under which the programs are run. Finally, it is relative to the particular machine and operating system upon which the programs are executed. This accounts for the use of "benchmarks"[1] to evaluate the effectiveness of a language.

Therefore, in comparing the effectiveness of two languages in a particular application area using a particular benchmark problem, the validity of the result will depend on how well the comparison is controlled. For this purpose, it is helpful to categories the "factors" that will guide our estimates of language effectiveness. These factors are indicated below.

1) Programming features :
   a) overall writing - compactness, uniformity, locality, and segmentation;
   b) overall reading - structural clarity and local clarity;
   c) application dependent functional support.

2) Implementation dependent features :
   a) compile-time diagnostics;
   b) execution-time diagnostics;
   c) other debugging facilities.

---

1 "Benchmarks" are computer programs designed to represent the user's primary workloads and are widely accepted as evaluation tests of computer installation and language implementations.

3) Efficiency :

a) compile speed;
b) execution speed;
c) execution storage requirements.

An evaluation of the programming features of a language determines how well it supports the writing and reading of programs in a given application area. An evaluation of its implementation dependent features determines how well a particular implementation of the language supports the debugging and maintenance of programs. While an evaluation of its efficiency determines how well the compilation and execution of programs utilize computer resources for a given implementation of the language.

The implementation dependent features of a language and efficiency are relatively easy to pinpoint. To judge how well an implementation supports debugging and program maintenance, we can evaluate the quality of its compile-time diagnostics, execution-time diagnostics, and other debugging facilities (such as run-time trace, programmable interrupt trapping, etc.). To judge the efficiency of an implementation, we can measure its compile speed (in seconds of CPU time), execution speed (in seconds of CPU time), and execution storage requirements (in thousands of bits or bytes). However, programming features are more difficult to localised; for this, we invoke a modification of G. Weinberg's studies (31). Weinberg identified and illustrated several qualitative characteristics that might be used to characterize the ease of overall reading and writing of a language.

The four main language characteristics that we may use to determine the ease of overall writing of a language in an application area are compactness, uniformity, locality and segmentation. The two main characteristics that determine the ease of overall reading (of a language) are structural clarity and local clarity. The third characteristic that influences the programming quality of a language is its built-in functional support. This is not to suggest that our characterisations are absolute and comprehensive standards for evaluating the programming features of a language. However, in the absence of any other viable bases, at this time of writing, for performing programming-language evaluation we ask that these characteristics to be accepted as reasonable and workable. We shall discuss these characteristics individually below.

Compactness : means that the language permits writing each element of a program so that a reasonably small number of symbols are used. Positive influences on the compactness of a language are such features as the ability to abbreviate words and expressions, default typing of variables, and the elimination of all program parts that are not essential in the description of the algorithm. Negative influences are the inability to abbreviate, the absence of defaults, and the required presence of program parts that have no influence on the algorithm being represented.

Uniformity : means that the language permits any particular action to be expressed in the same way, regardless of the context in which the action is specified, and with a minimum of "context-sensitive" restrictions. Positive influences on the uniformity of a language are such features as a simple and consistent rule for leaving embedded blanks within the program text, the absence of restrictions on the form of arithmetic expressions when they are used as subscripts, and a uniqueness of semantics for each single operational symbol (e.g., "="). Negative influences are unreasonable rules that deny the use of "reserved words" for ordinary variable names, different rules for blanks in different contexts, and dual meanings for an operational symbol.

Locality : means that the ability to write programs in such a way that the programmer can "localize" all aspects of a logical unit within the algorithm in a single textual position within the program. In other words, the programmer can structure the text to express an individual logical unit in a single writing space, and at the same time make visible the higher logical structure which contains that unit. Positive influence on locality are the ability to localize the "scope" of a variable by embedding its declaration within a "subblock" of code within the program and the ability to employ a "structured programming" style in writing a program. Negative influences on locality are a rigid, card-oriented syntax, the absence of dynamic storage allocation, and the forced separation of variable declarations from the text where they are referenced.

Segmentation : is locality in the larger sense. That is the ability to subdivide a large program into a number of subprograms, and the concurrent ability to pass information from one to another. Positive influences on segmentation are the ability to both define and activate subprograms, as well as reasonable and general provisions for designating values to be passed to and from subprograms when they are invoked. Negative influences on segmentation are rigid conventions for defining, invoking and/or passing values to a subprograms.

Local clarity : refers to the ability of an individual program statement to clearly convey its meaning when it is read.

Structural clarity : indicates the ability of an entire logical unit within the program to graphically display its semantic structure when the program is read. Structural clarity means the same to the reader of a program as locality does to the writer of the program.

REMARKS

When a language does not possess one or more of the four programming features (i.e. compactness, uniformity, locality, & segmentation), it tends to distract the programmer by imposing cumbersome, superfluous, and/or arbitrary rules of writing. On the other hand, when a language does sufficiently possess these four charateristics, it tends to expedite the programmer's tasks. Similarly, the features of local clarity and structural clarity tend to aid the reader of the program while he (or she) is either debugging it for the first time or incorporating a modification within it.

Finally, the most application-area-dependent language characteristic is what we called the "application dependent functional support". By this we mean the special collection of built-in functions and facilities that tends to make programming more natural in a given application area. For instance, in the scientific application area we are evaluating the built-in functions and facilities (of a language) for calculating trigonometric

functions and manipulating arrays. In the data processing area we are evaluating the built-in functional support for various common file-manipulation processes, such as merging and sorting.

It is appropriate at this point, as we have discussed in section 2.3, to enumerate the characteristics of an ideal methodology for representing and evaluating logical structures in a CAD environment. The characteristics are as follows.

1) Multilevel structure : It must be able to support a system specification capable of handling varying degrees of details, for instance, a concise and precise behavioral description at the top level, progressing downwards to an exact, detailed realization.

2) General application : It must be possible to describe a wide range of systems, including digital control, message switching, etc., as well as computer systems. The organization of the description should reflect the structure of the system, allowing easy communication among engineers and designers.

3) Technology independent : The modelling primitives must be independent of any particular technology and capable of realization in either hardware or software.

4) Easily assimilated : Because of the continual problem of training new designers and the immediate perturbation when working designers start to use it, the basic use of a design language should be easy to learn. That is to say, the beginner should produce correct and usable "programs" in a relatively short time, while the power and elegance in the use of the language come with practice.

5) Interactive : It must be possible to involve the designer in the design process, with appropriate feedback of errors and intermediate results including editing facilities. The design language must be simple and convenient to use.

6) Extendable : The system must be capable of being modified to embrace new developments and advances of technology and design procedures.

7) Self-documentation : It must be able to provide a readily understandable means of communicating ideas and detailed design information.

8) Parallel representation : Since most systems are essentially parallel in nature, concurrent operations must be easily handled.

---

[1] "Primitives", for example, refer to "operations" at the functional level (fig.2)

9) Easily machinable : The design language may be implemented on a computer system.

10) Portable : The implementation should be portable; i.e. it can be transferred to run on another machine without excessive perturbation and modification.

In the remainder of this section, we shall use the "bases of language evaluation" and these "CAD characteristics" for the selection of a suitable computer language for our proposed "Top-down Design Approach".

2.5.2 The Choice of APL (A Programming Language) for the "Top-down Design Approach"

From the above discussion (section 2.3 through 2.5), the choice of a language for the CAD would at first sight favours the use of special purpose or hardware design languages. Indeed, GPSS, SIMCRIPT, DDL (41) and CDL[1] (42, 43) have been considered as possible candidates. Among these languages, the syntax and semantics of CDL (a derivative of ALGOL 60) would have been an attractive language to adopt. However, it fails to fulfill many of the CAD characteristics discussed in section 2.5.1. Despite its availability on many machines, yet there is a great lack of technical support (which is essential for serious users) for the various versions. In addition, the language is not easily assimilated; especially by occassional users. The language also lacks many desirable "implementation dependent features", such as execution-time diagnostics. Furthermore, CDL[2] itself required excessive storage to load (45K bytes) and typically

---

[1] This author would like to acknowledge the CDC version received from the U.S. Naval Research through the assistance of Proof. Y.Chu of Maryland University. However, the version contained many implementation bugs which have prevented it from being successfully installed at the Computer Centre of Imperial College. Yet sufficient experience has been gained to permit an evaluation of the CDL.

[2] Based on the experience of the version of CDL received; this version was implemented in "extended FORTRAN IV" for the CDC 6000 CYBER series.

a total of 60K bytes to run. However, we are anxious to give all due credit to the positive features of CDL; example of these are the hierarchical and block-oriented structure, facilities to represent information flow in large variable systems, and the ability to handle the problems of "timing".

On the other hand, characteristics (CAD characteristics) of easy assimilation, easily machinable and portability, would be satisfied if one of the common high level language[1] could be directly used as a hardware design language (HDL). Unfortunately this is not possible as none of these languages offer all the syntactic elements and language constructs (e.g. multilevel structure general application & parallel representation) required in a HDL. Therefore, approximately half of all existing HDLs are genuine syntactic extensions[2] of (and frequently deviated from the) customary high level languages, while the other half are new and apparently exotic language creations (44). However, we shall show (section 2.6) that it is possible to achieve a HDL by augmenting a most suitable and widely used high level language in such a way that the original syntax is preserved. Thus, we may provide an implementation and a CAD environment for our proposed "Top-down Design Approach". The high level language chosen is APL (A Programming Language). In the process of selecting APL, we have also considered FORTRAN IV, ALGOL60, SNOBOL, and PL/1 as possible contenders. However, none of these language satisfy the above requirements (section 2.5.1) as well as APL.

---

1 e.g. BASIC, FORTRAN, COBOL, ALGOL, & PL/1

2 e.g. CDL is an extension of the ALGOL 60

APL was invented by Dr. K.E. Iverson as a notation, known as "Iverson's Notation" or "A Programming Language" (45). This Notation is rich in mathematical operations and symbolisms, and was initially formulated to handle scientific ideas.[1] There have been many developments of this Notation in the form of computer programming languages (46-56), APL/360 is the best known and is an easily available version. This implementation, widely accepted as "Basic APL" or simply "APL", is both precise and concise. The language is interactive and rich in primitive functions (see appendix 1); operations such as arithmetic, relation, set, logic, encode/decode and other mathematical functions are thus available. Both symbolic and character manipulations are also possible. The language is array[2] oriented and manipulations (at both the element or array levels) of vectors, matrices and arrays of higher dimensions is possible using the primitive functions provided; for example, string and array operations for concatenation and decomposition (masking), indexing & restructuring (e.g. by rotation) of arrays, generation of constant vectors, membership, encoding & decoding, etc. Other desired operations may be defined using the primitive and these in turn can operate as if they were themselves primitives (i.e. the language is uniform and contistent). In other words, the language structure allows the nesting of "procedures" ("functions" in the context of APL) with facilities for arbitrary "variable name scope control". It also contains "control structure" for "single" or "multiple-branching". Despite the richness in "programming features", APL is easily assimilated. APL/360 is widely accepted and well support (see section 2.5.4); this implies that "systems" developed with this language are "easily machinable" and "portable".

---

[1] At this time of writing, the commercial and data processing, applications of APL have become equally dominant.

[2] Data type maybe numeric, literal and boolean and associated memory storages are dynamic.

From the "implementation dependent features" view point, the language provides extension system commands (e.g. see appendix 1) and edit facilites for interactive usage, easy diagnosis of program definition and execution errors, powerful debugging facilities (e.g. execution trace and stop) and simple "workspace" (or memory) management. Later versions of APL, such as APLSV, VSAPL, APL*PLUS, XEROX APL, APLUM, CYBER APL, DEC10 APL & APL/700, have extended the power of the language. In these, file subsystems are provided to handle large data bases and to share variables or files. System commands may be embedded within programs (e.g. use of the "execute" operator, and the system function & variables in the form of "quad" functions & variables) so that system operation under program control is possible. Consequently many of these may be interfaced to other computer subsystems (e.g. data base management systems, language processors) and may initiate "background" or "concurrent" processing.

These properties of APL (& extended APL) have made the language ideally suitable as a design and simulation tool that are required for our "Top-down Design Approach". At the system design level, all primitives and user defined functions can be freely used to verify mathematical and algorithmic concepts. At the functional level, user defined functions can now be interpreted as functional blocks or as subsystems of the digital system architecture. Primitive functions (like encode/decode) and the symbol/character manipulative power of APL are useful in defining and simulating the information flow which are both internal and external to the subsystems. At the register transfer (RT) level, restricted use use of APL primitives and the structure of user defined functions (with some augmentations as indicated above), enable the description and simulation of system architecture details (e.g. the sbu-units in figure 1d). At the gate/logic level, restricted use of APL, as at the RT level, permits logic simulation (see chapter 4) and fault analysis (57) of detailed design. For logic synthesis and minimization (58), the full power of APL can be employed. The same comment applies to circuit level design (e.g. Spence - ref. 59, and MARTHA - ref. 39).

Even though logic simulation, & minimization programs and circuit analysis softwares and available in other computer language (e.g. DLS - ref. 60, LOGMIN - ref. 61, and ECAP - ref. 62), it is advisable to maintain the same language whenever possible. The use of only one language suggests working with one computer system and even with one of its subsystems (i.e. computer language processors); this is less distracting for the designer in his creative design task than the use of different languages and systems during various phases of the design process. On the other hand it should be noted that sometimes it is advantageous and economical to use non-APL packages; we shall show in section 2.6 that an "APL pre-processor" or an "APL operating system" may be designed so that the use of these non-APL packages may be "transparent" to the users.

Another important aspect of APL is that a design or analysis package can be formulated as a pseudo-code language, such as MARTHA & SIMLOG (see chapter 4), so that the language resembles the jargon familiar to the designer. Furthermore, the designer is also permitted to incorperate his defined APL functions into the pseudo-code language. This enables the software package to be adaptive and tailored to changing needs.

On the other hand, APL has many deficiencies (63,64). Semantically the ability of APL to handle rectangular homogoneous array is unsurpassed. Unfortunately, not all data are most naturally represented in this form. Many application problems, such as logic synthesis, require data structures that are non-rectangular as well as non-homogoneous. For example, a wiring name list may be represented by a character matrix; but a vector of character vectors might be a more convenient representation. Other examples are the need of additional primitives to handle structures such as sparse arrays, unordered sets and non-homogoneous arrays, lists, files graphs functions & expressions. Efforts have been proposed to overcome this deficiency by simulating a LISP-like structures and primitives. One approach is to write a complete LISP system, including statement interpretation, storage and name management. In this respect, all input is via "quote quad" so that the user may not be aware that the LISP system is written in APL. A second approach is to model the LISP structures as

---

[1] Quote-quad, ⍞, and quad, ⎕, are APL input operators; with the former for a literal array and the latter for a numeric array.

APL arrays and to build APL functions corresponding to the primitives of LISP. In this case, APL functions with names like *CAR* and *CDR* which may be employed within APL expressions. However, neither of these two methods is wholly satisfactory. The first is preferable in that the final result behaves exactly as desired. This, unfortunately, is attained at the expense of much programming efforts & a considerable amount of computer executions and often inevitably duplicate functions inherent to an APL system. With this approach, it is impossible to access both the APL and the LISP data in a single expression. On the other hand, the second approach, while simpler, engenders problems of name conflicts between the user's identifiers and those identifiers used in the modelled primitives. Furthermore, there is no simple way to distinguish objects of the new data type from ordinary APL arrays. In relation to data structure, APL also lacks primitives to handle "complex number structure" (i.e. real and imaginary parts) which is common in most electrical engineering concepts. However, as in modelling a LISP system, complex number arrays and corresponding APL functions may be defined. Likewise, this approach though simple requires a user accepting the possibility of name conflicts.

From the "control structure" view point, one of the essential requirements of the "stepwise refinement" or "top-down programming technique" is that within any program or system of programs, all sub-programs maybe viewed as "black boxes". The choice of local names, subprograms, programming techniques, and so forth within a box should not be constrained by choices made within higher level programs, and vice versa. One adverse consequence of APL's elegant name-localizing rule is that the black-box approach cannot generally be realized, because names cannot be made "strictly local" to a program. Although this defect hampers subprogram isolation, yet APL's localization rule provides flexibility in communication between calling and called programs that is unequalled in most other programming languages. Further, because the names accessible by a program are exactly those accessible at the point of call, APL has the significant advantage of allowing subprograms to be tested and observed "in vitro". There is also a lack of function "control mechanism" -- the only control mechanism to exit an APL function is a return to the point of call in the invoking function (or program). In addition, there is a lack of an "uplevel transfers"

of control to a specified point in the global environment, such as an "error return". Similarly, there is an absence of "sideways" control transfer; that is, the subroutine call and return are not sufficient to model parallel or cooperating processes. This kind of facility is found in discrete event simulation languages (e.g. SIMULA) in the form of coroutines. This problem, however, maybe reduced by the use of the "execute" (⍎ or ∈) and the use of the $\square LC$ (see section 2.6.3).

One of the most serious practical problems in the construction of practical systems and packages is the need to monitor and control the environment, notably for error conditions and external events. These facilities are absent in the basic APL; however, "Shared Variables" (Falkoff and Iveson; ref. 49) and "Side-tracking" facility of the Xerox APL (54) appear to offer some partial solution to a segment of the problem. It is eminently clear to application designers that some form of interrupt processing is required. This kind of facilities would permit recognition of and reaction to synchronous events such as errors, as well as to asynchronous events such as terminal conditions, timers, the state of concurrent programs, or other occurrences "external" to the immediate sequential environment.

Finally, APL has been implemented as "interpretors". It is believed that the inherent power of "dynamic storage allocation" and the "execute and evaluated input" are the features that prevent APL's implementation as a compiler. On the other hand, it is recognized that APL had been designed as an interactive and problem solving language. As such it is amenable to be implemented as interpretors. This leads to the emphasis on the developments of optimizing interpretors rather than compiliers.

Despite the seemingly inherent shortcomings of APL, yet it is still the most suitable language available. It is this author's contention that having recognized the strengths and shortcomings of the language, perhaps the next step is to devote ones efforts to utilize its power as intended by its inventor and advocates. APL maybe viewed as a "personal language"; its problem solving features are those of the digital system designers. Its popularity and the constant efforts in enhancing and extending the language

COMPUTER	INSTALLATION	COMMENTS
IBM 360 370	Boeing Computer Services, Inc., Virginia, U.S.A.	Uses IBM 370/168, providing VSAPL/CTS; U.K. office in Watford, England.
	BOC Datasolve, Middlesex, U.K.	Uses IBM 370/158, providing APL/SV (see table on Vendors)
Amdahl 470	IP Sharp & Associates, Toronto, Canada	Uses IBM 360/75 & Amdahl 470/V6, providing SHARP APL, formally known as APL PLUS but current developments have taken a diffent trend from APL PLUS; U.K. office.
	STSC (Scientific Time Sharing Corporation), Maryland, U.S.A.	Uses Amdahl 470/V5, providing APL PLUS; early developments in cooperation with IP Sharp; U.K. Office
Univac 1108	Computer Science Centre, University of Maryland, College Park, Md.,U.S.A.	Works with Model 35 Teletype; uses special set of characters. (ref. APL Quote-Quad, Vol.2, No.3, Sept. 1970)
Burroughs B 5500 B 6700 B 7700	Computer Science Group, University of Washington,, Seattle, Washington, U.S.A.	Uses B 5500, works with 8 remote Model 33 Teletypes; has many special features (ref. APL Quote-Quad, Vol. 2, No. 4, Nov.1970)
	Burroughs Corporation, Paoli, Pa., U.S.A.	Error recovery and environmental protection, and other special features for B6700, B7700.
CDC 6500 6600 7600 Star	Computer Centre, Michigan State University, East Lansing, Mich.,U.S.A.	CDC 6500; works with Model 33 Teletype; some file handling capability; 12 more header types for functions.
	Livermore Radiation Lab., Livermore, Cal., U.S.A.	CDC 6600, 7600, & Star; works with Model 33 Teletype; many special features, e.g. I/O, file manipulation, and system calls within the APL program.
CDC 6000 & CYBER series	Computer Centre, Univ. of Massachusetts, Amherst, Mas., U.S.A.	APLUM which is upgraded as CDC APL Ver.2, and is run under operating systems KRONOS & NOS; contains many special features, such as error-trapping, special file interfaces and the ability to communicate with other processors in the computer system.
	Computer Centre, Imperial College, London, U.K.	
Honeywell 635	National Security Agency, Ft. Meade, Md., U.S.A.	Special features and system commands.
PDP-10	Carnegi-Mellon Univ., Pittsburgh, Pa., U.S.A.	DEC-10 APL, runs under operating system TOP-10; contains many special features, error-trapping via the execute operator; implementation tends to be in a "non-standard" manner; it utilises the DECsystem file system and despite the ability of interfacing to non-APL systems, yet this unfortunately has led to some annoying restrictions. Later modification has features similar to the APLSV.
	On-Line System Computer Services, Pittsburgh, U.S.A.	
XEROX Sigma	Atkins, On-Line System Computing Services, Epsom, U.K.	Xerox APL, modified to Atkins APL, is run under the FOURMOST operating system; contains many special features, these features are "sidetracking", "Blind-I/O", text-string manipulating primitives, canonical representation of functions, and the ability of interfacing to a data base system, EDMS. In addition to Telefile's efforts of supporting and upgrading the hardware of Sigma series which was abandoned by Xerox, Xerox APL may also be run under the operating system and the mainframe of Honeywell Level 66 series computers.

NOTE : Apart from the APL implementations on mainframes, there exists many versions on "mini-computers"; e.g. Hewlett-Packard 3000, DEC PDP-11 and desk-tops such as IBM 5100 & 5110 and Leigh Data MCM 800.

Table 1 : A List of APL Implementation on various Computers

constructs and implementations warrant serious exposures to the language. Efforts spent in cultivating an indepth understanding and the acquisition of a working knowledge of the language are of long-lasting natures. This accounts for the spirit with which APL has been adopted for our "Top-down Design Approach". We shall conclude this section with a review of the availability of APL processors and a brief indication of their implications to the design of CAD systems.

2.5.3 Review of available APL processors

Since the invention of the "Iverson Notation" (45) in 1975, much effort have been devoted to implement the Notation as a computer language; examples of these were the implementations on the IBM 1130 and 1500. In 1966 an experimental time-sharing system for the IBM System/360 became available within IBM and it gradually became available as an IBM product. This version is commonly known as APL/360 and is implicitly & widely accepted as a "basic APL". Falkoff and Iverson (46) have provided much impetus to the development and implementation of this terminal language. Under their directions, an extension to APL/360 is implemented on the IBM 370. This extended version is the "APL Shared Variables" (APLSV - ref. 49). In addition to being available on the IBM product line, APL has been implemented on a number of other computers, as shown in table 1.

Many of these installations have implemented APL to support Model 35 Teletype. The fact that these Teletype models are cheaper and already in wider use than the IBM 2741 (or its equivalent) has undoubtedly made APL available to a wider audience. However, with the rapid advances in electronics and technology, terminals such as the DEC writer II, AJ-832, DIABLO, TI Model-770 and other VDU terminals (65,66) are providing robust terminals with "switchable character sets". There is every reason to believe that the cost of these terminals and their successors will

decrease with time.[1] The costs of these terminals are often off-set by their capabilities, such as "bi-directional printing", and "data logging" facilities. In this respect, switchable-character set is often considered as a "bonus" for a time-sharing usage. The availability of these "APL terminals" helps in promoting a better acceptance of the APL symbols and above all an effective use of the language. An APL function (or program) may now be easily read and the functions themselves maybe used to convey concepts which are capable of being executed as computer programs.

Each of the installations listed (in table 1) has experimented with and made certain additions and improvements. By making the language more powerful and removing some of its deficiencies (section 2.5.2), they help to promote its effective use. However, this blessing is not unmixed. The incorporation of nonstandard character sets[2] (necessary to accommodate Teletype) and of features for one computer that are not available on another has led to a growing standardization problem. By taking advantage of a particular implementation of APL, a user may find himself "locked into" that installation and unable to run his programs elsewhere.[3,4]

---

1 Integrated circuits and microprocessors are gradually replacing mechanical parts of terminals. With the rapidly decreasing cost of semi-conductor, it is reasonable to predict the correspondent cost reduction in terminals. In addition, new materials are actively developed and introduced into products; such as magnetic bubble memory for the TI Model-770 terminal. Furthermore, this "forecast" is reinforced by the popularity of personal computing which in turn stimulated the developments and productions of a host of cheap and inexpensive terminals and TV-terminals.

2 mnemonics instead of APL symbols; e.g. in XEROX APL, the "take" operator ↑ is represented by $TAK.

3 this apparant loss of "portability" does not affect our use of the language as basically we are using a subset of the APL/360. The extensions of APL, however, improves the CAD environment (section 2.6)

4 As a solution to portability among machine, the IBM Scientific Centre in Madrid proposed a portable APL interpreter. This interpreter is written in a specially designed system implementation language called "Intermediate Language" (IL). The APL interpreter is capable of running on a range of machines including systems 370 and series 1. — IBM Journal of Research & Development, July, 1978.

Recognizing this potential problem, APL user groups have been founded in North America and Europe to meet regularly to discuss and propose standards. Numerous topics have been discussed and guide lines issued by the STAPL (ACM SIGPLAN Technical Committee on APL); including the transfer of APL workspaces at the source level, general arrays, standardization of APL implementations, extensions to APL and secure packages written in APL. Detailed reports of the meetings are separately published in the APL Quote Quad[1]. On the other hand, the growth and vitality of the language are witnessed by the number of time-sharing bureaux heavily committed to the marketing and technical supports provided for APL. (see table 2)[2]. Our discussion thus far substantiates the remarks (section 2.5.2) that APL is readily available and a widely accepted & utilized[3] language. Furthermore, it also supports the reasons for our choice of the language for the proposed "Top-down Design Approach". To appreciate the implications of the various available APL processors to the design of CAD systems, we shall first consider the major and representative extensions of APL/360 as belows.

1) System Variables and System functions;
2) Error recovery/Event trapping;
3) Files capability;
4) Shared Files;

and 5) Shared Variables including virtual storage.

---

[1] A journal specially devoted to APL and is published four to five times a year, originally initiated by Prof. G. Foster, Link Hall of Science, Syracuse University, Syracuse, New York.

[2] Many of these bureaux have international networks extended over Europe; for example IP-SHARP's network spans North America, Europe and Australia.

[3] in commercial, scientific and engineering applications. This can be derived from the packages provided in the public domain; see for example, IP-SHARP and Scientific Time Sharing Corp. (STSC).

APL Services at U.K. Time-Sharing Bureaux

VENDOR	SERVICES	HARDWARE	LOCATION
Atkins, On-line System Services	Atkins APL	Two Xerox Sigma 9s	Epsom
Boeing Computing Services	VSAPL/CTS	IBM 370/168s	Virginia (USA)
BOC Datasolve	APL/SV	IBM 370/158 MP	London
CDC Data Services	APL 2.0	CDC CYBER 175 & CYBER 173s	Brussels (Europe)
Compower	VSAPL	IBM 370/148	Cannock
IP Sharp & Associates	SHARP APL	IBM 360/75 & Amhahl 470/V6	Toronto (Canada)
OLS Computer Services	APL 10-E	DECsystem 10s	Pennsylvania (USA)
Scientific Time Sharing Corp.	APL PLUS	Amdahl 470/V5	Maryland (USA)

NOTE : The above support APL fully, with marketing & technical supports. Other bureaux offer APL but less committed are : CSS, Comshare, CRC, and IBM Remote Computing Services.

Table 2 : APL Services at U.K. Time-Sharing Bureaux

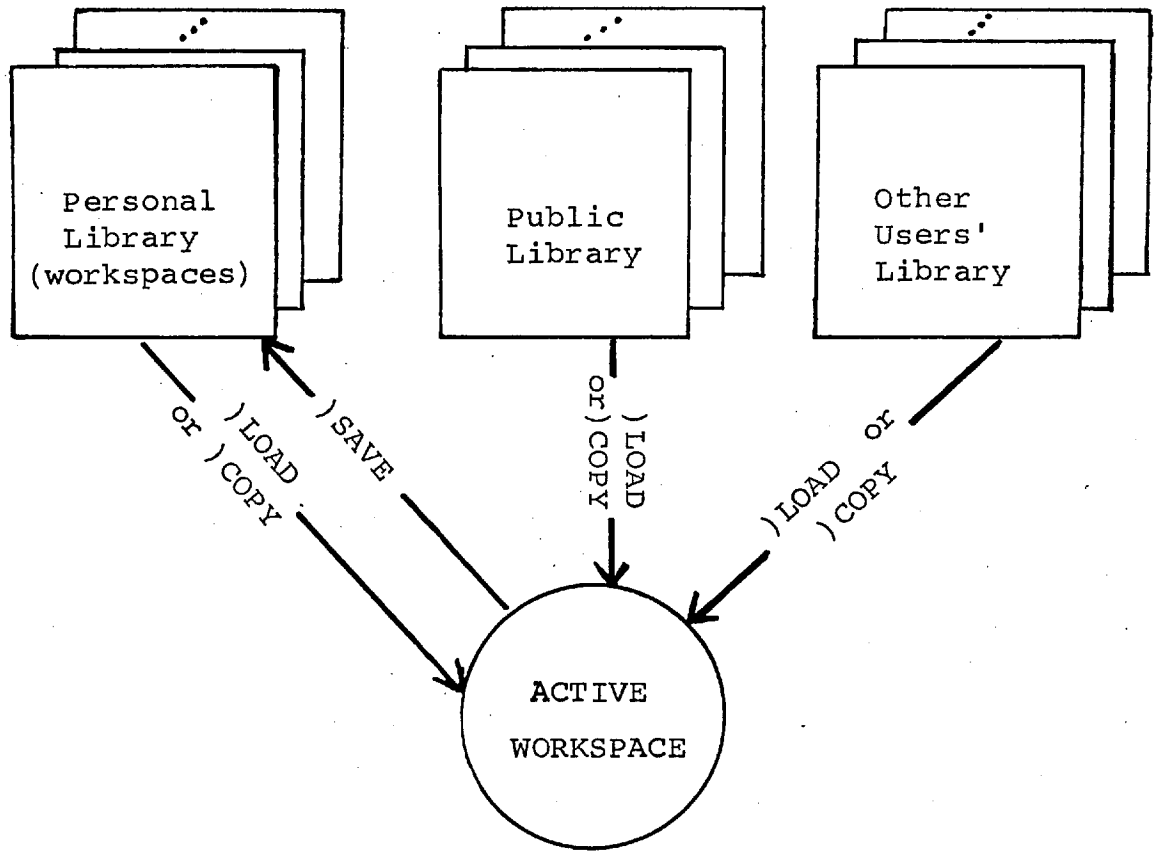
These three areas are inter-related; but for our present purpose, we shall treat them individually and conclude this section with a discussion of the consequence of these in the design of CAD systems.

1) System Variables and System Functions

In the APL/360 environment, transfer of information (functions or data) between workspaces is initiated by system commands)*LOAD* &)*COPY* ; as shown in figure 6. Furthermore, there exists some functions (I-beam function) which provide basic information about the state of the APL system; for example, the central processing unit (CPU) time used since sign-on and is usually expressed in 60ths of a second. These I-beam functions, unlike system commands, are dynamic and may appear in user-defined functions (e.g. *TIME*←⌶19). However, these functions only provide information on the system but cannot be used to modify the environment of it. The latter may be achieved via system commands such as those shown in appendix 1; while the classification of these commands is depicted in figure 7. For the APL/360, these commands must be entered by the user under the "immediate execution (or calculator) mode". Some systems, such as XEROX APL, permits these commands to be embedded in APL programs as "literal strings" and are invoked indirectly via the execute operator (⍎ or ⍷). Both the use of ⌶-beam function and the indirect execution of system commands, though useful, tends to mask the "overall reading and writing" of APL functions. As a standardization[1] to the language, system variables and system functions are introduced; with their names preceded by the symbol ⎕ or ⍞ . This distinction improves the overall reading and writing and above all avoids the possibility of name conflicts. System variables define, inform and control the APL environments and the values of the system variables can be changed between settings. For example, ⎕*AI*, which is a vector of accounting information, may be set by the user to any desired value, but the next time he requests its value, it will correctly reflect current accounting information - that is, the system resets the value of ⎕*AI* before it is read.

---

[1] Results of discussions and proposals of APL user group meetings.



NOTE : From a physical standpoint, the active workspace is a portion of the computer's primary storage, i.e. magnetic core, which is assigned to a user at the time of sign-on. While inactive workspaces (or workspaces) are subdivisions (i.e. "logical units") of a block of secondary storage (e.g. magnetic disk) associated with a computer account number. These workspaces may be private or public

Fig. 6 : Workspace management in the APL/360 Environment

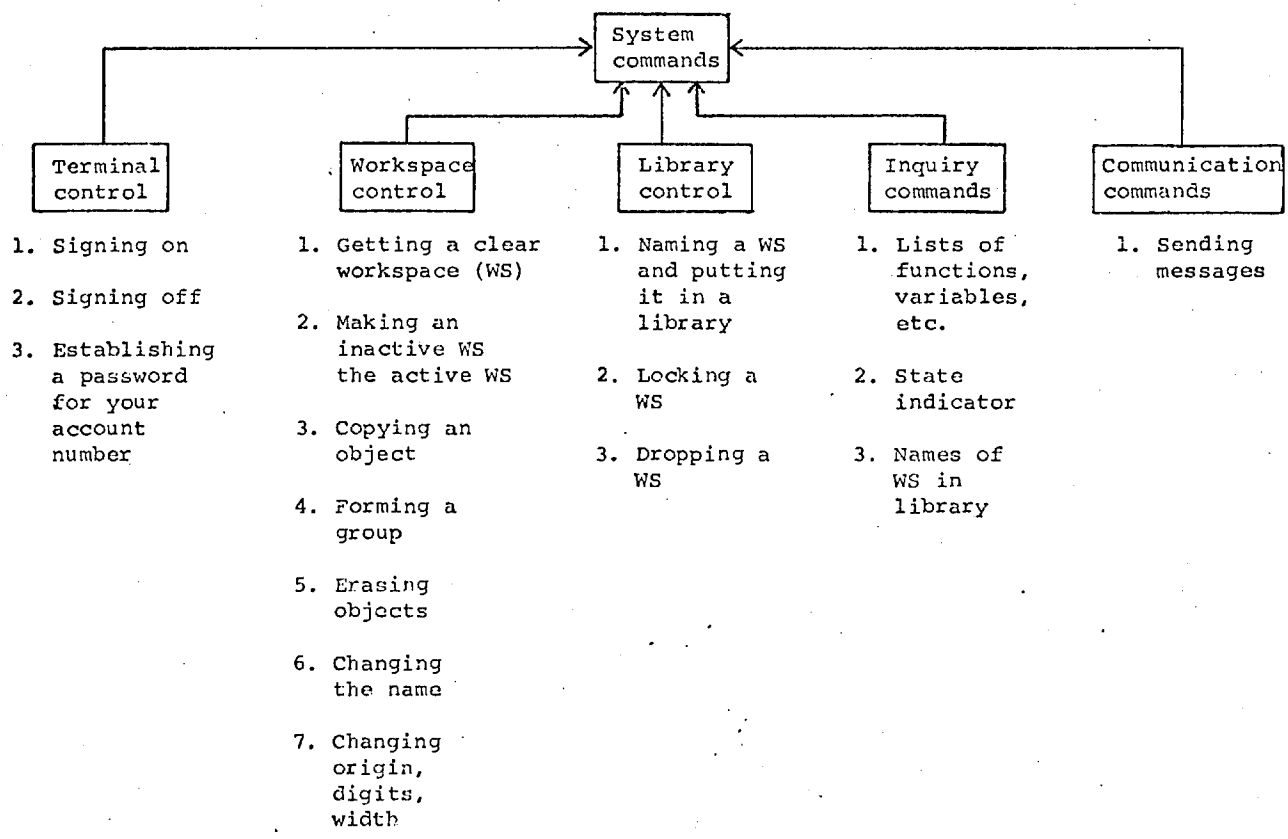


Fig. 7 : Classification of the APL System Commands

Similarly, □*AI* can be erased by the user, but the system gives it a value whenever its value is requested. While system functions, like APL primitives, permit a user communication with the APL system and (e.g. APLUM or APL Ver.2) in some extent with the operating system under the control of which the APL system runs. For example □*COPY* , copies objects from a workspace into a active workspace within an APL function. While □*LIB* <account, file name> inquires the existance of an non-APL file which could be created by a "background non-APL" computer job. With the *CSUBMIT* command (as in the case of APLUM or CDC APL Ver.2), a FORTRAN program could be submitted under APL while the results of this FORTRAN job may be retrieved by APL for further processing (see discussion follows and section 2.6).

Hence, system variables and system functions together provide significant capabilities; while in some cases (e.g. localised system variables) and completely new capabilities. For examples, IP-Sharp provides a set of "package" functions, (e.g. □*PDEF*,□*PACK* ,etc.) when properly used, will provide very powerful application packages. STSC's APL-PLUS introduces new features via the "quad function", such as □*SS* for efficient character and text manipulation. Atkins APL (a modification of the Xerox APL) provides a set of computer graphic primitives. For some systems (e.g. IP-SHARP, APL-PLUS, Atkins APL, APLSV & VSAPL/CTS) the largest family of system functions are those for file operations.

2) Error Trapping/Event Trapping[1]

When an error is encountered during the execution of an APL function(s), APL/360 system halts the operation (i.e. "suspends an execution") at the point where the error is detected. The system then prints the suspected error, unlocks the keyboard (if an IBM 2741 is used) and awaits for a

---

[1] Some implementors (e.g. STSC) prefer the term "Event trapping" to "Error trapping" as the former has a wider connotation. For example terminal type and operating conditions are not strictly errors; yet a sophisticated APL system needs to detect and account for these events.

new command to be entered by a user. In many extensions[1], facilities are provided to intercept an error detected and the error is then processed by the APL functions rather than the APL processor. That is, once an error is trapped, the function causes a forced branch to the "trap line". With a priori of the error condition, which the system designer has full knowledge, the APL function can adopt an appropriate action. For example, an erroneous data entry for a quad input causes the request for the input to be repeated. The facility of error trapping helps in designing easy-to-use software systems.

3) Files capability

Almost all commercial data processing and a good many scientific applications involve more data than may be accommodated in a single workspace. When there is adequate storage space (active workspace), APL/360 requires saving a complete workspace even when one out of the many objects (e.g. functions, variables) needs to be stored. One approach is to "Overlay" multi-workspaces as shown in figure 8; this corresponds to "overlaying programs", for example, with FORTRAN IV. For this, APL functions are either grouped according to operational tasks as "groups of objects" in an APL workspace or modules of workspaces. A monitor workspace is loaded by a user as an application system. Under the control of this monitor workspace, subsequent groups of objects (functions and/or variables) are "protected-copied" (⎕*PCOPY*) or "copied" (⎕*COPY*) into the active workspace. However, this approach has not eliminated the aforementioned problem; i.e. a complete workspace still needs to be saved in order to maintain a copy of the computed variable.

---

[1] e.g. "error trapping" (with single branch) implemented by APLUM, CDC APL Ver.2, & DEC - 10 and "sidetracking" (with multiple branches) implemented by Xerox APL & Atkins APL.

As an example, this need arises when a simulation run is to be saved for comparison in future computer sessions. Furthermore, copying objects under a program control is both time-consuming (in terms of response time) and expensive (in terms of computer units). On the other hand, loading workspaces under program control is cheaper (tests on the Xerox APL indicated typically 5 to 6 times faster and cheaper); but at this time of writing, saving a workspace cannot be achieved under a program control. It is necessary to direct the user to maintain a copy of the result by saving a workspace in the immediate execution mode.

To overcome this, file subsystems are introduced to enhance APL processors; figure 9 shows the alternative implementation of the "overlay" arrangement as depicted in figure 8. There are three school of thoughts in providing file subsystems; viz. non-shared files, shared files, and shared variables. The last two will be discussed respectively in paragraphs 4 and 5 below. For the non-shared files, there exist essentially three file structures[1]; namely, APL component file, APL non-component file, and non-APL file. We shall discuss these individually.

APL Component file : may physically be interpreded as a storage area on magnetic discs which also provide a storage for non-active APL workspaces. The storage area is subdivided into blocks called records which are of fixed sizes; typically 3000 bytes. For some installations, the size of the records may be defined at the time when a file is "created"; while some also permit this parameter to be modified at a later stage. In either event, the physical record size has much effect on the behaviour and efficiency of the file subsystem.

---

[1] This is a general classification and not every implementation provides all the three structures. Our classification assumes a direct access of these files within an APL, i.e. some primitives or system functions are provided for file access. For example, an installation such as IP-Sharp, provides both APL component file and non-APL file; but non-APL files may not be accessed by a user defined APL function. Instead, IP-Sharp provides a package to convert an IP-Sharp specified non-APL file into an APL component file which may subsequently be processed by a user's application APL system. In our context, we classify IP-Sharp as an installation providing only APL component file.

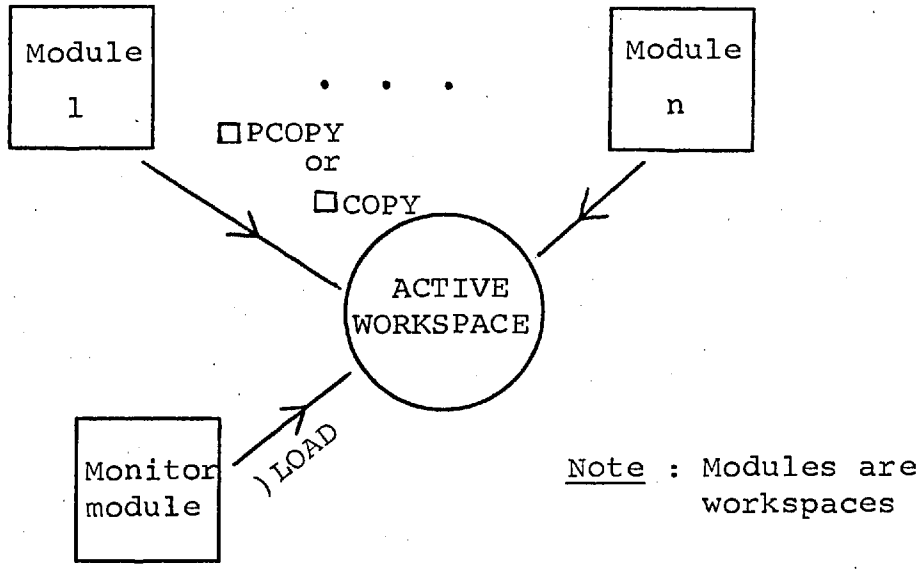


Fig. 8 : Overlaying multi-workspaces using □PCOPY or □COPY

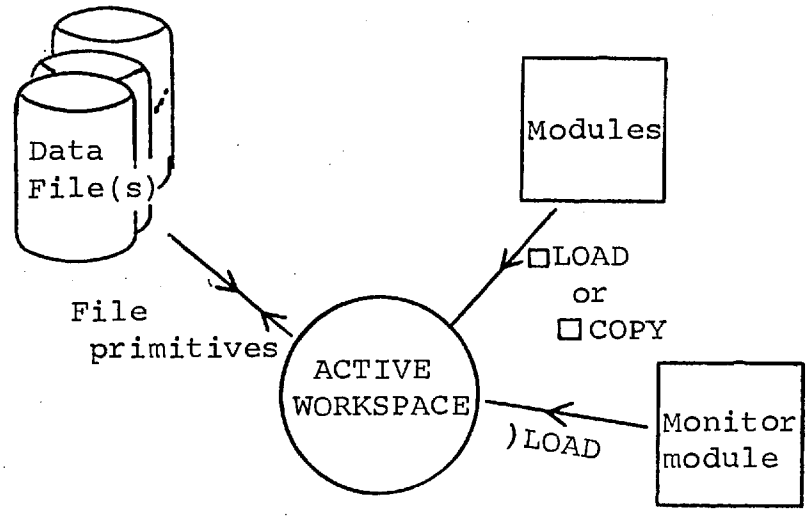


Fig. 9 : Overlaying multi-workspaces and data file(s) for large amount of data

From the application view point, we consider an APL component file having a name for identification and consisting of a collection of non-homogenous "data items". The structure of a data item corresponds to that of a variable in a workspace; e.g. scalar, vector, array, literal, numeric or boolean. Instead of referring by names, as for the case of variables, data items are referred to by numbers which specify the positions of the items in the file. Each data-item in the file is called a component (instead of a variable) and the associated number is called a component number, in the APL component file, consists of two physical records (as discussed in the above). The first record is an "identification record" and is divided into logical records providing information on the second record, such as record size (in bytes), data and time during which the record is created, and the computer account (hence ownership) which originated the component. The second record is the "data record" which holds an image of a variable in the workspace.

Hence APL component files relieve the user of concern for "housekeeping"; but the cost is in the extra file storage and input/output (I/O) accesses required for the identification record of each component. All APL processors with file subsystems offer component file by default; while some (e.g. Xerox APL, CDC APL.2, IP-Sharp APL, APL-PLUS, DEC-APL, etc.) provide file primitives to access the identification only. This has the extra degree of freedom to design a large "data base" system -- a priori information on the data record is attained without the need to retrieve it from the file and to process it subsequently in a workspace which may, temporary, have insufficient storage (recall that storage allocation is dynamic for APL).

APL non-component file : this is similar to a component file in that data records are APL variables and again the component number serves as an identification of the data record. The identification record (i.e. the first record of component file) is omitted. The user (or the application system) must maintain information concerning maximum record size, and no information is maintained concerning the source or time (of creation) of individual data record .

Non-component files reduce I/O by half compared with component files, and eliminate file space requirements for identification records. They should be preferred for use when file structures are fairly simple and detailed identification information on individual records is not valuable. However, non-component file structure is not implemented in many installations (e.g. IP-Sharp APL, APL-PLUS, CDC APL.2); Xerox APL & Atkins APL provide this structure.

Non-APL file : is a file created by language processors other than APL; examples are FORTRAN, BASIC, & COBOL files. Often it is desirable to be able to access these files within an APL system. In addition, it is equally important to be able to create non-APL files under APL for subsequent processing by processors such as FORTRAN, CDC APL.2, Xerox APL and DEC-10 APL provide this two-way communication facilities.

* * * *

It should be noted that "non-shared file" does not mean it may not be shared among a group of uses. With most installations, a file created by any computer account is by default a private file so that it may not be available to other computer account numbers. However, the originator may permit other users to access the file. The access may be a "read" and/or "write" access. By "non-share file" in the context of APL, we mean the facility whereby only one user may update (i.e. deleting, replacing, or writing) the file at any one time. While the file is being updated, the file may still be read by other users. "Shared files" will be discussed in paragraph 4.

4) Shared Files

Like "non-shared files" (also called "personal files"), more than one "shared file" may be active at any one time. However, when a file is first created, by default it is personal. When the file is to be declared a shared file (which permits simultaneous updates) the owner is required to specify the type of "access permission" among users. In the APL-PLUS and IP-Sharp file systems, all access rules are determined and recorded in an "access matrix" which associates with any file created. The matrix consists of three columns : the first is the user number of those who have access permission of some sort; the second is the sum of "access codes" (table 3) granted to that individual; and the third is an "access passnumber" which provides a level of security and is zero, by default, designating no access pass-number is required.

In figure 10, a file called *DATABASE* is share-tied to file tie number 20 the access matrix for *DATABASE* is displayed by □*RDAC* 20 . The matrix indicates that account 147589 has been designated the master account and is therefore declared complete access (Code = ¯1). While the permission code = 24 (8 for append plus 16 for replace) for account 394663 designates the abilities to append (□*APPEND*) & replace (□*REPLACE*) components in *DATABASE*, 273903 has been given the permission (access code = 1) to read, □*READ*, any component in *DATABASE* . For a multi-user environment, it is sometimes necessary to temporarily withhold the file exclusively until a critical sequence is completed; □*HOLD* has this effect when executed and only account 147589 has this capability. Furthermore, it may transfer the ownership of *DATABASE* by renaming (□*RENAME*) the file. The flexibilities offer by "shared file" and associated "file system functions" provide a framework for developing many practical and interesting systems, such as inventory control, many-person games, simulation studies and message switching.

ACCESS CODE	*FILE FUNCTIONS*
1	□*READ*, □*SIZE*
2	□*TIE*
4	□*ERASE*
8	□*APPEND*
16	□*REPLACE*
32	□*DROP*
64	□*HOLD*
128	□*RENAME*
256	□*RDAC*, □*STAC*
512	□*RDCI*
1024	□*RESIZE*

NOTE : *COMPLETE ACCESS TO A FILE IS DESIGNATED BY A PERMISSION (ACCESS CODE) CODE* = ¯1

Table 3 : Access Code (Permission) & corresponding File System Functions (APL-PLUS & IP-Charp)

```
      'DATABASE' □STIE 20
      □RDAC 20

147589      ¯1      0
394663      24      0
273903       1      0
```

Fig. 10 : Example of an Access Matrix

5) Shared Variables (including virtual storage)

In the above, we have discussed that file systems have been used to handle large database and shared database applications. In the case of Scientific Time Sharing (STSC), and its counterparts, the file system is called "Shared Files"; while that of the IBM is called "Shared Variables" (APLSV - ref. 49). The APLSV facility is a means by which "data communications" like those listed below is realised; viz.

1) The ability to transmit to and receive data from another APL user.

2) The ability to use the data management facilities of the host operating system (the non-APL system, such as OS/VS, within which APLSV operates).

3) The ability to manipulate a collection of data which is larger than can be accommodated in the workspace.

4) The ability to use input and output equipment other than the APL terminal.

It should be noted that a processor is anyone or anything that uses data. A processor accepts data input and produces data output. Under this definition a processor is an APL user, a non-APL system, such OS/VS (Operating System/Virtual Storage), or an equipment such as a disc drive or printer. Communication between processors involves the transmission of data between them over an established interface.

In APLSV the interface for communication between processors is the shared variable. While a communication is established when each processor, agreeing to share, uses the system function[1] $\square SVO$, naming the same variable as that which it wishes to share. This variable becomes the "shared variable" —

---

[1] The system functions for shared variables are : $\square SVO$, to establish sharing between processors; $\square SVQ$, to inquire about any sharing requests; $\square SVC$, to control sharing between processors; and $\square SVR$, to terminate sharing between processors.

shared because when sharing is established, it appears simultaneously in both processors hence providing a "link" between them.

Syntactically, the shared variable is not different from any other variable in the user's workspace: it may be local or global, and it may appear to either side of the "specification arrow". However, the value of the variable is that which is assigned last by either owner. For example, consider the following sequence with the variable *SHARE* as the communication interface for *USERA* and *USERB*, each operating out of his own workspace:

```
USERA:      SHARE←12     (shared variable set)
USERB:      SHARE        (shared variable used)
       12
            T←SHARE      (used)
            T
       12
USERA:      SHARE÷3      (used)
       4
            SHARE←7      (set)
USERB:      T=SHARE      (used)
       0
            SHARE←T      (set)
USERA:      SHARE        (used)
       12
```

Each processor may independently assign values to the variable. When the shared variable appears to the left of the specification arrow, its value is being "set". When it appears to the right of the specification arrow, its value is being "used". Either a set or a use of a shared variable is called an "access". A variable is defined to be a shared variable if it can be accessed by two concurrent processors.

Any processor may share several variables simultaneously. This sharing may be with the same processor or several different processors. But any one shared variable may have only two owners. That is, all sharing is "bilateral".

As an example of a communication system using shared variables, consider a reservation system in which students may sign up for exclusive use of one or more devices or rooms, such as a graphic terminal, audio visual equipment, or a rehearsal room. The reservation scheduler is one processor. It resides in an active APL workspace. Each registering student (himself a processor) makes his request to the scheduler at his APL terminal through a defined function that uses the shared variable communication link. The request may be an inquiry about the state of the schedule, a reservation for a particular time slot, or a change or deletion of a former entry made by this student. The scheduler acts on the request and returns a reply, again through the shared variable communication link. The relationship between the scheduler-processor and the student-processor is illustrated figure 11. Each of the names in APL type font represents a function which contains system functions for manipulating shared variables to affect a satisfactory communication between the processors described for the reservation system.

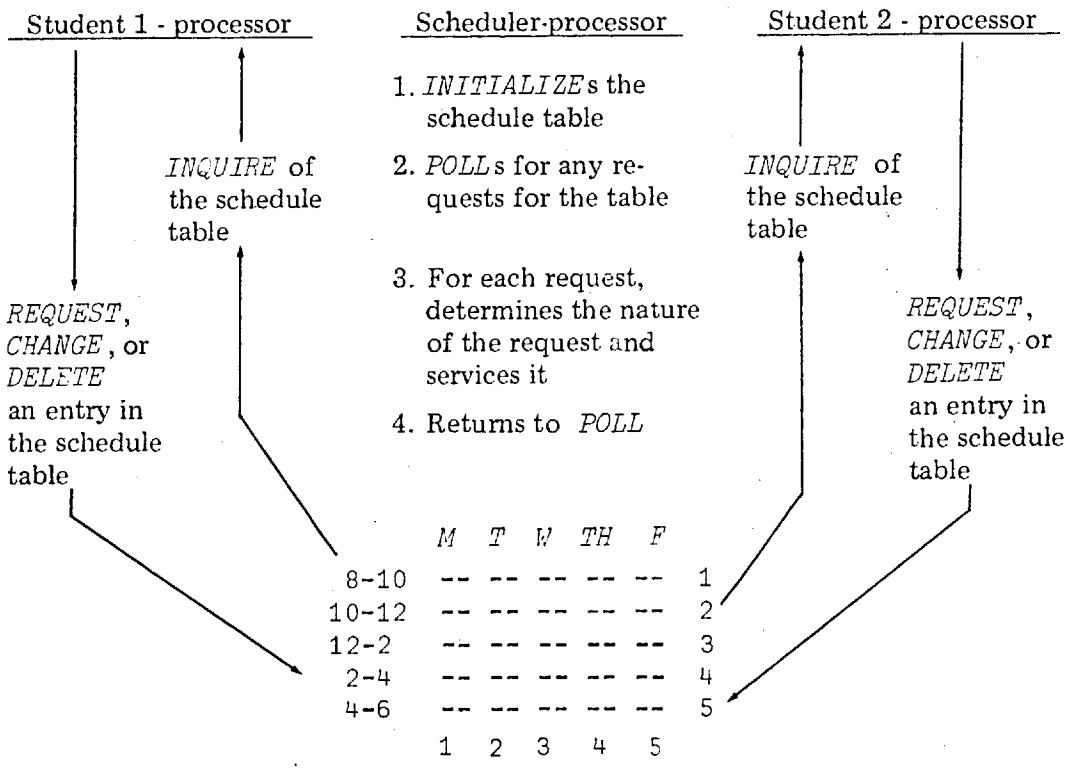


Fig. 11 : A Schedule Table — An example of a Communication using Shared Variables

2.5.4. The Implications of Enhanced APL Processors on the Design of CAD Systems

The available APL processors and their enhancements to the APL/360 have been reviewed and discussed in a considerable length (section 2.5.3). We are now in a position to consider their implications on the design of CAD systems.

1) Multi-workspace environment & data files : A typical CAD system for our proposed "Top-down Design Approach" may be that depicted in figure 12. With reference to the figure, the activities of design are shown on the left of the "Central data bank" while computer packages are listed on the right. The Central data bank holds intermediate and eventual data (or results) of a design. While packages are simplified in the diagram, each of these packages may itself be a complex system on its own (e.g. Chapters 3 and 4). It is apparent that a computer implementation of the CAD system, such as figure 12, neccessitates the realization of many interconnected programs (which form a package) and packages. On the other hand, the intricacy of the CAD system must be transparent to the user (recall the requirements of CAD system in sections 2.3 & 2.4). To conserve storage and resources of a computer installation, it is logical to partition the CAD system into functional modules which are invoked as demanded by a design task. In the context of APL, this imposes the need of a multi-workspace arrangement as shown in figure 8. From the above review, we are confident that multi-workspaces and files (non-shared files, shared files and shared variables) provide the capability required by the CAD system depicted in figure 12.

2) Simple Operational Procedure : With the capabilities offer by "system functions and variables". An "operating system" (see section 2.6) may easily be implemented, so that an occasional user may be assisted; while the operational procedure may be "abbreviated" for the experienced user (e.g. see figure 5 and section 2.4)

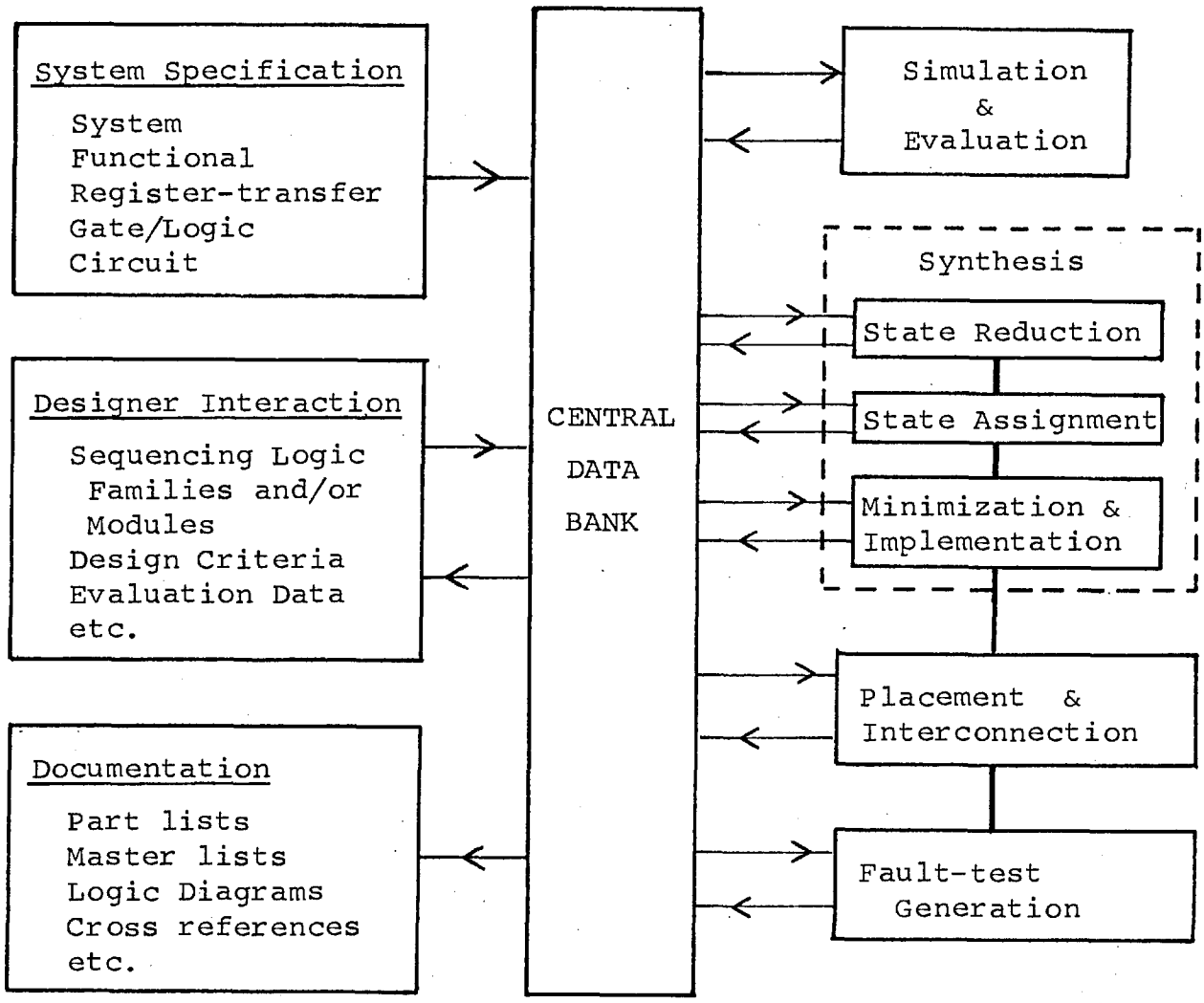


Fig. 12 : Functional Arrangement of a CAD system

3) Economical storage of packages : Systems like, Xerox APL, CDC APL.2, IP-Sharp & APL-PLUS, enable APL functions to be stored as "canonical forms" in data files. These canonical forms may later be recalled and reconverted into APL functions within another APL function in the active workspace. Furthermore, the functions may be erased, again under a program-control, to conserve storage in the active workspace. Retrieving APL functions with this method is faster (in terms of response time) and more economical (in terms of computer units) than loading a workspace under a program control[1]. In addition to this speed and cost advantage, there are advantages for "secondary storage" and maintenance of the CAD system. The storage required in a data file is proportional to the size of each function represented in canonical form (typically, a byte per character appearing in the function; while trailing spaces for each line of a function are ignored). Whereas, irrespective of the number of functions in a workspace, the storage requirement (even for one function) is that for a workspace (typically 32 K bytes). Recall that in a multi-workspace environment, it is more economical to load than to copy a workspace. By loading a workspace, one is required to save intermediate variables on a temporary file; the cost of saving and retrieving variables need to be considered. For efficient operations, as a rule of thumb, the storage required for functions in an active workspace should not exceed one third of the available storage in a "clear" (i.e. empty) workspace. Therefore, a package (or a module of the package) is best saved in a data file than to be stored in a partially filled workspace.

---

[1] Tests conducted by this author on the Xerox APL indicated there is a substantial saving with the number of functions to be less than 30. Beyond this figure, the speed and cost of retrieving functions from files and workspaces are almost equal. On the IP-Sharp system, the time required to load a workspace under program control increases during a session. Explanation offered by IP-Sharp representatives is that loading a workspace tends to require much computer resources. To prevent one user "hogging" the computer system, an increasing time penalty is imposed for this type of application.

Furthermore, a simple file organization (e.g. a directory consisting of function names and a cross-reference map indicating their occurrences in packages and positions on the file) will avoid duplicated copies of functions storing on the file. When a function is required to be modified, it is now unnecessary to make similar modification to all copies in different workspace of the CAD system. As a further bonus to maintenance, recall an APL component file, each data record is associated with an identification record giving the size, date & time of creation and the ownership of the data record. As the data record holds the canonical representation of an APL function, the history of each function is therefore updated by the APL processor for the programmer whom maintains the CAD system. A recent novel facility, called "package"[1] has been introduced by IP-Sharp. This makes storing application systems (e.g. CAD system) on data files even simplier.

4) Interface with other processors : Many APL processors (e.g. CDC APL.2, Xerox APL, & APLSV) enable the interface with other non-APL processors. This capability enables a better use of computer resources; for example, a logic minimization program (e.g. LOGMIN -- appendix 2) is unlikely to be modified on a regular basis. The reason being, the algorithms of minimization are becoming "standarized". Furthermore, the input specifications to such programs are less variant than those for a logic simulation packages (recall that for one state table specification, there may exist many logic nets). This type of packages are best implemented in a compiler language such as FORTRAN IV. Therefore, a better approach is to use an APL package to generate the necessary inputs for a compiler package (such as FORTRAN). By this, we fully utilize the interactive capability of an APL package while the speed & cost advantage of a FORTRAN package is attained. Furthermore, when a more efficient FORTRAN package is available, the user is not required to re-orientate himself to a new operating procedure for the CAD system, since the APL pre-processor may correspondingly be modified to accommodate any new feature introduced by the replacement package. This feature of "tranparency" is essential for a CAD system (see section 2.5.1).

---

[1] see IP-Sharp Technical Notes (SATN), "Packages", SATN No.14, 15th January, 1977.

5) Concurrent operation : With discrete event simulations, it is necessary for a CAD system to permit concurrent operations. Closely associated to the ability to interface with other processors APL (such as APLSV and to some extent Xerox APL, IP-Sharp, APL-PLUS and CDC APL.2) permits concurrent or "background tasks" to be executed. In discussing shared variables we have indicated such application (figure 11).

6) Multi-user environment : For a CAD system of considerable size (such as that depicted in figure 12), it is more economical[1] to make the system available to other designers. In addition, the feature of multi-access also establishes on-line dialogues among designers so that they may exchange results and compare their findings irrespective of their geographical separations. Through the shared files (e.g. IP-Sharp, APL-PLUS) and the shared-variables (e.g. APLSV & VSAPL) a multi-user environment can be introduced into a CAD system[2] (see section 2.6).

7) Easy Man-Machine Dialogue : This is a very important aspect for CAD systems. Both in section 2.5.2 and in this section (under paragraph entitled "Error Trapping/Event Trapping") we have discussed the need for error detection and correction by well designed CAD systems. We have remarked that error or event trapping as introduced, for example, by Xerox APL, CDC APL.2 and DEC-10 APL, will enable sophisticated CAD systems to be easily implemented. In addition, since the APL processor is used to detect error (but intercepted by the application program), we are making the best use of the resources provided -- otherwise, a considerable coding effort is required to perform every conceivable check before the intended operation (such as logic simulation) may be performed.

---

[1] In terms of the cost of developing and maintaining the CAD system.

[2] This author, has implemented three major CAD systems for financial & resources planning. These systems share the same feature as that for digital systems design.

To conclude this section, it should be noted that while the APL/360 has some limitations yet it has demonstrated its potentials and capabilities for digital system design. Enhancements and extensions to the language have reinforced the choice of it for implementing CAD systems. In the next section (section 2.6) we shall consider the use of APL for the computer - aided design (CAD) and the implementation of the environment for this purpose.

2.6 IMPLEMENTATION OF THE CAD AND ENVIRONMENT FOR THE "TOP-DOWN DESIGN APPROACH"

In this chapter we have attempted to rationalise the activities of a designer with the object of devising some means to assist him (sect. 2.1). This has led to the development of an "Interactive Model" (section. 2.2) and the proposal of a "Top-down design approach" (section 2.3), the success of which relies heavily on computer analyses and simulations. To this end, we have considered the criteria of designing suitable computer aids (section 2.4) and the choice of a suitable computer language (sect.2.5). Among other popular languages considered, APL has been found (sections 2.5.2 to 2.5.4) most suitable for our purpose. In this concluding section (section 2.6) we shall first describe the use of APL for the implementation of the CAD needs and end the chapter with proposing an implementation of the CAD environment[1] for our "Top-down design approach".

---

[1] The technique has been applied successfully on a financial and resources planning system for a major commercial organization. The system permits multiple-access, while the users are divided into different user groups; each of these groups uses a subset of the packages provided by the complete system. In addition to the data files associated with each user group, there exist some files shared by all user groups.

2.6.1 The Use of APL as a Hardware Design Language (HDL) : Syntax and Semantics

The use of APL for the system level (and to some extent the functional level) to represent and to analyse mathematical & algorithmic concepts presents little difficulties (e.g. see section 2.6.2 and chapter 3). At the circuit level, there have been careful studies and developments by P. Penfield (39) and R. Spence (59). In this section, we shall consider the use of APL as a hardware design language (HDL) : to clarify the application of APL at the functional, register-transfer and gate/logic levels. In order to satisfy as much as the postulates given in section 2.5.1 without violating the syntax of APL, we shall use the following devices:

1) Introduction of a dummy function for "execution-time" specifications;

2) Handling of "declarations" and "directives" by a "pre-processor"; which permits the use of the APL comment syntax for specifying declarations and directives, while this allows us the freedom of choosing the pre-processor language syntax.

On the other hand, the approach necessitates the allowance for a particular deviation from the APL syntax; but, as illustrated below, this can be absorbed and handled by the pre-processor. The resulting "version" of APL maybe considred as another language; but we choose to reference this subset of APL as just APL. Hence any future extension and enhancement to the APL language that are suitable for the hardware description will automatically extend the power of the subset. However, as only a subset (with some augmentation; see below) of APL is used, for the purpose of the discussion within this section, we shall identify this subset as APL/HDL (A Hardware Design Language expressed in APL).

It should be noted that the syntax and semantics of a computer language can usually be described in a precise way by a set of syntax equations which give the rules for recognizing and classifying legal symbol strings. However, the description of a language requires another language and this process of layers of precise description may progress indefinitely. We avoid this by resorting to an informal natural-language description called BNF (short for Backus-Nauer Form) and a note on the BNF is included in appendix 3. The description of the subset for hardware descriptions takes the following form:

A) Data types - Constants & Variables;
B) Expressions & Operators;
C) Statements;
D) Processes & Functions;
and E) Time Specifications.

A) Data Types - Constants & Variables : The data type in APL/HDL corresponding to that in APL is shown below.

DATA TYPE		
APL Language	APL/HDL	
	Hardware	Occur in APL/HDL Functions
Scalar	Flipflop	Boolean Variable
Vector	Register	Input or output vector of a network
Matrix	Memory	Domain matrix, code matrix, set of input vectors or set of output vectors of a combinational network

There is no logical need to distinguish between a register and a flipflop (which is a register of length one). The distinction is dictated by the APL language; where a vector with only one component is interpreted as a scalar unless especially declared as a vector (by ravelling the scalar; i.e. A ← 1 is scalar; but A←,1 is

a one element vector). Consequently, we have the following constant type:

<Scalar constant> ::= 0 | 1

<Vector constant> ::= <Boolean vector> | <encode vector>

It should be noted that in the hardware description context, we are concerned more with Boolean quantities (i.e. 0s and 1s). Special vector constants can be created with APL operators, such as the dyadic ρ (or RESHAPE, see appendix 1) operator. Encode vectors allow us to denote a boolean vector more concisely by writing the APL encode symbol ⊤ followed by an integer decimal number, the mapping of which into the binary number system yields the boolean vector.

Variables : denote in an "APL/HDL process" (discussed below); the content of registers, memories, and busses. A variable may also represent a signal (a semaphore). In "APL/HDL functions" (see below), variables represent the inputs and outputs of combinational networks.

All variables with the exception of bus variables can be indexed. Parts of variables may be "masked out" by applying the TAKE (↑), DROP (↓), or INDEXING operators. A memory cell content is given as a row of the matrix *M* representing the memory. Thus we have the notation *M*[⊥*MA*;] for the content of the cell whose binary address is given as the content of the address register *MA*.

B) Expressions & Operators : Constants, variables, operators, function calls, and parentheses can be combined to form expressions according to the syntactic rules of the APL language. The rules for "compound expressions" are also those given by APL, that is, execution is performed from "right to left" unless otherwise specified by parentheses. No precedence rules are given.

APL/HDL operators are all APL primitive functions, composite functions (see appendix 1), and mixed functions which are meaningful for the description of hardware behaviour and structure. This normally excludes all operators which cannot be applied to the data type boolean, that is, all primitive functions except the logical and relational operations.

C) Statements : A BNF description of the APL/HDL statement types is shown below.

<statement> ::= <process statement> | <function statement>

<process statement>::=<transfer statement>|<branch statement>|
<declaration statement>|<process directive>

<function statement> ::= <assignment statement>|<indexing statement>|
<branch statement>|<declaration statement>

Note that the statement syntax is identical to that of the APL; with one exception which we shall discuss in the context of "transfer statement". The semantics of the statement types are individually discussed below.

1) Transfer statements : the syntactic form of an APL/HDL statement is that of the assignment statement in APL defined functions, i.e., a label is followed by the assignment symbol ← and is followed by an expression. The meaning is then,

Right hand side expression : Denotes the source of a transfer, consisting either of the content of a register on which a monadic operation is performed (including the identity operation which is not explicitly denoted) or of the contents of several registers connected by operators.

Assignment symbol : Denotes a transfer.

Left hand side name : Denotes the destination element of the transfer.

A Syntactic Deviation : There is one case where we deem a syntactic deviation from APL as being inevitable, namely in the case that two (or more) registers shall be concatenated, that is, treated as if they were one register. Such an operation can be expressed by the APL catenation operator, connecting the variables which represent the registers. Such a register concatenation may be the source or destination of a transfer, i.e., in the latter case the destination operator occurs on the left hand side of a transfer statement. APL does not, however, permit operations (execept indexing) on the left hand side of an assignment statement.

One possible solution of this problem is the introduction of intermediate variables. However, it is desirable that any variable introduced must allow a hardware interpretation; the introduction of intermediate variables, in this case, only serves the purpose of obeying the syntactic rule of APL. Hence, from the hardware interpretation view point, one should avoid using intermediate variables as well as time variables.

On the other hand, we can allow the catenation as the only operation on the left hand side of a transfer statement and handle this special case in the pre-processor. This implies that the pre-processor has to check each statement for such a construct. When a left hand side catenation is recognized, the statement then has to be decomposed into the respective number of transfer statements; one for each variable in the catenation, in order to obtain an executable APL program. As a simple example, let us assume that two catenated registers, A & B, are the destination of a transfer from a register C; with

length (C) = length (A) + length (B).

In this case, the APL/HDL statement is,

$$A \ , \ B \leftarrow C$$

and the pre-processor has to replace it by the following two statements,

$$A \leftarrow (\rho A) \uparrow C$$

$$B \leftarrow (-\rho B) \uparrow C$$

2) Branching statements : Branch statements have the syntactic form of APL. The same holds for labels used as branch destinations. There are many techniques for branching in APL, however, these techniques may be summarized as below.

Example statement	Meaning
→ 0	Exit from program
→ *L1*	Branch to (statement with label) L1
→ *NULL*	No branch (or skip jump); null can be obtained in several ways, e.g. "iota zero"
→ *(ST)/L1*	If condition *ST* holds, branch to label L1, otherwise continue to next statement
	Example of "multiple-branch"
→ *(L1,L2,L3)[ST]*	Branch to L1 or L2 or L3 for condition *ST* = 1 or 2 or 3
→ *(BT)/L1,L2,L3*	Condition *BT* is a disjoint event, so that only a logical one occurring in one of the position. A branch is made to the corresponding label (i.e. any one of the label to the right of the compression operator, /)
→ *I ϕ L1,L2,L3*	(I) is a counter, specifying the number of left rotations for the labels which appear on the right of the rotate operator. A branch is then to the left-most label.

For the skip-jump branching, any APL mechanism which creates alternatively an empty vector (or a null) or a number can be used in the usual way. For multibranching, we recommend the use of either rotation or compression, as these forms are very easily readable and understandable, especially in connection with the boolean expressions by which usually the branch conditions are specified. Of course, the on-set of the boolean expressions must be mutually exclusive (disjoint) and collectively exhaustive, yielding one and only one branch destination at a time. Since the branch conditions

are usually obtained from a transition table, a state diagram, or a transition matrix (all equivalent forms) describing the process, this condition is by definition satisfied, and any violation indicates an error.

3) Declarations & Directives : "Declarations" and "Directives" are given in the form of APL comments, i.e. they must begin with the "lamp" symbol (⍝). A special symbol following the lamp — for which we suggest the APL symbol Δ — indicates to the pre-processor that the comment contains a message for it. In all other cases, it is truly a comment that can be discarded. In the case of a message, the characters following the Δ form a keyword that specifies to the pre-processsor the type of declaration or directive. For examples,

Ex. 1 : A process may be declared by,

```
⍝Δ PROCESS MUL: INPUT(X,Y); OUTPUT(Z); SIGNAL(S1)
```

Ex. 2 : Declaration of a 16-bit register consisting of Set-Clear (SC) flip-flops may be,

```
⍝Δ DECLARE AC := SC(16)
```

Examples for directives are:

```
⍝Δ INITIATE ( < LIST OF PROCESSOR NAMES > )
⍝Δ WAIT ( < EXPRESSION > )
```

It should be noted that these examples may be treated as suggestions. As remarked at the beginning of this section (sect. 2.6.1), the messages may take any syntactic form as one chooses; in particular if one is writing ones own pre-processor.

4) Assignment statements and Indexing statements :

Assignment statements occur only in APL/HDL functions. Their syntactic form is that of the APL assignment. Their hardware interpretation is that of connections between logical elements or arrays of logical elements. Functions for combinational networks with more than one output are described by matrices and matrix operations (45); iterative networks may be described by functions which contain loops. To this end, we need indexing statements which differ from the assignment statements. The difference is not a syntactical one; rather, there is no hardware interpretation for the indexing statements. Therefore, indexing statements have to be declared to the pre-processor.

D) Processes & Functions : APL has only one procedure type, called defined function. Defined functions are "external" (in the PL/1 sense) and can be used as function-type as well as statement-type procedure (or a combination of both). As a function-type procedure ("defined function with explicit result"), it can have none, one, or two arguments ("niladic", "monadic", or "dyadic" function -- see appendix 1). Note that the arguments may be vectors or matrices.

A process, on the other hand, may have an arbitrary number of inputs, outputs, and signals (semaphores). The signals announce the state the process is in (e.g. "dormant" or "busy") or the completion of certain phases of the process. Thus, we choose for a process the APL form of a defined niladic function without explicit result, that is, of a procedure which receives its parameters solely through the environment it has in common with other processes and which acts solely through changes of the environment. This implies that all such parameters must be global to the whole program (i.e. the main process). Similar situation applies to the content of registers or busses, i.e. the variables, representing central storing or transporting facilities, are shared by all processes. In terms of a hardware interpretation, a process can be equated with a module in the system which has its own control unit. There exists a hierarchy between processes, as there is usually a main process, representing the "master control", which invokes other asynchronous processes (which may totally or partly be concurrent).

In addition to the use of global variables, a process may be directed by certain internal control signals. The scope of names of these signals can be controlled throughout a nesting of subprocesses in the usual way as provided by the APL defined function mechanism. For readability of a program, and also for error detection and prevention, it is a good practice to request a declaration of all parameters in the process declaration heading. A declaration must be given in order to make a defined function and to represent an APL/HDL process (Examples one & two in the above illustrate such declaration.).

The pre-processor may then check the variable names listed in the declaration against the names of variables declared as local in the function header. No variable may occur in both lists. Therefore, the definition of the main program as a process is the only way to introduce labels and branch instructions as well as scope rules into the main program.

In connection with assignment and indexing statements, APL/HDL permits an arbitrary number of statements to be written on one line. This feature can be used for the declaration of indexing statements (as contrasted to assignment statements) in APL/HDL functions. Therefore, a line containing an indexing statement (or several such statements) may begin with the comment:

⍝Δ *I* < *STATEMENT SEPARATOR* >

While assignment statements (the normal case, as in APL) are declared by default.

E) Time Specifications : To avoid the introduction of time variables, we adopt from other hardware design languages the convention that each line in a program represents a process performed by hardware and executed within one clock period of the basic clock of the machine (being simulated), unless specified otherwise. We call the period of the basic clock an "elementary delay", or, in short, a delay, and a process executed within one delay is a pico-operation. As different pico-operations may be simultaneously executed, we must permit several statements representing pico-operations to be written on one line (of the simulation program). Such a possibility is not given in most existing APL implementations, such as APL/360, APLUM, CDC APL.2. However, extended version of APL, such as APL-PLUS, IP-Sharp APL, and Xerox APL, (see sections 2.5.3 & 2.5.4) do provide this facility. From APL-PLUS, we adopt the use of the "diamond" symbol[1] ◇ as a separator for statements written on the same line. On the other hand, if the version of APL being adopted does not recognize such a syntactic form, it becomes one of the tasks of the pre-processor to rearrange the statements of a multi-statement line into a sequence of single-statement lines. This can easily be done by virtue of the APL editing facilities.[2]

A register transfer consumes always one delay (by definition). The execution time of logical functions may be an integer number of delays. In this case, it is a simple and convenient scheme to define the function without any time considerations and specify the execution time in connection with the function call. In APL/HDL, this is done by introducing a dyadic dummy function called DELAY. The syntactic form is :

⟨integer⟩ DELAY ⟨expression⟩.

---

[1] Where the ◇ symbol is not available on the keyboard (appendix one), the "overstrike" ⌹ or ⌸ has the same effect. Xerox APL (or Atkins APL) has instead chosen ; as the statement separator.

[2] This can be done by APL using the editing facility of the processor in the immediate execution mode. However, the "canonical representation" of a function and the powerful string-search and manipulation facilities (sections 2.5.3 & 2.5.4) together provide editing capabilities under an APL program control. This,of course, requires the writing of such utility function.

The integer is non-negative and it specifies the number of delays required for the execution of the expression. For example, for an adder network requiring 17 delays, we write :

```
17  DELAY  X  ADD  Y
```

Processes may have synchronous and asynchronous parts. An asynchronous part is obtained by invocation of subprocesses which have their own internal control. Invoked processes can call other processes but not the invoking process; i.e. we have not adopted the "recursive" capability of APL. A process may simultaneously invoke several subprocesses. For the invocation and synchronization of subprocess, the two directives

```
⍝Δ 'INITIATE    < list of process names >
⍝Δ  WAIT        < expression >
```

are used. They provide a general and powerful tool for process synchronization by an arbitrary number of user-defined semaphores. We briefly explain the significance of each below.

INITIATE is necessary to begin the execution of parallel asynchronous streams of statements for which the total time requirement is unknown at the time of initiation. If the time requirement is known, then the sequence may be executed as additional statements on each line of the evoking process, obviating *INITIATE* Invoked processes communicate with invoking processes through signals. A signal may be zero (low) or one (high). We will consider altering the setting form high to low to constitute an event. All processes that have become blocked waiting for the event are notified of the occurrance and are given the opportunity to continue. Processes become blocked by having executed a *WAIT* statement. The ability to progress following an event is dependent on the total logical expression in the *WAIT* statement which caused the blockage.

As an example, we consider the following segment of a microprogram; the concurrent operations are schematically depicted in figure 13. The first 13 bits of the accumulator (AC) are transferred to the program counter (PC). Subsequently, two concurrent processes are initiated, one process performing a memory fetch while the other process adds the contents of the indirect address register (IA) to that of the accumulator,

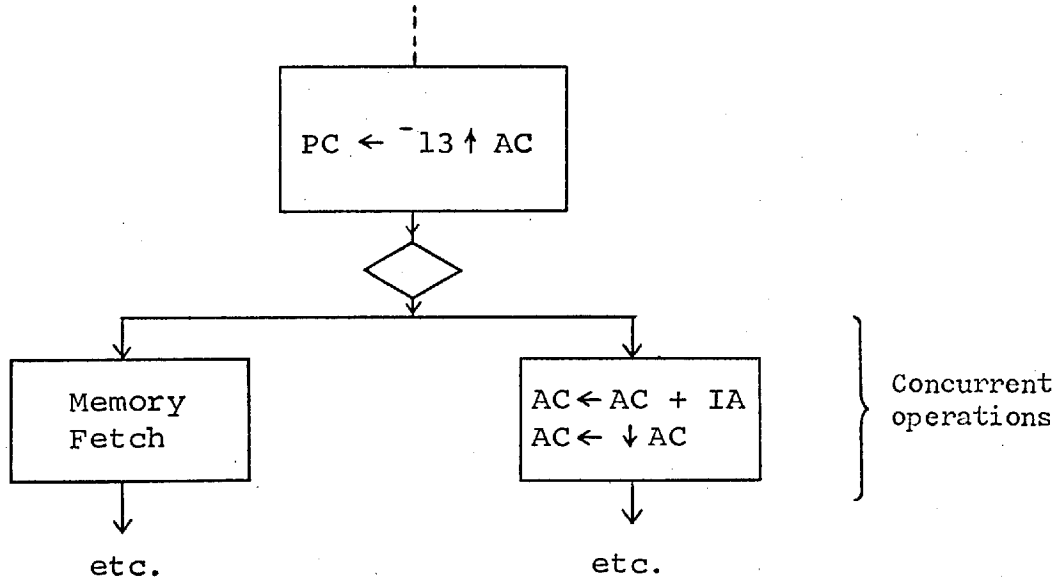


Fig. 13 : A Schematic of concurrent process (segment of a microprogram)

and the result is left-rotated. Therefore, the respective APL/HDL sequence in the main process may read,

```
[1]   PC←⁻13↑AC
[2]   A Δ  INITIATE(A,B)
[3]   A Δ  WAIT(S1∧S2)
```

The declarations of the processes A and B might then take the following forms.

```
   ∇ A
[1] A Δ PROCESS A; INPUT(PC); OUTPUT(MD)
        SIGNAL(S1)
[2]   S1←1
[3]   MA←PC
[4] A Δ INITIATE(M)
[5] A Δ WAIT(R)
[6]   S1← 0
    ∇
```

```
   ∇ B
[1] AΔ PROCESS B; INPUT(AC,IA)
       OUTPUT(AC); SIGNAL(S2)
[2] S2←1
[3] AΔ INITIATE(ADD)
[4] AΔ WAIT(ADDS)
[5] AC←↓AC
[6] S2← 0
    ∇
```

Several remarks are in order. First, INITIATE is by definition a 0 DELAY statement. Second, we have assumed that ADD has a signal ADDS and M (memory) has a signal R. The memory process may have the following form.

```
     ∇M ; MJ
[1]  ⍝Δ PROCESS M; INPUT(MA); OUTPUT(MD); SIGNAL(R,C)
[2]  R ← 1
[3]  C ← 1
[4]  MD ← MJ[;⊥MA]
[5]  R ← 3  DELAY  0
[6]  C ← 4  DELAY  0
```

It would be more correct, we believe, if process A recognized its need to be concerned about its ability to invoke M and execute WAIT(C) followed by INITIATE(M), instead of simply INITIATE(M).

Since the structure of ADD parallels that of A, B and M, we do not include its definition here. Observe, however, that the structure is quite general in that each process may declare and generate as many signals as is necessary. Notice, too, that signals are only altered within the process in which they are declared, although they may be sensed in any process.

Several points remain to be discussed; including the consideration of discrete event simulation, and the meaning of "step-wise refinement". In section 2.6.3, we shall discuss these aspects in conjunction with the proposal of a design system based on the APL/HDL and the implementation of the CAD environment for our "Top-down Design Approach". In the next section, we shall illustrate the design approach with some examples.

2.6.2 A Top-down Design Approach Illustrated by Examples

In the development of the Top-down Design Approach and the search for suitable computer-aids, we have attempted to consider as many points as we are aware of and worked through them with whichever tools available. These tools have mainly been computer languages, analysis programs and simulation programs or packages that are either evolved or accessible to us (e.g. CDL & DLS). One of the main motivations has been to devise techniques whereby digital signal processing designers and engineers may be assisted. In the course of time, we have been aiming to arrive at some standard techniques and computer programs that will attain the intended goal. Furthermore, the findings of the investigation should result a collection or library of tools for future applications and explorations. A selection of these tools is listed in table 4; while two major simulation packages (viz. DISPIC and SIMLOGS) developed are individually presented in chapters 3 and 4. Source coding of programs for those listed in table 4 and other additional developments are included in the respective appendices. Some non-APL programs are indicated on table 4; these were developed prior to the final decision of using APL as the main design tool. The conversion of these programs to APL (if necessary) should, however, present no major difficulties; bearing in mind the power and capability of APL.

However, the mixture of programming languages has indicated the need of adopting the most suitable computer language for the design task (section 2.5). On the other hand, it also reveals the importance of devising the capability of integration and interfacing among the mixture of computer languages and packages. The emphasis is on the ease of assimilation and "user transparency" in the eventual CAD design system. We shall be dicussing this aspect in section 2.6.3. In this section, we shall attempt to illustrate the Top-down Design Approach by examples.

Function Group	Function Names	Computer Language	Applications
FIRGP	FIRN, FCONTINUE, INFIR, OUTFIRN, PLOTF, PLOT, AND	APL\360	Compute, Tabulate & Plot Time-Response of Nth Order FIR Digital Filter
ELPFIRG	ELPFIR, ECON, EVFIRLP, OUTFIRN, PLOTF, PLOT, AND	-do-	-do- Even Order (Nth) Linear Phase FIR Digital Filter
FREQGP	FREQ, OUTPUT, PLOTFREQ, PLOT, AND	-do-	Compute Frequency Response of a Digital Filter
SIGNALGP	RAN, MEAN, SAMPLE	-do-	Generates a Randon Signal with Normal Distribution
RDUND	SINFA, INBIN, FDB, DEBINSM, SMONES, SMTWOS, ONESM TWOSM, FSMDEC, SMBINDEC, ONESDEC, TWOSDEC, DECONES, DECTWOS	-do-	SIGN-MAGNITUDE/DECIMAL ONE'S or TWO'S COMPLEMENT/DECIMAL DECIMAL/SIGN-MAGNITUDE, ONE'S & TWO'S COMPLEMENTS * * * * * (ROUNDOFF)
DIGITAL	ADDER, CARRYSAVE, DADD, BADD, LADD	-do-	RT-DESCRIPTIONS of ADDER/MULTIPLIER ADDER Types Simulation
BCOMP8GP	BCOMP8, ADD, MEMORY ASSEMBLER, CODE, BINARY, NUMERIC, WITH, ADDRESS, CHAR	-do-	BCOMP8 - 8-Bit Digital Computer ASSEMBLER for BCOMP8
DUMPGP	BCOMP8ΔT, ADD, MEMORY PDUMP MDUMP	-do-	BCOMP8 - with Trace Facility Program DUMP DUMP of Memory Contents
RNSGP	RNS, DROUND, RNSPT, ΔFMT, MODADD, MODMULT	-do	Some Residue Functions
LOGIC	DECODER, MULYIX, BCD, SERPAR, FLOAT, BUS, REGIS	-do-	Some RT Description of Logic Hardware
LOGMIN	/	FORTRAN IV	Logic Minimization Program
DLS	DIGITAL LOGIC SIMULATOR	Extended FORTRAN IV (CDC 6000 SERIES COMPUTER)	Simulation of Logic at the Gate/Logic Level

Table 4 : A Selection of Computer Aids for the Top-down Design Approach

A) System Level

At this level, a digital filter designer is assumed to have performed some filter design using techniques such as those reported in references 1 through 8. Design parameters (section 2.1.2), such as filter structures & coefficient quantization, that affect the performance of a digital filter are inter-related and often lead to complex problems. Techniques developed to meet problems associated are often too complex for manual solutions and beyond immediate assimilation by practising engineers. On the other hand, this type of problems are amenable to computer solutions and indeed much effort has been spent for this purpose (references 66 to 70).[1] Having arrived at a transfer characteristic, it is desirable to obtain a time-domain simulation of the filter. The reasons for this are many fold; simulation of the eventual implementation will provide much confidence of the design (even though, the design could have been achieved by CAD)[2]. Furthermore, the simulation also provides some test data for the eventual filter if implemented as a processor.

A group of APL functions, FIRGP (table 4), consisting of FIRN, FCONTINUE, OUTFIRN and PLOTF (see appendix 4 for these and associated functions) have been written to compute the time response of a N-th order FIR digital filter.[3] FIRGP accepts the coefficients of a N-th order FIR and a signal sequence as input data. The time response computed (i.e. output data) may either be tabulated or plotted. A Fast Fourier Transform (FFT, appendix 6) has been written in APL and this may be used to compute the corresponding frequency response from the time sequence computed by the FIRGP group. In addition, the FFT function also permits the inverse transform (i.e. a time domain may be obtained from the frequency domain specification). Alternatively, the frequency response of the filter

---

1 Computer programs are mostly written in FORTRAN IV and PL/1.

2 Afterall, a construction of a prototype may be considered as a hardware simulation.

3 An example of IIR simulation is considered in chapter 3 in conjunction with DISPIC's.

may be obtained directly using the function group called FREQGP (appendix 4). Illustrations (i.e. computer runs and printouts) for these groups of functions are included in the appendix.

It should be noted that all function groups assume the accuracy of computation defined by the implementation of a particular APL processor or the computer on which APL is supported. The effect of rounding errors due to the choice of finite word-length of the filter and various representation of number systems (e.g. sign-magnitude, one's complement and two's complement representations) may be taken into account in the simulation. In this connection, the operators (i.e. addition, subtraction & multiplication) for the function group, e.g. FIRGP and FREQGP, may be replaced by binary operators. A set of functions for binary arithmetic have been designed for this purpose and are included in appendix 7. Further developments on the algorithmic representation of computer arithmetic have been conducted and the results are documented in appendix 8.

B) Functional Level

Having considered the mathematical representation (e.g. transfer function) and the algorithms (e.g. filter structure) chosen for a digital filter, the designer is assumed to implement the filter as a programmable machine. On the one hand, the availability of microprocessors would have provided a solution. On the other hand, the variety of microprocessor families, and techniques like microprogramming have provided a multitude of possible solutions; some aids would be desirable to assist the designer. Simulators (either in software or hardware form) are often provided by manufacturers for this purpose; yet, the choice of these aids often presupposes the adoption of a particular product. It would therefore be desirable for a digital processor designer to be able to specify his own architecture which may be simulated prior to any technological commitment. While a subsequent computer aid may be used to convert

the requirements, such as instruction set chosen, into an existing microprocessor; while special operations may be designed as micro-programs.

As a first step[1], we have simulated a simple 8-bit computer, BCOMP8; details of which are reported in appendix 9. BCOMP8 is a single address machine. The arithmetic unit of BCOMP8 consists of a simple adder; while multiplication is required to be realised by programming the machine with the instruction set provided. The architecture is simple, yet the objective of the exercise was to investigate the techniques required for the simulation of a programmable machine using APL. It also demonstrates that through the simulation one may examine the sufficiency of an instruction set (of the machine, or target machine, under simulation) and the internal structure of an arithmetic unit.

The collection of APL functions written for BCOMP8 consists, essentially, of two groups. ASSEMBLER accepts a BCOMP8 program (which is expressed in mnemonic codes) and translates it into binary codes which may subsequently be loaded into the memory of the target machine. Function group, BCOMP8, simulates the operation of the machine and the execution of it is in accordance with the contents of the memory. To examine the states of the target machine during simulation, an additional group of functions, DUMPGP, was developed. It is composed of MDUMP, PDUMP, BCOMP8TΔ and DIGIT functions. DUMPGP also has the capability of reconverting the binary codes (i.e. the contents of the memory in BCOMP8) back to mnemonic codes. Furthermore, the contents of the memory and registers may be optionally printed during the execution of the BCOMP8 machine.

---

1 Subsequent work in this area is DISPIC's (chapter 3).

C) Register-Transfer Level

Multiplication is an important arithmetic operation of digital filtering. It is also a time consuming operation; since multiplication is performed with addition and shifting operations, any increase in speed of these can contribute to faster computations. Therefore, in a programmable machine, such that indicated by BCOMP8 in the above, a hardware multiplier is often essential as an element of the arithmetic unit. In choosing a suitable multiplier, the designer can be assisted by the speed/complexity study such as that reported in reference 14; in which serial, serial-parallel, parallel and iterative or cellular array multipliers were studied. However, a register-transfer simulation of a particular multiplier is fruitful for a detailed investigation of the internal structure and data flow of the implementation. In particular, when the product is to be truncated, or rounded, a simulation of the multiplier is useful to determine whether the performance is acceptable for the digital filter application.

As an exercise, a carry-save multiplier (figure 14) is described at the register-transfer level and illustrated with a simulation example in figure 15. The technique employed in the carry-save multiplier is described below. Since multiplication requires summing several numbers, an economy in time is possible for the overall carry propagation as compared with the sum of carry propagation times for individual additions. To understand how this economy is obtained, consider the example of figure 16 for decimal radix (for ease of reading). Addition is done in each cycle on three numbers -- the partial sum, one of the numbers to be added, and a carry number derived from the previous addition. The addition in a given cycle treats each digit independent of all others -- although a carry is obtained in each digit position in the usual way. This carry is not immediately applied to the next higher order; instead it is saved or stored in a form suitable for addition in the next cycle. Note that no carry propagation takes place in any addition but the last. Using the carry-save principle, multiplication time is determined by the time for carry propagation for a single addition and the delay for shift logic and single digit addition.

Other efforts in using APL for the architectural description and simulation at the register-transfer (RT) level are documentated in appendix 11.

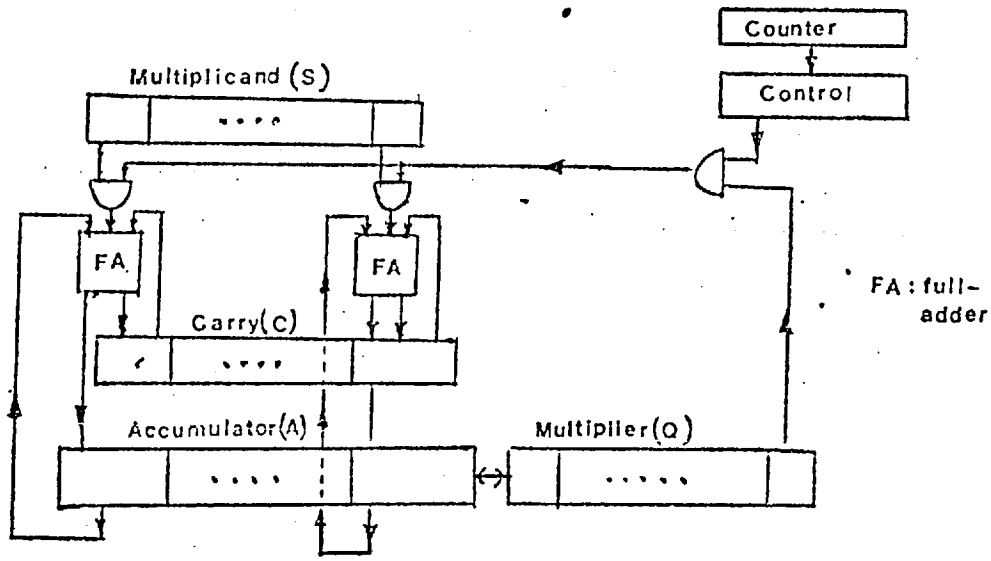


Fig. 14 : A Schematic Diagram of a Carry-Save Multiplier

```
     ∇ DECMSAVE
[1]   ⍝  DEFINING THE REGISTERS FOR CARRYSAVE
[2]    'ENTER REGISTER LENGTH :'
[3]    B←⎕
[4]    'ENTER NO. OF BITS IN SHIFT COUNTER :'
[5]    M←⎕
     ∇
```

```
     ∇ CARRYSAVE
[1]   ⍝  SIMULATION OF A SERIAL-PARALLEL CARRYSAVE MULTIPLIER
[2]   ⍝
[3]   ⍝Δ PROCESS CARRYSAVE : INPUT(S,Q); OUTPUT(A-Q)
[4]   ⍝Δ REGISTERS : S(B); Q(B); A(B); C(B)
[5]   ⍝ΔC,A ※  C←A←Bρ0
[6]   ⍝ΔK   ※  K←(Mρ2)⊤B
[7] LP: T←A≠(S∧Q[B-1])≠C
[8]    C←(A∧C)∨(A∨C)∧(S∧Q[B-1])
[9]    A←T
[10]   Q←(¯1↓A),¯1↓Q
[11]   A←0,¯1↓A
[12]  ⍝ΔK   ※  K←(Mρ2)⊤¯1+2⊤K
[13]   →(0≠∨/K)/LP
[14]   A ←(Bρ2)⊤(2⊥A)+2⊥C;'  ';Q
     ∇
```

```
     ⍝ LEGEND
     ⍝   0-ORIGIN INDEXING
     ⍝   B ↔ NO. OF BITS IN MULTIPLIER
     ⍝   K ↔ COUNTER OF REMAINING SHIFTS
     ⍝   Q ↔ REGISTER ORIGINALLY HOLDING MULTIPLIER
     ⍝       (RIGHT REGISTER; SEE FIGURE 14)
     ⍝   S ↔ REGISTER ORIGINALLY HOLDING MULTIPLICAND
     ⍝   A ↔ ACCUMULATOR (LEFT REGISTER; SEE FIG.14)
     ⍝   C ↔ CARRY REGISTER
```

Note :

1. Statement [14] is more correctly expressed as *A,Q ←(A BADD C), Q* where *BADD* is a parallel binary adder; present form is a convenience only.
2. Multiplier (Q) and multiplicand (S) need be defined before executing *CARRYSAVE*

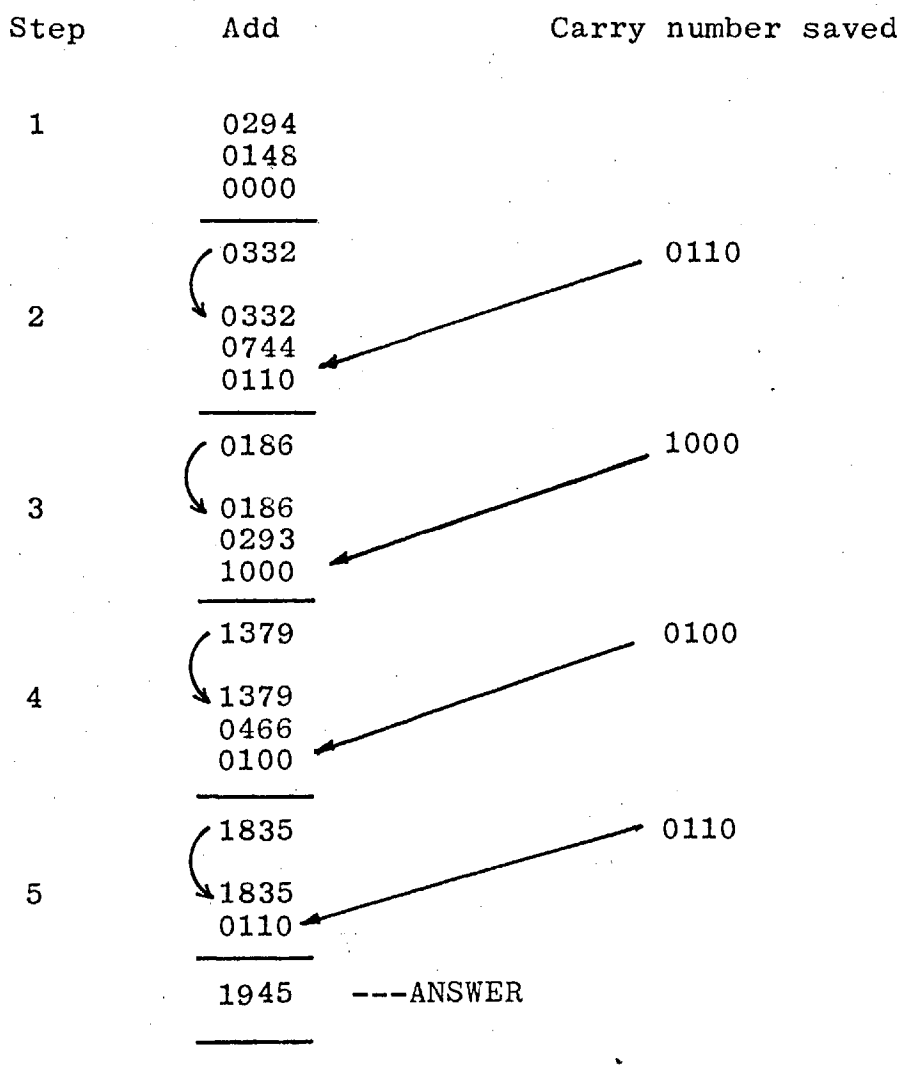
```
     DECMSAVE
ENTER REGISTER LENGTH :
⎕:
     5
ENTER NO. OF BITS IN SHIFT
REGISTER :
⎕:
     8
     S←1 1 0 0 1
     Q←1 0 0 0 1
```

```
     CARRYSAVE
0 1 1 0 1 0 1 0 0 1
```

Fig. 15 : RT Description & Simulation of a Carry-Save Multiplier

```
 0294
 0148
 0744
 0293
 0466
------
 1945
```

(a) An original problem of summing several numbers



(b) Example in decimal radix of carry-save principle

Fig. 16 : An illustration of the Carry-save Principle

D) Gate/Logic Level

At this level, detailed design of the filter is conducted by introducing technological concepts. This leads to logic synthesis and minimization of each part of the digital filter. The designer is concerned with system parameters (section 2.1.2), such as logic hazards, maintenance and cost requirements of the components (e.g. arithmetic unit, storage, etc.). If conventionally available modules are employed, the amount of design at this level is reduced. On the other hand, for special circuit requirements, a custom built approach is required. Therefore, the circuit designed must be thoroughly analyzed before integrated circuits (or chips) designs are committed to fabrication. In using MSI, LSI, programmed logic arrays (PLA) or microprocessors, a return to the RT level and augmented by analysis at this level is sometime necessary.

For the purpose of logic synthesis and minimization, we have attempted to develop a logic minimization program (called LOGMIN) and details of this are included in appendix 2. Illustrations on the use of LOGMIN are also included.

As a computer-aid towards logic verification and testing, we have initially adopted the "Digital Logic Simulator (DLS) in our investigation. DLS (60) is a computer program implemented with the "FORTRAN Extended Language" of the Control Data 6000 series of computers. The program simulates synchronous and asynchronous networks of digital logic elements. The inputs to DLS consists of (1) circuit elements and their interconnections, (2) circuit parameters, and (3) input sequences. The circuit may then be "exercised" (i.e. simulated) with the performance of the circuit printed (as computer outputs) as either timing diagrams (i.e. truth table) or graphs. By simulating a circuit in this way, the designer may be assisted to identify possible design faults (either conceptual or technological) before the circuit is committed to a hardware implementation. As an example, a divide-by-5 counter is simulated with the DLS. Figure 17 shows the logic diagram

of the counter while figure 18 shows the description (with the DLS syntax) and simulation of it using the DLS simulator.

DLS though a useful tool, yet it lacks many desirable features, such as a "macro-description" of the circuit whereby a designer may successively construct sub-circuits for a more complex circuit or functional block. Furthermore, any circuit defined in this way should be capable of being saved in a library for future applications and explorations in larger and more complex circuit. On the other hand, the simulation package should be both portable and adaptive (to even special cases) for a wider range of applications. A designer who is also skilful in computer programming may introduce into the package "primitives" as standard elements in addition to basic elements like AND, NAND, NOR, & Flip-flops. With these features in mind, we have investigated and developed SIMLOGS which will be presented in chapter 4.

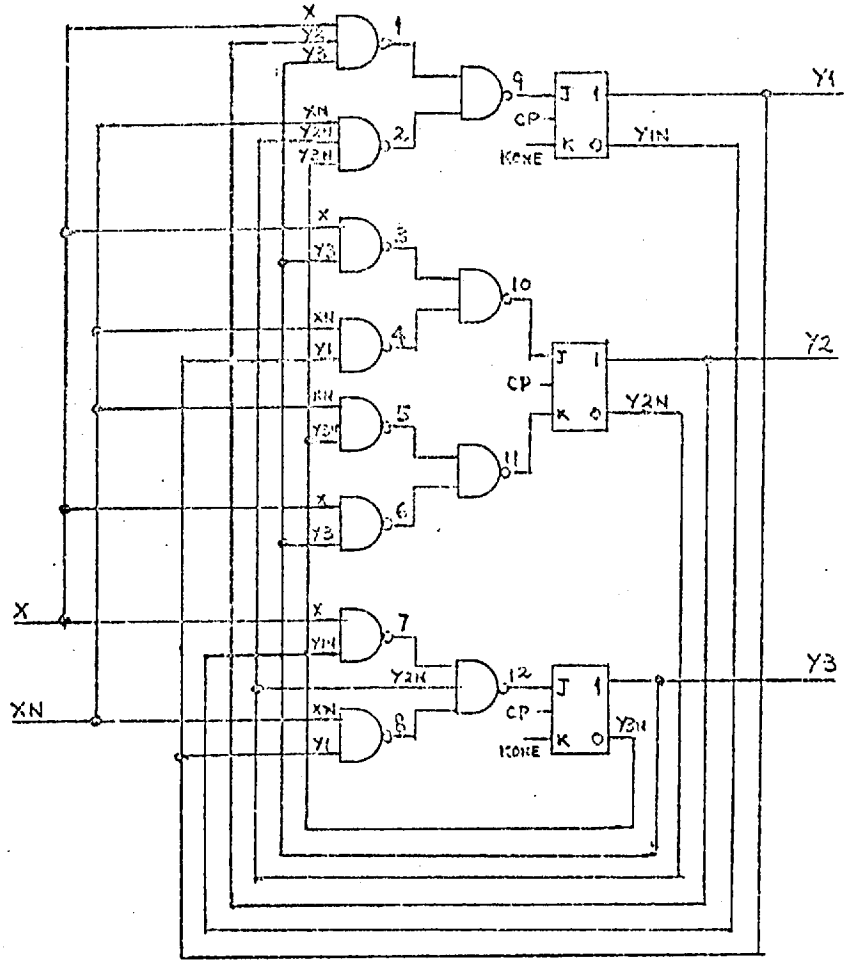


Fig. 17 : Logic Diagram of a Divide-by-5 (modulo-5) Up/Down Counter

```
£DLS MOD-5 UP/DOWN COUNTER
PATCH
*  SPECIAL LOGIC ELEMENTS AND INPUT / OUTPUT
       INPUT    X    XN
       PRINT    X    CP    9     10     11     12     Y1     Y2     Y3     KONE
KONE SWITCH
*
*  COMBINATIONAL ELEMENTS
1      NAND     X    Y2    Y3
2      NAND     XN   Y2N   Y3N
3      NAND     X    Y3
4      NAND     XN   Y1
5      NAND     XN   Y3N
6      NAND     X    Y3
7      NAND     X    Y1N
8      NAND     Y1   XN
9      NAND     1    2
10     NAND     3    4
11     NAND     5    6
12     NAND     7    8     Y2N
*
*  FLIP-FLOP ELEMENTS
Y1     JKFF     9    KONE  CP              Y1N
Y2     JKFF     10   11    CP              Y2N
Y3     JKFF     12   KONE  CP              Y3N
*
*  SPECIAL LOGIC ELEMENTS
CP     CLOCK
*
£MODE SYNC 5
£INITIAL CONDITIONS
*  PARAMETER OF INITIAL CONDITION
KONE   1
*
£CLOCK
CP           10     1
*
£RUN
```

(a) A DLS Description

TIMING TABLE - MOD-5 UP/DOWN COUNTER

	CP		10		12		Y2		KONE
TIME	X	9		11		Y1		Y3	
0	1	1	0	0	0	1	0	0	0	1
1	1	1	0	1	1	1	0	0	1	1
2	1	1	0	0	0	1	0	1	0	1
3	1	1	1	1	1	1	0	1	1	1
4	1	1	0	0	0	0	1	0	0	1
5	1	1	0	0	0	1	0	0	0	1
6	0	1	0	0	0	0	0	0	1	1
7	0	1	1	0	1	0	0	0	0	1
8	0	1	1	1	1	1	1	0	0	1
9	0	1	0	0	0	1	0	1	1	1
10	0	1	0	0	1	1	0	1	0	1
11	1	1	0	1	1	1	0	0	1	1
12	0	1	0	0	1	1	0	1	0	1
13	1	1	0	1	1	1	0	0	1	1

```
*£RECYCLE
```

(b) DLS Simulation

Fig. 18 : Description & Simulation of the Modulo-five Counter with DLS

Analysis of components, such as gates or flip-flops, at the circuit level has been thoroughly studied by various authors; for example Calahan (62). In the context of APL, P.Penfield, Jr. (39) and R. Spence (59) have conducted many careful studies and developments. Our attentions, however, are focussed at design levels over and above the circuit level; this is not to suggest that circuit level is less important. It will suffice to indicate that circuit level, like the other levels, may be introduced into the "Top-down Design Approach". Techniques for the integration is next to be discussed in section 2.6.3.

2.6.3 A CAD Design System and the User Environment for the Top-down Design Approach

Syntax and semantics of a subset of APL used as a hardware design language was presented in section 2.6.1. While the Top-down Design Approach has been illustrated by examples in section 2.6.2. We shall conclude section 2.6 and this chapter by focussing our attention on the following aspects,

1) Implementation of the Design System,
2) Users' Environment,

for the Top-down Design Approach. In particular, we shall be considering the question of discrete event simulation, and the meaning of "piece-wise refinement" in the Design Approach. In addition, the technique of interfacing CAD programs or packages implemented with non-APL programming languages and the control of a multi-user environment will be discussed.

1) Implementation of the Design System

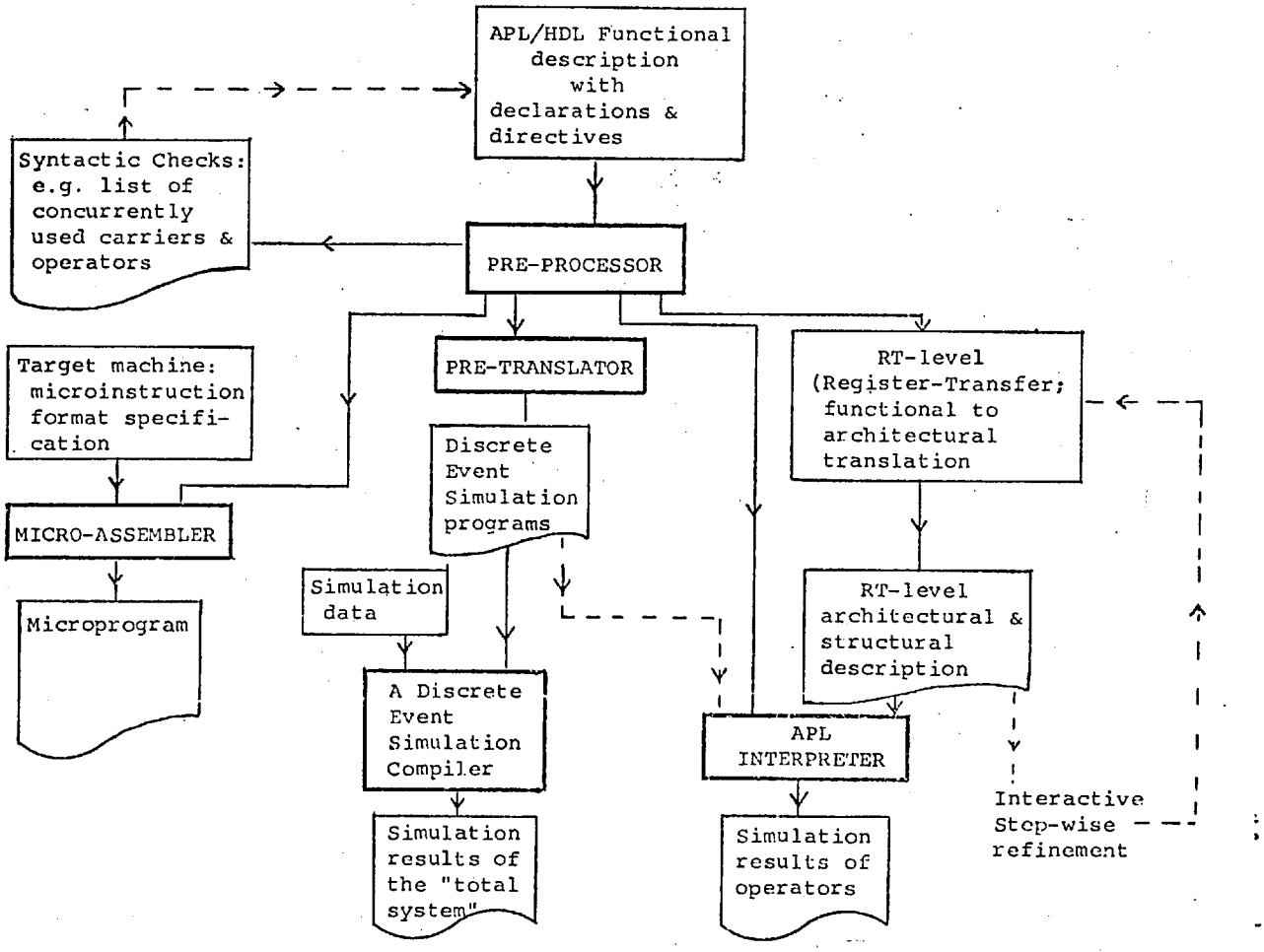


Fig. 19 : A CAD Design System based on a subset of APL (i.e. APL/HDL)

Figure 19 depicts a CAD Design System which is based on a subset of APL (i.e. APL/HDL -- section 2.6.1). According to our major goal, we have one single APL/HDL(functional and RT-level)description and may optionally act as source language for a pre-processor. This pre-processor translates the description into an architectural & structural RT-Level description, a micro-assembler, and a simulator. In any digital system there is a natural dichotomy, namely the distinction between the control part and the controlled part. The controlled part consists of a collection of operators. The control part governs the transfer of information between carriers and the operators. A

functional machine description may consist of a specification of the algorithms associated with each instruction and to which timing and control specifications are added. At this level, the semantics of the assignment symbol ← in APL/HDL is that of a transfer from carrier to carrier. The information may have to pass through a functional unit representing an operator if the content of a carrier is to be transformed in the transfer. Data routine functions (as realized by gating devices, multiplexers, demultiplexers, etc.) are not included in the description, for from an algorithmic point of view they perform only identity operations.

A) Direct Execution vs Discrete Event Simulation

A functional description is sufficient for performing a simulation of the machine behaviour, provided the input/output characteristics of the operators have been specified. Single processes and functions can be simulated by directly and individually executing their APL/HDL description (after pre-processing when necessary) as an APL program. If more than one process is invoked, the functional description will be automatically translated into an executable discrete event simulation program (such as those implemented by GPSS, SIMCRIPT or SIMULA).[1]

To avoid the need of utilizing a variety of programming languages, we have investigated the use of APL for discrete event simulation for our application (indicated with dotted lines in figure 19). This approach requires the capability of (i) syntax analysis of complex expressions (or statements), and (ii) executing co-routines. An exercise of compiling compound expressions (in this case APL expressions) is presented in appendix 10. We shall discuss the techniques of providing co-routines with APL below.

---

[1] At this time of writing, this technique has not yet been thoroughly investigated. However, based on a number of discussions with people experienced with discrete event simulation, such an approach is technically feasible.

Concurrent Execution or Coroutines with APL

Essentially there are two ways of achieving concurrent execution or coroutines with APL. One approach is the use of "shared variables" (section 2.5.2); an example, A Schedule Table, demonstrating the use of shared variables has already been given in figure 11. The other approach is the use of a "central dispatch" function for the control of current execution. These techniques will discussed with examples below.

a) Concurrent execution with shared variables : A system may be viewed as composed of a number of subsystems. These subsystems can have continuous existence and interacting among themselves. Therefore, it is difficult to describe them solely by the results they produce in response to individual presentations of an argument (e.g. an argument of an APL function). Falkoff, Iverson, and Sussenguth (49) coined the term "system program" to refer to functions that describe such continuously-operating subsystems. These system programs are able to communicate among themselves, so that although they behave independently much of the time, the action of one may alter the behaviour of another. The point of contact or "interface" between two system programs appears as a set of variables, shared variables, occurring in both. The continuous functioning of a system can therefore be depicted in the system program which represents it by making the definition an endless loop in which the shared variable takes on new values in response to external events. In this way, the shared variable is in effect the function's argument. Instead of being set once at the time the function is called, it is set or used repeatedly during the endless execution of the function.

As an example, let us consider a system having a collection of concurrent but interacting operations; e.g. a communication switching node capable of handling a number of incoming calls. The definitions (APL functions) below show an outline for the action of the selection mechanism, and the action of an interupt request made by subsystems (or processes) served by the central processing system.

```
      ∇ PROCESSX
[1]    L1:→(~∨/ΔX)/L1
[2]      P←(ρΔX)|P+(PϕΔX(ι1
[3]      QPN P
[4]      ΔX[P]←0
[5]      →L1
      ∇
```

```
      ∇ SELECTOR;S
[1]    L1:→(WAIT≤0)/L1
[2]      S← COND ι1
[3]      ΔX←ΔX∨S∈ιρΔX
[4]      WAIT←WAIT-1
[5]      →L1
      ∇
```

The first line of each of these functions (or definitions) is a "dwell". That is, some condition which is controlled by an "event" external to the program is tested, and if no exception is found, the program repeats the test. It dwells on that line indefinitely until an exception is encountered.

The logical array ΔX is its memory, indicating which action is to be taken. The function *PROCESSX* tests to see whether any option (or interrupt) has been requested; if none, statement 1 is repeated. The rest of the program is entered only after at least one selection has been made.

The variable P is a pointer indicating the number of the last request that was executed. Once any interrupt is detected, P is reset as the index of the next member of ΔX that is ON (i.e. has logical value of 1) following P. Statement [2] of *PROCESSX* shows that a "wrap-around" (i.e. a circular arrangement) stepping to the next member of ΔX that is ON; it is written in 0-origin indexing. Statement [3] of *PROCESSX* calls a function *QPN* whose definition is not shown, but which represents an action to be taken by *PROCESSX* after an interrupt is honoured. When the requested operation is completed, the P-th member of ΔX is turned OFF, and control is returned to statement [1], where the program checks anew to see if any other interrupt is requested (i.e. any member of ΔX is ON).

Meanwhile, the selector has a continuous existence of its own. Its first statement is a dwell until *WAIT* shows an acknowledgement. Then it set a variable S to indicate which of the incoming call has made an

interrupt request. The corresponding member of ΔX is set to logical one, and the register (or counter) *WAIT* is reduced by one.

It should be noted that to implement the above, at least two separate workspaces are required; one workspace holds *PROCESSX* and the other contains *SELECTOR*. Furthermore, as the programs are continuously being executed, the cost of computer time can be quite considerable. An alternative approach is the use of a central dispatch function. This technique is described below.

b) Coroutines within one workspace : The only control transfer mechanisms available on the exit from a defined function is the return to the point of call. This is a feature which APL has in common with most general-purpose languages. Coroutines or concurrent procedures are usually an idiosyncrasy of discrete event simulation languages, such as GPSS, SIMSCRIPT and SIMULA. However, unlike other general-purpose languages, APL enables us to program a control transfer scheme that is in effect equivalent to a coroutine mechanism. The constructs which allow this are the EXECUTE (⍎ or ⍷) function and the information system variable *⎕LC* (see sections 2.5.2 & 2.5.3).

As APL does not allow the transfer of control from one function to another but only back to the invoking function, we have to employ a "central dispatcher" function which calls the coroutines, and to which control is first returned on each exit from a coroutine. The existence of such a dispatcher function can practically be hidden from the user. Another prerequisite is to begin each coroutine with a jump instruction (section 2.6.1) which, on any later resumption of the function execution, transfer control to the right point. Furthermore, a special function must be provided which saves on each exit from a coroutine two information items :

1) the name of the coroutine to be started next,
2) the point of return for a later resumption of the function execution.

All these information items must be stored as global variables with reserved names (underscored to prevent the possibility of name conflicts).

In total, we have the following variables which act as carriers of information between the coroutines and the central dispatcher function:

CRN	(or coroutine name) contains the name (declared as a character string) of the coroutine to be started next;
CRL	(for coroutine return label) contains the label at which the execution of the coroutine to be invoked next shall be resumed;
CEL	(for coroutine exit label) contains the return label of a coroutine at the instant of exit from it.

Furthermore, the dispatch function must maintain a list of the return labels for all coroutines. Therefore, as many global variables as there are coroutines will be created. The (reserved) names of these variables are formed by catenating the letters '*RL*' (for return label) to each coroutine name.

It should be noted that the above is only a suggestion; while other schemes for constructing the internal data structure could be employed. In a system with many coroutines, for example, one could internally name the coroutines by integer numbers and then construct a matrix in which each row is associated with one of the coroutines, containing all the related information. In a system with a small number of coroutines, however, our approach (which is feasible only in APL) is reasonable simple and efficient.

The execution of a set of coroutines will be started by the calling sequence

'*C*' *CALL* '< coroutine name >' *OF* '< coroutine list >'

Here, <coroutine name> is the name of the coroutine that shall be activated first. <coroutine list> is a character matrix in which the names of all coroutines occurring in the program are listed. Each name is a row of the matrix. *C* is the condition for an execution of the call, i.e., control is transferred to the statement following the *CALL* construct if ~*C*. Since the parameter *C*

must be passed by name, the expression *C* must be entered as a character string.

For the construction of the coroutines themselves, three special functions are needed: *START*, *RESUME*, and *DETACH*. *START* transfers on a resumption (*RESUME*) of the control back to the point where the coroutine was previously left. Hence, each coroutine must begin with the statement →*START* The statement,

→ *RESUME* ' < name > '

interrupts the coroutine names as the argument. → *DETACH* can be used to transfer control back to the generator program. Thus the general form of a coroutine definition is shown below :

```
∇ COROUTINE1
  →START
    . .
    . .
    . .
  →RESUME 'COROUTINE2'
    . .
    . .
    . .
  →RESUME 'COROUTINE1'
    . .
    . .
    . .
  →DETACH
∇
```

The function definitions are :

```
     ∇ R←NAME OF LIST;I
[1]    R←NAME
[2]    I←1
[3]   LOOP:⍎LIST[I;],'RL←0'
[4]    →((⍴LIST)[1]≥I←I+1)/LOOP
     ∇
```

```
     ∇ R←C CALL NAME;IS
[1]    R←⍳0
[2]    CRN←NAME
[3]   LOOP:⍎'CRL←',CRN,'RL'
[4]    IS←CRN
[5]    ⍎CRN
[6]    ⍎IS,'RL←CEL'
[7]    ⍎'→(',C,')/LOOP'
     ∇
```

```
     ∇ LABEL←START
[1]    LABEL←2+CRL
     ∇
```

```
     ∇ OUT←RESUME NAME
[1]    CRN←NAME
[2]    CEL←(□LC)[2]-1
[3]    OUT←0
     ∇
```

```
     ∇ R←DETACH
[1]    R←CEL←0
     ∇
```

COMMENTS :

CRN, *CRL* and *CEL* are carriers of messages exchanged between the "dispatcher" *CALL* and the coroutines. *CRN* is the name of the coroutine to be called next, *CRL* is the return label on resumption of an execution, while *CEL* is the exit label. ⎕*LC* yields the label where *RESUME* occurs.

As an example, we write the well-known "quick-sort" procedure using coroutines for performing the required partitionings of the list to be sorted (71). This can be accomplished by the function *PARTITION* in connection with the two coroutines *MOVEI* and *MOVEJ* defined as follows.

```
      ∇ R←I PARTITION J;V
[1]     V←ID[J]
[2]     → 'I<J' CALL 'MOVEI' OF (2 5)⍴'MOVEIMOVEJ'
[3]     ID[J]←V
[4]     R←(I-1),J+1
      ∇

      ∇ MOVEI
[1]     → START
[2]     → REPEAT
[3]     → IF (ID[I]>V) DOTO END
[4]     ID[J] ← ID[I]
[5]     → RESUME
[6]    END:→ FI
[7]     I←I+1
[8]     → UNTIL I≥J
[9]     → DETACH
      ∇

      ∇ MOVEJ
[1]     → START
[2]     → REPEAT
[3]     → IF (V>ID[J]) DOTO END
[4]     ID[I]←ID[J]
[5]     → RESUME 'MOVEI'
[6]    END: →FI
[7]     J←J-1
[8]     → UNTIL I>J
[9]     → DETACH
      ∇
```

(DETACH is built-in for reasons of security.)

The sorting procedure itself can be written as recursive functions. Thus we complete the "quicksort" program by the following two APL functions.

```
      ∇ QSORT IDENT;M;N;ID
[1]     M←1
[2]     ⍎ 'N←⍴ID←',IDENT
[3]     M SORT N
[4]     ⍎ IDENT,'←ID'
      ∇
```

```
      ∇ M SORT N;S
[1]     → (M≥N)/0
[2]     S←M PARTITION N
[3]     M SORT S[1]
[4]     S[2] SORT N
      ∇
```

Auxilary functions called by *MOVEI* & *MOVEJ* are shown below.

```
      ∇ R← REPEAT
[1]     REPSTACK←(1+(⎕LC)[1+⍳1]),REPSTACK
[2]     R←⍳0
      ∇

      ∇ LABEL←UNTIL C
[1]    →(~C)/0,LABEL←(~C)/1↑REPSTACK
[2]     REPSTACK←1↓REPSTACK
      ∇
```

```
      ∇ LABEL← IF CLABEL
[1]     LABEL←(~CLABEL[⍳1])/1+CLABEL[1+⍳1]
      ∇
```

```
      ∇ CLABEL←C DOTO LABEL
[1]     CLABEL←C,LABEL
      ∇
```

As compared to the shared variable approach, this method is less expensive to execute in terms of computer run time (recall that computing resources are expanded even for the "idling" functions executing an endless loop while waiting for an acknowledgement from the controlling workspace). However, there is some overhead involved in executing the dispatcher functions and its associated programs. A certain amount of workspace needs to be assigned to maintain these functions and the global variables required to act as carriers of information. Unlike the shared variable

method, the dispatcher function technique simulates concurrent operations sequentially. Hence, the simulation may take longer to exercise than the shared variable approach. With is comment, we are to continue our discussion of the Design System (i.e. one based on APL/HDL).

B) Generation of Micro-program & Piece-wise Refinements

Continuing our discussion of the functional description (of a machine behaviour), the user may furnish the functional description separately for each instruction of the presumed instruction set. In non-trival cases, it may be rather tedious for the designer to determine which parts of the resulting partial structures can be integrated (or merged) into a whole. To this end, one must determine which parts of which processes are disjoint, time-wise or structure-wise, so that they may share devices such as registers, data paths, or functional units. This necessitates a sorting process, performed on the functional description, which in complex cases may become difficult for a manual solution which is error-prone. The computer, on the other hand, has the necessary prowess of performing this complicated sorting task. Hence, it is intended that the Design System supports the user in obtaining a refined functional description of the integrated system (one being designed & implemented by the digital designer).[1]

If control of the machine is obtained through micro-programming, a micro-assembler can be employed at this point for generating the micro-program in a format that is determined by a microinstruction format description as specified by the user for the respective machine (e.g. the use of existing microprocessors having microprogramming facilities).

---

[1] It should be noted that we have not claimed that such "sorting" process has been thoroughly considered and implemented. We are indicating the possibility of constructing this type of CAD tool. Hence, we make a provision for such facility in the Design System.

The next step in the development of a structural description requires the addition of the functional units for information flow control. This step (translating at the RT-level a functional to a structural description) can be automatically performed by a translator that operates by using certain standard components (e.g. those in appendix 11) which the user has specified. The user may repetitively carry out such a step as part of an interactive design process by which he optimizes the data path structure of the system. Such a process necessitates answering questions of the kind: How many direct register-register connections will be saved by providing an additional bus, and what is the total count of data flow control devices in each case?

In a structural RT-level description, the semantics of the assignment symbol ← has changed from an information transfer between carriers to a connection of carriers with functional units and vice versa. If control of the machine is realized by a hardwired control part, the detailed structural description of the control part has been obtained. As the remaining step we must obtain the detailed structural device level descriptions for all controlled parts.

It is not our intention to automate this process (as remarked in section 2.3.3 and depicted by figure 3), but we shall rather leave to the "man-in-the-loop" to carry it out in a process of "step-wise refinement". Let us clarify the notion of step-wise refinement by an example. In a functional description, the incrementation of the program counter content may be given by the statement

$$PC \leftarrow INC\ PC$$

Here, *INC* is a monadic function representing an incrementation network. In a functional description of that network, we may define the function *INC* as follows

```
     ∇ R ← INC X
[1]    R ← ((⍴X)⍴2) ⊤ 1+2 ⊤ X
     ∇
```

Whereas this definition describes the input/output behaviour of the incrementation networks, it certainly does not describe its structure. If a structural description is required, we may replace the definition given above by the following

```
     ∇ R ← INC X;I
[1]   ⍝ΔI ※ I←⍴X
[2]  LOOP: X[I]←X[I]≠(∧/I-1)↑X
[3]   ⍝ΔI ※ I←I-1
[4]    →(I>0)×LOOP
     ∇
```

The input/output behaviour of this function is the same as that of the above; but now *INC* assumes a form from which the structure of the respective iterative network can be derived. We might again replace this definition by one given in terms of a transistion matrix and an output matrix (73). It should be noted that for the purpose of simulation either definition of the function *INC* may be used. If an integrated device for this function were available, the first, purely functional description would be sufficient.

* * * *

We have seen that APL/HDL lends itself naturally to the process of step-wise refinement by employing one and the same language for functional as well as structural descriptions (in fact, at any level ranging from the System level to the circuit level). It is by this unification that APL has been found to be the most suitable programming language to adopt for our Top-down Design Approach. In the course of time, an APL/HDL based design system will encompass a library of standard processes and standard functions representing standard components. Whenever necessary, the user will have to add ad hoc written defined processes and functions, the development of which will be greatly facilitated by stepwise refinement and the possibility to verify each step by simulation. The mechanisms of concurrent processes and nested function calls in combination with the adequate "scope" control facilities, which the APL defined function mechanism provides, gives the user a flexible and powerful instrument.

2) Users' Environment

Through out this chapter, we have advocated the use of APL for the Top-down Design Approach; as the language has much to offer (section 2.5). However, this is not taken to suggest that APL is a panacea for all problems. Despite the power and capability APL provides, yet like any other programming language there exist some deficiencies (section 2.5). In time, with the constant extensions and enhancements on the language, these deficiencies may be reduced; if not overcome. On the other hand, in designing a CAD system it is essential to provide the capability to communicate or to invoke facilities offer by other non-APL processors and application packages. This will enable a more efficient use of computing resources and derive benefits from existing & established application packages (e.g. circuit analysis programs, optimization programs). In addition, as a measure of sharing the cost of developing and maintaining a CAD system, a multi-user environment for the CAD is a necessity. In the following, we shall be discussing methods[1] for the Top-down Design Approach.

A) Interface with Non-APL Processors & Application Packages

There are a number of approaches[2]; we shall be discussing a method based on the use of non-APL files. When the CDC APL.2 (55,56) is used (as at Imperial College), the technique is further simplified by the system function CSUBMIT. To illustrate this, consider that a logic minimization is to be performed by LOGMIN (appendix 2) which is implemented in FORTRAN. Let us also assume that the user is not expected to known FORTRAN and the input format for LOGMIN. A state-table, however, is obtained in a register-transfer level design and that a logic minimization is to be performed.

---

[1] The ideas conceived for the present studies have been implemented by this author in two large commercial systems.

[2] For example, the use of shared variables in APLSV. While in Xerox APL, when the ENQUEUE command is executed in an APL workspace, the invoking workspace is temporarily saved; while a non-APL program may be executed. When completed, the "suspended workspace" is recalled. Intermediate results, however, are still required in an interfacing file, which is non-APL.

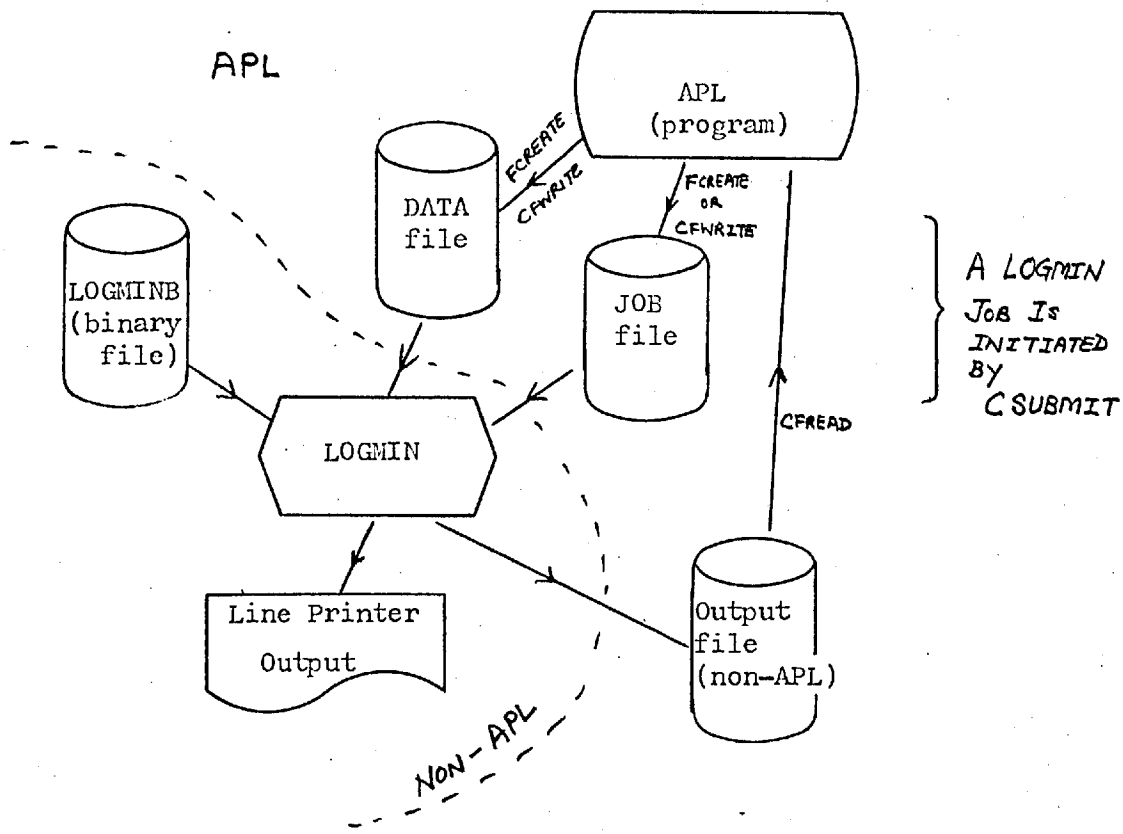


Fig. 20 : Interface with Non-APL Environments

Figure 20 depicts a method through which LOGMIN is invoked within APL. The data file (a coded file; non-APL) necessary for a logic minimization is created from the state-table under APL and is subsequently acting as an input data file for LOGMIN. A job file (a coded file) is then automatically created indicating the necessary job control cards (appendix 2) as indicated below.(These are job-control cards for the CDC-CYBER series running under the NOS operating system.)

```
JOB(UMEEB21, J4, CM20000, T10)
PASSWORD(SECRETE)
GET(LOGMINB,DATA)
LOGMINB(DAT A)
End-of-file
```

Upon completion, the APL application program issues the CSUBMIT, i.e.

Z ← CSUBMIT <job file name> , <type>

where, type ↔ 0 = if batch output produced by the job should be discarded
1 = it it should be printed/punched at the central batch site.

When LOGMIN is completed, the result may either be printed (type=1) or saved as a coded file (type=0; LOGMIN also need to output a FORTRAN file) which may again be subsequently interpreted by an APL function. The above process appears to be involved; however, this is "transparent" to the user. It is unnecessary for him to know the details of the internal operation of the CAD system. To this user, he has just submitted a batch job or waiting for a computation at the terminal with a CAD system which is APL based.[1]

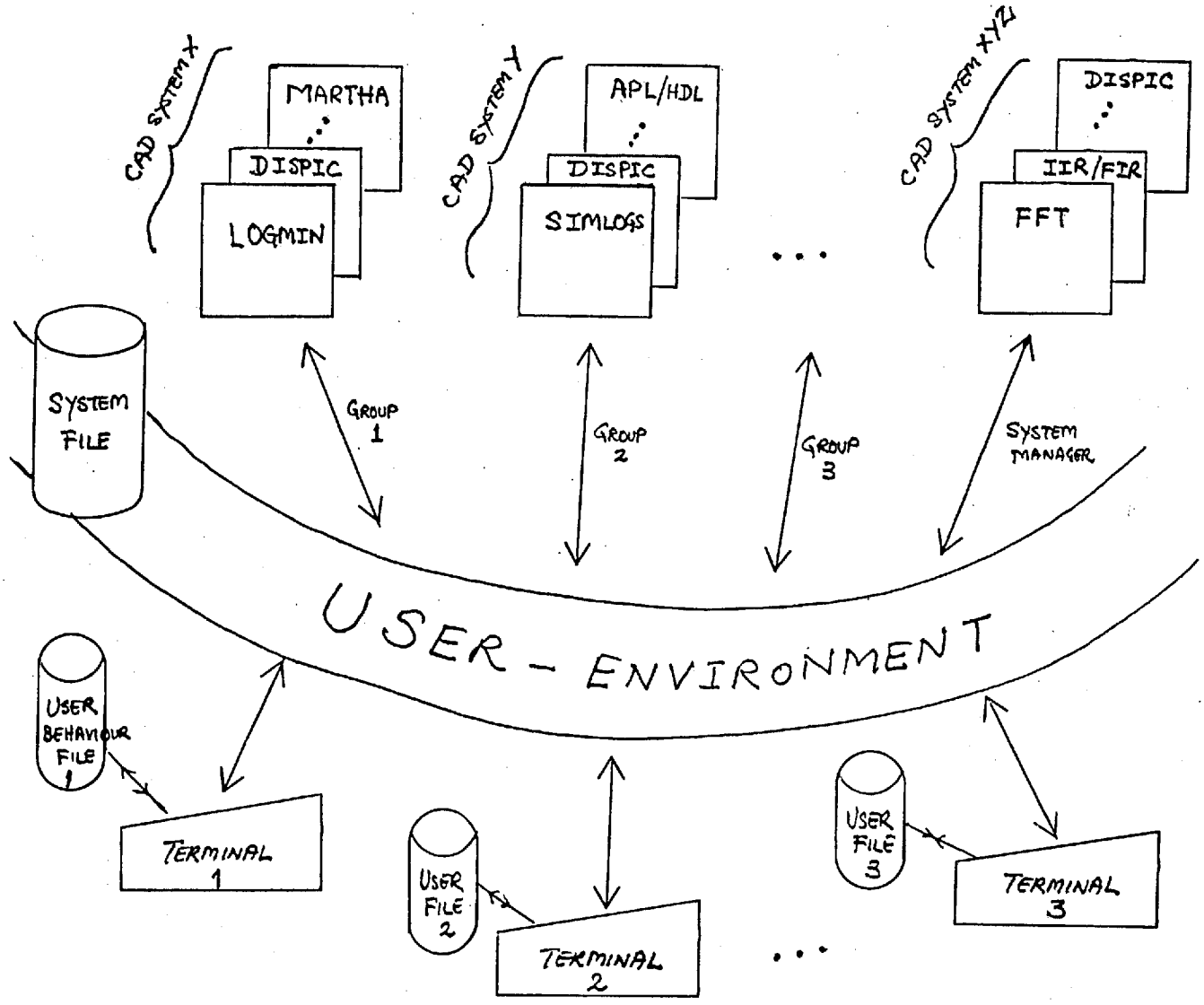
B) Multi-user Environment

Figure 21 shows an implementation of the Top-down Design Approach we envisioned.[2] The users are divided into various groups each of which is associated with a collection of workspaces forming a CAD system. For example, a version of the CAD system as depicted in figure 19 and DISPIC (chapter 3 & appendix 13) may collectively be classified as a CAD system for a user group. While other system groups (i.e. other user groups associated with a CAD system) may be formed out of a selection of the subsystems available to the multi-user Top-down Design System (here after for convenience, will be called Design System). The user-environment contains a set of user-behaviour files (one per terminal), a system file and a user-environment workspace. The user behaviour file contains information such as — identities of users (accessing the Design System), the subsystem used within a session for a user, cost of each session and the frequency of "system problems", such as computer down-time or break down in the communication link form a terminal to the computer. The

---

[1] This concept originated by this author has been implemented in a commercial bugetary system which consists of 4 levels of non-APL files; while each level consists of a set of 6 files set up individually by 6 geographically separated locations. An input validation system was designed in APL while the budgetary computations were implemented by a FORTRAN based financial package. To the users, they had a budgetary system with consistent & standard operating procedures.

[2] Concepts and technqiues were originally conceived for our Top-down Design Approach. These techniques have been applied to a commercial system by this author during his activity as a senior system consultant. This author realised that characteristics and requirements were identical to those of the Top-down Design Approach; hence the implementation.



NOTE :

— denotes a subsystem (or a subsystem workspace)

— denotes an accessing terminal or a computer account (i.e. not a physical terminal)

— denotes a 2-way communication

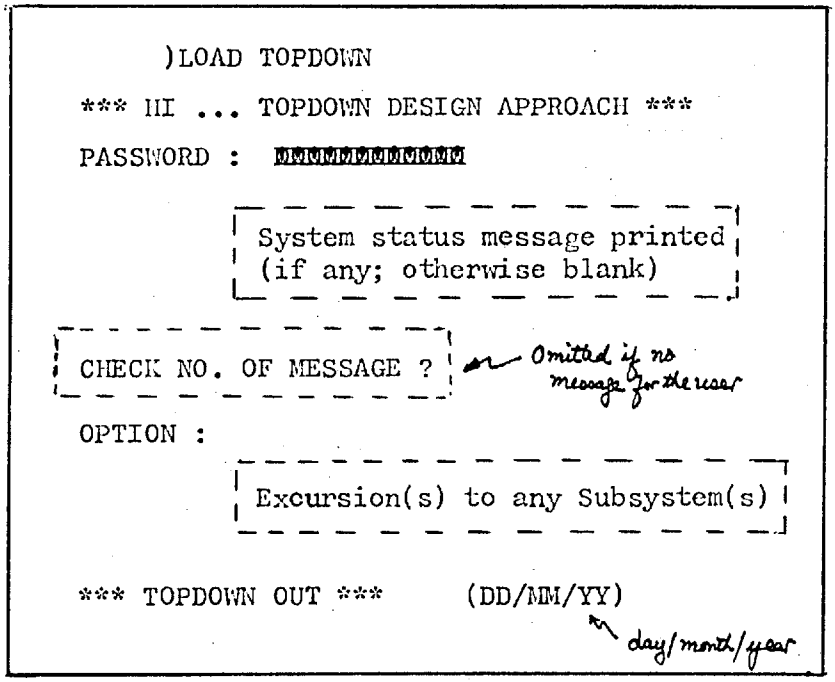
Fig. 21 : A Multi-user Environment for the Top-down Design Approach

system file, contains directories for user passwords, user group details, subsystems assigned for each user group, status of the system, user-group & terminal (strictly speaking, computer account) current accessing the system, and messages to & from the system manager. The user-environment workspace may be regarded as an "operating system" for the Top-down Design Approach. This workspace is loaded in the first instance by any users; in fact the user knows the existence of only one workspace for the Design System. The workspace derives all control information from the system file; such as requesting for a user-password (figure 22) at the first instant. From this, the user-group classification is derived; hence the number of subsystems available for this user. Next, it checks for the status of the system. That is, whether at that point in time, access to the system for a particular user-group may be temporarily suspended; while other groups may independently operate with the Design System. For a temporarily suspended group, a message (possibly a polite one) is displayed and the user is automatically signed off by the system. Otherwise, the user may proceed and the system requests for the options (i.e. a subsystem name). Note that a "help" facility is available; for example, a ? designates a help. The available options are displayed to remind the user. When a subsystem is chosen, the appropriate workspace for the subsystem is loaded into the active workspace and overwrites the user-environment workspace. On completion with the subsystem, the user-environment workspace is reloaded.

The system manager who controls the system, may interrogate the system file to examine the current status of the system — for example, which user group and indeed which particular user, is current in session (active user) with the Design System. He may also archives the message section to read any "mail" posted to him; hence he may communicate with any active or non-active users. A sketch of the user-dialogue and a sample of directories extracted from the system file are depicted in figure 22. It should also be noted that many operations, such as status interrogation, are restricted to the system manager.

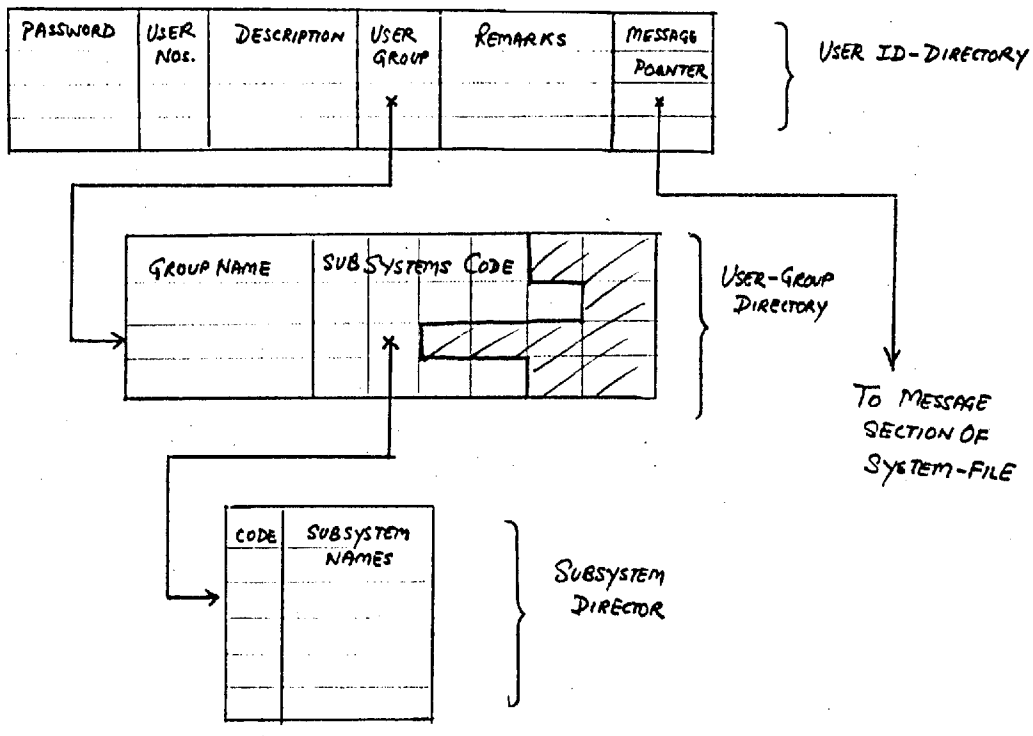
* * * *

This concludes the presentation of the Top-down Design Approach as an aid for the design and implementation of digital signal processors. In the next few chapters, we shall be considering efforts to provide subsystems for the Design Approach and thereby extending and enhancing the usefulness of this Design Approach.



Note the simplicity of the dialogue; sufficient information is derived for controlling the Top-down Design Approach

(a) A Schematic Dialogue



(b) A Sample of linkage between Directories

Fig. 22 : Structure of a multi-user environment dialogue

Chapter 3 THE DESIGN & SIMULATION OF A CLASS OF DIGITAL SIGNAL PROCESSORS —— DISPIC'S [1]

Summary

The "Top-down Design Approach" is discussed with special reference to the system, functional and register-transfer levels. To further illustrate and enhance the design process, a class of digital signal processors (DISPIC'S) have been designed and implemented with APL. The objectives of DISPIC are reviewed. This is followed by a presentation of the design and implementation of DISPIC; the application of which is demonstrated by the simulation of an IIR digital filter on this simulated processor (DISPIC). It should be noted that while DISPIC has been designed as an illustration of the Design Process, yet DISPIC may in itself be used as a tool to investigate filter structures and their performances. The chapter concludes with an indication of future developments for DISPIC.

3.1 INTRODUCTION

We have seen in chapter 2 that the design of digital signal processors or digital filters can be viewed 'conceptually' as having a number of levels (fig. 2): mathematical & algorithmic, architectural and realization. While the implementation of them consists of arranging and inter-connecting the overall system blocks (i.e. architecture) and choosing a network topology and hardware for the final circuitry. Digital filter hardware design consists of designing the individual circuit elements (e.g. adders, multipliers, shift-registers) while taking into considerations of special requirements of the processor/filter.

---

[1] This chapter is based on a report (75) and a paper (76) published by this author.

The design of an implementation can be considered as a 'translation process'. The 'description of the architecture' specifies the functions performed by the system being designed. This leads to an 'implementation description' which shows how the system can be built. Formal computer languages (e.g. APL, ALGOL68 & PL/1) maybe used to give a precise and concise description at each of the levels discussed above, but APL has been found the most suitable for the design process (Chapter 2, section 2.5). The description can further be used to verify the given architecture. Thus digital filters can be systematically implemented by this controlled design process.[1] This may assist a designer to quickly and flexibly achieve an implementation. Furthermore, this process maybe used to explore other less conventional ideas or concepts (e.g. the use of Residue Number Systems — Chapter 5).

An 8-bit computer (75 and appendix 9) has been implemented to investigate the techniques involved for the higher level, viz. Register-Transfer, Functional, and System Levels. To further illustrate and enhance the controlled design process, a more practical digital signal processor (i.e. DISPIC) has been implemented. This will be discussed in the following sections.

3.1.1 Scope of the Design

Before discussing the signal processor (or digital filter) in detail, it will be useful to clarify the focus of our attention; i.e. at which stage of the design process the implementation begins. We shall illustrate the discussion[2] with the design of an infinite impulse response (IIR) digital filter (a FIR filter was discussed in Chapter 2 and in appendix 4).

---

[1] The process is a controlled one because the designer approaches from the "Top-down" and gradually at each stage introduces just sufficient details to clarify his concepts. We have chosen the term "controlled design process" instead of "top-down design approach" because we are focussing our attentions specifically on the functional and register transfer levels.

[2] In the subsequent discussion, APL will be used when convenient.

3.1.2 System Level

A digital filter is characterized by a transfer function, while the realization of the filter may assume various structures; an example is an infinite impulse response filter (IIR) realised as a cascade of second order biquadratic sections (fig. 23). For the purpose of the discussion, let us assume that a 2nd-order filter has been obtained by a CAD means (mathematical concept) such as that discussed in reference 6. At this stage, the implementation at the system level begins. The mathematical and algorithmic concepts may be described by two APL functions (fig. 24) with IIRN2 being the main function which describes the algorithm of an Nth-order IIR filter realised by cascades of 2nd order biquadratic sections (IIRBQSN).

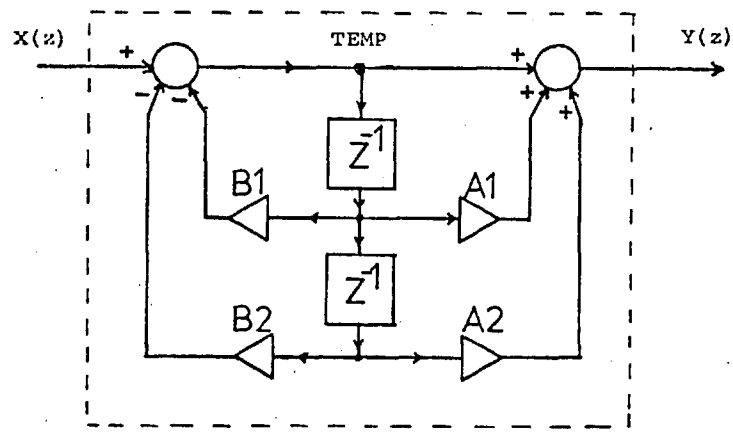


Fig.23: A Second order Biquadratic Section

As an example, the impulse response of a 2nd order IIR filter of the form:

$$H(z) = \frac{1 - 2 z^{-1} + z^{-2}}{1 - 1.5610 z^{-1} + 0.6414 z^{-2}}$$

is obtained by setting coefficients A's = -2, 1 and B's = -1.56, 0.6414. While the filter is excited by a unit impulse. In terms of APL, the

```
      ∇Y←IIRN2 X;N;I;SEC;XI;J;INTX
[1]     ⍝ N-TH ORDER  IIR (CASCADE OF BIQUADRATIC SECTIONS)
[2]     ⍝  COMPUTE THE NO, OF SECTIONS OF THE CASCADE FORM
[3]     SEC←⌈0.5×⍴AS
[4]     ⍝  SET INITIAL CONDITIONS (EG, RESET DEALYS OF FILTER)
[5]     N←⍴X
[6]     Y←N⍴0
[7]     I←0
[8]     XI←(⍴AS)⍴0
[9]     ⍝  COMPUTE THE OUTPUT OF THE FILTER FOR THE I-TH SAMPLING
[10]    ⍝   TIME (T)
[11]  L1:→(N<I←I+1)/END
[12]    ⍝  THE I-TH INPUT
[13]    INTX←X[I]
[14]    ⍝  MULTIPLEXING OF BIQUADRATIC SECTIONS
[15]    J←¯1
[16]  L2:→(SEC<J←J+1)/L3
[17]    ⍝  COMPUTE THE OUTPUT FOR THE J-TH SECTION
[18]    INTX←J IIRBQSN INTX
[19]    →L2
[20]    ⍝  THE I-TH OUTPUT OF THE CASCADE FILTER
[21]  L3:Y[I]←INTX
[22]    →L1
[23]  END:'** COMPLETED **'
[24]    ' '
      ∇
```

```
      ∇Y←I IIRBQSN X;TEMP
[1]     ⍝ COMPUTE SAMPLE FOR THE I-TH SECTION(BIQUADRATIC)
[2]     ⍝  INDEX DELAY/STATE OF THE I-TH SECTION
[3]     I←1 2+I×2
[4]     ⍝  COMPUTE THE OUTPUT OF THE I-TH SECTION FOR
[5]     ⍝  CURRENT SAMPLING TIME (T)
[6]     TEMP←X-+/BS[I]×XI[I]
[7]     Y←TEMP++/AS[I]×XI[I]
[8]     ⍝  UPDATE THE STATES OF THE I-TH SECTION FOR
[9]     ⍝   THE NEXT SAMPLING TIME (T)
[10]    XI[I]←TEMP,1↑XI[I]
```

Note : IIRN2 is an Nth-order IIR; realised by cascading 2nd order biquadratic sections which is simulated by function IIRBQSN

Fig. 24 : An APL Description of an Nth-order IIR Digital Filter

coefficients are defined as global variables for the functions IIRBQSN and IIRN2. The input to the filter, IIRN2, is defined by the right argument X. To perform the simulation, we initially set global variables AS & BS and the unit impulse (UNIT) as below.

```
  AS←  -2      1
  BS←  -1.56   0.6414
UNIT←  0 0 1   0 0 .....0
```

The corresponding impulse response (or filter output) is given by the explicit result (Y) returned by function IIRN2. A graphical display, however, maybe obtained using a PLOT function listed in appendix 5. Thus when executing

```
10  30  PLOT  IIRN2  UNIT
```

the impulse response as depicted in figure 25 is obtained. On the other hand, to obtain the frequency response (fig.26) of the filter, we may execute,

```
10  30  PLOT  1  FAST  IIRN2  UNIT .
```

FAST is a fast fourier transform function, the details of which are included in appendix 6.[1]

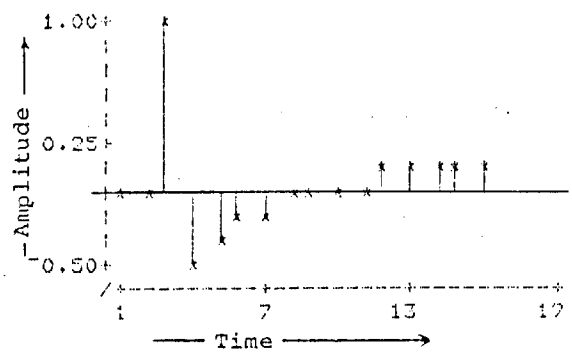


Fig. 25 : Impulse Response of a 2nd order Infinite Impulse Response (IIR) Digital Filter

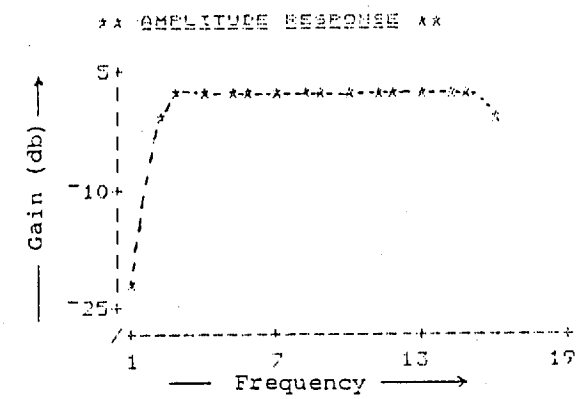


Fig. 26 : Frequence Response of the 2nd order IIR Filter (in figure 25)

[1] Number of points must not exceed 256 (see appendix 6); otherwise the right argument of FAST (result returned by IIRN2 in the above) must be truncated with an appropriate "window" function before using FAST.

When the designer is satisfied with the responses of the filter, the implementation process enters the functional and register transfer levels.

3.1.3 Functional and Register Transfer Levels

For the purpose of the discussion, let us assume that a programmable signal processor is chosen; so that the processor may be programmed to realize a given transfer function discussed in section 3.1.2. The design of a programmable signal processor differs from that of a general purpose computer and the following requirements need to be considered:-

(1) choosing an instruction set (i.e. OP-codes) tailored to the signal processing operations required,

(2) choosing an interconnection scheme for the hardwares to allow efficient execution of the instruction set,

& (3) in conjunction with (2) choose an architecture that is amenable to the introduction (even at a later stage) of parallelism in the various execution phases of the instructions.

With these requirements in mind, we shall discuss the digital signal processor mentioned at the beginning of this section.

3.2 THE DESIGN OF DISPIC'S

As a further investigation[1] of the controlled design process, we have attempted to design a programmable digital signal processor with the view of satisfying the requirements stated in section 3.1.3. For reference purposes, we christened this processor as DISPIC (a Digital Signal Processor at Imperial College).

---

[1] A preliminary investigation was conducted on designing an 8-bit programmable machine (appendix 9); while considerations of machine architecture using APL as a tool are documented in appendices 11 & 12.

3.2.1 Objectives of DISPIC's

Since DISPICS is expected to be a "subsystem" of the integrated "Top-down Design Approach" (Chapter 2, section 2.6), we have an additional requirement of DISPIC —— By its own right, it should facilitate a designer with a tool to investigate the structure and performance of a filter which is programmed on this processor. This implies a number of supporting facilities (not usually found in most small systems), such as residence editor, assembler, loader, which should not be so complex & difficult to use that distract the designer from his original task — designing a digital filter. To this end, under the name of DISPIC, we have designed and implemented three versions of DISPIC, each of which had been designed to emphasize some or all of the aforementationed requirements.

3.2.2 Architecture of DISPIC'S

The architecture of the three versions of DISPIC'S is depicted in figure 27; while the instruction set and system commands for version 1 are respectively listed in tables 5 & 6. Version 1 differs markedly from versions 2 & 3 by having one memory; while it emphasizes the user-system dialogue requirements (section 3.1.3). Hence it is expected to be used as a "close-ended" design tool (i.e. the user has little means of modifying the architecture other than some of the instruction sets). While versions 2 & 3 are designed for "open-ended" usages, so that the user (designer) may modify the structure[1] drastically to suit his requirements. This group of users is expected to have a good knowledge of the APL programming language.

---

[1] e.g. assembling the collection of APL functions and interconnecting them by the control function OPN (see appendix 12). The user may introduce his own functions to meet his particular requirements.

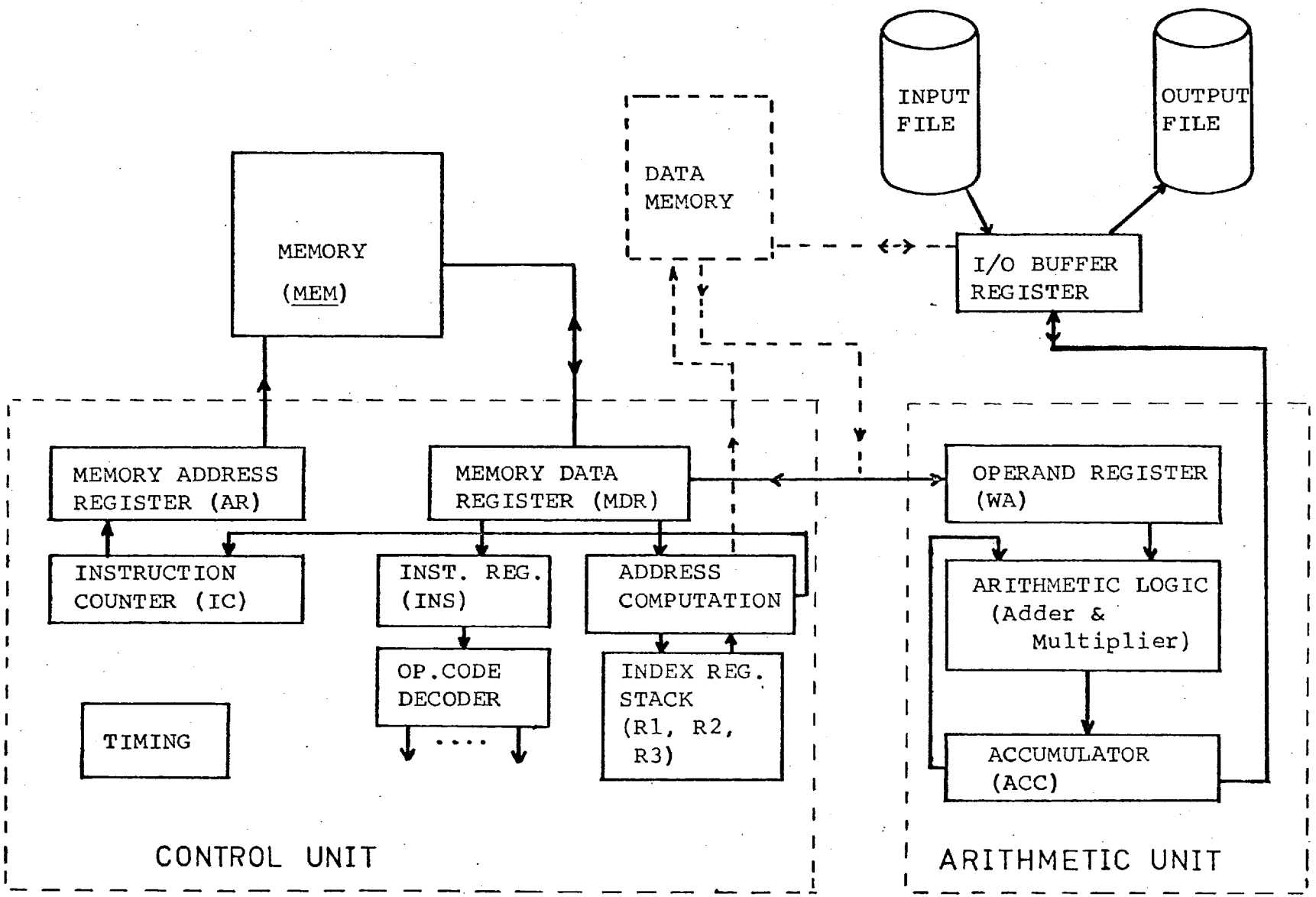


Fig. 27 : Architecture of DISPIC's

Essentially version 1 of DISPIC consists of a main memory with 500 locations,[1] each having a four digit address running from 0000 to 0499, a control unit, an arithmetic unit and an input/output buffer which acts as an interface between the arithmetic unit and the input/output files. The control unit and the arithmetic unit consist of the following registers:

Accumulator (ACC)	Stores a single DISPIC word which is one of the operands in arithmetic operations
Memory data register (MDR)	Contains a word obtained from the memory. The instruction is decoded while the address is used for address computation. Finally the data is fetched from the memory and subsequently transferred to the operand register
Operand register (WA)	Holds the operand of the memory data word; so that the arithmetic unit may compute while concurrently the next instruction maybe fetched from the memory.
Instruction Counter (IC)	Keeps track of the location of the next instruction to be fetched from the memory
Memory address register (AR)	Contains the final effective address obtained from the current instruction taking into account all address modification using index registers and indirect addressing

The salient features of DISPIC (Version 1) will be discussed in the following subsections (3.2.3 to 3.2.8).

---

[1] The memory size was arbitrarily chosen as 500 for the purpose of simulation. The memory size, however, may be increased or decreased using the system command MEMORY n; where n = the number of locations required.

Mnemonic OP code	Operation
	MEMORY REFERENCE INSTRUCTIONS
LDA	Load word addressed into Accumulator(ACC)
STO	Store contents of Accumulator in Location addressed
STA	Store Address Field of ACC in Address Field of Operand Register (WA)
	INDEXING OPERATIONS
LIα	Load last four digits of word addressed into Index Register (α), for α = 1,2,3
SIα	Store contents of Index Register α into last four position of location addressed
TIα	Decrement Index Register α by 1 and transfer control to location indicated, if register becomes zero
TXα	Same as *TIα* except branch only if register is not zero
AXα	Add last four digits of word addressed to Index register α
SXα	Substract last four digits of word addressed from Index register α
CIα	Skip next Instruction if Index Register equals the word addressed
CXα	Skip if Index Register does not equal to the word addressed
	ARITHMETIC INSTRUCTIONS
	INTEEGER OPERATIONS
ADD	Add contents of ACC to contents of the word addressed and result is deposited in the ACC
SUB	Substract contents or word address from ACC result to ACC
MPY	Multiply the contents of ACC with that of the word addressed, result to ACC
DIV	Divide contents of ACC by that of the word addressed, result to ACC
	FLOATING POINT OPERATIONS
FAD *FSB* *FMP* *FDV*	Similar to the integer operations; but require all operands to be in floating point format
FIX	Change contents of ACC to integer format
FLT	Change contents of ACC to floating format
	BRANCHING INSTRUCTIONS
BRU	Branch unconditionally to Location addressed
BMI	Branch if sign of the ACC is negative
BCα	Octal digit α represents a code, so that:- α =4 100 branch if ACC < 0 α =2 010 branch if ACC = 0 α =1 001 branch if ACC > 0
CSα	Compare ACC and word addressed using the octal code α , so that:- α =4 100 if ACC < WA, skip next instruction α =2 010 if ACC = WA, skip α =1 001 if ACC > WA, skip
BLα	Unconditional branch and placing the location of the next instruction in the Index Register α
CPA	Skip the next instruction if WA=ACC (character)
	MISCELLANEOUS INSTRUCTIONS
SHL	Shift ACC left as the number of position in the Address Field, leaving the sign fixed
SHR	Shift ACC right .....
HLT	Stop execution; typing start resumes execution at the location addressed
	INPUT AND OUTPUT INSTRUCTIONS
RWD	Read number from the user console and place it in the location addressed
RBF	Read a number from the Input File (*ΔINPF*) and place it in ACC via the I/O Buffer
WWD	Print the word addressed at the user console
WBF	Output the contents of the ACC to the Output File (*ΔOUTF*) via the I/O Buffer

Table 5 : Instruction Set of DISPIC's (Version 1)

SYSTEM COMMANDS

COMMAND	SYNTAX	USAGE
BUILD	BUILD	Creates a DISPIC application program (or data set)
DROP	DROP PRG	Drop the program PRG from the Library of programs
EDIT	EDIT PRG	Edit the program PRG
EXECUTE	EXECUTE N	Execute contents of the memory (corresponding a program) starting from location N
INITIALIZE	INITIALIZE	Initialize the environment of DISPIC ; this should be executed once before the DISPIC workspace is saved.
LIB	LIB	List the Library of DISPIC application programs
LINKMAP	LINKMAP	When the relocate command (RLOAD) is used, DISPIC keeps a record of all subroutines linked to the main program. LINKMAP lists the current status of linkages.
LIST	LIST PRG	Lists the DISPIC application program PRG
LISTFILE	LISTFILE	Lists contents of the Input file (ΔINPF) or the Output file (ΔOUTF); LISTFILE prompts the user to choose the required option
LOAD N	LOAD PRG	Loads the program PRG into the memory of DISPIC, starting from location N
LOADMAP	LOADMAP	Lists the programs currently loaded into the memory
MEMORY	MEMORY N	Changes the memory size to N locations (default=500) and blanks the entire memory. The Loadmap is cleared.
PATCH	PATCH N	Changes contents of the memory location N
RENAME	RENAME PRG	Changes the name of the program PRG; when executed the user is asked for the new name PRG is to assumed
RESTART	RESTART	Restarts execution of a DISPIC program after an interrupt (either due to an APL program suspension or a break which is initiated by pressing the ATTEN key at the user APL terminal)
RESET	RESET	Resets the record counter of the Input File (ΔINPF) to zero and empties the Input file
RLOAD N	RLOAD PRG	Relocate load; adjusts all address field with a ρ in the first position (i.e. bit 7 of the instruction format) with respects to the "origin" N. RLOAD also updates the Linkmap so that the 'absolute' address of PRG is entered in the map.
RUN	RUN PRG	Loads the PRG into the memory and executes it
STATES	STATES	Displays status of all registers (i.e. internal states) of DISPIC
TIMER	TIMER←ON TIMER←OFF	Turns the TIMER on/off, so that the CPU time is printed for every 100 instruction (N.B. CPU time refers to the APL CPU time) -- TIMER is ON by default
TRACE	TRACE←ON TRACE←OFF	Turns the Tracing Mechanism of DISPIC on, so that the internal states of DISPIC is displayed at the end of every memory cycle -- TRACE is OFF by default

[Note : ON & OFF are boolean variable; ON = 0, OFF = 1]

Table 6 : System Commands of DISPIC (Version 1)

3.2.3 Data Format : Instruction & Number

DISPIC words are of two types; instruction and number[1]; the format of these are depicted in figure 28. Strictly speaking they are denoted in character form[2]; with a total of 11 characters and, starting from left to right, they are labelled from 0 through 10.

A) Instruction format : The instruction format is divided into three sub-fields,

OP-code : digit 1 - 3, contains any of the instruction codes (table 5).

Index register: digits 4 - 6, and are used for address modification. (see sections 3.2.4 & 3.2.6)

Address : digits 7 - 10, and are used to designate the memory locations ranging from 0000 to 0499. This field may also be used to hold a constant for the immediate address operation.

Digit 0 has no meaning for the instruction format and is assumed to be blank. The instruction repertoire may be classified into six types, viz. Memory reference instructions, index operation, arithmetic operation, branching instruction, input & output instruction and miscellaneous instructions such as shifting and rotating the contents of the accumulator. The complete instruction set is listed on table 5. The instruction sets should be self explanatory; as an illustration consider adding four numbers which are held in memory locations 0 through 4 and the total is to be stored in location 0005.

---

[1] The term "number" is used to denote the data with which arithmetic operations are to be perfromed.

[2] Since DISPIC (Ver.1) could be used as an educational tool, we have adopted a decimal notation for the address field and number representation. Hence in the simulation the data formats are represented by literals. However, binary representations are used for versions 2 and 3 of DISPIC.

This program may assume the form below[1].

LOCATION	* / -	OPER	INDEX REG.	ADDRESS	REMARKS
		2 3 4	5 6 7	8 9 10 11	← CARD COL
0000		LDA	000	1001	(ACC) ← (1001)
0001		ADD	000	1002	(ACC) ← (ACC) + (1002)
0002		ADD	000	1003	(ACC) ← (ACC) + (1003)
0003		ADD	000	1004	(ACC) ← (ACC) + (1004)
0004		STØ	000	5000	(5000) ← (ACC)

B) Number Format : Integer and floating point numbers are defined. The formats for these are depicted in figure 28; either form consists of a sign and magnitude parts.

i) Integer number - is represented by a sign or a blank (which is interpreted as positive) as the first digit and is followed by ten digits or blanks (which designate zeros by default) as the magnitude part. The magnitude part is required to be right justified; since trailing zeros are associated with a value. Hence - 1234 should be written as

0	1	2	3	4	5	6	7	8	9	10
-							1	2	3	4

while

0	1	2	3	4	5	6	7	8	9	10
-	1	2	3	4						

designates the number -123, 400, 000.

---

[1] We have adopted the convention of using brackets to denote -- "the contents of". Hence (ACC) designates "the content of accumulator (ACC)". In addition, a coding form as shown is a convenient way of programming DISPIC; in fact any small system.

ii) Floating point number - is represented by two signs followed by nine digits or blanks as the magnitude parts. The first sign is the sign of the number and may be represented by a blank for positive. The second sign belongs to the exponent and therefore must be explicitly specified as either positive (+) or negative(-); that is, every floating point number requires a sign in the second position of the number format. Digits in positions labelled 2 and 3 (i.e. positions three and four) designate the exponent of the number. In other words, the power of tens the the mantissa must be raised to give the value of the number that is expressed in the floating-point format. The mantissa, appearing in digits 4 through 10, is a "normalised number"; i.e. the decimal point is implicitly implied to be on the left of the number. Hence - 12345.67 is represented by

-	+	0	5	1	2	3	4	5	6	7

It should be noted that, in contrast to the integer number format, trailling zeros are not significant.

3.2.4 Address modes

Designing an addressing system is often an involved task for a machine architecture. On the other hand, the addressing system reflects the power and capability of the machine.[1] Four modes of addressing exist in DISPIC, depending on the entries in the index register field of the instruction. The effective address (EA) is computed by the address computation block (Fig.27). This is a feature of parallelism; so that the arithmetic unit (AU)

---

[1] Consideration and description (using APL) of different address systems are involved in appendix 12.

may compute the signal sample while the address computation unit calculates the next effective address.

1) Direct Address : digits 4-6 are blanks, the EA=contents of the address field (i.e. digits 7-10)

2) Immediate address : digit 4 contains I, indicating that the contents of the address field in the instruction field is interpreted as an operand.

3) Indexing address : with an α in digit 6 (where α = 1,2,3) the EA = contents of the address field + contents of the index register Rα (e.g. R1, R2, R3).

4) Indirect address : with an * in digit 5, the EA = contents of the memory addressed by the contents of the address field of the instruction.

3.2.5 Arithmetic -- fixed and floating

The arithmetic instructions of DISPIC provide integer and floating - point operations. ADD, SUB, MPY & DIV respectively add, subtract, multiply and divide the contents of the accumulator (ACC) with that of the operand register (WA) the contents of which are fetched from the memory via the memory data register (MDR). It should be noted that for the division operation, the integer value of the accumulator is divided by that of MDR. The resulting quotient (which is an integer) is deposited on the accumulator while the remainder (or residue) is discaded. The corresponding arithmetic instruction codes for floating - point operations are FAD, FSB, FMP and FDV. Representations of the operands and results are in accordance with the floating-point format (discussed in section 3.2.3). Two additional instructions are provided to convert number representations between fixed and floating point format. These are FIX and FLT which respectively converts the contents of the accumulator "from floating to fixed point" and "from fixed point to floating".

DISPIC detects mixed type operations (i.e. operands are not both fixed or floating point representation). Other errors such as arithmetic overflow, arithmetic underflow and division by zero are detected by DISPIC. For all these cases, DISPIC halts the execution of the DISPIC program and reports the status of the operation to the user. (see section 3.2.8)

The collection of arithmetic instructions of DISPIC, especially those for floating point arithmetic, is devised to enable a simpler programming facility for the filter designer. In particular, during the initial stages of his work, programming in integer arithmetic (which is an option of DISPIC) will involve comparatively greater efforts. As an illustration, consider computing the following expression,

$$8.25 \times 3.5 - 17.5 = 11.375$$

This necessitates the alignment of decimal points between the product of 8.25 x 3.5 and 17.5 to yield the correct result of 11.375. For this we require the use of the shift operations SHL and SHR. However, the same effect maybe obtained from the floating operations, viz FMP and FSB.

3.2.6 Index Register and Subroutines

DISPIC has provided an index register stack[1] (e.g. R1, R2, R3), so that a memory address maybe modified during execution. Together with the collection of indexing instruction, they provide many programming capabilities for DISPIC; an example of this is facility of subroutines. We shall first show the use of index registers and then with an example illustrate their application to provide subroutine capability.

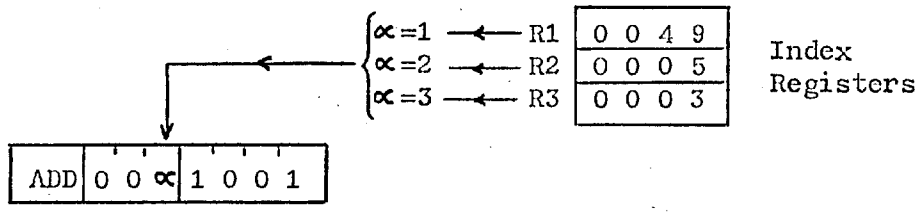
A) Use of Index Register

With reference to the instruction format (figure 28), an index register is referenced by non-zero number appearing in digit 6 of an instruction. Index registers[1] R1, R2, and R3 are respectively designated by numbers 1, 2 and 3.

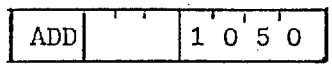
---

[1] Provisions are made for a maximum of 9 registers; even though 3 are incorporated.

When an index register is referenced, the value contains in the designated index register will be added to that of the address register before the instruction is performed. For example, let us depict the operation of an addition ADD 001 1001 as below.



which is executed as,



Where the "effective address" is given by 1001 + 0049 = 1050. Thus, the index registers permit the address of an instruction to be altered before execution and without changing the instruction as it appears in the memory. The value in any of the index registers may be modified by an "indexing operation" (table 5). For example, LI2 000 1072 will deposit the value 1072 into index register 2. Apart from the subroutine application, index registers also enable a program to address a larger size of memory than that provided by the address field (e.g. for a four digit address, the maximum location may be addressed is 9999).

B) Use of Subroutines

Digital filters are often realized in cascading sections of module (e.g. 2nd order biquadratic section) to form a higher order filter. This requires the realization of the sub-section (or module) as a sub-routine which is subsequently called by a mainprogram. With the availability of index registers, we may achieve the above. To avoid over-complicating the discussion, we shall illustrate the use of subroutines by a simple example, such as the computation of the following expression

$$X = 2 + 3A^{1/3} + B^{1/3}$$

where A, B are positive integers not greater than 1000. We assume that A & B appear as pairs and are supplied through an external medium, such as the input file (figure 27). As a first step one designs a subroutine for

the computation of the cube root with the following specifications:-

1) The subroutine will assume that the argument, an integer between 0 and 1000, inclusive, has been placed in the accumulator at the time the subroutine is involved.

2) The "return" address is the address of the next instruction to be executed in the main program following the completion of the subroutine. This return address should be saved by storing it in some convenient register before entering the subroutine. (We have chosen index register, R2 for this purpose).

3) The subroutine should place the cube root in the accumulator before branching back, i.e. returning to the main program.

A program to meet the above specification is shown in figure 29. The table is stored in locations 0006 to 1006; location 0000 is reserved for branching to the main program. Whenever the above "cube routine" is required, the following steps should be taken,

1) Place the argument in the accumulator,
2) Load the address of the next instruction after the subroutine into index register R2,
3) Branch to location 0001.

The cube root will be available in the accumulator upon exit from the subroutine. Since R1 and R2 are used by the subroutine, R3 is the only index register available for use in the main program.

The Main Program : A general flow chart is shown in figure 30 to compute the expression, $X = 2 + 3\ A^{1/3} + B^{1/3}$; while the program is shown in figure 31 and it resides in the memory locations 1007 through 1042. The program makes use of index register R3 to count the number of sets of data. Index registers, R1 & R2, are reserved for the subroutine. Notice that the constants located in 1033 and 1034 are used to load R2 with the address to which the subroutines should return. Other constants such 1, 2, and 3 are available at locations 1035 through 1037. Locations 1038 through 1042 are used to store the values M+1, M, A, B, and X.

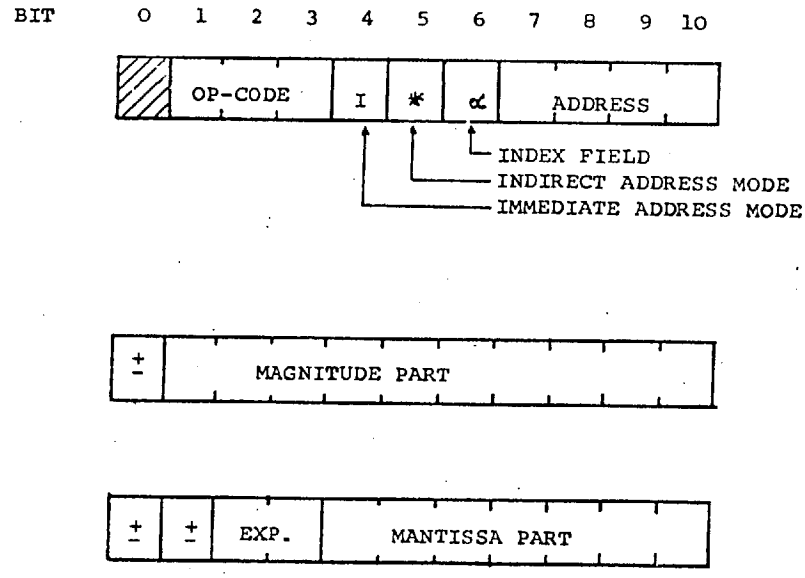


Fig. 28 : Instruction and Number Formats of DISPIC

LOCATION	±	OPER	INDEX REG.	ADDRESS	REMARKS
	1	2 3 4	5 6 7	8 9 10 11	← CARD COL
0000					RESERVED FOR BRANCHING
0001		STO	000	0005	(0005) ← (ACC)
0002		LI1	000	0005	LOAD R1 WITH THE VALUE OF n
0003		LOA	100	0006	(ACC) ← (0006 + (R1))
0004		BRU	010	0000	BRANCH TO ADDRESS IN (R2)
0005			n		ARGUMENT
0006	+	000	000	0000	$\sqrt[3]{0}$
0007	+	010	000	0000	$\sqrt[3]{1}$
0008	+	012	999	2105	$\sqrt[3]{2}$
					TABLE
1005	+	099	966	6556	$\sqrt[3]{999}$
1006	+	100	000	0000	$\sqrt[3]{1000}$

Fig. 29 : A Cube Root Subroutine (in DISPIC)

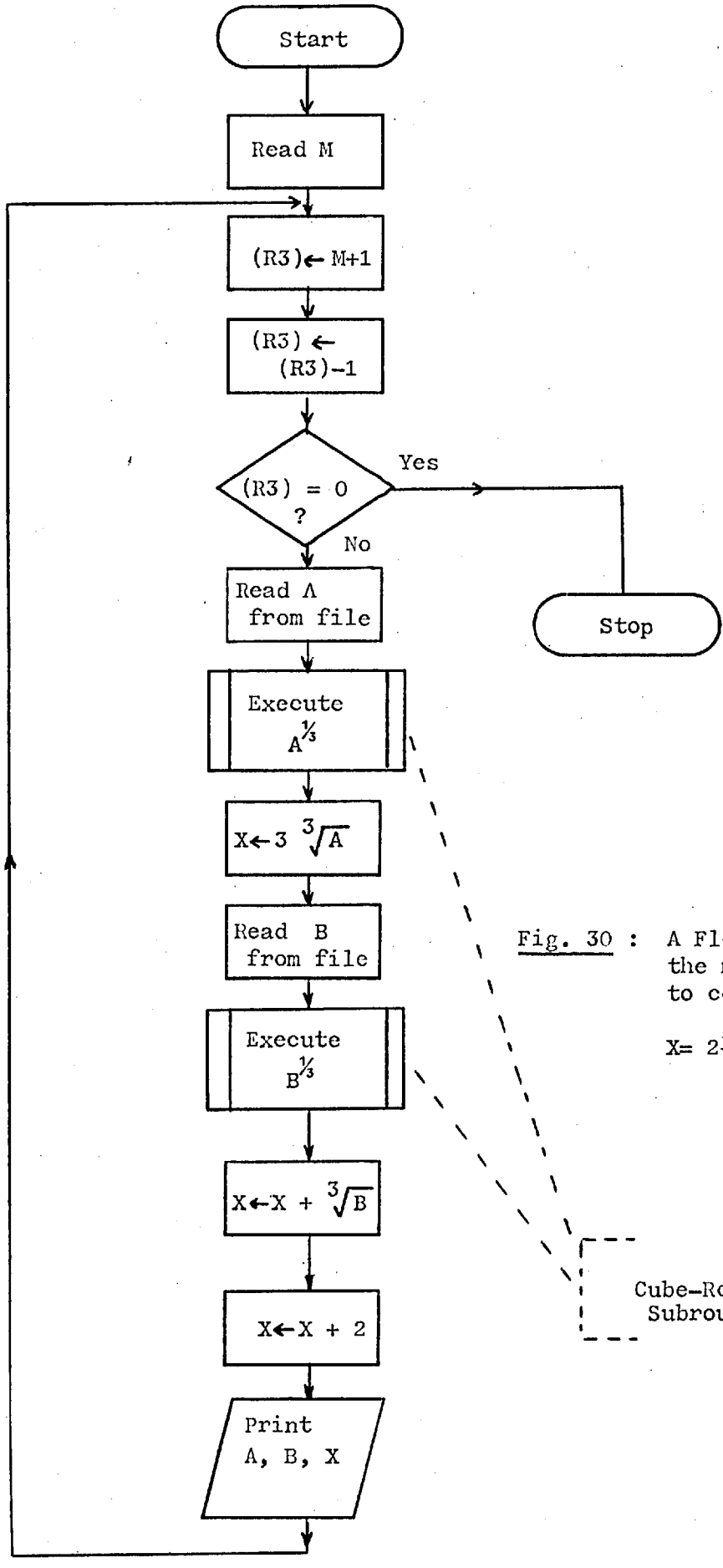


Fig. 30 : A Flow-chart of the main program to compute —

$X = 2+3A^{1/3}+B^{1/3}$

LOCATION	+/-	OPER	INDEX REG	ADDRESS	REMARKS
	1	2 3 4	5 6 7	8 9 10 11	← CARD COL
0000		BRU	000	1007	SUBROUTINE GOES IN PLACE OF HEAVY LINE
1037		PWD		1039	(1039) ← m
1038		LDA		1039	m ← m + 1
1039		ADD		1035	
1040		STØ		1038	
1041		LI3		1038	(R3) ← m + 1
1042		TI3		1031	DECREMENT AND TEST (X3)
1043		PWD		1040	(1040) ← a
1044		LDA		1040	(ACC) ← a
1045		LI2		1033	(R2) ← RETURN ADDRESS [1017]
1046		BRU		0001	BRANCH TO SUBROUTINE
1047		MPY		1037	(ACC) ← 3 x (ACC)
1048		STØ		1042	(1042) ← $3\sqrt[3]{a}$
1049		PWD		1041	(1041) ← b
1050		LDA		1041	(ACC) ← b
1051		LI2		1034	(R2) ← RETURN ADDRESS [1023]
1052		BRU		0001	BRANCH TO SUBROUTINE
1053		ADD		1042	(ACC) ← $\sqrt[3]{b} + 3\sqrt[3]{a}$
1054		ADD		1036	x ← $2 + \sqrt[3]{b} + 3\sqrt[3]{a}$
1055		STØ		1042	
1056		NWO		1040	PRINT a
1057		NWO		1041	PRINT b
1058		NWO		1042	PRINT X
1059		NWO		1032	PRINT A BLANK LINE
1060		BRU		1012	BRANCH BACK FOR ANOTHER SET
1061		HLT		1007	STOP

CONSTANTS:

LOCATION	+/-	OPER	INDEX REG	ADDRESS	REMARKS
	1	2 3 4	5 6 7	8 9 10 11	← CARD COL
1032	+				BLANKS FOR TYPEWRITER SKIPPING
1033	+	000	000	1017	FIRST SUBROUTINE RETURN ADDRESS
1034	+	000	000	1023	SECOND SUBROUTINE RETURN ADDRESS
1035	+	000	000	0001	+ 1
1036	+	020	000	0000	+ 2,00000000
1037	+	000	000	0003	+ 3
1038	+	000	000	0000	VALUE OF m + 1
1039	+	000	000	0000	m
1040	+	000	000	0000	a
1041	+	000	000	0000	b
1042	+	000	000	0000	x
					BLANK TRANSFER CARD

Fig. 31 : A DISPIC Program for :

$$X = 2 + 3A^{1/3} + B^{1/3}$$

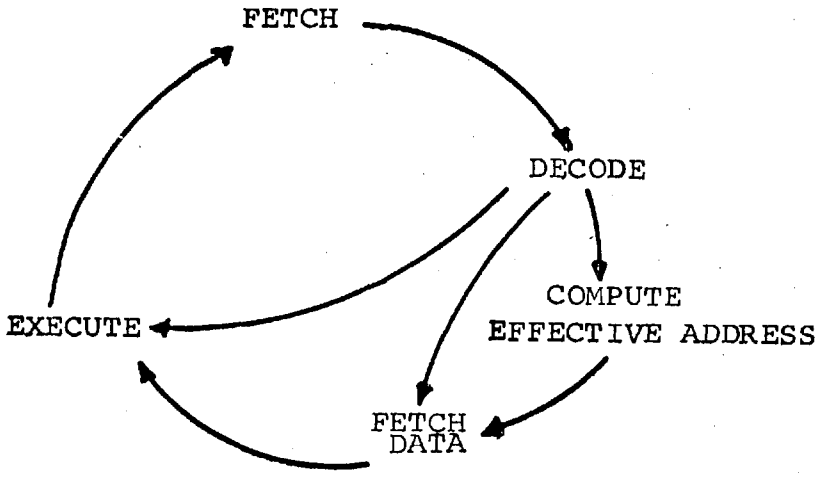


Fig. 32 : A FETCH-EXECUTE Cycle

3.2.7 Input/Output Facility

DISPIC is conceived as a real-time signal processor. The input sample enters the processor via the input/output (I/O) buffer register (which could have the facility of an A/D & D/A capability); likewise the output sample. Since we are simulating the architecture of DISPIC, it will be necessary to store all the signal samples in the input file prior to an simulation. The output samples, however, are written into the output file by DISPIC. System commands (see table 6) LISTFILE and CLEARFILE are provided to respectively list and empty either the input or the output files.

3.2.8 Instruction Cycle

A DISPIC instruction cycle consists of two phases; viz. the FETCH phase and the EXECUTION phase. During the FETCH phase, the word at the address indicated by the memory address register (AR) is obtained. The operation code (OP-code) is diverted to the instruction register (INS) and is subsequently decoded by the OP-code decoder. The address part is routed to the address computation block. The instruction counter (IC) is incremented by one. During the EXECUTION phase, the decoded instruction is executed using the accumulator and the word addressed by the address register if needed. The FETCH-EXECUTE cycle is depicted in figure 32.

3.2.9 Error-Messages

DISPIC detects instruction format and execution errors; in either case, the program (DISPIC) execution is halted and the user is advised of the status with an error message. These messages are listed below.

INSUFFICIENT MEMORY : More than 500 instructions and constants were submitted. The program should be reduced in size.

INSUFFICIENT FIELD LENGTH : The APL workspace is clustered with unused variables. Use MEMORY command to reduce DISPIC memory size, or erase inactive DISPIC files (i.e. files holding other DISPIC programs); use command LIB to catalogue inactive DISPIC files.

ILLEGAL INSTRUCTION : The mnemonic for the instruction currently being decoded does not match any in the DISPIC instruction set. (The instruction is printed with the program count.)

INVALID ADDRESS : Arithmetic was attempted on a non-numeric quantity. This may be due to reading a blank address field or an alpha character such as letter 'O' replacing a number zero.

ILLEGAL FIRST CHARACTER : The first character of an instruction should be a blank; while that for a number (or constant) should be any of -- negative or positive sign, or a blank.

ILLEGAL MEMORY ACCESS : An attempt was made to access a memory location with an address greater than 0499.

ILLEGAL INDEX DESIGNATION : A load or store index instruction was attempted which reference a non-existent index register.

ILLEGAL BRANCH OR SKIP : A branch or skip is made to a non-existance destination.

ILLEGAL INTEGER OPERATION : An integer operation has been attempted on non-numerical data fetched from the memory.

ILLEGAL FLOATING OPERATION : As "Illegal Integer Operation".

DIVISION BY ZERO ATTEMPTED : As indicated by the error message.

ARITHMETIC OVERFLOW : The result of an arithmetic operation exceeds the size of the accumulator. Execution, however, is not stopped; the error message is treated as a warning. (This is important for digital filtering, since temporary overflow in radix complement is tolerated; so long as the final result falls within the machine number range).

ARITHMETIC UNDERFLOW : As "Arithmetic Overflow."

ARGUMENTS NOT OF SAME TYPE : Mixed number representation was used. correct by using FIX or FLT.

3.2.9 System Commands

To assist the users in using DISPIC, system commands are incorperated. These commands are of three types; viz. text editing, file management and informative.

Commands associated with these are:-

A) Text Editing : BUILD, EDIT, PATCH

B) File Management : INIALISE, MEMORY, LOAD, EXECUTE, RUN, RESTART, RELOAD, RESET, RENAME, LISTFILE, DROP

C) Informative : LIB, LIST, LOADMAP, LINKMAP, STATES, TIMER, TRACE

The meanings and usages of these commands are explained and tabulated in table 6. As an illustration of using system commands, the following shows a program, TEMP, being listed by the command LIST TEMP is then executed to check the process of text editing.

```
      LIST TEMP
     ±OPC***□□□□
 0   RBF       0
 1   STO      20
 2   RBF       0
 3   STO      21
 4   RBF       0
 5   STO      22
 6   RBF       0
 7   STO      23
 8   BRU  3    5
 9   RBF       0
10   STO  3    2
11   LI1      32
12   HLT
```

```
      EDIT TEMP
DO YOU WISH TO ADD LINES, DELETE LINES,
CHANGE A LINE, PRINT LINES, OR EXIT?
ENTER: A,D,C,P, OR X      C
WHICH LINE IS TO BE CHANGED?   3
THE LINE NOW READS:      3□  STO     21
ENTER THE CORRECTED LINE  ±OPC***□□□□
                         3□ STO     21
ENTER: A,D,C,P, OR X      C
WHICH LINE IS TO BE CHANGED?   8
THE LINE NOW READS:      8□ BRU  3   5
ENTER THE CORRECTED LINE  ±OPC***□□□□
                         8□ BRU     35
ENTER: A,D,C,P, OR X      C
WHICH LINE IS TO BE CHANGED?   10
THE LINE NOW READS:     10□ STO  3   2
ENTER THE CORRECTED LINE  ±OPC***□□□□
                        10□ STO     32
ENTER: A,D,C,P, OR X      X
 ** COMPLETED **
```

```
      LIST TEMP
     ±OPC***□□□□
 0   RBF       0
 1   STO      20
 2   RBF       0
 3   STO      21
 4   RBF       0
 5   STO      22
 6   RBF       0
 7   STO      23
 8   BRU      35
 9   RBF       0
10   STO      32
11   LI1      32
12   HLT
```

The state of DISPIC is reset by executing RESET; the memory size is reduced from 500 to 300 using MEMORY. Program is loaded into DISPIC and is ready to run.

```
      RESET
** COMPLETED ... READY TO RUN


      MEMORY 300
MEMORY SIZE WAS : 500
      0 LOAD TEMP
DATA SET LOADED INTO MEMORY
LOCATIONS 0 THRU12
```

3.3 IMPLEMENTATION & APPLICATIONS OF DISPIC'S

3.3.1 Implementation of DISPIC'S on a Host Computer[1]

Each version of DISPIC consists of a collection of APL functions residing in one workspace. All versions of the signal processors are written with the basic APL and is therefore a "portable" package. Source coding of these functions are included in the appendix 13.

A) DISPIC VERSION 1

When DISPIC (Version 1) is first created, the following workspace parameters are required to be defined:-

WORKSPACE SIZE	:	32K bytes	
INDEX ORIGIN	:	0	
TERMINAL WIDTH	:	130	
CARRIAGE RETURN	:	CR	} set to the appropriate atomic vector component
BACKSPACE	:	$\overline{BS}$	
LINE FEED	:	$\overline{LF}$	

[1] DISPIC's were originally developed at Imperial College with the CDC APL.2 which is run under the operating system, NOS. They were subsequently loaded on the IBM 5100 desk top computer with some modifications to take advantage of the shared variables. For example, the input/output files of DISPIC (Ver.1) were defined as shared variables so that tape files could be used; resulting the capability of handling a large volume of data.

The initialize function, INITIALISE, should first be exccuted. The workspace identification, WSID, may then be defined as any name chosen by the user (we have identified version one as DISPIC 1). DISPIC 1, may now be used; that is, the user may create his DISPIC program with the BUILD command and subsequently save the workspace.

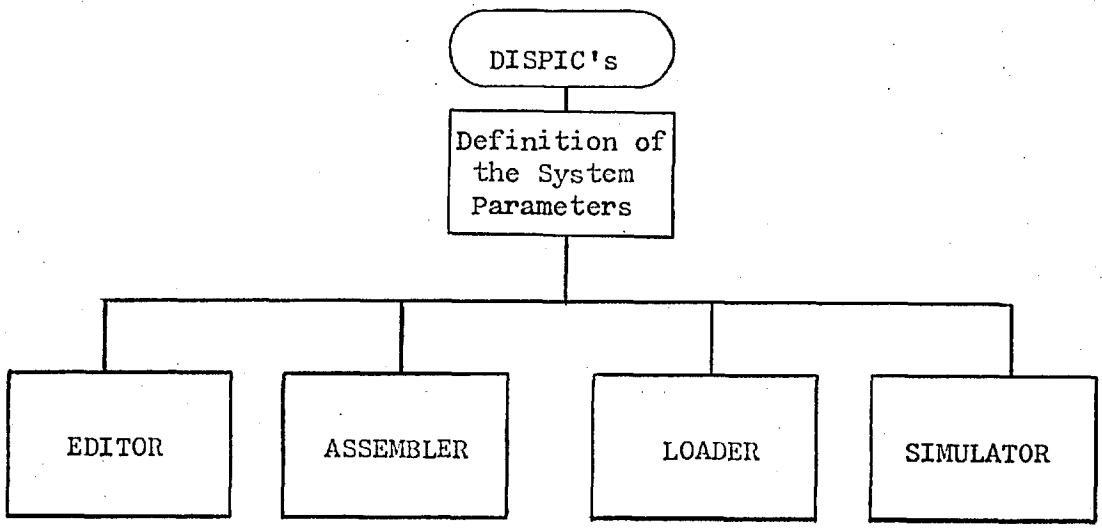
Organization of the workspace : APL functions and variables that are primarily for the DISPIC simulator are underscored (see appendix 13). For example OPN is the main control function which acts as a "decorder" for the operation codes of DISPIC and performs the appropriate operation by calling an associated set of APL functions; e.g. GWA is called to address a location of the memory and deposit the contents into the memory data register. System commands are implemented as APL functions having the same function names as those for the commands listed in table 6. For example, the function for adjusting the memory size of DISPIC is MEMORY n, where n is the number of locations required. This convenience of program naming convention is one of the reasons which has made APL the & most suitable computer language for the "Top-down Design Approach". The use of DISPIC will be illustrated with the simulation of an IIR in section 3.3.2.

B) DISPIC Versions 2 & 3

As remarked earlier, versions 2 & 3 of DISPIC[1] were intended for an "open-ended" usage (section 3.2.2). Hence, by interconnecting some standard functions (e.g. appendices 7-12) one may construct a required architecture. New functions will sometimes have to be written; but in this process, the standard set is subsequently enriched. Unlike DISPIC, versions 2 & 3 contain less supporting functions such as the system commands listed in table 6; however, these utilities are to be developed by the users as an integral part of the machine design. On the other hand, the machine words of DISPIC2 and DISPIC3 are represented in binary form. However, DISPIC3 consists two memories (one for program and one for data) while DISPIC2 is still a single-memory machine. The workspace parameters for these two versions are the same as those for DISPIC1. In general, the functions of DISPIC's may be organized as below.

---

[1] Subsequently referred to as DISPIC2 and DISPIC3



It should be noted that while the organization is functionally partitioned as above; yet these "functional modules" are inter-related so that the perturbation of a module can have a repercussion on the others. For example, the introduction of a new instruction code in the system parameter module requires subsequent modifications on the assembler and simulator modules. However, in terms of designing the simulation programs and later installation of the package (c.g. dividing the collection into function groups for a multi-workspace environment), it is advantageous to consider the organization as indicated in the above.

The system parameters are specified by a group of function such as INITIALIZE (in DISPIC1) and ΔPARAMS (in DISPIC2 & DISPIC3). Once these initializing functions are executed, the machine architecture to be simulated is defined. The functions forming the editor (e.g. text editing functions of DISPIC1, whereas BUILD, ΔSOURCE of DISPIC2 & DISPIC3), provide some convenience for the users (or digital system designers). The assembler module converts instructions which are closer for human comprehension to those more convenient for machine execution (e.g. mnemonic codes to binary codes). The loader module simulates the loading of a program into the machine (being simulated) as in an actual machine operation (e.g. Initial Program Load - IPL illustrated in appendix 12). The simulator module performs the operations

of the target machine (i.e. the machine to be simulated). In this connection, operations like decoding, branching, interrupts, memory reference, arithmetic, and input/output are simulated. It is through this simulation, a designer may model his architecture and gain sufficient experience to determine on an eventual implementation of the machine.

A selection of the source code listing for DISPIC2 and DISPIC3 are included in appendix 13. As an illustration, the execution of an APL "scan" (i.e. \ ) operation is simulated DISPIC2, and is depicted in figure 33. While an example of a "system definition" of DISPIC3 is shown in figure 34.

3.3.2 Simulation of an IIR filter with DISPIC

An application of DISPIC is best illustrated by an example; again, we shall make use of the second-order biquadratic IIR filter discussed in section 3.1.1. A program (a DISPIC program) for the filter is shown in figure 35. This program assumes the partitioning of the memory into two parts; viz. program area and data area. The memory map after the loading of IIR2 is depicted in figure 36. When the program is executed, the impulse response (corresponding to a unit impulse) is stored in the output file. With the use of the fast fourier transform APL function, FAST, (appendix 6) the frequency response (as figure 26) may be obtained.

The example also illustrates the importance of using one formal computer language throughout all the levels of the controlled design process. This avoids much of the unnecessary details of computer; while these details are distracting for the filter designer.

3.3.3 Future Developments of DISPIC

During the development of DISPIC it was envisioned that the signal processor would eventually be implemented by an existing microprocessor, such as INTEL 8080. In this connection some work have been conducted in simulating the operation codes of the INTEL 8080; in view of the development not fully completed, we have decided to omit the presentation in this writing. In

spite of this, we feel work in this direction should be pursued to enrich the collection of "standard operation" for DISPIC (as discussed in section 3.3.1). With this as a starting point, one may extend the efforts on other families of microprocessor so that ultimately one may attain a "universal micro-assembler" for the APL/HDL based design system as depicted in figure 19.

Other short term enhancements of DISPIC are the introduction of a "timing mechanism" in the control function (e.g. OPN, section 3.3.1) of DISPIC. Therefore a designer may assign a delay time for each operation. When a DISPIC program is run, the execution time for each instruction is recorded on a file so that a statistical studies of the DISPIC program performance may be conducted. The object of this exercise is twofold -- a "benchmark" studies of different architectures may be obtained. On the other hand, DISPIC may also be used to study the performance of a filter being programmed and to some extent monitor the progress of a designer in his design task. For the less experienced user, DISPIC will provide a feedback of his attempts and thereby acting as a training and exploration tool.

Microprogramming techniques have not yet been investigated throughly for DISPIC's. With the flexibility for design offered by these techniques, we feel they should be incorporated into the "library" of DISPIC'S.

The limitation of resources (both human, material and financial) have not enabledus to pursue the above developements to some satisfactory conclusion. Despite this, DISPIC's have demonstrated the flexibility and opportunity the "Top-Down Design Method may provide. It is our hope that others will extend the capacility and thereby enhance the usefulness of DISPIC'S. In the next chapter, we shall traverse to the lower level and consider the design and implementation of a gate level simulator -- SIMLOGS.

```
      ΔDEFINE
** MACHINE DEFINED ... READY TO RUN
```

Fig. 33(a) : DISPIC2 is Initialized by executing function ΔDEFINE

```
      Δ PROGAM 1  ------- P001 ; A SCAN PROGRAM

      ΔSOURCE P001

CCCC □□□□□
INP
STO   LIMI
CST   1
STO   X
STO   Z
OUT   Z
CST   1
ADD   X
STO   X
ADD   Z
STO   Z
OUT   Z
LAD   X
BRH   6
HLT
```

Fig. 33(b) : A DISPIC2 Source Program (P001) to simulate an APL "scan" operator; P001 is listed by function ΔSOURCE

```
      MP001←ASSEMBLE P001
SYMBOL TABLE NAME?   SMP001
** END OF ASSEMBLING ...
```

Fig. 33(c) : Program P001 is Assembled; by executing function ASSEMBLE (assembled program is called MP001; while the symbol table is SMP001)

```
      ΔSYMTAB SMP001

NAME        LOCATIONS
LIMI           23
X              22
Z              21
```

Fig. 33(d) : Symbol Table SMP001 is displayed by executing ΔSYMTAB

```
      ⍞ DISPLAY MEMORY

      MP001
0 0 1 1 0 0 0 0 0
0 0 1 0 1 0 1 1 1
0 1 1 0 0 0 0 0 1
0 0 1 0 1 0 1 1 0
0 0 1 0 1 0 1 0 1
0 1 0 0 1 0 1 0 1
0 1 1 0 0 0 0 0 1
0 1 0 1 1 0 1 1 0
0 0 1 0 1 0 1 1 0
0 1 0 1 1 0 1 0 1
0 0 1 0 1 0 1 0 1
0 1 0 0 1 0 1 0 1
0 0 0 1 1 0 1 1 0
0 1 1 1 0 0 1 1 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
```

Fig. 33(e) : Display the Memory of DISPIC after MP001 is loaded

```
      +\⍳6

1 3 6 10 15 21
```

Fig. 33(f) : A Plus-scan of 1 2 3 4 5 6 in APL operation

```
      DISPIC      MP001
⎕:

      6

1
3
6
10
15
21
```

Fig. 33(g) : Simulation of the plus-scan

Fig. 33 : Simulation of an APL "Plus-Scan" on DISPIC2

CODE	BINARY	REMARKS
HLT	0000	Terminate program
LAD	0001	Load Accumulator(ACC) from Memory addressed
STO	0010	Store ACC to Memory addressed
INP	0011	Input data from console to Accumulator
OUT	0100	Output data from Accumulator to console
ADD	0101	Add Memory addressed to ACC, result to ACC
CST	0110	Load Address-field of instruction to ACC
BRH	0111	Branch to location addressed if ACC=0
MLT	1000	Multiply memory addressed by ACC, result in ACC
	...	
	1111	(Not yet defined)

OP-Code of DISPIC3

```
        ΔPARAMS

** PARAMETERS OF THE ASSEMBLER/MACHINE DATA FORMATS **

1.   INSTRUCTION FORMAT : LABEL, OPCODE, INDEX REG., ADDRESS
   ΔIFI    0 4 0 5

2.   SYMBOL TABLE :  VARIABLE NAME, MEMORY LOCATION
   ΔSI     5 5

3.   INSTRUCTION/OP-CODE LENGTH (ΔOPI) : 4

4.   MEMORY SIZE : NO. OF MEMORY, NO. OF WORDS, WORD LENGTH,
                   OPCODE FIELD, ADDRESS FIELD
   ΔMZ      1 24 9 4 5

4B.  DATA MEMORY : NO. OF WORDS, WORD LENGTH
   ΔMZ2     20 9

5.   OP-CODE TABLE (ΔOPTAB) :
HLT
LAD
STO
INP
OUT
ADD
CST
BRH
MLT

6.   IMMEDIATE OPERATION CODES (ΔIMOP) :
BRH
CST
```

Fig. 34 :

Defining System Parameters for DISPIC3

```
      BUILD
ENTER LINES OF DATA SET; PRESS RETURN AT END

     ±OPC***□□□□
   0□ RBF       0    A READ COEFFICIENTS TO DATA AREA FROM INPUT FILE
   1□ STO      50    A  STORE A1
   2□ RBF       0    A  READ  A2 AND STORE A2
   3□ STO      51
   4□ RBF       0    A  READ AND STORE B1
   5□ STO      52
   6□ RBF       0    A  READ AND STORE B2
   7□ STO      53
   8□ LDAI      0    A ZEROLISE  X1,X2
   9□ STO      54    A  X1←0
  10□ STO      55    A  X2←0
  11□ RBF       0    A READ NO. OF SAMPLES FROM INPUT FILE
  12□ STO      56    A  SET END-OF-INPUT FILE FLAG IN INDEX REG.:R1
  13□ LI1      56
  14□ RBF       0    A START BIQUADRATIC SECTION SIMULATION;
  15□ FLT            A  READ INPUT SIGNAL AND CONVERT IT TO FLOATING
  16□ STO      57    A  X← INPUT SIGNAL
  17□ LDA      52    A COMPUTE  TEMP
  18□ FMP      54    A  ACC←B1×X1
  19□ STO      58    A  STORE  B1×X1
  20□ LDA      53
  21□ FMP      55    A  ACC←B2×X2
  22□ STO      59    A  STORE  B2×X2
  23□ LDA      57    A  LOAD INPUT SIGNAL(X)
  24□ FSB      58    A  ACC←X-B1×X1
  25□ FSB      59    A  ACC←X-(B1×X1)-B2×X2
  26□ STO      56    A  TEMP←ACC; STORE TEMP
  27□ LDA      50    A COMPUTE OUTPUT SAMPLE
  28□ FMP      54
  29□ STO      60    A  STORE A1×X1
  30□ LDA      51    A
  31□ FMP      55    A  ACC←A2×X2
  32□ STO      61    A  STORE  A2×X2
  33□ FAD      60    A  ACC←A1×X1 + A2×X2
  34□ FAD      56    A  ACC←TEMP + A1×X1 + A2×X2
  35□ WBF            A WRITE OUPUT SAMPLE TO OUTPUT FILE
  36□ LDA      54    A SHIFT SIGNAL IN SAMPLE DELAYS
  37□ STO      55    A  X2←X1
  38□ LDA      56
  39□ STO      54    A  X1←TEMP
  40□ TX1      14    A TEST FOR END-OF-INPUT FILE; OTHERWISE GOTO 14
  41□ HLT            A HALT, SIMULATION ENDS
  42□                A CARRIAGE IS PRESSED TO END 'BUILD'

WHAT SHOULD THE NAME OF THIS DATA SET BE?   IIR2
```

Note : The program(also known as "Data set") is now created and is called *IIR2*

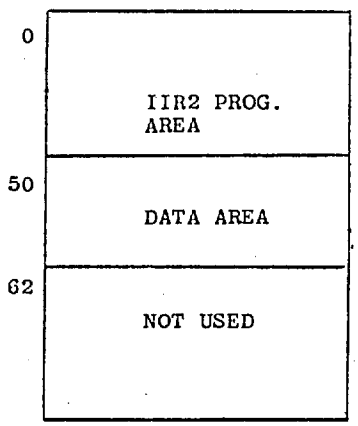
Fig. 35 : A 2nd Order IIR Filter (IIR2) Programmed on DISPIC (Ver.1)

Data Area

Location	Contents	Designation
50	*A1*	Coefficient of Z^{-1}
51	*A2*	Coefficient of Z^{-2}
52	*B1*	Coeff.(pole) of Z^{-1}
53	*B2*	Coeff.(pole) of Z^{-2}
54	*X1*	Output of delay Z^{-1}
55	*X2*	Output of delay Z^{-2}
56	*TEMP*	Temp. value (see figure)
57	*X*	The Ith input sample
58	*B1* × *X1*	
59	*B2* × *X2*	
60	*A1* × *X1*	
61	*A2* × *X2*	
62 etc.	scratch ..	Scratch areas, or memory not used.

Organization of the DISPIC Memory

Organization of the Memory



Data Area -- Coeff Coefficients and Intermediate Results

Fig. 36 : Memory Map of DISPIC after IIR2 is Loaded

Chapter 4 MODELLING & SIMULATING LOGIC CIRCUITS AND SYSTEMS WITH *SIMLOGS* [1]

Summary

When the conventional approach, in formulating and verifying digital designs, is augmented by a suitable computer design aid, design ideas may be 'thoroughly', 'speedily' and 'economically' tested prior to the construction of a prototype. To attain these objectives and to provide a subsystem for an integrated computer-aid to designing digital systems, this author has developed *SIMLOGS* (a Simulator for Logic Systems) which is implemented in the APL computer language. The design philosophy and the environment of SIMLOGS are presented.

4.1 INTRODUCTION

4.1.1 The Necessity of Employing Logic Simulation in Design - Verification & Testing.

The trend of increasing capabilities and decreasing cost of semiconductor circuits (e.g. microprocessors, LSI, MSI, etc.) have enlarged the scope of design and facilitated the realization of many digital systems (e.g. signal processors, PCM system, etc.). These systems could not be implemented in the past because of the limitations associated with descrete components and the high material and operational costs involved. The same technological advancement in semiconductor has also brought about difficult problems of design verification and testing. The present conventional approach in formulating and verifying digital systems by the construction of prototypes (i.e. physical simulation) have been suffering from excessive technological dependence and inflexible modification during the early stage of the design. Suitable computer design aids (CAD) will reduce if not obliterate these difficulties. Systems which are based on LSI/MSI, often accentuate the difficulties of testing, in particular when it is necessary to access gates or intra-connections within these packages.

---

[1] This chapter is based on papers (77, 78) published by this author.

If this conventional approach is augmented by *SIMLOGS* (A SIMULATOR for LOGIC SYSTEMS), the design idea may be 'thoroughly', 'speedily' and 'economically' tested prior to the construction of a prototype. *SIMLOGS* is a table-driven gate-level simulator with macro-description facilities to simulate a digital system of various degrees of (or increasing) complexity. Certain physical properties of the system and the associated technological dependence can be considered during different stages of the simulation. Details of a selected module (for example, corresponding to the LSI package) may be examined for the purpose of fault diagnosis. Hazards in the system are reported in the printout. Thus the fault testing facilities may also serve as an aid to maintaining a piece of manufactured hardware. *SIMLOGS*, however, has been developed as a subsystem for the Top-down Design Approach (Chapter 2).

4.1.2 *SIMLOGS* as a Subsystem for the Top-Down Design Approach

As discussed in Chapter 2, the success of the Top-Down Design Approach rests heavily on computer simulations; the formulation, design and implementation of good software packages for this purpose are therefore of paramount importance. *SIMLOGS* has been designed to enable the simulation at the gate level. This is, of course, not correct to assume that the Gate/Logic Level is necessarily the most important in the Interactive Model. At the Gate Level, the designer is concerned with the interconnection of primitive elements which are combinational logic elements (e.g. *AND*, *OR*, *NAND*, *ROM*'s, etc. as in *SIMLOGS*) and memory elements (e.g. JK Flip-Flop, shift registers, etc.). The level also includes functional modules such as encoder, decoder, multiplexer, and counters. *SIMLOGS* has been developed to facilitate simulation at this Gate Level; it is therefore a subsystem for the Top-down Design Approach.

4.2 THE DESIGN OF SIMLOGS — MODELLING & IMPLEMENTATION

The fundamental requirement of a simulator is to simulate the behaviour of physical devices accurately; even under fault conditions. On the other hand, the user's interaction with the simulator is of equal importance. It is obvious that exact simulation and modelling of a sizeable physical system is

impractical or impossible because of the infinite number of possible fabrication errors that associate with physical devices. A compromise is therefore necessary in the choice of techniques and algorithms to model and implement a simulator. The considerations of accuracy can be thought of as being primarily theoretical, that is the development of techniques for accurate modelling. While the cost-effective considerations have primarily been practical implementation of software. Thus the evolution of cost-effective digital simulators has been achieved via the development of accurate modelling and of increasing complex, sophisticated and efficient software implementations. *SIMLOGS* has been developed as a computer software for the above purposes (modelling and analyzing logic circuits). In the following sections we shall discuss the philosophy adopted in designing *SIMLOGS*.

4.2.1 Description/Specification of Circuits

The development of good simulators has been expensive; usually measured in man years of effort. Hence an important starting point is the complete definition of the present and possible future use of the system. For example, the system to be developed should not be restricted to a particular type of logic (e.g. synchronous, asynchronous, combinational, sequential). 'Equivalent models' for devices will not be used - for example, *AND*, *OR*, *NOT* logic will not be used to describe systems composed of *NAND* or *NOR* logic. Furthermore, flip-flops will not be expanded to their gate equivalents (e.g. cross-coupled *NOR* gates are used to describe a set-reset flip-flop). Although equivalent modelling simplifies the implementation of the simulator, yet the accuracy of the simulator and efficiency of the software are greatly reduced. These considerations necessitate *SIMLOGS* to have primitives consisting of basic gates, including *AND*, *OR*, *NOT*, *NAND*, *NOR* and elements including flip flops (*JK*), and composite elements such as a shift registers (*SHR*), a 1-bit half adder (*HADD*) and a read only memory (*ROM*). (Table 7) The use of some modules repetitively through out the system is an inherent features of a digital system. Hence a macro-description of standard modules (MSI/LSI commercially available packages) or custom built modules is necessary in *SIMLOGS*. Boolean description has been conceived as a pre-processor for *SIMLOGS* to avoid complicating the implementation unnecessarily.

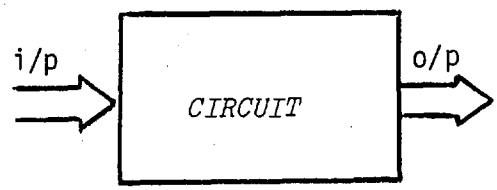
Primitive Elements

AND	5 inputs	AND	gates
NAND		NAND	
OR		OR	
NOR		NOR	
NOT			
EXOR		Exclusive-OR	
JK		Clocked JK flip-flop with Set and Preset	
SHR		4-bit shift register with shift right & parallel transfer facility	
HADD		1-bit half Adder	
ROM		a 64 x 8 (8-bit word) Read-Only Memory with 6 address bits & 1 enable ('*PROGRAMMING*' & reading the *ROM* is affected by commands *PROGRAM* and *PROBE*)	

Table 7 :

Primitives of *SIMLOGS*

User's (or designers') interaction with *SIMLOGS* is a major consideration; the purpose of a simulator is to provide a design aid for engineers. Therefore details of computing should not distract the designers's activities. A circuit may be conceived as a block having inputs and outputs as depicted in figure 37.

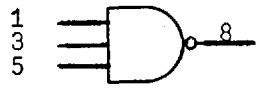


(a) Block Structure

<OUTPUTS> *CIRCUIT* <INPUTS>

Z *ID* *X*

where: *X* = input terminals
Z = output terminals
ID = identification



(b) Schematic Diagram

8 *NAND* 1 3 5

(c) Symbolic Description

Figure 37 : Block Structure, Schematic Diagram and Symbolic Description of a Circuit

Symbolically, it may be described as follows :

< Output terminals > *CIRCUIT* < Input terminals >

and syntactically as :

Z *IDENTIFICATION* *X*

An example of the schematic and symbolic description of a *NAND* gate is shown in figure 37(b). Generalising this concept, we have the block description (figure 38) of a *CIRCUITX*, having its specification as the first statement, *INPUTS* as a second statement, and *OUTPUTS* as the last statement.

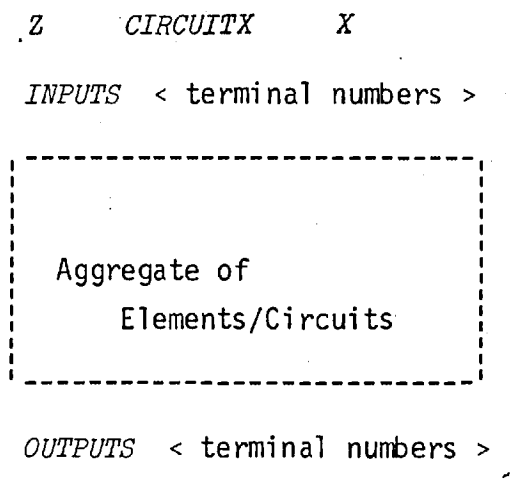


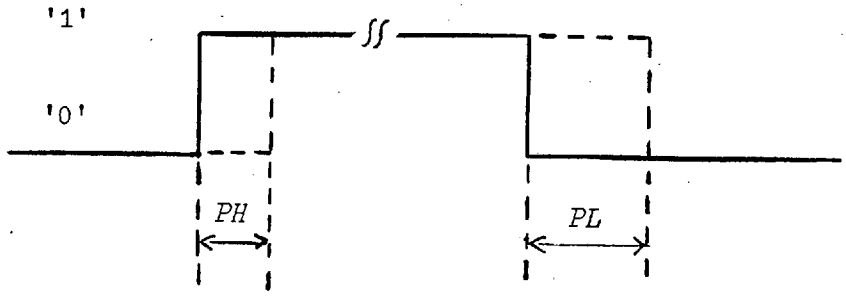
Figure 38 : General Interpretation of
Z ID X

Between the *INPUTS* and *OUTPUTS* specifications, there may exist any number of statements, each of which symbolically describes the gate/element being used and its input/output connections. These statements together specify an aggregate of elements/circuits forming the *CIRCUITX*. By 'aggregate' we also imply that the ordering of these statements are not important; but the relative position (inferred by the input/output terminal connections) of each with respect to *CIRCUITX* is important. Observe that *CIRCUITX* statement has the same syntax as *Z IDENTIFICATION X* ; thus *CIRCUITX* may be referred to as a composite element between the *INPUTS/OUTPUTS* statements of another circuits, say *CIRCUITY*. Observe that the syntax of the *CIRCUITX* statement is dyadic, while that for the main circuit is nyadic. The main circuit invokes

a sub-circuit (e.g. *CIRCUITX*) by name, and is itself referenced as system (table 8) by *SIMLOGS* as *SYSTEM*.

4.2.2 Physical Considerations of the Model

Timing (propagation delay, rise/fall time), electrical loading/rating, noise-margin, temperature tolerance are physical properties associated with digital circuits. Differences in technology, say *TTL* and *ECL*, in the same network may result in accentuating the variation of these physical attributes. Zero or unit delay simulators can be used for logic verification, but are not effective for design verification (design verification refers to accurate timing analysis as opposed to simply verifying logical functions). Therefore, this necessitates the implementation of assignable or precise timing models. Furthermore, spike analysis (the case where a set of inputs changes to an element are attempting to force the element to perform a signal transition more rapidly than its delay characteristics might allow), can be accomplished using assignable delays. *SIMLOGS* permits the specification of delays for a transition (*PH*) from logical "0" to "1" and for a transition (*PL*) from logical "1" to "0" (Fig. 39).



PH ↔ propagation delay from logical '0' to '1'

PL ↔ propagation delay from logical '1' to '0'

Figure 39 : Delay Parameters

Furthermore, to provide more precise timing, *SIMLOGS* allows the specification of nominal (*NOM*), minimal (*MIN*) and maximal (*MAX*) values for both *PH* and *PL*; resulting a total of 6 timing parameters. These parameters are specified via

SYSTEM	Identifies the system or main circuit to be simulated N.B. Names other than these chosen for the SIMLOGS command set maybe used to identify subsystems/ sub-circuits (e.g. *SN*5472 being a *TTL=GATED JK FLIP-FLOP*)
INPUTS	Specifies the input terminal of the associated subsystem/system
OUTPUTS	Specifies output terminals; corresponds to command *INPUTS*
SETDATA	Specifies the initial conditions & dynamic (i.e. the input signals during simulation) conditions of the system being simulated Used in conjunction with commands *CLOCK*
CLOCK	Specifies clock leads when used in the system; also used to specify mark/space (or *ON/OFF* durations) ratios
CYCLE	Specifies the cycling rate of the terminals; i.e. whenever the system reaches a steady state, all terminals referenced reversed their states
PDELAY	Specifies the propagation delays or timing parameters of the circuit/circuit elements (Total of 6 parameters : Maximal, Norminal, Minimal parameter for 0→1 and 1→0)
TIMINGS	Sets all terminals (where *PDELAY* are specified) to any of : *MAX*,(maximum) *MIN*,(minimum) or *NOM*(norminal)
DELAY	Sets selected terminals to *MAX*, *MIN* or *NOM*
HOLD	Sets selected terminals to *MAX*, *MIN* or *NOM* hold time
SELECT	*DELAY/HOLD*, if specified, are effected
RANDOM	Sets all timings to *MAX* or *MIN* at random
RESET	Reset all simulation condition to initial conditions and all timings to norminal values
BUILD	Constructs the simulation model specified by the user (i.e. compile specification to data structures ready for simulation)
ON	Starts exercising the simulation
PRINT	Specifies State Table printout for selected terminals
PRINTALL	Specifies State Table printout for all terminals
WAVE	Specifies wave forms to be printed out
BLANK	Used in conjunction with *PRINT* or *PRINTALL* format state tables with blank columns
PROGRAM	Programs a selected *ROM*
PROBE	Examine contents of a selected *ROM*

Table 8 : *SIMLOGS* Commands

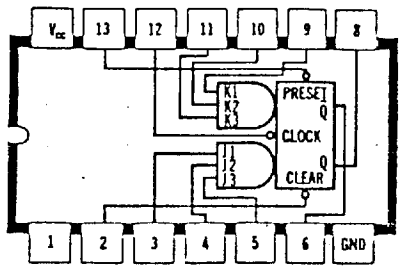
the *SIMLOGS* command *PDELAY*. The syntax of *PDELAY* is :

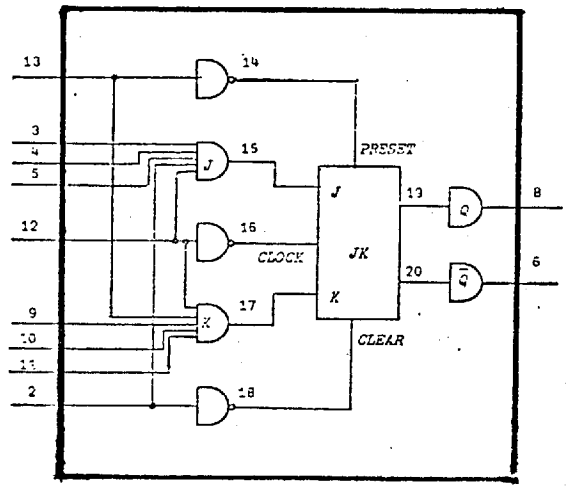
PDELAY $\underbrace{< PH\ PL >}_{NOMINAL}$ $\underbrace{< PH\ PL >}_{MINIMAL}$ $\underbrace{< PH\ PL >}_{MAXIMAL}$

As an illustration, the following specifies a *NAND* gate having delays of 20, 20, 10, 10, 30, 30 units :

```
6 NAND 4 5, PDELAY 20 20 10 10 30 30
```

Figure 40 shows the specification of a *JK* master-slave flip-flop with *AND* inputs (*SN*5472).






```
∇ Z SN5472 X
[1] INPUTS 13 3 4 5 12 9 10 11 2
[2] 14 18 NAND 13 2,  PDELAY 0
[3] 15 17  AND  3 4 5 12 2 13 9 10 11,
    PDELAY 16 25 10 10 25 40
[4] 16  NAND 12, PDELAY 0
[5] 8 6 AND  19 20, PDELAY 16 25 10 10 25 40
[6] 19 29 JK 15 16 17 14 18
[7] OUTPUTS 8 6
∇

⍝ NOTE:
⍝   X[1 2 ... 8 9] ↔ INPUTS 13 3 ... 2
⍝   Z[1 2] ↔ OUTPUTS 8 6
```

Figure 40 : Specification of *SN*5472 Using *SIMLOGS*

A further refinement of timing involves the capability of assigning all gates or elements to the nominal, or minimal, or maximal delays; alternatively, a mixture of these delays for the complete circuit or for some of the elements in the circuit. *SIMLOGS* facilitates these by :

TIMINGS { *MAX* / *MIN* / *NOM* } for setting all elements to { maximal / minimal / nominal } values

By default, all timings are initially set to nominal values. The following

< terminal no. > { *DELAY* / *HOLD* } { *MAX* / *MIN* / *NOM* }

sets the terminal(s) in the main module to the appropriate delay (i.e. *PL*) or hold (i.e. *PH*) times. When *HOLD* and *DELAY* are used in conjunction with command *SELECT* < module no. >, then *HOLD* & *DELAY* apply to the specified module. Note that *PDELAY* is used during the specification of the circuit/system prior to simulations. While *TIMINGS*, *DELAY*, *HOLD* and *SELECT* are used immediately before a simulation-run. Finally, *SIMLOGS* allows timing to be randomly varying between maximal and minimal values by the command *RANDOM*.

The choice of delay parameters directly affects the number of values used in the simulators. Zero unit, or nominal delay models can use either two or three values for simulations. The values would be 0, 1 and X; where X designates an indeterminate value or unknown. Two-value simulator has the major drawback of 'initialization' of a logical circuit to be simulated; since only two values exist, all signals would be initially set to 0 or 1. Another drawback is that X's due to spikes, unknown, etc. cannot be simulated because signals can assume only binary values (0 or 1). A third-value X may be introduced to overcome the first difficulty. *SIMLOGS* uses 5 values to represent :

0	↔	steady at logical '0'
1	↔	steady at logical '1'
↑	↔	transition from '0' to '1'
↓	↔	transition from '1' to '0'
?	↔	reversed while rising or falling (indicating timing hazard)

Although it is desirable to provide the facility to model and to simulate other physical faults such as electrical loading etc., yet during the initial stage and in particular the exploratory stage of a design verification, timing analysis is the most important factor to be considered. Provisions for other physical attributes are of secondary value but will increase the complexity of *SIMLOGS* unnecessarily.

4.2.3 Implementation & User Environment

APL (A Programming Language) (45-56) has been chosen as a computer language to implement *SIMLOGS*. The richness of the language, the array oriented and its dynamic data structure enable *SIMLOGS* to be implemented easily. The syntax of the language provides a close correspondence between circuits and the language as discussed in the last section. Furthermore, *SIMLOGS* commands (Table 8) may be formulated to resemble the jargons which are familiar to designers. The terminal implementations of *APL* (e.g. *APL*\360, *XEROX APL*, *SHARP APL*, *APL PLUS*, *APLUM*, etc.) allow *SIMLOGS* to be 'portable'. Source code listing of the basic version (see below) of *SIMLOGS* is included in appendix 14.

The environment of *SIMLOGS* is depicted in figure 41. Version 1 Level 0 consists of one workspace containing [1] Logic Description Module[1], [2] Simulation Module, [3] *SIMLOGS* Circuit Library and [4] User Circuit Library. The *SIMLOGS* Circuit Library contians a number of symbolic descriptions of commercially available MSI/LSI packages (e.g. *SN*74155, *SN*74184, *SN*74194). These packages are defined using the primitives and definition commands (e.g. *AND*, *NOR*, *ROM*, *JK*1, *SHR*, *PDELAY*, *INPUTS*, *OUTPUTS*) in the Logic Description Module. Specialized circuits may similarly be defined and are stored in the User Circuit Library by the user for future applications. Hence, during the digital system definition process, [1], [3] and [4] are interrelated. [1] refers and retrieves predefined circuits stored in libraries [3] and [4] in order to construct the digital system to be simulated. The circuits are only copied from any or both librarios. The correctness of wiring up the system is then checked by [1], when the user issues the system command *BUILD*. The Simulation Module [2], simulates the system which has already been checked by *BUILD* in [1]. As such, [2] performs simulations as an independent module and is related to [1], [3] and [4] by processing their results.

The integration of [1], [2], [3] and [4] as one workspace (Version 1, Level 0) reduces the otherwise computing details and routine operations that are required of the user when they reside in separate workspaces. On the other hand, they are separated (Version 1 Level 1) to overcome memory constraints of the active workspace (typically 32K bytes) as in the previous level (Level 0). However, level 0 is implemented because, for a simple system simulation, it is the least costly in terms of overheads and the need of accessing different workspaces. Since only basic APL (or APL\360) is used, Level 0 has an additional advantage of being a 'portable' package. Level 1, however, takes advantage of the "overlay" capability of an extended APL (e.g. XEROX APL,

---

[1] [ ] designates a component of *SIMLOGS* and corresponds to that depicted in figure 41. For example, [2] designates the "Simulation Module" of

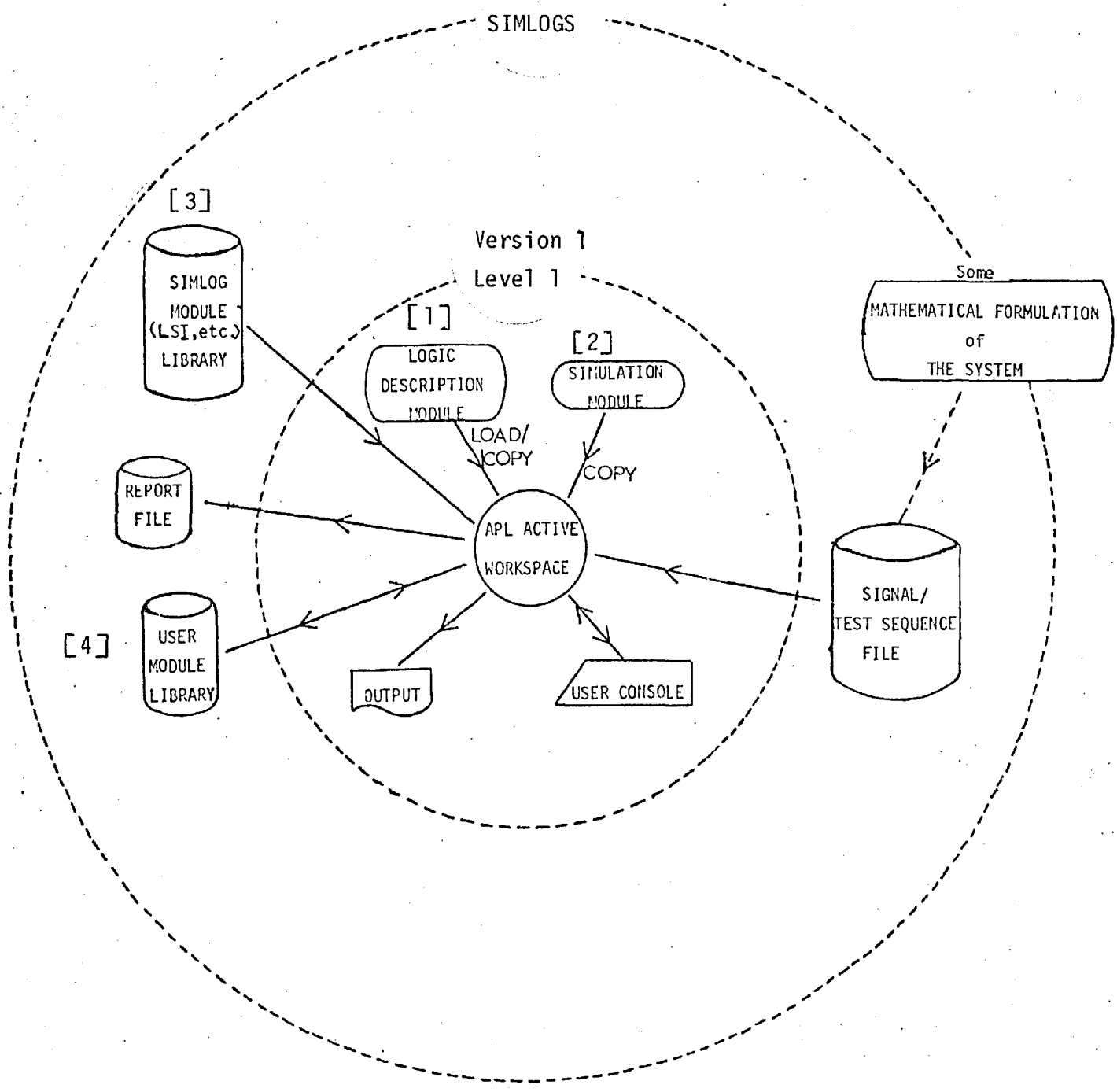


Fig. 41 : The Environment of SIMLOGS :
A Simulator for Logic System

APLUM). To facilitate the possibility of handling a large amount of data, APL files are used in Level 2 in lieu of workspaces for libraries [3] and [4]. Signal and report files are also incorporated as interfaces to the other subsystems of the "Top-down Design Approach".

4.3 APPLICATIONS & FUTURE DEVELOPMENTS OF *SIMLOGS*

4.3.1 Logic Design Aided by *SIMLOGS*

Verification and testing are the two major areas of the logic design process in which *SIMLOGS* can assist the designer in evaluating the preconstructed system. Although logic verification is the emphasis in this writing, it is fruitful to see the role of verification in the logic design process. The realization of a digital system has been obtained as the result of a register transfer level (Fig. 2) design or followed by some form of logic minimization. Timing (i.e. the finite propagation delays associated with a digital circuit) is the most important consideration to implement boolean functions which describe a digital system. By introducing timing, the correctness of an expected implementation is assured. When the timing information of individual elements (e.g. *NAND*, *NOR*, *EXOR*, Flip-Flops, *ROM*) is provided, *SIMLOGS* can predict the possible existence of critical races, spikes or hazards in the overall system. The delays of real logic elements fall within a range of values. This is considered by the propagation delay (*PDELAY*) command in *SIMLOGS*. A circuit module (e.g. decoder, counter) of a commissioned digital system is often necessary to be replaced by a spare. This spare component may have different delay characteristics due to a manufacturing tolerance. This gives rise to the problem of interchangeability. By including delay tolerances in the verification, this problem can be reduced.

One approach of test verifictions is to define diagnostic sequences which will intentionally cause partial or total operational failure in a digital system (79). The number of these 'fail modes' theoretically approaches infinity. In practice, however, under certain situation, the number of failure modes can be reduced to a manageable size. This approach then lends itself nicely

to the analysis by a simulator such as *SIMLOGS*.

Verification and testing often initiate repeated adjustments (or modifications) to the original design. This is briefly depicted in figure 42.

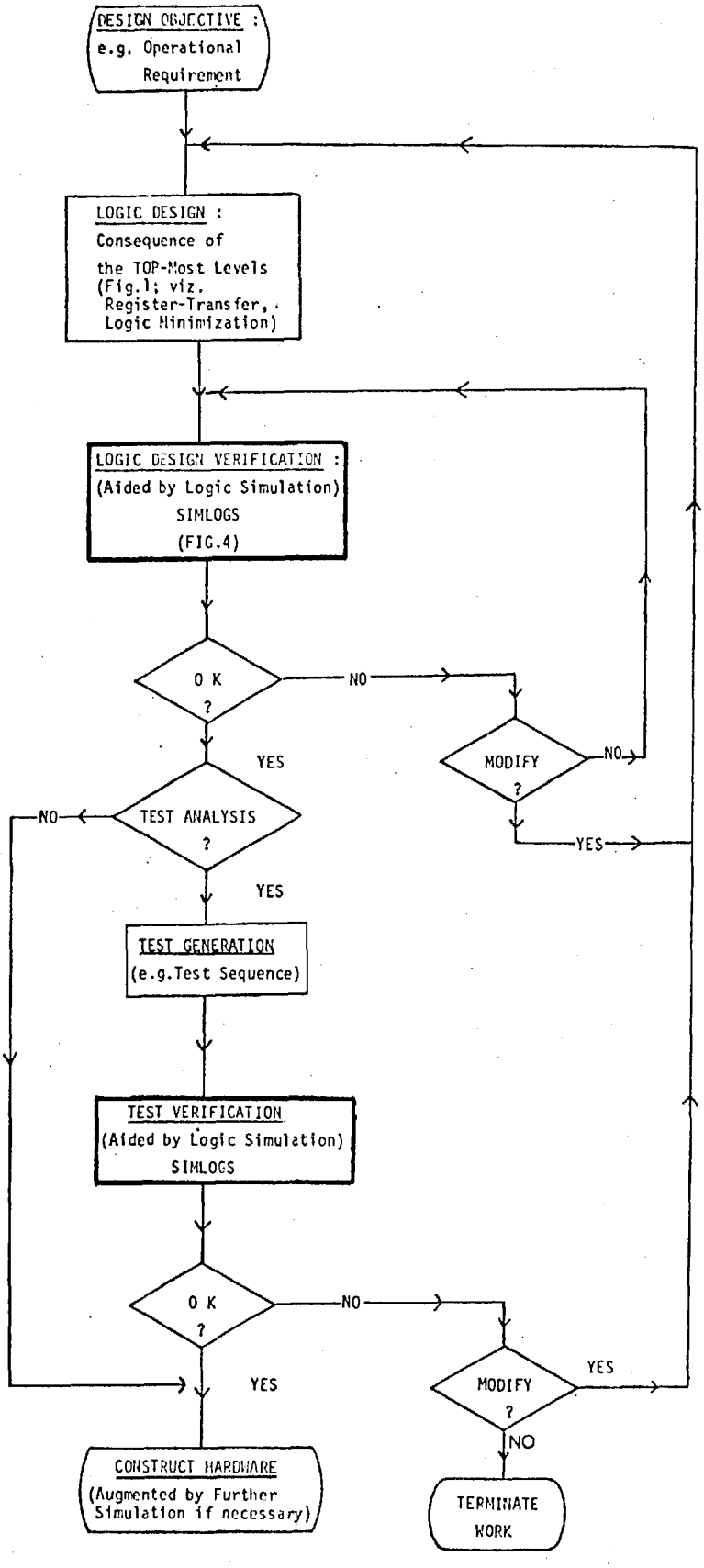
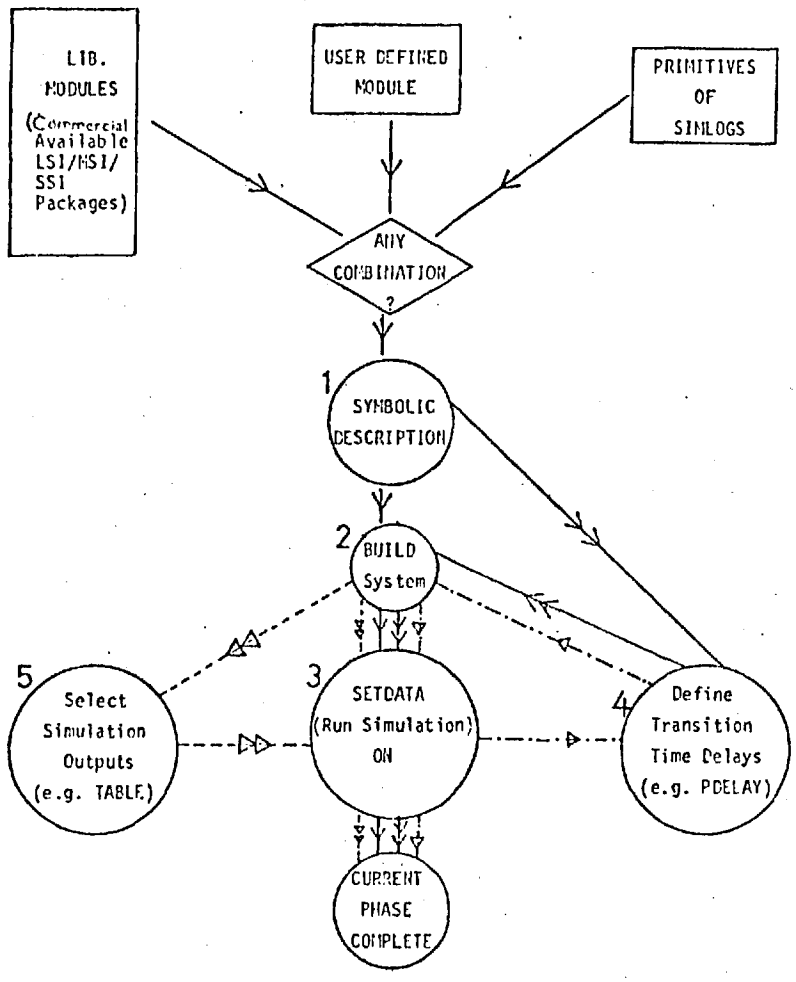


Fig. 42 :

A Digital System Design using Logic Simulation

4.3.2 *SIMLOGS* as an Aid to Verify a Digital System Design

The dual roles of *SIMLOGS* in the logic design process (fig. 42) are verification and testing. The significance of verification is further elaborated in figure 43, while testing serves a complementary function in the rest of the design process. Verification commences when the designer issues the command *)LOAD SIMLOGS* at the user's console. In response to the question *WHICH LEVEL?* the designer can choose any of the options (Level 0, 1 or 2) discussed in section 4.2.3. Library Modules, User Defined Modules and Primitives of *SIMLOGS* in figure 43 correspond to a single workspace, or multiple workspaces, or APL files and workspaces as discussed in section 4.3.1 and depicted in figure 41. With reference to figure 43, the procedures of verification (common to all levels) will be discussed.



Note :

Numbers 1-5 refer to Step 1-5 in the text

Fig.43 :

Logic Design Verification using *SIMLOGS*

STEP 1 : Symbolic Description of a System

The structure of the Symbolic Description consists of 1) Identification, 2) Input terminals declaration, 3) Internal or intra-connections of circuit elements (e.g. *NAND*, *NOR*, *ROM*), and 4) Output terminals declarations. This structure corresponds to the black-box structure of a system (or circuit module) being simulated. The general form of Identification (*ID*) is : *Z ID X* where *X*,*Z* are respectively input, output terminal numbers which are referred to in the system. If *ID* was the complete system, it is explicitly indicated as *SYSTEM* in lieu of *Z ID X*. Input and Output declarations assume the form of :

INPUTS } <List of terminal numbers>
OUTPUTS }

While the intra-connection of a circuit element takes the same form as *Z ID X*. Figure 44b indicates the description of the gated J-K flip-flops shown in figure 44a. Command *PDELAY* specifies the norminal, minimum, and maximum propagation delay times (i.e. from 0-to-1 and 1-to-0, or respectively PH, PL). The first two numbers following *PDELAY* define

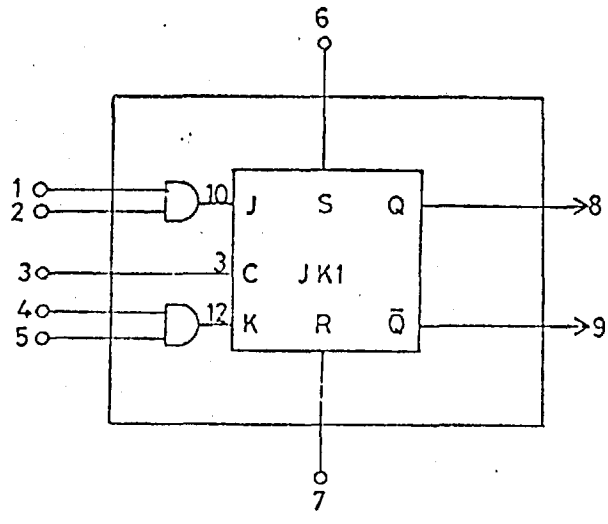


Fig. 44(a) :
A Gated JK Flip-flop

```
     ▽ SYSTEM
[1]    INPUTS 1 2 3 4 5 6 7
[2]    8 9 JK1 10 3 12 6 7
[3]    10 12 AND 1 2 4 5,PDELAY 1 1 1 1 1 1
[4]    OUTPUTS 8 9
     ▽
```

Fig. 44(b) :
Symbolic Description of the Flip-flop in (44a)

```
    ∇ SETDATA
[1]   X[3]←1 CLOCK 1
[2]   X[1 2]←2 CLOCK 2
[3]   X[4 5]←4 CLOCK 4
[4]   X[6]←10 CLOCK 10
[5]   X[7]←20 CLOCK 20
    ∇
```

Fig. 44(c) : Input Signal to the JK Flip-flop

TERMINALS 6	7	1	2	4	5	3	8	9	EFFECTIVE TIME
0	0	0	0	0	0	0	1	0	1
0	0	0	0	0	0	1	1	0	2
0	0	1	1	0	0	0	1	0	3
0	0	1	1	0	0	1	1	0	4
0	0	0	0	1	1	0	1	0	5
0	0	0	0	1	1	1	1	0	6
0	0	1	1	1	1	0	1	0	7
0	0	1	1	1	1	1	0	1	8
0	0	0	0	0	0	0	0	1	9
0	0	0	0	0	0	1	1	0	10
0	1	1	1	0	0	0	1	0	11
0	1	1	1	0	0	1	1	0	12
0	1	0	0	1	1	0	1	0	13
0	1	0	0	1	1	1	1	0	14
0	1	1	1	1	1	0	1	0	15
0	1	1	1	1	1	1	1	0	16
0	1	0	0	0	0	0	1	0	17
0	1	0	0	0	0	1	1	0	18
0	1	1	1	0	0	0	1	0	19
0	1	1	1	0	0	1	1	0	20
1	0	0	0	1	1	0	0	1	21
1	0	0	0	1	1	1	0	1	22
1	0	1	1	1	1	0	0	1	23
1	0	1	1	1	1	1	0	1	24
1	0	0	0	0	0	0	0	1	25
1	0	0	0	0	0	1	0	1	26
1	0	1	1	0	0	0	0	1	27
1	0	1	1	0	0	1	0	1	28
1	0	0	0	1	1	0	0	1	29
1	0	0	0	1	1	1	0	1	30
1	1	1	1	1	1	0	0	0	31
1	1	1	1	1	1	1	0	0	32

Fig. 44(d) : State Table (Output of *SIMLOGS*)

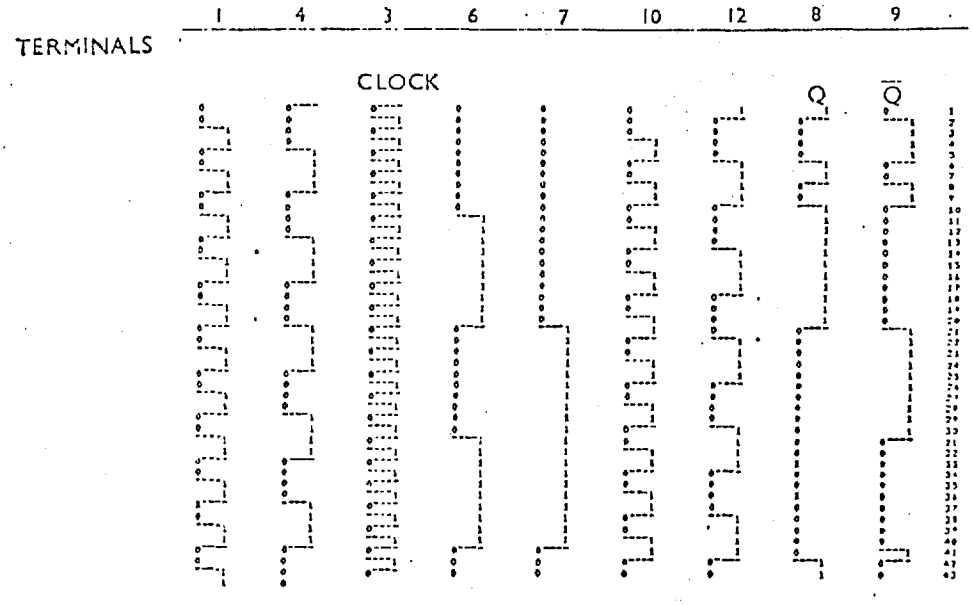


Fig. 44(e) : Timing Diagram (An alternative output of *SIMLOGS*)

Fig. 44 : Description and Simulation of a Gated JK Flip-flop

the norminal PH, PL; the next two for the minimum PH, PL and the last two for the maximum PH, PL. Propagation delay (or timing) is further discussed in Step 4.

STEP 2 : Construct and Check Correctness of Inter-Connection

The correctness of inter-connections (i.e. consistency in terminal number specifications) is checked by entering command *BUILD*. If no error is detected, *SIMLOGS* will report with the message ... *SYSTEM CHECKED. READY.* Interconnection errors are indicated by ** *ERRORS IN* : followed by a list of errors (terminal-module number pairs) detected by the Logic Description Module (fig. 41). After the appropriate amendments by the designer, the system may be reconstructed by issuing *BUILD*.

STEP 3a : Set Initial Conditions and Input Signals

The initial conditions may be set by the *SET* command or specified along with the input signal using *SETDATA*. Figure 44c shows the input signals to the gated J-K Flip Flop; where *X* defines the input terminals (e.g. *X*[6]↔ input terminal 6 in the *SYSTEM*). The *CLOCK* command requests for a binary ON-OFF signal (e.g. 2 *CLOCK* 1 specifies a mark/space ratio of 1:2).

STEP 3b : Run Simulation with Default Printout

The simulation is started by issuing the *ON* command. *SIMLOGS* by default, prints out the state table for each of the terminals in the system.

STEP 4 : Define Transition Time Delays and Timing Evaluation

Hazards (occuring at any instant of time) in the system are indicated by a * at the associated terminal and after the 'Effective Time' column of the state table or timing diagram. The designer may then adjust the *CLOCK* speed or the propagation delays using *PDELAY*, *TIMINGS*, *SET*, *HOLD*. Nominal propagation delays are, by default, used in the simulation. Minimum or maximum timings for all elements in the *SYSTEM* may be chosen by command

TIMINGS MIN or *TIMINGS MAX* respectively. Timing values for a particular circuit module may be defined by first typing *CHOOSE* < module number > following by *SET*, *HOLD* commands. Timing may also be varied at random (according to a uniform distribution) by the command *RANDOM*.

STEP 5 : Select Simulation Output Options

Selected terminal may be examined by typing *TABLE* < a list of terminal numbers > before typing *ON*. A timing diagram output, instead of a state table, can be obtained by typing *WAVE* < a list of terminal numbers > before entering *ON*. Figure 44d & figure 44e respectively shows the state table and timing diagram of the gated J-K Flip Flop simulation.

Not every step in the simulation procedure (discussed above) is required to be executed. Figure 43 in fact indicates a number of alternatives in using *SIMLOGS*. The macro-description facility of *SIMLOGS* enables a designer to define and to verify a system which may later be used as a subsystem to construct a more complex digital system. Figure 45 shows a synchronous counter which employs the gated J-K Flip Flop as one of its components.

4.3.3 Future Development of *SIMLOGS*

The philosophy and environment of *SIMLOGS* have been presented. We believe that the fundamental requirement of *SIMLOGS* is to simulate the system "faithfully" even in the presence of component faults. It therefore can play its important dual roles (verification and testing) in logic system design. However, it is only a design tool and therefore user's requirements should be considered in the formulation and implementation of the Simulator. Since simulation techniques have become an increasingly important design tool, further development on *SIMLOGS* should remain focusing on extending the diagnostic facilities in logic verification and in particular, testing. With many systems, increased assurance of design correctness before (and even during) the manufacturing process begins is an economic necessity.

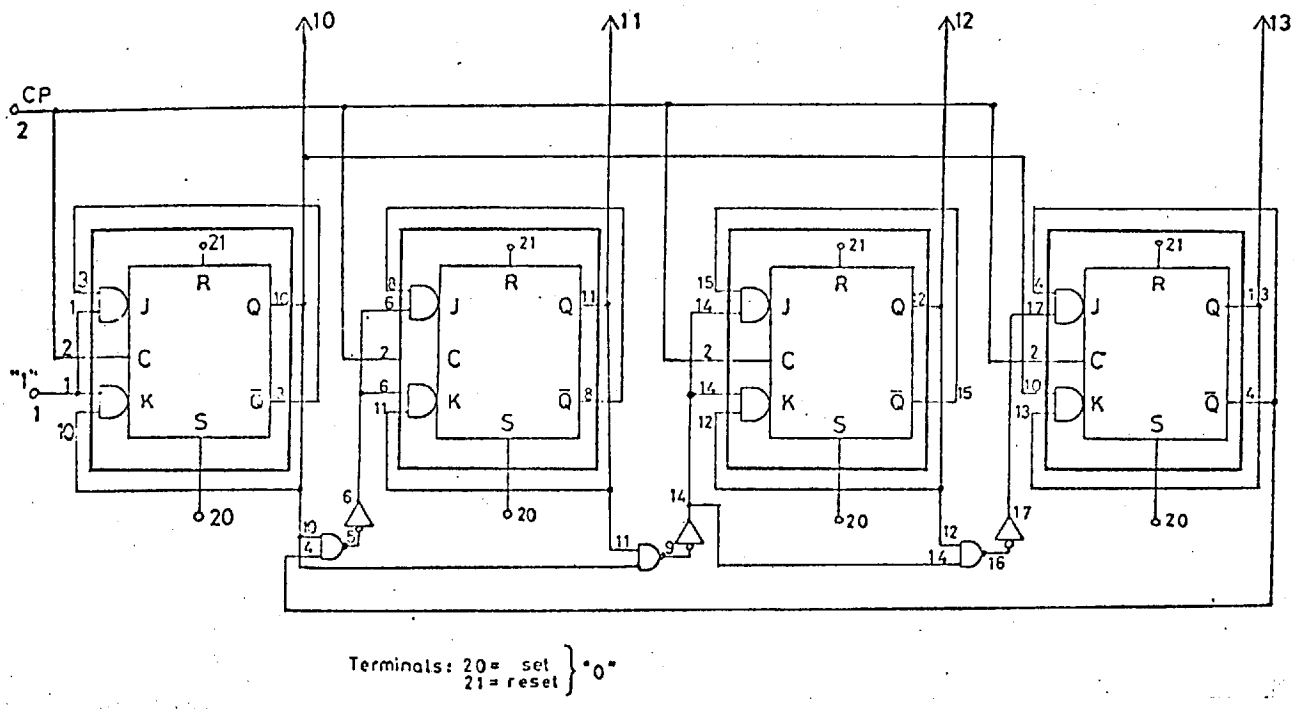


Fig. 45(a) : A Synchronous Decade Counter

```
    ∇ SYSTEM
[1]   ⍝
[2]   ⍝  SYNCHRONOUS DECADE COUNTER
[3]   ⍝
[4]   ⍝ REQUIRES:
[5]   ⍝  (1) 4 GATED JK FLIP-FLOPS
[6]   ⍝  (2) 3 NAND GATES
[7]   ⍝  (3) 3 NOR GATES
[8]   ⍝
[9]   ⍝ INPUTS:
[10]  ⍝  1    ↔  INPUT TO 1ST STAGE='1'
[11]  ⍝  2    ↔  CLOCK INPUT TO COUNTER
[12]  ⍝ 20    ↔  SET COUNTER TERMINAL='0'
[13]  ⍝ 21    ↔  RESET COUNTER TERMINAL='0'
[14]  ⍝
[15]  ⍝ OUTPUTS:
[16]  ⍝ 10,11,12,13  ↔  COUNTER OUTPUTS
[17]  ⍝                  (LSB  TO  MSB)
[18]  ⍝
[19]  INPUTS 1 2 20 21
[20]  10 3 CELL 3 1 2 1 10 20 21
[21]  5 NAND 10 4,PDELAY 16 25 10 10 25 40
[22]  6 NOR 5,PDELAY 16 25 10 10 25 40
[23]  11 8 CELL 8 6 2 6 11 20 21
[24]  9 NAND 11 10,PDELAY 16 25 10 10 25 40
[25]  14 NOR 9,PDELAY 16 25 10 10 25 40
[26]  12 15 CELL 15 14 2 14 12 20 21
[27]  16 NAND 12 14,PDELAY 16 25 10 10 25 40
[28]  17 NOR 16,PDELAY 16 25 10 10 25 40
[29]  13 4 CELL 4 17 2 10 13 20 21
[30]  17 NOR 16,PDELAY 16 25 10 10 25 40
[31]  OUTPUTS 10 11 12 13
    ∇
```

Fig. 45(b) :

Symbolic Description of the Synchronous Decade Counter

```
    ∇ SETDATA
[1]   X[2]←1 CLOCK 1
[2]   X[1 3 4]←1 0 0
    ∇
```

Fig. 45(c) :

Input Signal to the Synchronous Counter

```
    ∇ Z CELL X
[1]   ⍝
[2]   ⍝   GATED JK FLIP-FLOP
[3]   ⍝
[4]   ⍝ REQUIRES:
[5]   ⍝   (1)  DUAL 2-INPUT AND GATES
[6]   ⍝   (2)  1 JK1 FLIP-FLOPS; R=S=0 GIVES
[7]   ⍝        OUTPUTS OF (0,0)
[8]   ⍝
[9]   ⍝ INPUTS:
[10]  ⍝   1,2  ↔ 1 ⍝ 2 = J
[11]  ⍝   4,5  ↔ 4 ⍝ 5 = K
[12]  ⍝     3  ↔ CLOCK INPUT
[13]  ⍝     6  ↔ SET
[14]  ⍝     7  ↔ RESET
[15]  ⍝
[16]  ⍝ OUTPUTS:
[17]  ⍝   8 ↔ Q
[18]  ⍝   9 ↔ Q' (COMPLEMENT OF Q)
[19]  ⍝
[20]  INPUTS←7
[21]  8 9 JK1 10 3 12 6 7
[22]  10 12 AND 1 2 4 5,PDELAY 16 25 10 10 25 40
[23]  OUTPUTS 8 9
    ∇
```

Fig. 45(d) :

Description of a Gated JK Flip-flop used as a "building block" -- to construct the Decade Counter in 45(a)

TERMINALS	10	11	12	13
	0	0	0	0
	0	0	0	0
	1	0	0	0
	1	0	0	0
	0	1	0	0
	0	1	0	0
	1	1	0	0
	1	1	0	0
	0	0	1	0
	0	0	1	0
	1	0	1	0
	1	0	1	0
	0	1	1	0
	0	1	1	0
	1	1	1	0
	1	1	1	0
	0	0	0	1
	0	0	0	1
	1	0	0	1
	1	0	0	1
	0	0	0	0
	0	0	0	0

Fig. 45(c) :

State Table (Output of *SIMLOGS*)

Fig. 45 : Description & Simulation of a Synchronous Decade Counter

Chapter 5 STUDIES OF DIGITAL FILTER IMPLEMENTATION USING RESIDUE NUMBER SYSTEMS [1]

Summary

The residue number systems (RNS) were independently discovered by Garner (86) and Svoboda (93) towards the end of the fifties. Since then there has been considerable theoretical interest in the application of the RNS to implement digital arithmetic processors. The interest stems from the fact that binary operations of addition, subtraction, and multiplication can be performed very quickily because of the separability of operations on each residue digit and if redundant residue methods are employed, then error checking and correcting structures can be implemented. RNS, however, is essentially an integer number system; as such the notion of radix point has no meaning. Consequently there are the associated difficulties of implementing operations such as division & scaling and sign or magnitude determination. Despite much efforts (81-102) in solving these problems, it is not clear that residue number systems offer any striking advantages in the implementation of general purpose arithmetic units.

On the other hand, one may reformulate the application of RNS to a particular class of problems. Such a class is one consisting mainly of the binary operations of addition, subtraction and multiplication with as few as possible scaling operations; while explicit sign or magnitude determination is not allowed. An example of a digital correlator (97,98) which permitted only easy operations while the number range was so large that eliminated the need for scaling. For more general digital signal processing problems, however, the need for scaling is essential; yet the requirements are not so stringent as the general purpose arithmetic units that signal processing applications still fall within the special class.

---

[1] This chapter is based on two reports (15,16) written by this author.

This has prompted efforts spent in studying the application of RNS to implement digital filters. This chapter is an effort in considering the characteristics and algorithms of RNS. The design of special logic networks and the use of read-only-memory technology to implement RNS operations are respectively presented in appendices 16 and 17. While the fundamentals of the RNS are included in appendix 15. The intention of presenting our studies of RNS in this manner is to avoid a possible disjoint appearance of the chapter. As remarked in chapter one, the emphasis of the research is on a systematic approach of implementing digital filters. Therefore within this frame work, our studies in this chapter is one aspect of the Top-down design approach.

Notes

1. Equations are numbered either in decimal numeric or a roman numeric and a decimal number; e.g. (4.2) or (II.2). The former designates equations in the text of chapter 15 while the latter references those in appendix 15. The roman number refers to the section within the appendix.

2. Integer numbers enclosed by brackets designate literature in the reference section of this thesis.

5.1 INTRODUCTION TO RESIDUE NUMBER SYSTEMS (RNS)

In any arithmetic operation, there implicitly assumes two concepts: the quantities being operated upon and the rules according to which the quantities are being manipulated. Mathematically, we speak of the representation of the numbers and the structure of the number system. By "structure" we mean the set of rules or algorithms that can be used to yield the representation of any number of interest in the system defined. The familiar examples are the decimal number system and the binary number system and the structure of each consists of addition, subtraction, multiplication and division. Although these two systems are different in form (i.e. the representation of the numbers) yet they have identical characteristic; more precisely, they are both weighted (or positional or polyadic) number systems. Contrast to them, there exist other systems; namely the mixed-radix number system (XRNS) and the residue number system (RNS).

Residue number system is the main interest of this chapter; but we can have better insight of RNS if we first briefly review the characteristics of other number systems. This chapter is compiled from references 81 through 102.

5.1.1 The weighted and the mixed-radix number systems

A weighted number system is one in which there exists a set of weights R_i such that any number X, in the system, can be expressed as,

$$X = \sum_{i=1}^{\infty} A_i R_i \qquad (1.1)$$

where $0 \leq A_i \leq R_i - 1$ and A_i are known as the digits while R_i are also known as radices or bases of the system under consideration. Two immediate comments of (1.1) are: (1) X has been assumed to be an integer and (2) the system is infinite which means X can be represented with any desired accuracy. In

practical computation, the finite (or truncated) integer system is assumed, so that (1.1) is modified to:

$$X = \sum_{i=1}^{n} A_i R_i \qquad (1.1a)$$

where n is the maximum number of digits allowed to represent X. In terms of machine usage, n denotes the word-length of the machine. Rational numbers (such as 0.5 in the decimal number system) are represented by introducing a radix-point to the system. If the value of R_i are successive powers of the same number then the system is said to be fixed-base or fixed radix; so that (1.1) modifies to,

$$X = \sum_{i=1}^{n} A_i R^{i-1} \quad ; \quad R^0 = 1 \qquad (1.1b)$$

In equation (1.1) through (1.1b), positive integers are assumed. We shall discuss the representation of negative numbers or signed-number representation in section 5.3.

When the weights are not powers of the same radix but the product of a selected set of numbers, the system is called mixed-radix number system. In this system corresponding to equation (1.1b), we have

$$X = P_n\left(\prod_{i=1}^{n-1} R_i\right) + \cdots + P_3 R_1 R_2 + P_2 R_1 + P_1 \qquad (1.2)$$

where R_i are the radices and P_i are the mixed-radix digits and are bounded by $0 \leqslant P_i < R_i$. A more comprehensive discussion of this system appears in appendix 15 (section II).

5.1.2 The residue number system (81-102)

The residue number system is based on the concept of linear congruence in a branch of mathematics called the theory of numbers.(77-80) An integer x is represented by,

where

$$x = m\left[\frac{x}{m}\right] + |x|_m$$
$$0 \leqslant |x|_m \leqslant m-1 \qquad (1.3)$$

m is a positive integer called the modulus, $\left[\frac{x}{m}\right]$ denotes the largest integer less than or equal to $\frac{x}{m}$ and $|x|_m$ represents the least positive residue of x modulo m. $|x|_m$ is termed the residue of x with respect to modulus m. In Congruence notation, (1.3) is expressed as:

$$|x|_m \equiv x \pmod m \qquad (1.4)$$

If we now define an n-tuple of moduli $\mathcal{M} = (m_1, m_2, \ldots, m_n) \triangleq (m_i)_{i=1}^{n}$ with a corresponding ordered n-tuple of residues $\{|x|_{m_1}, |x|_{m_2}, \cdots, |x|_{m_n}\} \triangleq (|x|_{m_i})_{i=1}^{n}$ to represent x, then the n-tuple of residues constitutes a residue code of x. The representation is unique (theorem I.2; appendix 15). Therefore, we have a number system, called the RNS, which can uniquely represent a range of integers as in the weighted number systems; provided,

$$
\begin{aligned}
&(1)\quad \mathcal{M} = (m_i)_{i=1}^{n} \\
&\qquad \text{where } m_i \text{ are pairwise relatively prime;} \\
&(2)\quad 0 \leq |x|_{m_i} \leq m_i - 1; \\
&(3)\quad 0 \leq x \leq M - 1 \\
&\qquad \text{where } M = \prod_{i=1}^{n} m_i
\end{aligned}
\qquad (1.5)
$$

Concepts in this sub-section, together with other fundamentals of RNS, are reviewed in appendix 15 (section I). In section 5.2 we shall discuss the structure or the basic arithmetic of the RNS. To conclude section 5.1, we shall review the advantages and disadvantages which are consequences of the number representations inherent in the number systems.

5.1.3 Characteristics of the number systems considered (81-102)

For a given range of numbers, all number systems have unique number representations. The systems can be non-redundant or redundant. A non-redundant system is one in which every combination of the A_i digits (or the $|x|_{m_i}$ residues) represents a number and no two different sets of digits (or residues) correspond to the same number. In a redundant system, there are fewer numbers than the combination of digits (or residues).

The mixed-radix number system (XRNS) is essentially a weighted number system; hence possessing all the characteristic of the latter. Consequently the XRNS can be used as an intermediate system during the conversions between fixed-base number system and the RNS. Fixed-base number system can be considered as a special case of the XRNS (see appendix 15). The important

advantages of weighted number systems are:-

(1) Ease of relative-magnitude comparision of two numbers;
(2) Multiplication and division of a base can be achieved by shifting the digits in the storage; correspond to moving the radix point in a pencil-paper method;
(3) Extension of number range is easily realised by adding more digit positions;
(4) The logic required for performing addition is essentially identical for all digit positions;
(5) Overflow detection is easily implemented;

The main disadvantage, inherent in the number system is the interdependence of the digits. This implies that a truely parallel arithmetic operation is not feasible; because the carry-propagation from the least-significant digit to the most-significant digit must be allowed for. Hence there is a fundamental limitation of the speed of arithmetic.

To overcome this speed-limitation, we can employ (1) special circuitry to reduce carry-propagation time or (2) a number system with special carry characteristics.

A system with special carry characteristics is the RNS. As could be observed in section 5.1.2, the RNS is inherently carry-free; that is, each digit of the result is a function of only one digit from each operand (section 5.2), hence independent of all the other digits for addition, subtraction and multiplication. These arithmetic operations can be implemented using combinational logic.

The RNS, however, is an integer number system; consequently, rational number representation and operations are more involved. We shall discuss this in sections 5.3 and 5.4. RNS does not have all the inherent advantages of the weighted number systems; hence it is more suitable for applications which require less frequent use of the stated operations (i.e. the inherent advantages of the weighted number system). It is this special application of RNS which prompted the study in this research.

5.2 BASIC ARITHMETIC IN RNS

In appendix 15 (section I), the fundamentals of RNS were reviewed: attention was focussed on the algebra of residue and the manipulation of individual residue digits. In this section we shall consider the basic arithmetic performed on groups of residue digits representing a residue number. As stated in section 5.1, RNS is more suited for addition, subtraction and multiplication. Division is more involved but simplier for special cases. For special applications, such as digital filtering, division in its general form may be avoided hence its complexity is not a great impediment.

5.2.1 Addition and subtraction in RNS

Theorem 2.1: For an RNS having moduli $(m_1, m_2, \ldots, m_n)$ let X and Y be bounded by $0 \leq X, Y \leq M-1$, then $|X \pm Y|_M$ is given by

$$X \leftrightarrow \{ |X|_{m_1}, |X|_{m_2}, \cdots, |X|_{m_n} \}$$
$$Y \leftrightarrow \{ |Y|_{m_1}, |Y|_{m_2}, \cdots, |Y|_{m_n} \}$$
$$|X \pm Y|_M \leftrightarrow \{ ||X|_{m_1} \pm |Y|_{m_1}|_{m_1}, ||X|_{m_2} \pm |Y|_{m_2}|_{m_2}, \cdots, ||X|_{m_n} \pm |Y|_{m_n}|_{m_n} \} \qquad (2.1)$$

The result is unique if bounded by $0 \leq |X \pm Y|_M \leq M-1$.

Proof: By definition of residue,

$$|X \pm Y|_M \leftrightarrow \{ |X \pm Y|_{m_1}, |X \pm Y|_{m_2}, \cdots, |X \pm Y|_{m_n} \}$$

From equation (I.19),

$$\therefore |X \pm Y|_M \leftrightarrow \{ ||X|_{m_1} \pm |Y|_{m_1}|_{m_1}, ||X|_{m_2} \pm |Y|_{m_2}|_{m_2}, \cdots, ||X|_{m_n} \pm |Y|_{m_n}|_{m_n} \}$$

which is identical to equation (2.1). The uniqueness of $|X \pm Y|_M$ follows directly from theorem (I.2); this completes the proof.

This theorem shows that addition and subtraction of the residue code is isomorphic to addition and subtraction in the weighted number system.

5.2.2 Multiplication in RNS

Theorem 2.2: Let X,Y be bounded by $0 \leq X,Y \leq M-1$ in a RNS of moduli $(m_1, m_2, \ldots, m_n)$, then $|X \cdot Y|_M$ is given by

$$X \leftrightarrow \{ |X|_{m_1}, |X|_{m_2}, \ldots, |X|_{m_n} \}$$
$$Y \leftrightarrow \{ |Y|_{m_1}, |Y|_{m_2}, \ldots, |Y|_{m_n} \}$$
$$|X \cdot Y|_M \leftrightarrow \{ |\,|X|_{m_1} \cdot |Y|_{m_1}|_{m_1}, |\,|X|_{m_2} \cdot |Y|_{m_2}|_{m_2}, \ldots, |\,|X|_{m_n} \cdot |Y|_{m_n}|_{m_n} \} \quad (2.2)$$

if $0 \leq |X.Y|_M \leq M-1$ then $|X.Y|_M$ is unique.

Proof : By the definition of residue,

$$|X \cdot Y|_M \leftrightarrow \{ |X \cdot Y|_{m_1}, |X \cdot Y|_{m_2}, \ldots, |X \cdot Y|_{m_n} \}$$

Using equation (I.20),

$$\therefore |X \cdot Y|_M \leftrightarrow \{ |\,|X|_{m_1} \cdot |Y|_{m_1}|_{m_1}, |\,|X|_{m_2} \cdot |Y|_{m_2}|_{m_2}, \ldots, |\,|X|_{m_n} \cdot |Y|_{m_n}|_{m_n} \}$$

which is identical to equation (2.2). The uniqueness within the interval $[0, M-1]$ follows from theorem (I.2).

Hence multiplication in RNS is also isomorphic to that in the weighted number system.

5.2.3 Division in the RNS

Division in the RNS can be classified into:-

(A) Division Remainder Zero: the dividend is an integer multiple of the divisor which is relatively prime to M.

(B) Scaling: is the division of an arbitrary dividend by any factor of M which is a product of the moduli m_i; only products of the first power of the moduli are permissible.

(C) General Division: division of an arbitrary integer by an arbitrary divisor.

Items (A) and (B) are special cases with restrictive applicability, however the associated algorithms are simplier than (C). As "division" is not a basic operation in digital filtering, we shall not consider the general case (i.e.item C). Scaling is to be discussed in 5.3.2.

Division remainder zero

Subtraction can be performed by adding the "additive inverse"; likewise, division can be realised by multiplication of the "multiplicative inverse".

Theorem 2.3: Division by multiplicative inverse

$$\left|\frac{X}{Y}\right|_{m_i} = \left|\frac{1}{Y}\cdot X\right|_{m_i}$$

for all m_i, if and only if Y divides X (i.e. with no remainder or $a \backslash b$) and $(Y, m_i) = 1$.

(equation 2.3)

Proof: Assume $(Y, m_i)=1$ and $Y \backslash X$, then $(\frac{X}{Y})$ is an integer and represented by:

$$\left|\frac{X}{Y}\right|_M \leftrightarrow \left\{\left|\frac{X}{Y}\right|_{m_1}, \left|\frac{X}{Y}\right|_{m_2}, \cdots, \left|\frac{X}{Y}\right|_{m_n}\right\}$$

also $$|Y|_M \leftrightarrow \{|Y|_{m_1}, |Y|_{m_2}, \cdots, |Y|_{m_n}\}$$

Computing $(\frac{X}{Y}\cdot Y)$ in residue codes,

$$\therefore \left|\left(\frac{X}{Y}\cdot Y\right)\right|_M \leftrightarrow \left\{\left|\left|\frac{X}{Y}\right|_{m_1}\cdot |Y|_{m_1}\right|_{m_1}, \left|\left|\frac{X}{Y}\right|_{m_2}\cdot |Y|_{m_2}\right|_{m_2}, \cdots, \left|\left|\frac{X}{Y}\right|_{m_n}\cdot |Y|_{m_n}\right|_{m_n}\right\}$$

But,

$$\left|\left(\frac{X}{Y}\cdot Y\right)\right|_M = |X|_{m_i}$$

therefore

$$\left|\left|\frac{X}{Y}\right|_{m_i}\cdot |Y|_{m_i}\right|_{m_i} = |X|_{m_i}$$

By theorem I.3, the uniqueness of multiplicative inverse,

$$\therefore \left|\frac{X}{Y}\right|_{m_i} = \left|\frac{1}{Y}\cdot X\right|_{m_i}$$

which proves the theorem.

If Y does not divide X (i.e. $Y \not\backslash X$), $\left|\frac{X}{Y}\right|_{m_i}$ is not defined, consequently,

$$\left|\frac{X}{Y}\right|_{m_i} \neq \left|\frac{1}{Y}\cdot X\right|_{m_i}$$

Hence division in RNS, under the stated conditions, is also isomorphic to that in the weighted number system. Notice that the restriction (i.e. $Y \backslash X$) is inherent for an integer number system and the same applies to a finite integer weighted number system. In the next section we shall discuss the operations of rational number and is followed by a discussion of fractional multiplication in section 5.4.

5.3 GENERAL COMMENTS ON RATIONAL NUMBER OPERATIONS IN RNS

In section 5.1, we saw that RNS was an integer number system consequently multiplicative inverses were not defined (section 5.2). The problem can be reduced by using scaling to represent fractional quantities. It should be noted that, even with scaling, the operands in RNS must be integers and the results of the arithmetic must be expressible as integers. This maintains the advantage of computational speed of RNS but results two inherent disadvantage consequences: (1) quotients must generally be rounded to the closest integer value and (2) in most computations, the absolute value of the result will be larger than the input values used for calculations.

The latter fact is particularly important; the computer is required to have a number range significantly larger than that of the input information to keep the arithmetic result within range of the machine. For this reason, the concept and techniques of base-extension is required. Base-extension is discussed fully in appendix 15. We shall see how these techniques are used in Fractional multiplication in section 5.4.

Thus far in the discussion, we have implicitly assumed positive integer operations; purposely so, to simplify presentation. In section 5.3.1 we shall discuss sign representation; but in subsequent discussion, without lost of generality, we shall revert to the positive number representation for brevity.

5.3.1 Sign representation

There are, in general, two approaches to represent signed-numbers in RNS,

(1) Explicit sign representation (sign and magnitude),
& (2) Implicit sign representaion (complement).

Explicit sign representation: defines the situation where information apart from the number itself is used to denote the sign. This corresponds to using a "sign" and "magnitude" to denote a number in the weighted number system.

Implicit sign representation: this includes the sign as part of the number itself; corresponding to the "radix" and "radix-minus-one" complements in the weighted number system. Under this scheme, a number X within range,

$$0 \leqslant X \leqslant M - 1$$

may be partitioned into two parts; viz. $\left[0, \frac{M}{2}-1\right]$ and $\left[\frac{M}{2}, M-1\right]$ so that positive numbers are assigned to the first interval while negative numbers to the second interval.

5.3.2 Scaling

In the weighted number system, scaling up (or down) is to multiply (or divide) by a power of the radix. These operations are implemented by shifting (right or left) the operand and are fast and economical operations. In residue arithmetic; multiplication is a simple operation, consequently scaling up presents little or no problem. Division by a predetermined set of constants is faster than by an arbitrary divisor (sec. 5.2.3); in particular, when the constant set is the product of any moduli which comprise the total range M. This restricted operation of division is generally referred to as scaling (or scaling down) and the operation time is essentially one mixed-radix conversion or one macrocoefficient extraction time. Scaling by mixed-radix conversion and by macrocoefficient extraction will be considered individually.

The scaling problem:

From equation (I.3), division may be defined by:

$$X = \left[\frac{X}{Y}\right] Y + |X|_Y$$

where X is the dividend, Y is the divisor, $\left[\frac{X}{Y}\right]$ the integer of X over Y and $|X|_Y$ is the least positive integer remainder. Scaling is the process to find $\left[\frac{X}{Y}\right]$ given Y being a product of the first powers of the moduli. Rewriting the equation,

$$\left[\frac{X}{Y}\right] = \frac{X - |X|_Y}{Y} \qquad (3.1)$$

so that in residue representations,

$$\left[\frac{X}{Y}\right] \leftrightarrow \left\{ \left|\frac{X-|X|_Y}{Y}\right|_{m_1}, \left|\frac{X-|X|_Y}{Y}\right|_{m_2}, \cdots, \left|\frac{X-|X|_Y}{Y}\right|_{m_n} \right\} \qquad (3.2)$$

A. Scaling

Comparing equations (3.1), (3.2) with the tabulation or matrix (equation II.17) in appendix 15, we observe that the scaling algorithm consists of two parts, viz. division remainder zero and extension of base using mixed-radix conversion. We shall illustrate this by two numerical examples.

(1) Scaling by a single modulus

Given moduli $m_1 = 2$, $m_2 = 3$, $m_3 = 5$ and $m_4 = 7$; also $X \leftrightarrow \{1,2,4,3\} \leftrightarrow 59$. Scale X by $Y = m_3 = 5$.

Procedure : (i.e. we are required to find $\left[\frac{X}{5}\right]$)

1) Division remainder zero:

(i) Determine the residue representation of the number which is divisible by Y and is closest to X without exceeding X, i.e. $X - |X|_Y$. Since $|X|_Y = |X|_{m_i} = |X|_5$ and is given, therefore,

$$|X|_Y = |X|_{m_3} = 4$$

and the corresponding residue code is $\{0,1,4,4\}$

$$\therefore X - |X|_Y = X - |X|_{m_3} \leftrightarrow \{1,2,4,3\} - \{0,1,4,4\} = \{1,1,0,6\} \qquad (3.3)$$

(ii) Divide by $|Y|_{m_i}$ or multiply by $\left|\frac{1}{Y}\right|_{m_i}$: Division is possible if we temporarily ignore digit in position m_i . Therefore,

$$|Y|_{m_i} = |5|_{m_i} = \left|\frac{1}{5}\right|_{m_i} \leftrightarrow \{1,2,-,3\}$$

$$\therefore \left|\frac{X-|X|_Y}{Y}\right|_{m_i} = \left| |X-|X|_Y|_{m_i} \cdot \left|\frac{1}{Y}\right|_{m_i} \right|_{m_i} \qquad (3.4)$$

i.e. $$\left| |X-|X|_{m_3}|_{m_i} \cdot \left|\frac{1}{m_3}\right|_{m_i} \right|_{m_i} \leftrightarrow \{1,1,-,6\} \cdot \{1,2,-,3\}$$

OR $$\left|\frac{X-|X|_5}{5}\right|_{m_i} \leftrightarrow \{1,2,-,4\}$$

The original range for the full set of moduli was $[0, M-1]$. Since $0 \leq X \leq M-1$, therefore $0 \leq \left[\frac{X}{Y}\right] \leq \frac{M}{Y}-1$. Hence,

$$\left.\begin{aligned} 0 \leq \left(\left| \frac{X - |X|_Y}{Y} \right|_{m_i} \right)_{\substack{i=1 \\ i \neq j}}^{n} \leq \frac{M}{Y} - 1 \\ \text{where} \qquad m_j = Y \end{aligned}\right\} \qquad (3.5)$$

hence the residue representation is unique. To find the digit corresponding to $m_j = Y$ we use the base-extension conversion described in appendix 15 (section IV.2.1; particularly numerical example therein.).

2) Base-extension

	m_1	m_2	m_3	m_4
	2	3	5	7
$\left\| \frac{X - \|X\|_5}{5} \right\|_{m_i}$	(1)	2	0	4
SUBTRACT 1	1	1	1	1
	0	1	4	3
MULTIPLY BY $\left\|\frac{1}{2}\right\|_{m_i}$		2	3	4
		(2)	2	5
SUBTRACT 2		2	2	2
		0	0	3
MULTIPLY BY $\left\|\frac{1}{3}\right\|_{m_i}$			2	5
			0	(1)
SUBTRACT 1			1	1
			(4)	0

Let the digit corresponding to $m_j = Y$ after conversion be $Z(Y)$. As in numerical example of section IV.2.1 (appendix IV),

$$\left| \left[\frac{X}{Y} \right] \right|_Y = \left| \left| \frac{-1}{\hat{M}} \right|_Y \cdot Z(Y) \right|_Y \quad \text{where} \quad \hat{M} = \frac{M}{Y}$$

$$\therefore \quad \left| \left[\frac{X}{5} \right] \right|_5 = \left| |-6|_5 \times 4 \right|_5$$

$$= |4 \times 4|_5 = 1$$

The residue representation of the scaled value is thus,

$$\left[\frac{X}{5} \right] \leftrightarrow \{1, 2, 1, 4\}$$

(2) Scaling by serveral moduli

Let the scaling factor Y be a product of two moduli; $P=m_1 m_2$, say. The scaling procedureis similar to (1): $\left[\frac{X}{m_1}\right]$ is first computed, followed by $\left[\frac{X}{m_1 m_2}\right]$ and the digits corresponding to moduli m_1, m_2 positions are then obtained using base-extension conversion --- yielding the rounded-down scaled value. Note that the order of moduli in $\left[\frac{X}{m_1 m_2}\right]$ or $\left[\frac{X}{m_1 m_2 \cdots m_{n-1}}\right]$ in general, is arbitrary hence order of division by moduli is also arbitrary.

Problem: Given moduli $m_1=2$, $m_2=3$, $m_3=5$, $m_4=7$; X 1,2,4,5 ; and scaling factor $Y=m_2 m_3 = 3 \times 5 = 15$. Find $\left[\frac{X}{m_2 m_3}\right]=\left[\frac{X}{15}\right]$. The solution is tabulated below.

Explanation	Residue Operations	m_1 2	m_2 3	m_3 5	m_4 7
To Obtain $\left[\frac{X}{m_2}\right]=\frac{X-\lvert X\rvert_{m_2}}{m_2}$	Residue Representation of X	1	(2)	4	5
	Subtract $\lvert X\rvert_3$	0	2	2	2
	Multiply By $\lvert\frac{1}{3}\rvert_{m_i}$	1	0	2	3
		1	–	2	5
To Obtain $\left[\frac{X}{m_2 m_3}\right]=\frac{1}{5}\left(\frac{X-\lvert X\rvert_{m_2}}{m_2}-\lvert X\rvert_{m_3}\right)$		1	–	(4)	1
	Subtract $\lvert X\rvert_5$	0	–	4	4
	Multiply By $\lvert\frac{1}{5}\rvert_{m_i}$	1	–	0	4
		1	–	–	3
Base Extension To Obtain Digits Corresponding To m_2, m_3		(1)	0	0	5
	Subtract 1	1	1	1	1
		0	2	4	4
	Multiply By $\lvert\frac{1}{2}\rvert_{m_i}$		2	3	4
			1	2	(2)
	Subtract 2		2	2	2
			2	0	0

Let $Z=\left[\frac{X}{15}\right]$, then

$$\left|\lvert\tfrac{1}{2}\rvert_3\cdot\lvert\tfrac{1}{3}\rvert_3\cdot\lvert\tfrac{1}{5}\rvert_3\cdot\lvert\tfrac{1}{7}\rvert_3\cdot\lvert Z\rvert_3+\lvert\tfrac{1}{3}\rvert_3\cdot\lvert\tfrac{1}{5}\rvert_3\cdot 2\right|_3=0$$

i.e. $\left|\lvert\tfrac{1}{14}\rvert_3\cdot\lvert Z\rvert_3+2\right|_3=0$

or $\left|\lvert\tfrac{1}{2}\rvert_3\cdot\lvert Z\rvert_3+2\right|_3=0$ Giving $\lvert Z\rvert_3=2$

Similarly, $\left|\lvert\tfrac{1}{14}\rvert_5\cdot\lvert Z\rvert_5+0\right|_5=0$

or $\left|\lvert\tfrac{1}{4}\rvert_5\lvert Z\rvert_5+0\right|_5=0$ Giving $\lvert Z\rvert_5=0$

Therefore the residue representation of $\left[\frac{X}{m_2 m_3}\right]$ is:

$$\left[\frac{X}{15}\right]\leftrightarrow\{1,2,0,5\}$$

Observations (on the two examples):

1) Excluding the time required for solving the equations for z, the algorithm requires N subtractions and N multiplications.

2) Irrespective of the scaling factor being one modulus or product of several moduli, the same number of operations (stated in 1) is required.

3) Scaling factor (Y) can only be product of first power of moduli.

4) Scaling by a higher power of a modulus can only be obtained indirectly through repeated scaling operations.

B. Scaling using macrocoefficient extraction

Scaling using macrocoefficient extraction assumes knowledge of "macrocoefficient extraction" and "base-extension" discussed respectively in section III.3.4 (appendix 15) and IV.2.2.2 (appendix 15; IV). We shall briefly consider below scaling by : (1) one modulus, (2) two moduli and (3) three moduli in an RRNS containing four non-redundant moduli.

(1) Scaling by one modulus

Recall equations (3.1) and (3.2)

$$\left[\frac{X}{Y}\right] = \frac{X - |X|_Y}{Y} \tag{3.1}$$

and

$$\left[\frac{X}{Y}\right] \leftrightarrow \left\{ \left|\frac{X-|X|_Y}{Y}\right|_{m_1}, \left|\frac{X-|X|_Y}{Y}\right|_{m_2}, \cdots \left|\frac{X-|X|_Y}{Y}\right|_{m_n} \right\} \tag{3.2}$$

Here $Y=m_j$ (where j=1 or 2, or etc.). With reference to equation II.17 of appendix 15, perform the first two steps of the mixed-radix conversion process. The remaining residues of the scaled quantity are then obtained by the Three Residue Base-extension (3BR-E) algorithm discussed in section IV.2.2.2. of appendix 15.

(2) Scaling by two moduli

In appendix 15, (section III.1), we defined macrocoefficient for the (K, K-1, K+a) RRNS's. Part of equation IV.2 is repeated here for convenience.

$$\left.\begin{aligned} \gamma_x &= \left[\frac{x}{K}\right] = \left[\frac{x}{m_1 m_2}\right] \\ \beta_x &= \left[\frac{x}{K-1}\right] = \left[\frac{x}{m_3 m_4}\right] \\ \text{\&} \quad \Gamma_x &= \left[\frac{x}{K+a}\right] = \left[\frac{x}{m_5 m_6}\right] \end{aligned}\right\} \qquad (3.7)$$

These macrocoefficients are the scaled values of X by two moduli and the problem of scaling is thus a problem of macrocoefficient extraction. Methods of finding complete residue representations of (3.7) were discussed in section III.3.4 entitled "macrocoefficient extraction algorithms" (appendix 15).

(3) Scaling by three moduli

We shall illustrate this by an example; suppose we require $\left[\frac{x}{m_1 m_2 m_3}\right]$. From equations (II.16) and (III.20) of appendices II and III respectively, we have

$$\left|\left[\frac{x}{m_1 m_2 m_3}\right]\right|_{m_4} = \left[\frac{x}{m_1 m_2 m_3}\right]_{m_4} = \left|\frac{|\gamma_x|_{m_4} - |\gamma_x|_{m_3}}{m_3}\right|_{m_4} \qquad (3.8)$$

Since $0 \leq x \leq m_1 m_2 m_3 m_4 - 1$ and $0 \leq |x|_{m_4} \leq m_4 - 1$, therefore

$$0 \leq \left[\frac{x}{m_1 m_2 m_3}\right]_{m_4} \leq m_4 - 1$$

and $\left[\frac{x}{m_1 m_2 m_3}\right]_{m_4}$ uniquely represent $\left[\frac{x}{m_1 m_2 m_3}\right]$. Hence we only required to compute

$$\left[\frac{x}{m_1 m_2 m_3}\right]_{m_4, m_j} \triangleq \left|\left[\frac{x}{m_1 m_2 m_3}\right]_{m_4}\right|_{m_j} \qquad (3.9)$$

where $j = 1, 2, 3, 5, 6$.

Equation (3.9) can obtained using combinational logic.

This concludes our discussion of scaling.

5.4 FRACTIONAL MULTIPLICATION (89,90,93-96,100-102)

Fractional arithmetic was first proposed by Svoboda (93 - 96); the concept of rational system of residue classes (RSRC) was introduced. In these classes, fractions could be defined and arithmetic performed.

Fractional addition/subtraction corresponds to adding/subtracting numbers of the same scaling factor. The addition/subtraction is thus identical to that for integers as discussed in section 5.2 of this chapter. In this section we shall discuss fractional multiplication which is more involved than the operation depicted by equation (2.2). Like base-extension and scaling, there are two approaches for fractional multiplication; viz. algorithms using mixed-radix conversion (93-96) and those using macrocoefficient extraction. (89,90,100-102) "Simple algorithm" for fractional multiplication was discussed in references 81 and 91; but this algorithm could result undefine representations. Therefore it is not discussed herein. Unless stated to the contrary, algorithms which we shall discuss are applicable to both the non-redundant residue number system (N-RNS) and the redundant residue number system (RRNS). Before discussing the algorithms, we shall first review the notions of (1) rational number representation and (2) technique of systematic error correction and rounding off.

5.4.1 Rational number representation

Fractions in the RSRC can be defined by,

$$\left.\begin{aligned} x &\triangleq \frac{X}{M} \\ \text{where integer } X \text{ is bounded by,}& \\ 0 &\leq X \leq M-1 \\ \text{and} \quad M &= \prod_{i=1}^{n} m_i \end{aligned}\right\} \qquad (4.1)$$

The residue representation of integer X and fraction x are,

$$X \leftrightarrow \left(|X|_{m_1}, |X|_{m_2}, \cdots, |X|_{m_n} \right) \qquad (4.2)$$

$$x \leftrightarrow / |X|_{m_1}, |X|_{m_2}, \cdots, |X|_{m_n} / \qquad (4.3)$$

Hence x in the RSRC and X in the RNS are represented by n residue digits and therefore are of the same order of magnitude. Illustrating this by an example, let X=3799 and the RNS (or RSRC) consists of moduli $\mathcal{M} \leftrightarrow (3\ 4\ 5\ 7\ 11)$, so that M=3.4.5.7.11=4620. Residue representations of X and x are,

$$X = 3799 \leftrightarrow (1\ 3\ 4\ 5\ 4)$$

and

$$x = \frac{3799}{4620} \leftrightarrow /1\ 3\ 4\ 5\ 4/$$

Condition for accurate representation

Since $0 \leqslant X \leqslant M-1$, therefore x is bounded by

$$0 \leqslant x \triangleq \frac{X}{M} < 1 \tag{4.4}$$

As the number of residue digits is finite, x is in fact approximated by $x' = \frac{A}{B}$ and x' is defined if we can find

$$X = x' M = \frac{A}{B} \cdot M \tag{4.5}$$

and this is only possible iff

$$A.M \equiv 0 \pmod{B} \tag{4.6}$$

5.4.2 Systematic error correction and rounding off

Rounding off in the RSRC means approximating x by x'. For clarity of presentation, we shall write x_i for $|x|_{m_i}$ and x' for $|x'|_{m_i}$ in this section. For

$$M^+ \leftrightarrow (m_1\ m_2 \cdots m_n, m_{n+1} \cdots m_{n+k}) \tag{4.7}$$

a fraction x, is given by

$$x \triangleq \frac{X}{M^+} \leftrightarrow /x_1, x_2, \cdots, x_n, x_{n+1}, \cdots, x_{n+k}/ \tag{4.8}$$

can be approximated by

$$x' \triangleq \frac{X'}{M'} \leftrightarrow / x'_{n+1}, x'_{n+2}, \cdots, x'_{n+k} / \tag{4.9}$$

where integer X' is expressed in relation to the reduced base M' given by

$$M' \leftrightarrow (m_{n+1}\ m_{n+2} \cdots m_{n+k}) \tag{4.10}$$

The original base M^+ is therefore reduced to M', so that

$$M' = \frac{M^+}{M} \tag{4.11}$$

where

$$M \leftrightarrow (m_1\ m_2 \cdots m_n) \tag{4.12}$$

Therefore by rounding-off we reduce the number of residue digits expressing the fraction x; analogous to rounding in the weighted number system. To avoid systematic error in rounding, we can apply a correction C defined by,

(i) for M even, $C = \frac{1}{2} M$ (4.13)

(ii) for M odd, $C = \frac{1}{2} (M-1)$ (4.14)

With this correction, we can define,

$$X^+ = X + C \tag{4.15}$$

and the residue representation of it is,

$$X^+ \leftrightarrow (x_1^+, x_2^+, \cdots, x_n^+, x_{n+1}^+, \cdots, x_{n+k}^+) \tag{4.16}$$

By reducing the base, we have $0 \leq X^+ \leq M-1$, hence

$$X^+ (mod\, M) \leftrightarrow (x_1^+, x_2^+, \cdots, x_n^+) \tag{4.17}$$

For this to be expressed in the original system defined by equation (4.7), additional digits such as

$$x_{n+1}^*, x_{n+2}^*, \cdots, x_{n+k}^*$$

must be computed, so that $X^+ (mod\, M)$ in the original system is given by,

$$X^+ (mod\, M) \leftrightarrow (x_1^+, x_2^+, \cdots, x_n^+, x_{n+1}^*, \cdots, x_{n+k}^*)$$

Now, integer $T = X^+ - \{ X^+ (mod\, M) \}$ is divisible by M and T has the form,

$$T \leftrightarrow (\underbrace{00 \cdots 0}_{n \text{ ZEROS}} \; t_{n+1}, t_{n+2}, \cdots, t_{n+k}) \tag{4.19}$$

where

$$t_{n+i} = x_{n+i}^+ - x_{n+i}^* \; (mod\, m_i) \tag{4.20}$$

and M has the form,

$$M \leftrightarrow (\underbrace{00 \cdots 0}_{n \text{ ZEROS}} \; p_{n+1}, p_{n+2}, \cdots, p_{n+k}) \tag{4.21}$$

where p_{n+i} are different from zero. This gives,

$$X' = \frac{T}{M} \leftrightarrow (x'_{n+1}, x'_{n+2}, \cdots, x'_{n+k}) \tag{4.22}$$

and finally,

$$x' = \frac{X'}{M'} \leftrightarrow / x'_{n+1}, x'_{n+2}, \cdots, x'_{n+k} / \tag{4.23}$$

The error of approximation or rounding off is given by

$$| x - x' | < \frac{C}{M'} \leq \frac{1}{2M'} \tag{4.24}$$

To conclude, let us illustrate rounding-off with systematic error-correction by a numerical example.

Numerical example

Given $M^+ \leftrightarrow (3\ 4\ 5\ 7\ 11)$ we wish to round off $x = \frac{3799}{4620}$ by suppressing $m_1=3$, $m_2=4$, and $m_3=5$.

From data,

$M = 3.4.5.7.11 = 4620$
$M = 3.4.5 = 60$
$M' = 7.11 = 77$

Explanations	m_1 3	m_2 4	m_3 5	m_4 7	m_5 11
$x = \frac{X}{M^+} = \frac{3799}{4620}$	/ 1	3	4	5	4 /
X	(1	3	4	5	4)
Correction $C = \frac{1}{2} M = 30$	(0	2	0	2	8)
$X^+ = X + C$	(1	1	4	0	1)
$X^+ (\text{mod } M) = 49$	(1	1	4)		
$X^+ (\text{mod } M)$ extended to M'	(1	1	4	0	5)
$T = X^+ - \{X^+ (\text{mod } M)\}$	(0	0	0	0	7)
$M = 60$	(0	0	0	4	5)
$X' = \frac{T}{M}$			(0	8)	
Therefore $x' = \frac{X'}{M'}$			/ 0	8)	

5.4.3 Fractional multiplication using mixed-radix conversion [17-20]

A. The basic concept

Using equation (4.1), two fractions x and y can be represented by,

$$x = \frac{X}{M} \leftrightarrow / |x|_{m_1}, |x|_{m_2}, \cdots, |x|_{m_n} / < 1 \qquad (4.25)$$

$$y = \frac{Y}{M} \leftrightarrow / |Y|_{m_1}, |Y|_{m_2}, \cdots, |Y|_{m_n} / < 1 \qquad (4.26)$$

The product z = x.y is given by

$$x \cdot y = \frac{X \cdot Y}{M^2} = \frac{\frac{x \cdot y}{M}}{M} < 1$$

By equation (4.1),

$$\mathfrak{z} = \frac{z}{M} \leftrightarrow /|z|_{m_1}, |z|_{m_2}, \cdots, |z|_{m_n}/ < 1 \qquad (4.27)$$

Therefore z can be computed if we first find

$$z = \frac{x \cdot y}{M} \qquad (4.28)$$

That is the fractional product, x.y, is given by computing the integer multiplication of integers X and Y and dividing the integer product by the range M. To avoid temporary overflow during the intermediate computation, we require to insert a temporary (or "redundant") range to account for the "double word length" resulting in multiplying X by Y. The range is then reduced to the "single length M" after division of X.Y by M. The added range must therefore be at least of length M. To prevent a systematic error in the procedure we can include correction C as defined by equations (4.13) and (4.14).

This is the basic concept behind the algorithm using mixed-radix conversion. We shall examine the algorithm in greater detail and conclude section 5.4.3 with an outline of the whole algorithm.

B. Details of the algorithm

RRNS is assumed in the discussion; algorithm for the N-RNS can be obtained from that of the RRNS by discarding calculations for the redundant moduli m_c. Notice that m_c is not to be confused with the temporary redundant moduli m_i; the former redundancy is for error correction or detection applications (100 - 102) while the latter is for preventing intermediate overflow during multiplication. Using equation (I.3), equation (4.28) becomes:

$$z \triangleq \frac{x \cdot y}{M} = \left[\frac{x \cdot y}{M}\right] + \frac{|x \cdot y|_M}{M} \qquad (4.29)$$

By neglecting the last term, we approximate Z to,

$$z \approx \left[\frac{x \cdot y}{M}\right] \qquad (4.30)$$

The maximum error involved is

$$\Delta_{max} = \left(\frac{|x \cdot y|_M}{M}\right)_{max} = \frac{M-1}{M} = 1 - \frac{1}{M} \qquad (4.31)$$

The problem now remains is to compute a complete residue sequence for equation (4.30). Let us define,

(i) redundant moduli m_c : $c = n+1, n+2, \cdots, n+r$

& (ii) temporary redundant moduli m_s : $s = n+r+1, n+r+2, \cdots, n+s$

so that the new redundant number system consists of,

moduli m_ℓ : $\ell = 1,2,\cdots,n,\ n+1,\cdots,n+r,\ n+r+1,\cdots,n+s$

with, m_a : $a = 1,2,\cdots,n,\ n+1,\cdots,n+r$

m_c : $c = \quad\quad n+1,\cdots,n+r$

and $m_{c'}$: $c' = \quad\quad n+1,\cdots,n+r,\ n+r+1,\cdots,n+s$

All moduli are understood to be pairwise relatively prime. The algorithm of the fractional multiplication is thus:

(1) Compute sequences $(|X|_{m_s})$ and $(|Y|_{m_s})$

Sequences $(|X|_{m_s})$ and $(|Y|_{m_s})$ may be computed with $(|X|_{m_i})$ and $(|Y|_{m_i})$ as inputs to a base-extension algorithm, discussed in section IV.2.1 of appendix 15. Subscripts i and s respectively denote 1,2, ..., n and n+r+1, ..., n+s.

(2) Procedure of computing $\left[\frac{X\cdot Y}{M}\right]$ modulo m_a ; a=1,2,...,n+r

Let W=X.Y then by equation (I.3) we have

$$W = |W|_M + M\left[\frac{W}{M}\right] \tag{4.32}$$

Taking residue of modulo $m_{c'}$,

$$\therefore\ |W|_{m_{c'}} = \left||W|_M + M\left[\frac{W}{M}\right]\right|_{m_{c'}} \tag{4.33}$$

where $$M = \prod_{i=1}^{n} m_i$$

Since moduli are pairwise relatively prime, therefore $\left|\frac{1}{M}\right|_{m_{c'}}$ exist and are unique. With notation depicted by equation (I.6) we have $\left||W|_M\right|_{m_{c'}} = |W|_{M,m_{c'}}$. Rearranging equation (4.33),

$$\therefore\ \mathcal{Z}_{m_{c'}} = \left[\frac{X\cdot Y}{M}\right]_{m_{c'}} = \left[\frac{W}{M}\right]_{m_{c'}} = \left|\frac{|W|_{m_{c'}} - |W|_{M,m_{c'}}}{M}\right|_{m_{c'}} \tag{4.34}$$

Interpretation of equation (4.34):

1) $|W|_{m_{c'}} = |X\cdot Y|_{m_{c'}}$ can be computed by integer multiplication.

2) Since m_i is a factor of M, by the Composite Modulus Theorem (I.25), therefore $|W|_{M,m_i} = |W|_{m_i}$ and residue sequence $(|W|_{m_i})$, $i = 1,2,\cdots,n$ uniquely represents $|W|_M$.

Hence $|W|_{M,m_{c'}}$ can be computed using base-extension (discussed in appendix 15, section IV.2.1) from the sequence $(|W|_{m_i})_{i=1}^{n}$.

3) Multiplicative inverses $\left|\frac{1}{M}\right|_{m_{c'}}$ exist and are unique.

Therefore we can compute $\left[\frac{W}{M}\right]_{m_{c'}}$ in equation (4.34).

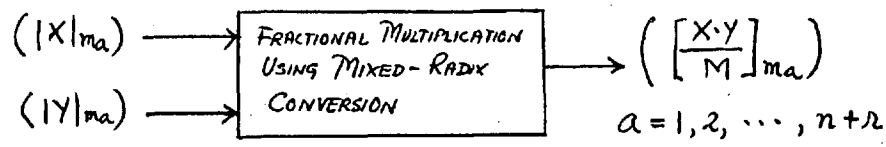
(3) Compute $\left(\left[\frac{W}{M}\right]_{m_i}\right)_{i=1}^{n}$

With $\left[\frac{W}{M}\right]_{m_c}$ known, we can compute $\left(\left[\frac{W}{M}\right]_{m_i}\right)_{i=1}^{n}$ by base-extension algorithm.

(4) Finally, we can obtain $\left(\left[\frac{W}{M}\right]_{m_a}\right)_{a=1}^{n+r}$ by discarding digits corresponding to moduli n+r+1, ..., n+s. (i.e. moduli m_s).

The approach offers a means of detecting multiplicative overflow because if $\left[\frac{W}{M}\right]_{m_{c'}} = 0$, then $0 \leq X \cdot Y \leq M-1$ and therefore no overflow has occured. This is an advantage over those algorithms of the next section. To conclude this section we shall outline the algorithm of fractional multiplication using mixed-radix conversion.

C. Outline of Fractional multiplication algorithm using Mixed-radix conversion



Procedure:

1) Compute sequence $(|X|_{m_s})$, $(|Y|_{m_s})$; s= n+r+1,...,n+s

With $(|X|_{m_i})$, $(|Y|_{m_i})$; i=1,2, ..., n as inputs to base -extension algorithm (see section IV.2.1) compute $(|X|_{m_s})$ and $(|Y|_{m_s})$.

2) Compute $|X\cdot Y|_{m_\ell}$; $\ell = 1,2, \ldots, n+s$

Use integer multiplication compute $|X\cdot Y|_{m_\ell}$.

3) Compute $(|X\cdot Y|_{M,m_{c'}})$; c'= n+1, ..., n+r

Use $(|X\cdot Y|_{m_i})$; i=1,2,...,n as input to base-extension algorithm, compute $(|X\cdot Y|_{M,m_{c'}})$.

4) Compute $\left[\frac{X\cdot Y}{M}\right]_{m_{c'}}$; c' = n+1, ..., n+r

Use $|X\cdot Y|_{m_{c'}}$ and $|X\cdot Y|_{M,m_{c'}}$ computed respectively in steps 2 and 3 and formula (4.34) compute

$$\left[\frac{X\cdot Y}{M}\right]_{m_{c'}} = \left| \frac{|X\cdot Y|_{m_{c'}} - |X\cdot Y|_{M,m_{c'}}}{M} \right|_{m_{c'}}$$

5) Compute $\left[\frac{X\cdot Y}{M}\right]_{m_i}$; i=1,2, ..., n

Use $\left[\frac{X\cdot Y}{M}\right]_{m_{c'}}$ just calculated in step 4 as inputs to base-extension algorithm, compute $\left[\frac{X\cdot Y}{M}\right]_{m_i}$.

6) Obtain $\left[\frac{X\cdot Y}{M}\right]_{m_a}$; a=1,2, ..., n+r

$\left[\frac{X\cdot Y}{M}\right]_{m_\ell}$; ℓ =1,2, ..., n+s is formed from $\left[\frac{X\cdot Y}{M}\right]_{m_i}$ and $\left[\frac{X\cdot Y}{M}\right]_{m_{c'}}$, obtained respectively in steps 5 and 4. Discarding $\left[\frac{X\cdot Y}{M}\right]_{m_s}$, s = n+n+1, ..., n+s in $\left[\frac{X\cdot Y}{M}\right]_{m_\ell}$ we therefore have as required. EXIT of algorithm.

5.4.4 Fractional Multiplication using Macrocoefficient Extraction (references: 89,90,100-102)

Fractional multiplication algorithms using macro-coefficient extractions were proposed and developed at Honeywell by Hastings, Vessey, Moore[89] and Watson.[100] Two algorithms will be discussed; these we shall call algorithm one and algorithm two. Algorithm two time-shares the available hardwares more efficiently and therefore reduces the computation time and the hardwares required. However, the algorithm has a larger possible error than algorithm one.

The (K, K-1, K+a) RRNS is assumed; where M=K(K-1), and $0 \leq X,Y \leq M-1$. Unlike the algorithm based on mixed-radix conversion, temporary redundant moduli (to account for intermediate temporary multiplication overflow) is provided by the redundant moduli of (K+a). For hardware economy, "a" is chosen to be a=1 (reference 90).

A. Algorithm One

With reference to section III.2 of appendix 15, for integers X and Y we can define macrocoefficients,

$$\left.\begin{aligned} \delta_x &= |X|_K \quad , \quad \delta_y = |Y|_K \\ \gamma_x &= \left[\frac{X}{K}\right] \quad , \quad \gamma_y = \left[\frac{Y}{K}\right] \\ \text{where} \quad & 0 \leq \delta_x, \delta_y \leq K-1 \\ & 0 \leq \gamma_x, \gamma_y \leq K-2 \end{aligned}\right\} \qquad (4.35)$$

so that X and Y can be expressed as,

$$\left.\begin{aligned} X &= K\gamma_x + \delta_x \\ Y &= K\gamma_y + \delta_y \end{aligned}\right\} \qquad (4.35a)$$

Therefore,

$$X \cdot Y = K^2\gamma_x\gamma_y + K\gamma_x\delta_y + K\gamma_y\delta_x + \delta_x\delta_y$$

Adding and subtracting $K\gamma_x\gamma_y$,

$$\therefore \quad X \cdot Y = (K^2-K)\gamma_x\gamma_y + K\gamma_x\gamma_y + K\gamma_y\delta_x + K\gamma_x\delta_y + \delta_x\delta_y$$

Dividing by $M=K(K-1)$,

$$\therefore \quad \frac{X \cdot Y}{M} = \gamma_x\gamma_y + \frac{\gamma_x\gamma_y + \gamma_x\delta_y + \delta_x\gamma_y}{K-1} + \frac{\delta_x\delta_y}{M} \qquad (4.36)$$

From equations (4.35a) we can express an integer W as,

$$\frac{W}{K-1} = \left[\frac{W}{K-1}\right] + \frac{|W|_{K-1}}{K-1} = \beta_W + \frac{\epsilon_W}{K-1} \qquad (4.36a)$$

Using this, we can express equation (4.36) as,

$$\frac{X \cdot Y}{M} = \gamma_x\gamma_y + \left[\frac{\gamma_x\gamma_y + \gamma_x\delta_y + \gamma_y\delta_x}{K-1}\right] + \frac{|\gamma_x\gamma_y + \gamma_x\delta_y + \gamma_y\delta_x|_{K-1}}{K-1} + \frac{\delta_x\delta_y}{M} \qquad (4.37)$$

By the same argument as in section 4.3.1 we can approximate X.Y to $Z = \frac{X \cdot Y}{M}$ and neglect the last two terms if maximum possible error $(\Delta)_{max}$ is acceptable. Fractional product x.y is then obtained from $x \cdot y \triangleq \frac{Z}{M}$ (see equation 4.27 of section J.4.3A). Therefore we can define,

$$\frac{X \cdot Y}{M} = \gamma_x\gamma_y + \left[\frac{\gamma_x\gamma_y + \gamma_x\delta_y + \gamma_y\delta_x}{K-1}\right] \qquad (4.38)$$

The maximum possible error $(\Delta)_{max}$ is given by,

$$(\Delta)_{max} = \left(\frac{|\gamma_x\gamma_y + \gamma_x\gamma_y + \gamma_y\delta_x|_{K-1}}{K-1} + \frac{\delta_x\delta_y}{M}\right)_{max}$$

Substituting the bounds for δ and γ (also $0 \leq |W|_{K-1} \leq K-2$),

$$\therefore \quad (\Delta)_{max} = \frac{K-2}{K-1} + \frac{K-1}{K} \quad \text{or} \quad 2 - \frac{2}{K-1} + \frac{1}{M} \tag{4.39}$$

Equation (4.38) can be computed by first calculating the second term which can be written as,

$$\left[\frac{S}{K-1}\right] = \left[\frac{\gamma_x\gamma_y + \gamma_x\delta_y + \delta_x\gamma_y}{K-1}\right] \tag{4.40}$$

Before considering the procedure of calculating equation (4.40) let us first estimate its bounds. We have, from equation (III.12) of appendix 15,

$$0 \leq \delta_x, \delta_y \leq K-1$$
$$0 \leq \gamma_x, \gamma_y \leq (K-1)-1 \quad \text{or} \quad K-2$$

Therefore

$$0 \leq \gamma_x\gamma_y \leq K^2-4K+4$$
$$0 \leq \gamma_x\delta_y \leq K^2-3K+2$$
$$0 \leq \delta_x\gamma_y \leq K^2-3K+2$$
$$\therefore \quad 0 \leq S \triangleq (\gamma_x\gamma_y + \gamma_x\delta_y + \delta_x\gamma_y) \leq 3K^2-10K+8$$

The bounds of equation (4.40) are thus,

$$0 \leq \left[\frac{S}{K-1}\right] \leq \left[\frac{3K^2-10K+8}{K-1}\right]$$

or

$$0 \leq \left[\frac{S}{K-1}\right] \leq 3K-7 \tag{4.41}$$

Furthermore, from equation (III.4) we have

$$S = (K-1)\left[\frac{S}{K-1}\right] + |S|_{K-1} = (K-1)\beta_S + \epsilon_S \tag{4.42}$$

therefore we can define

$$\beta_S \triangleq \left[\frac{S}{K-1}\right] \tag{4.43}$$

and the procedure of computing equation (4.40) or equation (4.43) is as follows,

(1) Compute the complete residue sequence for

$$\gamma_x, \gamma_y, \delta_x \text{ and } \delta_y$$

corresponding to moduli m_i ($i = 1,2,3,4,5,6$), by the macro - coefficient extraction algorithm discussed in section III.3.4 of appendix 15.

(2) Compute the complete residue representation of β_s

Use the macrocoefficient extraction algorithm (described in section III.3.4) compute:

(i) $|\beta_s|_{m_i}$ where i = 1,2 in the (K, K-1) RNS

& (ii) $|\beta_s|_{m_c}$ where c = 5,6 in the (K-1, K+a) RNS

so that we have the sequence of residue,

$$(|\beta_s|_{m_1}, |\beta_s|_{m_2}, |\beta_s|_{m_5}, |\beta_s|_{m_6}) \qquad (4.44)$$

and this uniquely represent β_s since from equation (4.41) β_s is bounded by, $0 \leq \beta_s = \left[\frac{S}{K-1}\right] \leq 3K-7$

The range consisting of moduli m_1, m_2, m_5 and m_6 is:

$$\begin{aligned} \text{range} &= m_1 \cdot m_2 \cdot m_5 \cdot m_6 \\ &= K(K+1) \end{aligned}$$

If β_s is to be uniquely represented by (4.44) we require,

$$Q = \{K(K+1) - 1\} - \{3K-7\}$$

to be positive. Now,

$$\begin{aligned} Q &= K^2 + K - 1 - 3K + 7 \\ &= K^2 - 2K + 6 \\ &= (K-1)^2 + 5 > 0, \text{ for all } K \geq 0. \end{aligned}$$

Hence equation (4.44) does represent β_s uniquely. By this very fact, we can use the base-extension algorithm of section IV.2.2.1 in appendix IV to compute

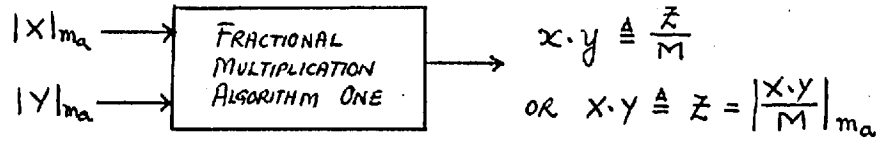
$$|\beta_s|_{m_3} \quad \text{and} \quad |\beta_s|_{m_4}$$

(3) Compute $\frac{X \cdot Y}{M}$

The fractional product $x \cdot y \triangleq \frac{z}{M}$ where $z = \frac{X \cdot Y}{M}$ depicted by equation (4.38) can now be computed by adding the residues of $\gamma_x \gamma_y$ and $\beta_s \triangleq \left[\frac{S}{K-1}\right]$.

An outline of the complete algorithm is shown below.

Outline of the algorithm one



where $a = 1, 2, 3, 4, 5, 6$

Procedure:

1) Use macrocoefficient extraction compute the complete residue representations for γ_x, γ_y, δ_x and δ_y.

2) Compute the complete residue sequence for (by integer multiplication) $\gamma_x * \gamma_y$.

3) Compute the complete residue sequence for $\gamma_y + \delta_y$.

4) " " " " " " $\gamma_x(\gamma_y + \delta_y)$.

5) " " " " " " $\gamma_y * \delta_x$

6) " " " " " " $S = \gamma_x(\gamma_y + \delta_y) + \gamma_y \cdot \delta_x$

7) Use macrocoefficient extraction compute
$|\beta_s|_{m_1}$, $|\beta_s|_{m_2}$, $|\beta_s|_{m_3}$, and $|\beta_s|_{m_6}$.

8) Use residues obtained in step 7 as inputs to a base-extension algorithm, compute $|\beta_s|_{m_3}$ and $|\beta_s|_{m_4}$.

9) Compute the complete residue representations for,
$$X \cdot Y \triangleq Z = \frac{X \cdot Y}{M} = \gamma_x \gamma_y + \beta_s$$
for m_a (a = 1,2,3,4,5,6). EXIT of algorithm.

B. Algorithm Two

This algorithm assumes the same (K, K-1, K+a) RNS as in algorithm one; but differs in the definition of X and Y. (compare the following with equation 4.35). Let X and Y be bounded by $0 \leq X, Y \leq M-1$ so that

$$\left.\begin{aligned} X &= K\gamma_x + \delta_x \\ \text{and} \quad Y &= (K-1)\beta_y + \epsilon_y \end{aligned}\right\} \qquad (4.45)$$

Therefore,

$$X \cdot Y = K(K-1)\gamma_x \beta_y + (K-1)\beta_y \delta_x + K\gamma_x \epsilon_y + \delta_x \epsilon_y$$

Dividing by $M=K(K-1)$,

$$\frac{X\cdot Y}{M} = \gamma_x \beta_y + \frac{\beta_y \delta_x}{K} + \frac{\gamma_x \epsilon_y}{K-1} + \frac{\delta_x \epsilon_y}{M} \tag{4.46}$$

Using equation (I.3) or equation (4.36a), therefore

$$\frac{X\cdot Y}{M} = \gamma_x \beta_y + \left[\frac{\beta_y \delta_x}{K}\right] + \left[\frac{\gamma_x \epsilon_y}{K-1}\right] + \frac{|\beta_y \delta_x|_K}{K} + \frac{|\gamma_x \epsilon_y|_{K-1}}{K-1} + \frac{\delta_x \epsilon_y}{M} \tag{4.47}$$

Again, as in algorithm one, we can approximate $X\cdot Y$ to $Z \triangleq \frac{X\cdot Y}{M}$ and neglecting the last three terms in equation (4.47) so that

$$\begin{aligned}\frac{X\cdot Y}{M} &= \gamma_x \beta_y + \left[\frac{\beta_y \delta_x}{K}\right] + \left[\frac{\gamma_x \epsilon_y}{K-1}\right] \\ &= \gamma_x \beta_y + \gamma(\beta_y \delta_x) + \beta(\gamma_x \epsilon_y)\end{aligned} \tag{4.48}$$

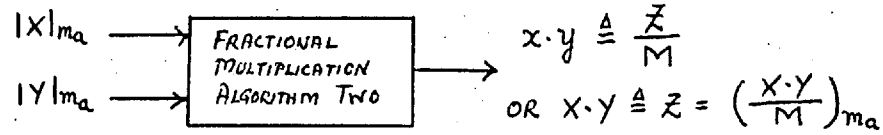
where γ and β are functions of $\beta_y \delta_x$ and $\gamma_x \epsilon_y$ can be computed simultaneously (the advantage of this algorithm). The maximum possible error involved is,

$$\begin{aligned}(\Delta)_{max} &= \left(\frac{|\delta_y \delta_x|_K}{K} + \frac{|\gamma_x \epsilon_y|_{K-1}}{K-1} + \frac{\delta_x \epsilon_y}{M}\right)_{max} \\ &= \frac{M-1}{M} + \frac{K-1}{K} + \frac{K-2}{K-1} \\ &= 3 - \frac{2}{K-1}\end{aligned} \tag{4.49}$$

which is larger than that for algorithm one (equation 4.39).

As this algorithm is essentially the same as algorithm one, we shall only outline the procedure but in slightly more detail.

Outline of algorithm two



where $a = 1, 2, 3, 4, 5, 6$

AND $X = K\gamma_x + \delta_x$

$Y = (K-1)\beta_y + \epsilon_y$

Procedure:

1) Use macrocoefficient extraction algorithm (section III.3.4) compute simultaneously a complete residue sequence (i.e. for m_a) for

$$\gamma_x \ , \ \delta_x \ , \ \beta_y \text{ and } \epsilon_y .$$

2) Compute the complete residue sequence (i.e. for m_a) for $\gamma_x \beta_y$.

3) " " " " " " " " " $\beta_y . \delta_x$.

4) " " " " " " " " " $\gamma_x \epsilon_y$.

5) Use the scaling for two moduli (section 3.2.2) or by the macrocoefficient definitions (in section III.1 of appendix III), compute <u>simultaneously</u> the complete residue sequence (i.e. for m_a) for,

$$\gamma(\beta_y \delta_x) \quad \text{and} \quad \beta(\gamma_x \epsilon_y)$$

according to equation (4.48).

6) Add the complete residue sequence of

$$W = \gamma(\beta_y \delta_x) + \beta(\gamma_x \epsilon_y)$$

7) Add the complete residue sequence of

$$\gamma_x \beta_y + W = \frac{X \cdot Y}{M} = Z \triangleq X \cdot Y$$

EXIT of algorithm.

This concludes our discussion of fractional multiplication using macrocoefficient extraction and the whole of section four on fractional multiplication.

5.5 IMPLEMENTATION OF DIGITAL FILTERS USING RNS [1]

The advancements of technologies have made much impact on the philosophy of designing digital systems. Logic gate network has been favoured in the design of RNS systems. However, the amount of circuitry increases considerably with the increase of moduli (see appendix 16). Recent developments of high density read-only-memories (ROM) have revived the use of table look-up techniques for implementing RNS operations (e.g. appendix 17). These techniques have been suggested in the past (99), however the implementations with core memory provided little or no advantage over logical gates. The availability of low-cost integrated microprocessors has also stimulated interest for constructing digital signal processors by using multiple microprocessors that are individually limited in wordlength and computing power. To make effective use of a multiple processor architecture, it is essential that the algorithm can be decomposed into subtasks which may occur simultaneously with very little intercommunication. This line of reasoning has promoted the use of residue number systems which have the inherent parallel structures.

Based on the use of ROM's and microprocessors, many architectures have been proposed and reported in references 103 through 121. It is not the intention of this author to reiterate these reported works; it will suffice to indicate the different approach for which these technologies have been applied to take advantage of the properties of RNS. Jullian, (119,120) have adopted the use of ROM's as basic building blocks. Based on the "base extension algorithm" and Svoboda's "metric vector estimates" (93), for scaling (discussed above), recursive filtering and FFT algorithms have been implemented and studied.

---

[1] The implementation of digital filters using RNS was originally the main concern of this research. Hardware studies, such as those reported in appendices 16 & 17 were conducted before the emphasis was shifted to investigating the "Top-down Design Approach." This approach was evolved out of the work on RNS (CAD methods were used for the hardware studies). It was intended to develop the design approach for a thorough study of the RNS. In the intervening period, much careful studies & developments have been conducted. It will suffice to reference these contributions in this section.

Noise performance has also been evaluated. Their findings have demonstrated the feasibility of implementing high speed digital signal processing with RNS. The structure of digital hardware (using arrays of identical ROM's) also simplifies control logic and the use of latches with ROMs provides automatic pipelining with little increase of complexity in the control logic.

Soderstrand and Jenkins, on the other hand, exploited both the ROM's and microprocessors. Soderstrand utilized the high-speed addition and subtraction of the residue number and avoided multiplication by a fraction in residue numbers. Instead, a table-lookup for high-speed multiplication by a fraction was adopted. Comparisons had been made with the Peled & Liu's approach of implementing filter functions by table lookup techniques. It was concluded (105) that Soderstrand's approach offered improvements over that of the Peled & Liu. Prompted by this, Jenkins investigated and showed (118) that full precision (unrounded) integer multiplication in the residue code was simpler and faster than an equivalent full precision integer multiplication in the 2's complement code. However, with rounded multiplication, residue multipliers became more complicated because of the need for scaling. Whereas, the 2's complement multiplier tended to become simplified as precision was sacrificed.

Jenkins and Leon proposed a hybrid algorithm that combined features of a combinatorial vector multiplication technique and a residue number architecture (113) for both recursive and non-recursive digital filters (116). The vector multiplication eliminated general multiplication and the residue structure decomposed each operation into simpler subtasks that were appropriate for multiple microprocessor implementations. The hybrid algorithm could be used for recursive filters although the need to choose the moduli for efficient scaling limited the choice of possible residue systems. On the other hand, the algorithm appeared to be ideal for non-recursive structures where there was more flexibility in the choice of moduli.

In conclusion, the feasibility of applying RNS to implement digital filters have been demonstrated by developments of a number of authors (103-121). In some cases, such as unrounded integer multiplications, offer advantages over the conventional number-systems. As semiconductor technology continues to develop the implementation of digital signal processors with RNS will become more attractive. On the other hand, the extended dimension and added

flexibility for digital filter or signal processor implementation reinforces the need for design aids for system designers. To this end, CAD approach may gradually become a necessity.

Chapter 6 CONCLUSION

6.1 SUMMARY & CONCLUDING REMARKS

In the last few chapters and their associated appendices we have presented our efforts in providing a computer-aided approach for implementing digital filters. The object of this chapter is not to reiterate the discussion already presented but endeavour to highlight the salient features and to place our research into perspective. The chapter concludes with a recommendation for future works and related studies.

The objective of devising some aids to assist digital filter designer prompted our attempts in rationalizing the activities of a digital system designer. This has resulted the formulation of an "Interactive Model" which in turn has led to the development of a "Top-down Design Approach". The success of this Approach, however, rests heavily on computer analyses and simulations; and the formulation, design and implementation of a good design software package(s) for this purpose is therefore of paramount importance. To this end, the criteria of designing suitable computer aids and the bases for the evaluation and comparison among computer programming languages have been treated at great length.

APL (A Programming Language) has been found to be the most suitable language for our "Top-down Design Approach". The advantages and disadvantages of choosing APL are discussed. This is followed by a review of available APL processors and their implications on the design of CAD systems. In this connection, a CAD system based on a subset of APL is proposed and the implications of it are carefully considered. To ensure an economical use of the CAD system, we have discussed and proposed methods of interfacing APL with non-APL processors which may provide facilities lacking in APL. As a measure of sharing the cost of developing and maintaining a CAD system, we have proposed a method for a multi-user environment for the CAD system.

As an effort of providing subsystems for the "Top-down Design Approach", two extensive packages have been developed — DISPIC's and SIMLOGS. DISPIC's facilitate the design and simulation of programmable digital

filters (and signal processors) at the functional and register transfer levels. While SIMLOGS provides the tools for analysis and simulation of logical circuits/systems at the gate/logic level. Possible extensions for these packages are indicated in the pertinent chapters.

Within the framework of the Top-down Design Approach, we considered the mathematical and algorithmic studies of the residue number system to implement digital filters. Hardware studies of such implementation, aided by computer tools developed, such as LOGMIN, are conducted. Significant progress in this area by other researchers is briefly reviewed.

Our efforts have demonstrated the feasibility and advantages of the proposed Top-down Design Approach. With the limitation of resources (human, material and financial) it has not been able to fully implement and thoroughly evaluated the Top-down Design Approach. However, it is not too bold to claim that we have made a starting point and a step towards the right direction for such a system. It is hoped that in its present form it will assist designers and that efforts of others will enhance the usefulness of the Top-down Design Approach.

6.2 A RECOMMENDATION FOR FUTURE STUDIES

The results of the present research have laid a foundation for extended studies which could proceed in several directions. The following represents a number of items which have arisen but not addressed or adequately considered during the present studies.

6.2.1 Further Developments of the CAD System (fig. 19)

In section 2.6.1 (chapter 2), we have considered the syntax and semantics of a subset of APL that is useful for a hardware design language. In the light of this, we have developed a collection of APL (and non-APL) programs which are presented in the appendices. To enhance the usefulness of the design system, a library of standard components should be extended to provide devices such as a variety of buses, multiplexers, coders & decoders,

channel adaptors and interrupt devices. The library should be supported by an auxiliary software system which provides facilities for inquiry, archives and retrieval of components.

Having implemented such facility, a pre-translator may then be fully specified and implemented. Ultimately, a scheme for automatic translation within (and hopefully between) levels of design and the generation of discrete event programs may be improvised. As discussed in section 2.6.3 (chapter 2), the discrete event program may assume various forms; that is, problem oriented languages or APL may be adopted. The evaluation and a subsequent choice of the discrete event programming language is in itself an important research topic.

6.2.2 Subsystems for the Design Approach

LOGMIN, DISPIC's and SIMLOGS have been developed to provide subsystems for the CAD system; hence the Top-down Design Approach. Other important subsystems are — micro-assembler, and a universal microprocessor cross-assembler & simulators for associated microprocessors. The purpose of these has already been discussed in connection with DISPIC's in chapter three (section 3.3.3).

6.2.3 Interface with Non-APL Processors and Application Packages

In section 2.6 of chapter 2 (echoed at various places in the thesis), we have indicated the use of non-APL processors and application packages. It is uneconomical to develop the entire CAD system using only APL. Despite the power and capability APL provides, yet like any other language there exists some deficiencies. In time, these may be reduced, however it is essential to provide the capability to communicate or to invoke the facilities offered by other non-APL processors and application packages. This will provide an efficient use of computing resources and derive benefits from existing and established application packages. We have discussed the technical feasibility of such arrangement; however, for the purpose of meeting user requirements (e.g. ease of assimilation) and consistent

operating procedure, a precise interface should be carefully studied and specified. The definition of such interface must not be confined to a particular operating system or computer installation.

6.2.4 Evaluation of the Top-down Design Approach

Our research has been motivated by the objective of providing some aids to assist the digital system designer. To this end, we have proposed and implemented the Top-down Design Approach within the constraints of available resources. In the absence of substantial evidence pointing to the contrary, we believe that the Design Approach will provide the intended aids. However, it is important that this educated opinion should be proven. For this, we recommend extensive efforts expanded on item one (i.e. section 6.2.1) above and followed by conducting a program of evaluative research. With "unbiassed subjects", one may assess the ture usefulness of the Design Approach.

6.2.5 Application of RNS to Implement Digital Signal Processors

In chapter 5, we have reported on the efforts of studying the implementation of digital filters using RNS. At this time of writing, sufficient evidence has indicated the feasibilities and in some cases the advantages of applying RNS to implement digital filters. Much efforts are still required to exploit the properties of RNS to provide high-speed signal processing. Prototypes of a complete system need to be constructed and controlled field trials conducted to evaluate the performance of an RNS digital filter.

Redundant RNS codes could be (and have been) used for error detection and correction. It will be interesting to apply jointly the parallel structure and error detection & correction properties of RNS to digital signal processing. With the capabilities of semi-conductor technology, such as read-only-memories and microprocessors, there seems to be a possibility of achieving economical but sophisticated high speed digital signal processors.

On the other hand, design parameters and techniques for the implementation of RNS digital signal processors are involved and not easily assimilated by designers; in particular industrial designers who are working with tight schedules. Computer-aids may be devised to assist designers in this respect. Indeed, this has motivated our research on a computer-aided design for implementing digital filters.

REFERENCES

[1] Gold, B. & C.Rader (69), Digital Processing of Signals, N.Y.: McGraw Hill.

[2] Constantinides, A.G. & R.E.Bogner (Ed.) (75), Introduction to Digital Filtering London : John Wiley.

[3] Oppenheim, A.V. & R.N.Schafer (75), Digital Signal Processing, N.J.: Prentice - Hall.

[4] Stanley, W.D. (75), Digital Signal Processing, V.A. Reston Pub.

[5] Liu, B. (Ed.) (75), Digital Filters and the Fast Fourier Transform, P.A., Stroudsbury : Dowden, Hutchinson Ross.

[6] M.H. Ackroyd (73), Digital Filters, London: Buterworths.

[7] Rabiner, L.R. and B. Gold (75), Theory and Application of Digital Signal Processing, Englewood Cliffs., N.J.: Prentice-Hall.

[8] Oppenheim, A.V. (Ed.) (78), Applications of Digital Signal Processing, N.J.: Prentice Hall.

[9] Chao, J.T. (77), "The Digital Signal Processor Industry - Its Past Performance, Present Status & Future Prospects" IEEE International Conf. on ASSP, Hartford, May 1977

[10] Allen, J. (75), "Computer Architecture for Signal Processing" Proc.IEEE, Vol.63, No.4, April 1975.

[11] Salazar, A.C. (Ed.) (78), Digital Signal Computers & Processors, N.Y. : IEEE Press Book.

[12] Kratz, G.L., W.W.Sproul, & E.T.Walendziewiz, (74), "A Microprogrammed Approach to Signal Processing", IEEE Trans. on Computers, Vol,C-23, August 1974, pp. 808-817.

[13] Wu, Y.S. (75), "Microprogramming Applications to Signal Processing Architecture" in R.W.Hartenstein & R. Zaks (Ed.) Microarchitecture of Computer Systems, N.Y.: Euromicro, North-Holland.

[14] Lai, P.K.M. (72), A Preliminary Study of Hardware Implementation of Digital Filters, DIC thesis, Elect.Eng.Dept., Imperial College of Science and Technology, London.

[15] Lai, P.K.M. (73), Studies of Digital Filter Implementation using Residue Number System (RNS) -- an overview, Internal Rept., Elect.Eng.Dept., Imperial College.

[16] Lai, P.K.M. (74), Residue Arithmetic and Some Algorithms for Rational Number Operations, Internal Report, Elect.Eng.Dept, Imperial College.

[17] Lai, P.K.M. (76), "A Systematic Approach to the Design of Hardware Digital Signal Processors" Proc. 19th Midwest Symp. on Circuits & Systems, Milwaukee, USA.

[18] Peled, A. & B.Liu (74), "On the Ordering Problem of Cascade Realization of Non recusive Digital Filters" Proc. NTC 74, USA, pp. 266-271

[19] Breuer, M.A. (Ed.) (72), Design Automation of Digital Systems: Theory & Techniques, Englewood Cliffs, N.J.: Prentice-Hall.

[20] Breuer, M.A. (Ed.) (75), Digital System Design Automation: Languages, Simulation & Data Base, Woodland Hills, CA.: Computer Science Press.

[21] Lewin, (77), Computer-Aided Design of Digital Systems, London: Edward Arnold.

[22] Reiner W. Hartenstein (77), Fundamentals of Structured Hardware Design (A Design Language Approach at Register Transfer Level) Amsterdam: North-Holland Pub.

[23] Minksy, M.L.(67), Computer - Finite and Infinite Machines, Englewood Cliff, N.J.: Prentice-Hall.

[24] Weber, H.(74), "Ein Programmiersystem Zur Unterstutzung der Rechnerentwicklung" in H.Hasselmeier & W.G.Spruth: Rechnerstrukturen, Oldenbourg, Munich.

[25] Iverson, K.E.(72), Algebra: An Algorithmic Treatment, Reading, Mass.: Addison-Wesley.

[26] R.W.Floyd (67), "Assigning Meanings to Programs" in Mathematical Aspects of Computer Science, edited by J.T.Schwartz, pp. 19-32, American Mathematical Society, Providence, R.I.,

[27] Buxton, J.M.etal. (Ed.) (76), Software Engineering; Concepts & Techniques N.Y.: Petrocelli/Charter.

[28] IEEE (74), "Hardware Description Languages," Special Issue of Computer, Vol 7, No.12.

[29] Glazer, E.L. (69), "Computer aided design for computing systems," IEEE Conf. on Computer Science and Technology, No.55, pp.13-19.

[30] Elson, Mark (73), Concepts of Programming Languages, Chicago: Science Research Associates, Inc.

[31] G.M.Weinberg (71), The Psychology of Computer Programming, N.Y.: Van Nostrand Reinhold.

[32] J.Maynard (72), Modular Programming, London: Butterworths.

[33] Dahl, B.E.Dijkstra, & C.A.Hoare (72), Structure Programming, London: Academic Press.

[34] McGowan, C. & J.Kelley (75), Top-Down Structured Programming Techniques N.Y.: Petrocelli/Charter.

[35] Myers, G. (76), Reliable Software Through Composite Design, N.Y.: Petrocelli/Charter.

[36] Katzan, H. Jr. (76), System Design and Documentation (An Introduction to the HIPO Method), N.Y.: Van Nostrand.

[37] Jackson, M. (76), Basic Structured Program Design: Workshop, London: Michael Jackson Systems, Ltd.

[38] Jackson, M.A. (75), Principles Of Program Design, London: Academic Press.

[39] Penfield, P. Jr. (71), Martha User's Manual, Cambridge, Mass: MIT Press.

[40] Lewin, M.H. (67), "An Introduction to Computer Graphic Terminals", Proc. IEEE, Vol.55, pp.1544-1552.

[41] Dietmeyer, D.L. (71), Logic Design of Digital Systems, Boston: Allyn and Bacon.

[42] Y. Chu (70), Introduction to Computer Organization, Englewood Cliff, N.J.: Prentice-Hall.

[43] Y. Chu (72), Computer Organization and Microprogramming, Englewood Cliff, N.J.: Prentice-Hall.

[44] IEEE (74), "Hardware Description Languages", Special Issue of Computer, Vol.7, No.12.

[45] Iverson K.C. (62), A Programming Language, N.Y.: J. Wiley & Sons.

[46] Falkoff, A.D. & E.E. Iverson & E.H. Sussenguth, (64), A Formal Description of System/360,IBM System Journal, Vol.3.

[47] Gilman L. & A.J. Rose, (73), APL - An Interactive Approach (2nd Ed.) N.Y.: J. Wiley & Sons.

[48] Katzan, H. Jr. (71), APL User's Guide, N.Y.: Van Nostrand Reinhold.

[49] Falkoff A.D. & K.E. Iverson, (73), APLSV User's Manual, IBM, Pub. No. SH20-1460. Philadelphia.

[50] IBM, (75), IBM 5100 APL Reference Manual (2nd Ed.), IBM, Pub. No. SA21-9213.

[51] Driscoll, H.A. (A) User Documentation for APL*PLUS System Functions Maryland:
Scientific Time Sharing Corp. (STSC), Working Memo. No.119, June, 1976.
(B) APL*PLUS File Subsystem Instruction Manual STSC, 1971.

[52] BOC (77), A General Guide to the APL Service, Middlesex: BOC Datasolve, Issue 1.0.

[53] Digital Equip. Corp. (76), DEC System 10: APL Programmer's Reference Manual Maynard, Mass: Digital Equip. Corp, Pub. No. DEC-10-LALA-B-D.

[54] Xerox, (75), Xerox APL (language & Operations Reference Manual, U.S.A.: Xerox.

[55] Wiedman, C. (75), APLUM Reference Manual (2nd Ed.), Amherst: Univ. of Massachusetts.

[56] Control Data, (76), APL Version 2 Reference Manual, Control Data Corporation, Pub. 60454000.

[57] Bastib, D.A.,et al. (72), "A Set of Interactive APL Programs for Fault Detection and Location in Digital Circuits", 1972 International Symposium on Fault-Tolerant Computing, Newton, Mass.

[58] G.J. Klir, (72), Introduction to the Methodology of Switching Circuits, N.Y.: Van Nostrand.

[59] Spence, R. (74), Resistive Circuit Theory, N.Y.: McGraw-Hill.

[60] Niederhauser, J.R. (71), Digital Logic Simulator (DLS), Ohio: AFIT Thesis.

[61] Lai, P.K.M. (74), Logic Minimization and a FORTRAN IV Program for Logic Minimization, Internal report, RNS-CAD/25-4-74, Elect. Eng. Dept., Imperial College.

[62] Calahan, D.A. (72), Computer Aided Network Design (revised edition), N.Y.: McGraw-Hill.

[63] Abrams, P.S. (75), "What's Wrong With APL?", APL 75, Pisa, Italy, pp.1-8

[64] Earnshwas, R.A. (75), "Is APL a Viable Programming Language?" The Computer Journal, Vol.18, No.4, pp.313-323.

[65] DEC (73), DEC writer II Printer Terminal User Manual, Digital Equipment Corp. Maynard: Massachusetts, U.S.A.

[66] Anderson Jacobson, (74), AJ 832 Keyboard Printer Terminal Operator's Manual, Sunnyvale, California: Aderson Jacobson, Inc. (USA)

[67] Taylor, F. & Smith, S.L. (66), Digital Signal Processing In FORTRAN, Lexington, Mass: Lexington Books, 402pp.

[68] Steiglitz, K. (70), "Computer-Aided Design of Recursive Digital Filters", IEEE Trans. Audio Electroacoust., Vol.AU-18, pp 123-129.

[69] Nagle, H.T. Jr. & M.M. Edgeworth (71), Computer Aided Design of Digital Filters, TR. No.14, NAS8-20163, George C. Marshall Space Flight Centre, NASA, Redstone Arsenal, Ala.

[70] McClellan J.H., T.W.Parks, & L.R.Rabiner (73), A Computer Program for Designing Optimum FIR Linear Phase Digital Filters, IEEE Trans. on Audio and Elec., Vol.AU-21, No.6, pp.506-526.

[71] Childers D. & A.Durling (75), Digital Filtering & Signal Processing, St. Paul, Mn: West, 539 pp.

[72] Knuth, D.E. (74), Structured Programming with GOTO Statements, Standford University Technical Report, STAN-CS-74-416.

[73] Hoare, C.A.R. (61), "Algorithm 64, QUICKSORT", CACM, No.4.

[74] Giloi, W.K. and H.Liebig, (73), "A Formalism for Description and Synthesis of Logical Algorithms and their Hardware Implementation", IEEE Trans. on Computer, Vol.C-23, No.9, pp. 897-906

[75] Lai, P.K.M. (75), Simulation of an 8-bit Digital Computer using APL/360, Internal rept., Elect. Eng. Dept., Imperial College.

[76] Lai, P.K.M. (78), "A Controlled Design Process of Implementing Digital Filters — Illustrated by 'DISPIC' ", 21st Migwest Symposium on Circuits & Systems, Iowa State University, Iowa, USA, August, 1978.

[77] Lai, P.K.M. (77), "The Development of SIMLOGS as an Aid to Verify Digital Systems", 20th Midwest Symposium on Circuits & Systems, Lubbock, U.S.A.

[78] Lai, P.K.M. (77), "Modelling and Simulating Logic Circuits by SIMLOGS" Journees D'Electronique, EPFL, Lausanne, Switzerland.

[79] Freedman, A.S. & P.R.Menon (71), Fault Detection in Digital Circuits, Englewood Cliffs: Prentice Hall.

[80] Hardy A.H. and E.M. Wright (54), An Introduction to the Theory of Numbers, Oxford University Press.

[81] Jones (64), Modular Arithmetic, N.Y.: Blaisdell Publishing.

[82] Salz, N.P. (60), "Number — the Language of Engineering " Electro-Technology, pp.128-142.

[83] Vinogradov, I.M. (54), An Introduction to the Theory of Numbers, N.Y.: Dover.

[84] Aiken, H. and W. Semon (59), Advanced Digital Computer Logic; Tech. Rept. No.WAC TR-59-472, Harvard University, Cambridge, Mass, U.S.A.

[85] Harvard University (61), Notes on Modular Number Systems, Tech. Rept. No.ASD 61-12, Harvard University, Cambridge, Mass.

[86] Chu, Y. (62), "Residue Number Arithmetic" in Digital Computer Design Fundamentals. N.Y. : McGraw-Hill.

[87] Driese, E.C., G.E.Glen and R.E.Young (61), Computer Applications of Residue Class Notations, Tech. ASD-TR 61-189 (AD-269 123), Wright-Patterson A.F.B.

[88] Flores, I. (63), "Modular Arithmetic" in The Logic of Computer Arithmetic, N.J. : Pretice-Hall.

[89] Garner, H.L. (59), "The Residue Number System," IRE Trans, on Electronic Computers, Vol.EC-8, No.2, pp. 140-147.

[90] Garner, H.L., et al. (61), Residue Number Systems for Computers, Tech. No. ASD-TR 61-483, University of Michigan, Ann Arbor, Michigan.

[91] Garner, H.L. (65), "Number Systems and Arithmetic" in Advances in Computers, Vol.6, N.Y. : Academic Press, pp. 131-194.

92. Hastings, C.W. and T.A.Vessey (63), Residue Arithmetic Computing Techniques, MPG Res. Doc. R-Rd 6261, Honeywell, St.Paul Minnesota.

93. Hastings, C.W. (66), "Automatic detection and correction of errors in digital computers using residue arithmetic", Proc. IEEE Conference (Region Six), pp. 429-464.

94. Levin, M., J.Marx and S.Levy (60), "New Techniques in Residual Arithmetic," Conf. Proc. of the 4th National Convention on Military Electronics, pp. 183-189.

95. Merrill, R.D. Jr. (64), "Improving Digital Computer Performance using Residue Number Theory", IEEE Trans. on Electronics Computer, Vol. EC-13, pp. 93-101.

96. Svoboda, A. (57), "Rational Numerical System of Residue Classes," Stroje na Zpracovani Informaci (Machines for the Processing of Information), Sbornik 5 (Symp. 5), Nakl. CSAV, Praha, pp. 9-37.

97. Svoboda, A. (58), "The Numerical System of Residual Classes in Mathematical Machines," Congreso Internac. de Automatica Actas, Madrid, pp. 388-397.

98. Svoboda, A. (59), "The numerical system of residue classes in mathematical machines," Proc. of the International Conference on Information Processing, UNESCO, Paris, pp. 419-422, October 15-20 June, 1959, London : Butterworth.

99. Svoboda, A. (62), "Computer Progress in Czechoslovakia II: The Numerical System of Residue Classes" in Digital Infromation Processors edited by W.Hoffmann, N.Y. : Interscience Publishers, pp. 543-574.

100. Lockheed Missiles and Space (61), Interim Technical Report on Modular Arithmetic Techniques, Tech. Rep. No. ASD-TR 61-472, 151pp. Wright-Patterson Air Force Base, Ohio.

101. Lockheed Missiles and Space (63), Modular Arithmetic Techniques, Tech. Rept. No. ASD-TDR 62-686, Wright-Patterson AFB, Dayton, Ohio.

102. Szabo, N.S. and R.I.Tanaka (67), Residue Arithmetic and its Applications to Computer Technology, N.Y. : McGraw-Hill.

103. Watson, R.W. (65), Error Detection and Correction and other Residue Interacting Operations in a Redundant Residue Number System, Ph.D. dissertation, University of California, Berkeley.

104. Watson, R.W. (66), "Theoretical foundations for reliable machine design using a redundant residue number system code," Proc. IEEE Conference (Region Six), pp. 465-490.

105. Watson, R.W. and C.W.Hastings (66), "Self-Checked Computation using Residue Arithmetic," Proc.IEEE,Vol.54, pp.1920-1931.

[119] Jenkins, W.K. (77), "A Technique for High Precision Digital Filtering with Multiple 8-bit Microprocessors," 20th Midwest Symp. on Circuits, Lubbock, Texas.

[120] Jenkins, W.K. (78), "Techniques for Residue-to-Analog Conversion for high data rate Digital Filtering," 1978 IEEE International Conference on Acoustics, Speech & Signal Processing, Tulsa, Oklahoma, pp.804-807.

[121] Jenkins, W.K. (78), "A Comparison of Residue Number Multipliers and 2's-complement Multipliers Implemented by stored Multiplication tables," 1978 IEEE International Symposium on Circuits and Systems, N.Y.

[122] Jullien, G.A., et. al. (77), "Hardware Realization of Digital Signal Processing Elements using the Residue Number System," 1977 IEEE International Conference on a Acoustics, Speech and Signal Processing, Hartford, CT.

[123] Jillien, G.A. (78), "Residue Number Scaling and Other Operations using ROM Arrays," IEEE Trans on Computers, Vol C-27, No/4, pp.325-336.

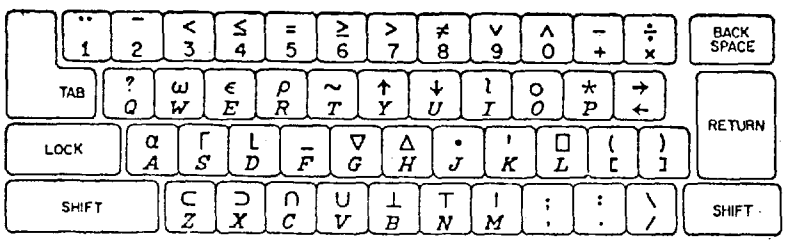
[124] A. Peled and Bede Liu, (74), "A New Hardware Realization of Digital Filters," IEEE Transactions on Acoustics, Speech, and Signal Processing, Vol. ASSP-22, pp.456-462

Appendix 1 :

SUMMARY OF APL/360 FEATURES [1]

---

[1] Extracted from: Sandra Pakin(1968), "APL/360" Reference Manual" (2nd edition), Chicago: Science Research Associates, Inc., 1972.



An APL Keyboard

	Explict Result	No Explict Result
Niladic	∇R← NAME	∇ NAME
Monadic	∇R← NAME X	∇ NAME X
Dyadic	∇R←Y NAME X	∇ Y NAME X

APL Function Types

Monadic form *fB*		f	Dyadic form *AfB*	
Definition or example	Name		Name	Definition or example
+*B* ↔ 0+*B*	Plus	+	Plus	2+3.2 ↔ 5.2
-*B* ↔ 0-*B*	Negative	-	Minus	2-3.2 ↔ ¯1.2
×*B* ↔ (*B*>0)-(*B*<0)	Signum	×	Times	2×3.2 ↔ 6.4
÷*B* ↔ 1÷*B*	Reciprocal	÷	Divide	2÷3.2 ↔ 0.625
B | ⌈*B* | ⌊*B* 3.14 | 4 | 3 ¯3.14 | ¯3 | ¯4	Ceiling	⌈	Maximum	3⌈7 ↔ 7
	Floor	⌊	Minimum	3⌊7 ↔ 3
⋆*B* ↔ (2.71828..)⋆*B*	Exponential	⋆	Power	2⋆3 ↔ 8
⍟⋆*N* ↔ *N* ↔ ⋆⍟*N*	Natural logarithm	⍟	Logarithm	*A*⍟*B* ↔ Log *B* base *A* *A*⍟*B* ↔ (⍟*B*)÷⍟*A*
|¯3.14 ↔ 3.14	Magnitude	|	Residue	Case | *A*|*B* *A*≠0 | *B*-(|*A*)×⌊*B*÷|*A* *A*=0, *B*≥0 | *B* *A*=0, *B*<0 | Domain error
!0 ↔ 1 !*B* ↔ *B*×!*B*-1 or !*B* ↔ Gamma(*B*+1)	Factorial	!	Binomial coefficient	*A*!*B* ↔ (!*B*)÷(!*A*)×!*B*-*A* 2!5 ↔ 10 3!5 ↔ 10
?*B* ↔ Random choice from ⍳*B*	Roll	?	Deal	A Mixed Function (See Table 2)
○*B* ↔ *B*×3.14159...	Pi times	○	Circular	See Table at left
~1 ↔ 0 ~0 ↔1	Not	~		
		∧	And	*A* *B* | *A*∧*B* | *A*∨*B* | *A*⍲*B* | *A*⍱*B* 0 0 | 0 | 0 | 1 | 1 0 1 | 0 | 1 | 1 | 0 1 0 | 0 | 1 | 1 | 0 1 1 | 1 | 1 | 0 | 0
		∨	Or	
		⍲	Nand	
		⍱	Nor	
		<	Less	Relations Result is 1 if the relation holds, 0 if it does not: 3≤7 ↔ 1 7≤3 ↔ 0
		≤	Not greater	
		=	Equal	
		≥	Not less	
		>	Greater	
		≠	Not Equal	

(-*A*)○*B*	*A*	*A*○*B*
(1-*B*⋆2)⋆.5	0	(1-*B*⋆2)⋆.5
Arcsin *B*	1	Sine *B*
Arccos *B*	2	Cosine *B*
Arctan *B*	3	Tangent *B*
(¯1+*B*⋆2)⋆.5	4	(1+*B*⋆2)⋆.5
Arcsinh *B*	5	Sinh *B*
Arccosh *B*	6	Cosh *B*
Arctanh *B*	7	Tanh *B*

Table of Dyadic ○ Functions

Table A1.1 : Primitive Scalar Functions

Name	Sign[1]	Definition or example[2]
Size	⍴A	⍴P ↔ 4 ⍴E ↔ 3 4 ⍴5 ↔ ⍳0
Reshape	V⍴A	Reshape A to dimension V 3 4⍴⍳12 ↔ E 12⍴E ↔ ⍳12 0⍴E ↔ ⍳0
Ravel	,A	,A ↔ (×/⍴A)⍴A ,E ↔⍳12 ⍴,5 ↔ 1
Catenate	V,V	P,⍳2 ↔ 2 3 5 7 1 2 'T',⊂'HIS' ↔ 'THIS'
Index[3,4]	V[A] M[A;A] A[A;...;A]	P[2] ↔3 P[4 3 2 1] ↔7 5 3 2 E[1 3;3 2 1] ↔ 3 2 1 11 10 9 E[1;] ↔ 1 2 3 4 ABCD E[;1] ↔ 1 5 9 'ABCDEFGHIJKL'[E] ↔ EFGH IJKL
Index generator[3]	⍳S	First S integers ⍳4 ↔ 1 2 3 4 ⍳0 ↔ an empty vector
Index of[3]	V⍳A	Least index of A P⍳3 ↔2 5 1 2 5 in V, or 1+⍴V P⍳E ↔ 3 5 4 5 4 4⍳4 ↔ 1 5 5 5 5
Take	V↑A	Take or drop |V[I] first 2 3↑X ↔ ABC (V[I]≥0) or last (V[I]<0) EFG
Drop	V↓A	element of coordinate I ¯2↑P ↔ 5 7
Grade up[3,5]	⍋A	The permutation which ⍋3 5 3 2 ↔ 4 1 3 2
Grade down[3,5]	⍒A	would order A (ascending or descending) ⍒3 5 3 2 ↔ 2 1 3 4
Compress[5]	V/A	1 0 1 0/P ↔ 2 5 1 0 1 0/E ↔ 1 3 5 7 9 11 1 0 1/[1]E ↔ 1 2 3 4 ↔ 1 0 1/E 9 10 11 12
Expand[5]	V\A	1 0 1\⍳2 ↔ 1 0 2 1 0 1 1 1\X ↔ A BCD E FGH I JKL
Reverse[5]	⌽A	DCBA IJKL ⌽X ↔ HGFE ⌽[1]X ↔ ⊖X ↔ EFGH LKJI ⌽P ↔ 7 5 3 2 ABCD
Rotate[5]	A⌽A	BCDA 3⌽P ↔ 7 2 3 5 ↔ ¯1⌽P 1 0 ¯1⌽X ↔ EFGH LIJK
Transpose	V⍉A	AEI Coordinate I of A 2 1⍉X ↔ BFJ becomes coordinate CGK V[I] of result 1 1⍉E ↔ 1 6 11 DHL
	⍉A	Transpose last two coordinates ⍉E ↔ 2 1⍉E
Membership	A∈A	0 1 1 0 ⍴W∈Y ↔ ⍴W E∈P ↔ 1 0 1 0 P∈⍳4 ↔ 1 1 0 0 0 0 0 0
Decode	V⊥V	10⊥1 7 7 6 ↔ 1776 24 60 60⊥1 2 3 ↔ 3723
Encode	V⊤S	24 60 60⊤3723 ↔ 1 2 3 60 60⊤3723 ↔ 2 3
Deal[3]	S?S	W?Y ↔ Random deal of W elements from ⍳Y

Table A1.2 : Primitive Mixed Functions
(see notes on next page)

1. Restrictions on argument ranks are indicated by: S for scalar, V for vector, M for matrix, A for Any. Except as the first argument of S⍳A or S[A], a scalar may be used instead of a vector. A one-element array may replace any scalar.

2. Arrays used in examples: P ↔ 2 3 5 7 E ↔ 1 2 3 4 / 5 6 7 8 / 9 10 11 12 X ↔ ABCD / EFGH / IJKL

3. Function depends on index origin.

4. Elision of any index selects all along that coordinate.

5. The function is applied along the last coordinate; the symbols ⌿, ⍀, and ⊖ are equivalent to /, \, and ⌽, respectively, except that the function is applied along the first coordinate. If [S] appears after any of the symbols, the relevant coordinate is determined by the scalar S.

Notes to table A1.2

SYSTEM COMMANDS

COMMAND	PURPOSE
)*CLEAR*	Activates a clear workspace (160)
)*CONTINUE*	Terminates a work session and stores the active workspace in *CONTINUE* (171)
)*CONTINUE HOLD*	Like)*CONTINUE*, but holds the dial-up connection (171)
)*COPY* WSID key	Copies all functions, variables, and groups from a stored workspace (163)
)*COPY* WSID key NAME	Copies a function, a variable, or a group from a stored workspace (163)
)*DIGITS* 1-16	Sets number of significant digits to be displayed (169)
)*DROP* WSID	Deletes a stored workspace (161)
)*ERASE* NAME(S)	Erases objects listed (167)
)*FNS* letter	Lists names of defined functions (152)
)*GROUP* NAME LIST	Defines a group NAME whose members are LIST (162)
)*GROUP* NAME	Disperses group NAME (153)
)*GRP* NAME	Lists members of group NAME (153)
)*GRPS* letter	Lists names of groups (153)
)*LIB* number	Lists names of workspaces in designated library (154)
)*LOAD* WSID key	Activates a copy of a stored workspace (155)
)*MSG* PORT text	Sends text to designated port; keyboard locks (170)
)*MSGN* PORT text	Like)*MSG*, but the keyboard unlocks (170)

NOTE :

1. Object indicated by capital roman type is a required part of the command; object indicated by lowercase roman type is an optional part of the command.
2. Number in parentheses refers to the page on which the command is discussed in Sandra Pakin.

System commands -- continued

)*OFF* lock	Terminates a work session (171)
)*OFF HOLD* lock	Like)*OFF*, but holds the dial-up connection (171)
)*OPR* text	Sends text to APL operator; keyboard locks (170)
)*OPRN* text	Like)*OPR*, but the keyboard unlocks (170)
)*ORIGIN* 0-1	Sets index origin (168)
)*PCOPY* WSID key	Like)*COPY*, but protects the contents of the active workspace (166)
)*PCOPY* WSID key NAME	Like)*COPY* NAME, but protects the contents of the active workspace (166)
)*PORTS*	Lists port number and associated user code for all terminals signed on (170)
)*PORTS* CODE	Lists port number(s) associated with designated user's code (170)
)*SAVE*	Restores a copy of the active workspace (155)
)*SAVE* WSID lock	Stores a copy of active workspace (155)
)*SI*	Lists state indicator (121)
)*SIV*	Lists state indicator plus local variables (122)
)*SYMBOL* number	Sets size of symbol table (170)
)*VARS* letter	Lists names of global variables (152)
)NUMBER key	Sign on (140)
)*WIDTH* 30-130	Sets width of output line (169)
)*WSID*	Identifies active workspace (160)
)*WSID* NAME	Changes identification of active workspace (160)

Appendix 2 :

LOGMIN : A LOGIC MINIMIZATION PACKAGE IMPLEMENTED IN FORTRAN IV

LOGMIN : A LOGic MINimization Package Implemented in FORTRAN IV[1]

General Description of the Package

Package LOGMIN consists of a main program LOGMIN, a subroutine MINTE and a function IAND. All programs are written in standard FORTRAN IV codes; but IAND is in "extended FORTRAN" code and is therefore a compilier-dependent function. IAND is accepted by two compilers, viz. FTN & MNF, at the Imperial College Computer Centre (ICCC). All programs of LOGMIN will run on the CDC 6000 and CYBER series of computers and under the operating system KRONOS and NOS.

LOGMIN examines the logical "True", "False" and "Redundant" (i.e. "Don't Care") minterms that describe a single-output combinational function. These minterms are coded, in order as stated, as "1", "0" and "-1". Subroutine MINTE is based on a programmed algorithm developed by Carroll and Jordan (1) and modified by Rhyne (2).

Prime implicants describing the single-output function are generated and essential implicants are selected by the program. When cyclic condition occurs, an approximate selection technique developed by Bowman and McVey (3) is used to select a suitable set from the cyclic set. Although the approximate method does not guarantee minimal cost, yet it offers a reasonable compromise between computational expense and reduction of gating cost.

---

[1] Extracted from a writing of this author; entitled "Logic Minimization and a FORTRAN IV Program for Logic Minimization", Internal report, RNS-CAD/25-4-74, Electrical Engineering Dept., Imperial College.

Source code for LOGMIN is included and information for using the package is detailed in the next section. For brevity reason, detailed description and flow-charts of LOGMIN are not included. At this time of writing, LOGMIN is dimensioned for a maximum of six variables for the J1 (instant turn-around, and time sharing) job catagory and a maximum of ten variables for catagories such as J4 at the ICCC. An option is available for performing both "Sum of Product" (SOP) and "Product of Sum" (POS) minimization in one computer run.

Information for using LOGMIN

1) Program Card Structure

Card Structure for the program deck is as follows:

a) With source deck

```
JOB card
PASSWORD card
FTN(OPT=0,R=1)
LOG.
END-OF-RECORD card

      LOGMIN source deck

END-OF-RECORD card

      Logic (state-table) Data deck

END-OF-FILE  card
```

b) With object deck stored as permanent file (on disc)

```
JOB card
PASSWORD card
GET(LOGMIN1)
LIBRARY(FTNLIB)
LOGMIN1.
END-OF-RECORD card

      Logic Data deck

END-OF-FILE card
```

(Note : Version LOGMIN1 is dimensioned for a maximum of 6 variables & stored as a private file)

2) Input data format

The first card specifies the number (N) of variables of the function being minimized and the option (OPT) for "SOP only" or "SOP and POS" minimization in one computer run. Each of the next and subsequent data cards specifies a minterm of the single-output combinational function. The last data card also indicates end of data card (NCARD) deck. Data card formats are:

a) First Card (specifies "N" & "OPT")

Column 1-2 : N; integer decimal numbers 1 to 6 punched on column two.

Column 8 : OPT; blank for "SOP only" , decimal 9 for "SOP & POS"

b) Subsequent data cards (specify minterms and NCARD)

Column 1-2 : minterm codes with,

1 = True
0 = False
-1 = Redundant or Don't cares

Column 6 : NCARD, blank except the last card on which a decimal integer 9 is punched.

NOTE : For easier interpretation of output, order of minterm entry is recommended as that on the truth or state table.

3) Output of LOGMIN

With reference to example one (see section on examples), segments of the output are :-

a) **SOP** Option only

1) Input data in array-D for **SOP** minimization

All input data cards, each corresponding to a minterm, are listed in form "number of data card = minterm code"

2) Boolean minimization program table

Headings of the table is self-explanatory. The first set of listing corresponds to the minterms of the truth table, i.e. minterm number = minterm".

3) Prime Implicant table

The first column (No.) indicates the number of prime implicant generated after minimization. The second column shows the corresponding cost (first-level gating cost). The next "column" of the table is the prime implicant matrix. Columns of the matrix correspond to Boolean variables and labelled as A, B, C, ... etc.; with A corresponding to the "most significant bit" depicted by a truth table. Each row of the matrix corresponds to a prime implicant. The last column of the table indicates all essential implicants and chosen implicants when a cyclic pattern exists.

4) Constraint table

When a cyclic pattern arises, a constraint table is printed indicating all the cyclic implicants and their minterm being covered. From the table, one can construct the cyclic chart as an aid for branching method to "check" the computer solution. Alternatively one can use the Petrick's method (4) for the same purpose.

b) **SOP & POS** Option

Two outpus are given, with the first corresponding to **SOP** and the second corresponding to the **POS** minimization. Interpretation of both outputs are as that described in section (a) above.

Limitations of LOGMIN

LOGMIN has been designed to utilize computer resources as efficiently as possible; however, this does result some restriction on the application of LOGMIN. It is advisable to bear these limitations in mind when using LOGMIN.

1) The present version of LOGMIN handles a maximum of six variables for the time-sharing version (i.e. LOGMIN1); while 10 variables for batch-mode operation. This restriction can be removed by re-dimensioning variable arrays in the source version of LOGMIN.

2) Particular case, like example four — full adder design, cannot be minimized by LOGMIN. In fact neither the Kanaugh Map nor the Quine-McCluskey method will minimise such function. Special technique, such as the use of EXCLUSIVE-OR function, is required to solve this particular case.

3) A short-coming of the Kanaugh Map and the Quine-McCluskey method is that both minimize the boolean function for AND/OR logic. NAND/NOR logic subsequently has to be derived from a minimized AND/OR logic function; the solution is not necessarily a minimal solution. Hence LOGMIN will not guarantee minimal NAND/NOR minimization.

4) Many combinational design problems have multiple-outputs. The simplest approach to multiple-output design problem is to consider each output separately — minimize and implement each individually. This approach ignores the the possibility of some "common terms" which can be shared by all outputs to yield further reduction in cost of implementation. Further reduction, however, can be accomplished by informally comparing the various output functions and identifying the common terms. Again this process is tedious and error-proned; hence computerized assistance is desirable. LOGMIN in its present state unfortunately does not handle multiple-output minimization problems.

Some Illustrative Example - LOGMIN solutions

Example one : Function $F(A,B,C,D,E) = \sum (0,3,4,6,7,15, 21,23,26,28, X_2, X_8, X_{12}, X_{14}, X_{17}, X_{24}, X_{31})$ is to be minimized. The computer solution is,

$$F = \overline{A}\,\overline{B}\,D + B\,\overline{D}\,\overline{E} + A\,B\,\overline{C}\,\overline{E} + \overline{A}\,\overline{D}\,\overline{E} + C\,D\,E + A\,\overline{B}\,\overline{D}\,E$$

(first three terms: ESSENTIAL IMP.; last four terms: CHOSEN IMP.)

A cyclic pattern which exists is indicated by "CHOSEN" and "CONSTRAINT TABLE" in the computer printout. From the constraint table we can derive all covering sets by the Petrick's method, so that

$$\begin{aligned}\text{Covering set} &= (1+2)(1+2+4)(5+6)(8+9)(6+9)\\ &= (1+2)(5+6)(9+6.8)\\ &= 1.5.9 + 1.6.8 + 1.6.9 + 2.5.9 + 2.6.8 + 2.6.9\end{aligned}$$

The cost units (c.u.) for all these covering sets turns out to be the same for this example; viz. 10 c.u. LOGMIN has chosen the set (1.6.8).

Example two : This example demonstrates the option **SOP & POS** minimizations. Function $F(A,B,C,D) = \sum(1,4,6,7,8,9,10,15)$ is to be minimized.

(1) For the **SOP** solution

$$F = \underbrace{\overline{B}\,\overline{C}\,D + \overline{A}\,B\,\overline{D} + A\,\overline{B}}_{\text{ESSENTIAL}} + \underbrace{B\,C\,D}_{\text{CHOSEN}}$$

$$\begin{aligned}\text{Covering set} &= (3+4)(4+6)\\ &= 4 + 3.6\end{aligned}$$

Cost units for (4) is 3 c.u., while (3.6) has 6 c.u.; hence (4) should be selected and this was in fact the solution of LOGMIN.

(2) For the **POS** solution

$$\overline{F} = \overline{A}\,B\,\overline{D} + \overline{A}\,B\,\overline{C} + B\,\overline{C}\,D + A\,B\,\overline{D}$$

By the DeMorgan's Theorem,

$$F = (A+B+D)(A+B+\overline{C})(\overline{B}+C+\overline{D})(\overline{A}+B+D)$$

Since the SOP solution gives a total cost of 11 c.u. and that for the POS gives 12 c.u., the former solution is prefered.

Example three : Comments and interpretations are as those for example two. Example three is included to show that LOGMIN also handles the "Don't care" terms.

Example four : Full Adder. This is a special case where formal minimization fails. This example is included to indicate careful interpretation of results is advisable; when printout shows all prime-implicants are essential, one should check whether they are identical in form and in number with the input minterms.

Example five : Modulo–4 Adder. The operation table for a modulo–4 adder and the encoding of residue codes in binary codes are shown below.

⊕	0	1	2	3
0	0	1	2	3
1	1	2	3	0
2	2	3	0	1
3	3	0	1	2

Encoding in Binary

Residue	Binary	
	A	B
0	0	0
1	0	1
2	1	0
3	1	1

With binary encoding, the complete output table for the sum of modulo–4 adder is shown below.

ADDEND		AUGEND		SUM	
A	B	C	D	C_1	C_0
0	0	0	0	0	0
0	0	0	1	0	1
0	0	1	0	1	0
0	0	1	1	1	1
0	1	0	0	0	1
0	1	0	1	1	0
0	1	1	0	1	1
0	1	1	1	0	0
1	0	0	0	1	0
1	0	0	1	1	1
1	0	1	0	0	0
1	0	1	1	0	1
1	1	0	0	1	1
1	1	0	1	0	0
1	1	1	0	0	1
1	1	1	1	1	0

The computer (or LOGMIN) solutions are:

For C_0 output : $C_0 = \overline{B}\,D + B\,\overline{D}$ at 4 c.u.

For C_1 output : $C_1 = \overline{A}\,C\,\overline{D} + \overline{A}\,\overline{B}\,C + \overline{A}\,B\,\overline{C}\,D + A\,\overline{C}\,\overline{D} + A\,\overline{B}\,\overline{C} + A\,B\,C\,D$

or

$C_1 = BD(\overline{A}\,\overline{C} + AC) + (\overline{B} + \overline{D})(\overline{A}C + A\overline{C})$

at 20 c.u.

REFERENCES

1 Carroll, C.C., G.E. Jordan (1968) ; A Fast Algorithm for Boolean Function Minimization, Technical Report, No.AU-T-3, Project THEMIS, Auburn University, Auburn, Alabama, December 1968.

2 Rhyne, V.T. (1973) ; Fundamentals of Digital Systems Design, New Jersey: Prentice Hall, 1973.

3 Bowman, R.M., E.S. McVey (1970) ;"A Method for the Fast Approximation Solution of Large Implicant Charts", IEEE Trans. on Computers, Vol.C-19 No.2, February 1970, pp.167-173.

4 Petrick, S.R. ; A Direct Determination of the Irredundant Forms of a Boolean Function from the Set of Prime Implicants; Technical Report No.56-110, A.F. Cambridge Research Centre, Bedford, Mass., April 1956.

COMPUTER OUTPUTS & LISTINGS

Computer outputs for the above examples and source listing of LOGMIN are shown below.

```
INPUT DATA IN ARRAY-D FOR **SOP** MINIMIZATION

 1 =  1      2 =  0      3 = -1      4 =  1      5 =  1
 6 =  0      7 =  1      8 = -1      9 = -1     10 =  0
11 =  0     12 =  0     13 = -1     14 =  0     15 = -1
16 =  1     17 =  0     18 = -1     19 =  0     20 =  0
21 =  0     22 =  1     23 =  1     24 =  1     25 = -1
26 =  1     27 =  0     28 =  0     29 =  1     30 =  0
31 =  0     32 = -1
```

```
*** BOOLEAN MINIMIZATION PROGRAM ***

THIS FUNCTION CONTAINS    5 VARIABLES

A LISTING OF THE INPUT DATA FOLLOWS
TRUE MINTERMS = 1
FALSE MINTERMS = 0
REDUNDANT MINTERMS (DONT CARES) = -1

 0 =  1,     1 =  0,     2 = -1,     3 =  1,     4 =  1,
 5 =  0,     6 =  1,     7 =  1,     8 = -1,     9 =  0,
10 =  0,    11 =  0,    12 = -1,    13 =  0,    14 = -1,
15 =  1,    16 =  1,    17 = -1,    18 =  0,    19 =  0,
20 =  0,    21 =  1,    22 =  1,    23 =  1,    24 = -1,
25 =  0,    26 =  1,    27 =  0,    28 =  1,    29 =  0,
30 =  0,    31 = -1,
```

THE FOLLOWING IS A LIST OF THE PRIME IMPLICANTS OF THE MINIMIZED FUNCTION.

ESSENTIAL PRIME IMPLICANTS ARE SC LABELED, AND PRIME IMPLICANTS SELECTED FROM A CYCLIC CHART ARE LABELED AS CHOSEN.

NO.	COST	PRIME IMPLICANTS A	B	C	D	E	
1	3	X	0	X	0	0	CHOSEN
2	3	0	0	X	X	0	
3	3	0	0	X	1	X	ESSENTIAL
4	3	0	X	1	X	0	
5	3	0	X	1	1	X	
6	3	X	X	1	1	0	CHOSEN
7	3	X	1	X	0	1	ESSENTIAL
8	4	1	0	X	0	1	CHOSEN
9	4	1	0	1	X	1	
10	4	1	1	0	X	0	ESSENTIAL

X INDICATES A MISSING VARIABLE,0 INDICATES A COMPLEMENTED VARIABLE AND 1 INDICATES A TRUE VARIABLE.
THE FUNCTION IS REPRESENTED BY THE SUM OF BOTH THE ESSENTIAL AND THE CHOSEN PRIME IMPLICANTS.

CONSTRAINT TABLE

COVERED MINTERM	COVERING PRIME IMPLICANTS		
0	1	2	4
4	5	6	
15	6	8	
21	3	9	
23	5	9	

```
16.01.56.JOB(UMEE021,J1)
16.01.55.P.KAMLAI S.E.MEP*.
16.01.58.PASSWOR
16.01.58.COMMENT.   VERSION ONE OF **LOGMIN**
16.01.58.COMMENT.     (**X, SIX VARIABLES)
16.01.59.COMMENT.
16.01.59.COMMENT.   EXAMPLE ONE  (**SOP** ONLY)
16.01.59.PURGE(LOGMIN1)
16.02.01.FTN(OPT=2,R=1)
16.12.19.          44.218 CP SECONDS COMPILATION TIME
16.02.19.CATALOG(LGO)
16.02.19.   CATALOG GIVES THE CONTENT OF A FILE.
16.02.19.   CATALOG COMPLETE.
16.02.19.REMIND(LGO)
16.02.19.SAVE(LGO=LOGMIN1)
16.02.21.LGO.
16.02.24.STOP
16.02.25.CP          4.811 SEC.
16.02.25.CM          0.032 KWU.
16.02.25.MS          0.567 KCS.
16.02.25.UN          3.084 UNITS
16.03.24.LP          0.615 KLN.
```

Example 1 : Minimization of Function

$$F(A,B,C,D,E) = \sum (0,3,4,6,7,15,21,23,26,28, X_2, X_8, X_{12}, X_{14}, X_{17}, X_{24}, X_{31})$$

```
INPUT DATA IN ARRAY-D FOR **SOP** MINIMIZATION

 1 =  0      2 =  1      3 =  0      4 =  0      5 =  1
 6 =  0      7 =  1      8 =  1      9 =  1     10 =  1
11 =  1     12 =  1     13 =  0     14 =  0     15 =  0
16 =  1
```

```
*** BOOLEAN MINIMIZATION PROGRAM ***

THIS FUNCTION CONTAINS    4 VARIABLES

A LISTING OF THE INPUT DATA FOLLOWS
TRUE MINTERMS = 1
FALSE MINTERMS = 0
REDUNDANT MINTERMS (DONT CARES) = -1

 0 =  0,     1 =  1,     2 =  0,     3 =  0,     4 =  1,
 5 =  0,     6 =  0,     7 =  1,     8 =  1,     9 =  1,
10 =  1,    11 =  1,    12 =  0,    13 =  0,    14 =  0,
15 =  1,
```

THE FOLLOWING IS A LIST OF THE PRIME IMPLICANTS OF THE MINIMIZED FUNCTION.

ESSENTIAL PRIME IMPLICANTS ARE SC LABELED, AND PRIME IMPLICANTS SELECTED FROM A CYCLIC CHART ARE LABELED AS CHOSEN.

NO.	COST	PRIME IMPLICANTS A	B	C	D	
1	3	X	0	0	1	ESSENTIAL
2	3	-	1	X	0	ESSENTIAL
3	3	0	1	1	X	
4	3	X	1	1	1	CHOSEN
5	2	1	0	X	X	ESSENTIAL
6	3	1	X	1	1	

X INDICATES A MISSING VARIABLE,0 INDICATES A COMPLEMENTED VARIABLE AND 1 INDICATES A TRUE VARIABLE.
THE FUNCTION IS REPRESENTED BY THE SUM OF BOTH THE ESSENTIAL AND THE CHOSEN PRIME IMPLICANTS.

CONSTRAINT TABLE

COVERED MINTERM	COVERING PRIME IMPLICANTS	
7	3	4
15	4	6

```
** MODIFIED DATA IN ARRAY-C FOR **POS** MINIMIZATIONTION **

 1 =  1      2 =  0      3 =  1      4 =  1      5 =  0
 6 =  1      7 =  0      8 =  0      9 =  0     10 =  0
11 =  0     12 =  0     13 =  1     14 =  1     15 =  1
16 =  0
```

```
*** BOOLEAN MINIMIZATION PROGRAM ***

THIS FUNCTION CONTAINS    4 VARIABLES

A LISTING OF THE INPUT DATA FOLLOWS
TRUE MINTERMS = 1
FALSE MINTERMS = 0
REDUNDANT MINTERMS (DONT CARES) = -1

 0 =  1,     1 =  0,     2 =  1,     3 =  1,     4 =  0,
 5 =  1,     6 =  0,     7 =  0,     8 =  0,     9 =  0,
10 =  0,    11 =  0,    12 =  1,    13 =  1,    14 =  1,
15 =  0,
```

THE FOLLOWING IS A LIST OF THE PRIME IMPLICANTS OF THE MINIMIZED FUNCTION.

ESSENTIAL PRIME IMPLICANTS ARE SC LABELED, AND PRIME IMPLICANTS SELECTED FROM A CYCLIC CHART ARE LABELED AS CHOSEN.

NO.	COST	PRIME IMPLICANTS A	B	C	D	
1	3	0	0	X	0	ESSENTIAL
2	3	0	-	1	X	ESSENTIAL
3	3	X	1	0	1	ESSENTIAL
4	3	1	1	X	0	ESSENTIAL
5	3	1	1	0	X	

X INDICATES A MISSING VARIABLE,0 INDICATES A COMPLEMENTED VARIABLE AND 1 INDICATES A TRUE VARIABLE.
THE FUNCTION IS REPRESENTED BY THE SUM OF BOTH THE ESSENTIAL AND THE CHOSEN PRIME IMPLICANTS.

Example 2 : Minimization of Function

$$F(A,B,C,D) = \sum (1,4,5,7,8,9,10,15)$$

INPUT DATA IN ARRAY-D FOR **SOP** MINIMIZATION

```
 1 =  1      2 =  0      3 = -1      4 =  1      5 =  1
 6 =  0      7 =  1      8 = -1      9 = -1     10 =  0
11 =  0     12 =  0     13 = -1     14 =  0     15 = -1
16 =  1     17 =  0     18 = -1     19 =  0     20 =  0
21 =  0     22 =  1     23 =  0     24 =  1     25 = -1
26 =  0     27 =  1     28 =  0     29 =  1     30 =  0
31 =  0     32 = -1
```

*** BOOLEAN MINIMIZATION PROGRAM ***

THIS FUNCTION CONTAINS 5 VARIABLES

A LISTING OF THE INPUT DATA FOLLOWS
TRUE MINTERMS = 1
FALSE MINTERMS = 0
REDUNDANT MINTERMS (DONT CARES) = -1

```
 0 =  1,     1 =  0,     2 = -1,     3 =  1,     4 =  1,
 5 =  0,     6 =  1,     7 =  1,     8 = -1,     9 =  0,
10 =  0,    11 =  1,    12 = -1,    13 =  0,    14 = -1,
15 =  1,    16 =  1,    17 =  0,    18 =  0,    19 =  0,
20 =  0,    21 =  1,    22 =  0,    23 =  1,    24 = -1,
25 =  0,    26 =  1,    27 =  0,    28 =  1,    29 =  0,
30 =  0,    31 = -1,
```

THE FOLLOWING IS A LIST OF THE PRIME IMPLICANTS OF THE MINIMIZED FUNCTION.

ESSENTIAL PRIME IMPLICANTS ARE SO LABELED, AND PRIME IMPLICANTS SELECTED FROM A CYCLIC CHART ARE LABELED AS CHOSEN.

NO.	COST	PRIME IMPLICANTS A	B	C	D	E	
1	3	X	X	0	1	1	CHOSEN
2	3	0	0	X	X	0	
3	3	0	0	X	1	X	ESSENTIAL
4	3	0	X	1	X	0	
5	3	0	X	1	1	X	
6	3	X	X	1	1	1	CHOSEN
7	3	X	1	X	0	0	ESSENTIAL
8	4	1	0	X	0	1	CHOSEN
9	4	1	0	1	X	1	
10	4	1	0	0	X	0	ESSENTIAL

X INDICATES A MISSING VARIABLE,0 INDICATES A COMPLEMENTED VARIABLE AND 1 INDICATES A TRUE VARIABLE.
THE FUNCTION IS REPRESENTED BY THE SUM OF BOTH THE ESSENTIAL AND THE CHOSEN PRIME IMPLICANTS.

CONSTRAINT TABLE

COVERED MINTERM	COVERING PRIME IMPLICANTS		
0	1	2	
4	1	2	4
15	5	6	
21	8	9	
23	5	9	

```
16.01.55.JOB(UMEE021,J1)
16.01.55.P.K.M.LAI E.E.DEPT.
16.01.55.PASSWOR
16.01.55.COMMENT.   VERSION ONE OF **LOGMIN**
16.01.55.COMMENT.   (MAX. SIX VARIABLES)
16.01.55.COMMENT.
16.01.55.COMMENT.   EXAMPLE ONE  (**SOP** ONLY)
16.01.55.PURGE(LOGMIN1)
16.02.11.FTN(OPT=2,R=1
16.12.14.     4.219 CP SECONDS COMPILATION TIME
16.02.14.CATALOG(LGO)
16.02.19.   CATALOG GIVES THE CONTENT OF A FILE.
16.02.19.   CATALOG COMPLETE.
16.02.19.REWIND(LGO)
16.02.19.SAVE(LGO=LOGMIN1)
16.02.21.LGO.
16.02.24.STOP
16.02.25.CP          4.833 SEC.
16.02.25.CM          8.032 KWW.
16.02.25.MS          9.567 KW3.
16.02.25.UN          0.086 UNITS
16.03.24.LP          0.615 KLN.
```

Example 1 : Minimization of Function

$$F(A,B,C,D,E) = \sum (0,3,4,6,7,15,21,23,26,28,) X_2, X_8, X_{12}, X_{14}, X_{17}, X_{24}, X_{31})$$

INPUT DATA IN ARRAY-D FOR **SOP** MINIMIZATION

```
 1 =  0      2 =  1      3 =  0      4 =  0      5 =  1
 6 =  0      7 =  1      8 =  1      9 =  1     10 =  1
11 =  1     12 =  1     13 =  0     14 =  0     15 =  0
16 =  1
```

*** BOOLEAN MINIMIZATION PROGRAM ***

THIS FUNCTION CONTAINS 4 VARIABLES

A LISTING OF THE INPUT DATA FOLLOWS
TRUE MINTERMS = 1
FALSE MINTERMS = 0
REDUNDANT MINTERMS (DONT CARES) = -1

```
 0 =  0,     1 =  1,     2 =  0,     3 =  0,     4 =  1,
 5 =  0,     6 =  1,     7 =  1,     8 =  1,     9 =  1,
10 =  1,    11 =  1,    12 =  0,    13 =  0,    14 =  0,
15 =  1,
```

THE FOLLOWING IS A LIST OF THE PRIME IMPLICANTS OF THE MINIMIZED FUNCTION.

ESSENTIAL PRIME IMPLICANTS ARE SO LABELED, AND PRIME IMPLICANTS SELECTED FROM A CYCLIC CHART ARE LABELED AS CHOSEN.

NO.	COST	PRIME IMPLICANTS A	B	C	D	
1	3	X	0	0	1	ESSENTIAL
2	3	[illegible]	1	X	[illegible]	ESSENTIAL
3	3	0	1	1	X	
4	3	X	1	1	1	CHOSEN
5	2	1	0	X	X	ESSENTIAL
6	3	1	X	1	1	

X INDICATES A MISSING VARIABLE,0 INDICATES A COMPLEMENTED VARIABLE AND 1 INDICATES A TRUE VARAIBLE.
THE FUNCTION IS REPRESENTED BY THE SUM OF BOTH THE ESSENTIAL AND THE CHOSEN PRIME IMPLICANTS.

CONSTRAINT TABLE

COVERED MINTERM	COVERING PRIME IMPLICANTS	
7	3	4
15	4	6

** MODIFIED DATA IN ARRAY-C FOR **POS** MINIMIZATIONTION **

```
 1 =  1      2 =  0      3 =  1      4 =  1      5 =  0
 6 =  1      7 =  0      8 =  0      9 =  0     10 =  0
11 =  0     12 =  0     13 =  1     14 =  1     15 =  1
16 =  0
```

*** BOOLEAN MINIMIZATION PROGRAM ***

THIS FUNCTION CONTAINS 4 VARIABLES

A LISTING OF THE INPUT DATA FOLLOWS
TRUE MINTERMS = 1
FALSE MINTERMS = 0
REDUNDANT MINTERMS (DONT CARES) = -1

```
 0 =  1,     1 =  0,     2 =  1,     3 =  1,     4 =  0,
 5 =  1,     6 =  0,     7 =  0,     8 =  0,     9 =  0,
10 =  0,    11 =  0,    12 =  1,    13 =  1,    14 =  1,
15 =  0,
```

THE FOLLOWING IS A LIST OF THE PRIME IMPLICANTS OF THE MINIMIZED FUNCTION.

ESSENTIAL PRIME IMPLICANTS ARE SO LABELED, AND PRIME IMPLICANTS SELECTED FROM A CYCLIC CHART ARE LABELED AS CHOSEN.

NO.	COST	PRIME IMPLICANTS A	B	C	D	
1	3	0	0	X	0	ESSENTIAL
2	3	[illegible]	[illegible]	1	X	ESSENTIAL
3	3	X	1	0	1	ESSENTIAL
4	3	1	1	X	0	ESSENTIAL
5	3	1	1	0	X	

X INDICATES A MISSING VARIABLE,0 INDICATES A COMPLEMENTED VARIABLE AND 1 INDICATES A TRUE VARAIBLE.
THE FUNCTION IS REPRESENTED BY THE SUM OF BOTH THE ESSENTIAL AND THE CHOSEN PRIME IMPLICANTS.

Example 2 : Minimization of Function

$$F(A,B,C,D) = \sum (1,4,6,7,8,9,10,15)$$

```
INPUT DATA IN ARRAY-D FOR **SOP** MINIMIZATION

 1 =  1      2 =  0      3 =  0      4 =  1      5 =  0
 6 =  0      7 =  1      8 =  0      9 =  0      10 =  1
 11 = -1     12 = -1     13 = -1     14 = -1     15 = -1
 16 = -1
```

```
** MODIFIED DATA IN ARRAY-C FOR **POS** MINIMIZATIONTION **

 1 =  0      2 =  1      3 =  1      4 =  0      5 =  1
 6 =  1      7 =  0      8 =  1      9 =  1      10 =  0
 11 = -1     12 = -1     13 = -1     14 = -1     15 = -1
 16 = -1
```

```
*** BOOLEAN MINIMIZATION PROGRAM ***

THIS FUNCTION CONTAINS    4 VARIABLES

A LISTING OF THE INPUT DATA FOLLOWS
TRUE MINTERMS = 1
FALSE MINTERMS = 0
REDUNDANT MINTERMS (DONT CARES) = -1

 0 =  1,     1 =  0,     2 =  0,     3 =  1,     4 =  0,
 5 =  0,     6 =  1,     7 =  0,     8 =  0,     9 =  1,
 10 = -1,    11 = -1,    12 = -1,    13 = -1,    14 = -1,
 15 = -1,

THE FOLLOWING IS A LIST OF THE PRIME IMPLICANTS OF THE
MINIMIZED FUNCTION.

    ESSENTIAL PRIME IMPLICANTS ARE SO LABELED, AND
    PRIME IMPLICANTS SELECTED FROM A CYCLIC CHART ARE
    LABELED AS CHOSEN.

 NO.   COST      PRIME IMPLICANTS
                  A   B   C   D
  1      4        0   0   0   0   ESSENTIAL
  2      3        X   0   1   1   ESSENTIAL
  3      3        X   1   1   0   ESSENTIAL
  4      2        1   X   X   1   ESSENTIAL

X INDICATES A MISSING VARIABLE,0 INDICATES A COMPLEMENTED
VARIABLE AND 1 INDICATES A TRUE VARAIBLE.
THE FUNCTION IS REPRESENTED BY THE SUM OF BOTH THE
ESSENTIAL AND THE CHOSEN PRIME IMPLICANTS.
```

```
*** BOOLEAN MINIMIZATION PROGRAM ***

THIS FUNCTION CONTAINS    4 VARIABLES

A LISTING OF THE INPUT DATA FOLLOWS
TRUE MINTERMS = 1
FALSE MINTERMS = 0
REDUNDANT MINTERMS (DONT CARES) = -1

 0 =  0,     1 =  1,     2 =  1,     3 =  0,     4 =  1,
 5 =  1,     6 =  0,     7 =  1,     8 =  1,     9 =  0,
 10 = -1,    11 = -1,    12 = -1,    13 = -1,    14 = -1,
 15 = -1,

THE FOLLOWING IS A LIST OF THE PRIME IMPLICANTS OF THE
MINIMIZED FUNCTION.

    ESSENTIAL PRIME IMPLICANTS ARE SO LABELED, AND
    PRIME IMPLICANTS SELECTED FROM A CYCLIC CHART ARE
    LABELED AS CHOSEN.

 NO.   COST      PRIME IMPLICANTS
                  A   B   C   D
  1      3        0   X   0   1   ESSENTIAL
  2      3        X   0   1   0   ESSENTIAL
  3      2        X   1   0   X   ESSENTIAL
  4      2        X   1   X   1   ESSENTIAL
  5      2        1   X   X   0   ESSENTIAL

X INDICATES A MISSING VARIABLE,0 INDICATES A COMPLEMENTED
VARIABLE AND 1 INDICATES A TRUE VARAIELE.
THE FUNCTION IS REPRESENTED BY THE SUM OF BOTH THE
ESSENTIAL AND THE CHOSEN PRIME IMPLICANTS.
```

Example 3 : Minimization for "Dectecting 8421 BCD codes Divisible by 3"

$$F(A,B,C,D) = \sum (0,3,6,9, X_{10}, X_{11}, X_{12}, X_{13}, X_{14}, X_{15})$$

```
INPUT DATA IN ARRAY-D FOR **SOP** MINIMIZATION

 1 =  0      2 =  1      3 =  1      4 =  0      5 =  1
 6 =  0      7 =  0      8 =  1
```

```
** MODIFIED DATA IN ARRAY-C FOR **POS** MINIMIZATIONTION **

 1 =  1      2 =  0      3 =  0      4 =  1      5 =  0
 6 =  1      7 =  1      8 =  0
```

```
*** BOOLEAN MINIMIZATION PROGRAM ***

THIS FUNCTION CONTAINS    3 VARIABLES

A LISTING OF THE INPUT DATA FOLLOWS
TRUE MINTERMS = 1
FALSE MINTERMS = 0
REDUNDANT MINTERMS (DONT CARES) = -1

 0 =  0,     1 =  1,     2 =  1,     3 =  0,     4 =  1,
 5 =  0,     6 =  0,     7 =  1,

THE FOLLOWING IS A LIST OF THE PRIME IMPLICANTS OF THE
MINIMIZED FUNCTION.

    ESSENTIAL PRIME IMPLICANTS ARE SO LABELED, AND
    PRIME IMPLICANTS SELECTED FROM A CYCLIC CHART ARE
    LABELED AS CHOSEN.

 NO.   COST      PRIME IMPLICANTS
                  A   B   C
  1      3        0   0   1   ESSENTIAL
  2      3        X   1   0   ESSENTIAL
  3      3        1   X   0   ESSENTIAL
  4      3        1   1   1   ESSENTIAL

X INDICATES A MISSING VARIABLE,0 INDICATES A COMPLEMENTED
VARIABLE AND 1 INDICATES A TRUE VARAIELE.
THE FUNCTION IS REPRESENTED BY THE SUM OF BOTH THE
ESSENTIAL AND THE CHOSEN PRIME IMPLICANTS.
```

```
*** BOOLEAN MINIMIZATION PROGRAM ***

THIS FUNCTION CONTAINS    3 VARIABLES

A LISTING OF THE INPUT DATA FOLLOWS
TRUE MINTERMS = 1
FALSE MINTERMS = 0
REDUNDANT MINTERMS (DONT CARES) = -1

 0 =  1,     1 =  0,     2 =  0,     3 =  1,     4 =  0,
 5 =  1,     6 =  1,     7 =  0,

THE FOLLOWING IS A LIST OF THE PRIME IMPLICANTS OF THE
MINIMIZED FUNCTION.

    ESSENTIAL PRIME IMPLICANTS ARE SO LABELED, AND
    PRIME IMPLICANTS SELECTED FROM A CYCLIC CHART ARE
    LABELED AS CHOSEN.

 NO.   COST      PRIME IMPLICANTS
                  A   B   C
  1      3        0   1   0   ESSENTIAL
  2      3        0   1   1   ESSENTIAL
  3      3        1   0   1   ESSENTIAL
  4      3        1   1   0   ESSENTIAL

X INDICATES A MISSING VARIABLE,0 INDICATES A COMPLEMENTED
VARIABLE AND 1 INDICATES A TRUE VARAIBLE.
THE FUNCTION IS REPRESENTED BY THE SUM OF BOTH THE
ESSENTIAL AND THE CHOSEN PRIME IMPLICANTS.
```

Example 4 : Full Adder (Only The Sum Output is Considered)
-- note the "cyclic result"

```
 INPUT DATA IN ARRAY-D FOR **SOP** MINIMIZATION

 1 = 0      2 = 1      3 = 0      4 = 1      5 = 1
 6 = 0      7 = 1      8 = 0      9 = 0     10 = 1
11 = 0     12 = 1     13 = 1     14 = 0     15 = 1
16 = 0

*** BOOLEAN MINIMIZATION PROGRAM ***

THIS FUNCTION CONTAINS    4 VARIABLES

A LISTING OF THE INPUT DATA FOLLOWS
TRUE MINTERMS = 1
FALSE MINTERMS = 0
REDUNDANT MINTERMS (DONT CARES) = -1

 0 = 0,     1 = 1,     2 = 0,     3 = 1,     4 = 1,
 5 = 0,     6 = 1,     7 = 0,     8 = 0,     9 = 1,
10 = 0,    11 = 1,    12 = 1,    13 = 0,    14 = 1,
15 = 0,

THE FOLLOWING IS A LIST OF THE PRIME IMPLICANTS OF THE
MINIMIZED FUNCTION.

     ESSENTIAL PRIME IMPLICANTS ARE SO LABELED, AND
     PRIME IMPLICANTS SELECTED FROM A CYCLIC CHART ARE
     LABELED AS CHOSEN.

  NO.   COST      PRIME IMPLICANTS
                   A   B   C   D
   1     2         X   0   X   1   ESSENTIAL
   2     2         X   1   X   0   ESSENTIAL

X INDICATES A MISSING VARIABLE,0 INDICATES A COMPLEMENTED
VARIABLE AND 1 INDICATES A TRUE VARAIBLE.
THE FUNCTION IS REPRESENTED BY THE SUM OF BOTH THE
ESSENTIAL AND THE CHOSEN PRIME IMPLICANTS.
```

Example 5(a) : Modulo-four Adder (Sum output = C_o)

```
 INPUT DATA IN ARRAY-D FOR **SOP** MINIMIZATION

 1 = 0      2 = 0      3 = 1      4 = 1      5 = 0
 6 = 1      7 = 1      8 = 0      9 = 1     10 = 1
11 = 0     12 = 0     13 = 1     14 = 0     15 = 0
16 = 1

*** BOOLEAN MINIMIZATION PROGRAM ***

THIS FUNCTION CONTAINS    4 VARIABLES

A LISTING OF THE INPUT DATA FOLLOWS
TRUE MINTERMS = 1
FALSE MINTERMS = 0
REDUNDANT MINTERMS (DONT CARES) = -1

 0 = 0,     1 = 0,     2 = 1,     3 = 1,     4 = 0,
 5 = 1,     6 = 1,     7 = 0,     8 = 1,     9 = 1,
10 = 1,    11 = 0,    12 = 1,    13 = 0,    14 = 0,
15 = 1,

THE FOLLOWING IS A LIST OF THE PRIME IMPLICANTS OF THE
MINIMIZED FUNCTION.

     ESSENTIAL PRIME IMPLICANTS ARE SO LABELED, AND
     PRIME IMPLICANTS SELECTED FROM A CYCLIC CHART ARE
     LABELED AS CHOSEN.

  NO.   COST      PRIME IMPLICANTS
                   A   B   C   D
   1     3         0   X   1   0   ESSENTIAL
   2     3         0   1   1   X   ESSENTIAL
   3     4         X   0   0   1   ESSENTIAL
   4     3         1   X   0   1   ESSENTIAL
   5     3         1   0   0   X   ESSENTIAL
   6     4         1   1   1   1   ESSENTIAL

X INDICATES A MISSING VARIABLE,0 INDICATES A COMPLEMENTED
VARIABLE AND 1 INDICATES A TRUE VARAIBLE.
THE FUNCTION IS REPRESENTED BY THE SUM OF BOTH THE
ESSENTIAL AND THE CHOSEN PRIME IMPLICANTS.
```

Example 5(b) : Modulo-four Adder (Sum output = C_1)

```
LOGMIN                          CDC 6400 FTN V3.0-P340 OPT=2  74/04/24. 16.32.00.     PAGE     1
      PROGRAM LOGMIN(INPUT,OUTPUT,TAPE5=INPUT,TAPE6=OUTPUT)
C
C     LOGIC MINIMIZATION BASED ON QUINE-MC CLUSKEY METHOD
C       **NOTE**
C                 THIS VERSION IS DIMENSIONED FOR
C                 A MAXIMUM OF SIX VARIABLES
C
      INTEGER C(64),D(64),OPT
C
C     ****  TO ENSURE PRINTING 8 LINES/INCH USE FORMAT 5000  ****
C     ****  FOR 6 LINES/INCH USE FORMAT 5001                 ****
      WRITE(6,5000)
 5000 FORMAT(1HH)
 5001 FORMAT(1HS)
C
C     READ INPUT DATA .... 'N' AND 'ARRAY-C'
      READ (5,1000) N,OPT
 1000 FORMAT(I2,6X,I1)
      I=0
   10 I=I+1
      READ (5,1001) C(I),NCARD
      D(I)=C(I)
 1001 FORMAT(I2,3X,I1)
      IF(NCARD.EQ.0) GO TO 10
      WRITE(6,5000)
 3000 FORMAT(1H1,//10X,'INPUT DATA IN ARRAY-D FOR **SOP** MINIMIZATION')
      WRITE(6,3001) ((J,C(J)),J=1,I)
 3001 FORMAT(1X,// 5(I5,3H = ,I2,3X),1X)
      WRITE (6,3002)
 3002 FORMAT('-----------------------------------------------',
     1       '-------------------------------')
C
C     PERFORM  ** SOP **  MINIMIZATION
C
      CALL MINTE (N,0)
C
C     TEST OPTION SECTION (I.E.  **SOP** OR ** SOP AND POS **)
      IF (OPT.EQ.0) GO TO 20
C
C     PERFORM  ** POS **  MINIMIZATION
C
C     *** INTERCHANGE  1'S  AND  0'S  IN ARRAY-C  ***
C
      DO 31 J=1,I
      IF(C(J)) 30,31,32
   31 C(J)=1
      GO TO 31
   32 C(J)=0
   31 CONTINUE
      WRITE (6,4000)
 4000 FORMAT(1H1,//12X,'*** MODIFIED DATA IN ARRAY-C FOR **POS** MINI',
     1       'MIZATION ***'//)
      WRITE(6,3001)  ((J,C(J)),J=1,I)
      WRITE (6,3002)
C
C     INTERCHANGE COMPLETED, PERFORM **POS** MINIMIZATION
C
      CALL MINTE (N,0)
   20 CONTINUE
      STOP
      END
```

```
      C
            SUBROUTINE MINTS(N,C)
            REAL TX(1024),TY(512)
            INTEGER X(10),X1,X2,X3,DIF,XE(3),XC(3),Y(10)
            INTEGER C(64),MINT(64),NUM(64)
            INTEGER PRIMI(256),PRIMJ(256),COST(256),E(256)
            INTEGER TABI(520),TABLE(6250)
            COMMON TABLE
            DATA X1/4H X  /
            DATA X2/4H 1  /
            DATA X3/4H 2  /
            DATA XE/4HESSE,4HNTIA,4HL   /
            DATA XC/4HCHOS,4HEN   ,4H    /
            DATA Y/4H A   ,4H B   ,4H C   ,4H D   ,4H E   ,4H F   ,4H G   ,
           1        4H H   ,4H I   ,4H J   /
            A2=ALOG(2.0)
            NT=2**N
            MK=F
            M=0
            DO 125 I=1,NT
            TABI(I)=1
            IF(C(I).EQ.0) GO TO 125
            M=M+1
            MINT(M)=I-1
        125 CONTINUE
      C     ****  SKIP 3 LINES  ****
            WRITE(6,4000)
            WRITE(6,995) N
            MM=F
            DO 3 I=1,NT
          3 NUM(I)=I-1
            WRITE(6,7?0) (NUM(I),C(I),I=1,NT)
      C
      C     SOLVE FOR PRIME IMPLICANTS
      C
            DO 4 I=1,M
            II=MINT(I)
            M1=M-I+1
            DO 5 J=1,M1
            NN=0
            JJ=MINT(M-J+1)
            IF (IAND(II,JJ).NE.II) GO TO 5
            ORET=1.0
            DIF=JJ-II
            IF(C(II+1).GT.01 NN=NN+1
            IF (C(JJ+1) .GT.1)  NN=NN+1
            IF(DIF-EQ.0) GO TO 9
            ICNT=1
            NUM(1)=II
            DO 6 K=1,N
            TZ=TAND(2**(K-1),DIF)
            IF(TZ.EQ.0.0) GO TO 6
            IC2=TCNT
            DO 8 L=1,IC2
            ICNT=ICNT+1
            NUM(ICNT)=NUM(L)+IZ
            LL=NUM(ICNT)+1
            IF(C(LL))8,5,30
         30 NN=NN+1
          8 CONTINUE
          6 CONTINUE
            CIST=ICNT+1
          9 IF(NN.EQ.0) GO TO 5
            IF(MP.EQ.0) GO TO 10
```



```
            DO 11 L=1,MP
            IF(PRIMJ(L).LT.JJ) GO TO 11
            IF(IAND(PRIMJ(L),JJ).NE.JJ) GO TO 11
            IF(IAND(PRIMI(L),II).EQ.PRIMI(L)) GO TO 5
         11 CONTINUE
         10 MP=MP+1
            PRIMI(MP)=II
            PRIMJ(MP)=JJ
            T(MP)=0
            ICOST=ALOG(CIST)/A2
            COST(MP)=N-ICOST
          4 CONTINUE
            IF(C(II+1).LT.0) GO TO 4
            TABI(II)=MK+1
            DO 13 L=1,MP
            IF(PRIMJ(L).LT.II) GO TO 13
            IF(IAND(PRIMI(L),II).NE.PRIMI(L)) GO TO 13
            IF(IAND(PRIMJ(L),II).NE.II) GO TO 13
            MK=MK+1
            TABLE(MK)=L
         13 CONTINUE
            TABI(II+1)=MK+1
            IC=MK-TABI(II)
            IF(IC.EQ.0) GO TO 15
            A=IC
            TX(I)=1./A
            GO TO 4
      C
      C     SET ESSENTIAL TERM
      C
         15 J=TABLE(TABI(I))
            T(J)=-1
            NUM(1)=PRIMI(J)
            LL=NUM(1)+1
            C(LL)=-1
            ICNT=1
            DIF=PRIMJ(J)-PRIMI(J)
            DO 15 K=1,N
            IZ=IAND(2**(K-1),DIF)
            IF(IZ.EQ.0) GO TO 16
            IC2=ICNT
            DO 17 L=1,IC2
            ICNT=ICNT+1
            NUM(ICNT)=NUM(L)+IZ
            LL=NUM(ICNT)+1
         17 C(LL)=-1
         15 CONTINUE
          4 CONTINUE
      C
      C     ALL PRIME IMPLICANTS ARE FOUND -- CK FOR CYCLIC CHART
      C
            LL=0
            DO 27 L=1,M
            IF(C(MINT(L)+1)) 27,27,25
         25 LL=LL+1
            NUM(LL)=L
         27 CONTINUE
            IF(LL.EQ.0) GO TO 31
```

```
C
C   SOLVE CYCLIC CHART
      RTY=0.0
      DO 66 I=1,LL
      IF(TY.GE.TX(NUM(I))) GO TO 28
      RTX=TX(NUM(I))
      II=NUM(I)
   28 CONTINUE
   48 DO 49 I=1,MP
   49 TY(I)=0.0
      DO 50 I=1,LL
      I=TAB(NUM(I))
      JT=TAB(NUM(I)+1)-1
      DO 55 J=IT,JT
      TY(TABLE(J))=TY(TABLE(J))+TX(NUM(I))
   55 CONTINUE
   56 RTY=0.0
      IT=TAB(II)
      JT=TAB(II+1)-1
      DO 57 J=IT,JT
      IF(TY(TABLE(J))-ETY) 60,56,57
   56 IF(RTY.EQ.0.0) GO TO 60
      IF(COST(JJ).LE.COST(TABLE(J))) GO TO 60
   57 RTY=TY(TABLE(J))
      JJ=TABLE(J)
   60 CONTINUE
C
C   CHOOSE PRIME IMPLICANT
C
      Y(JJ)=1
      RTX=0.0
      DO 65 I=1,LL
      IJ=MINT(NUM(I))
      IF (PRIM(JJ).LT.IJ) GO TO 61
      IF(IAND(PRIM(JJ),IJ).NE.IJ) GO TO 61
      IF(IAND(PRIM(JJ),IJ).NE.PRIM(JJ)) GO TO 61
      TX(NUM(I))=1.0
   61 IF(RTX.GE.TX(NUM(I))) GO TO 65
      RTX=TX(NUM(I))
      II=NUM(I)
   65 CONTINUE
      IF(RTX.NE.0.0) GO TO 48
   71 CONTINUE
C
C   WRITE OUTPUT
C
C ****  SKIP 3 LINES  ****
      WRITE(6,4000)
      WRITE(6,1009) (Y(K),K=1,N)
      DO 45 J=1,MP
      II=PRIM(J)
      K1=PRIM(J)-I3
      DO 35 K=1,N
      K1=K-1
      X(N-K1)=X2
      IF(IAND(I3,2**K1).NE.0) X(N-K1)=X2
      IF(IAND(K3,2**K1).NE.0) X(N-K1)=X1
   35 CONTINUE
      IF (Y(J)) 38,37,38
   37 WRITE(6,1003) J,COST(J),(X(I),I=1,N)
      GO TO 40
   38 WRITE(6,1001) J,COST(J),(X(I),I=1,N),XE
      GO TO 40
   39 WRITE(6,1003) J,COST(J),(X(I),I=1,N),XC
   40 CONTINUE
      WRITE(6,2001)
      IF(LL.EQ.0)  RETURN
C       SKIP 5 LINES
      WRITE(6,4000)
      WRITE(6,1004)
      DO 70 I=1,M
      IT=TAB(I)
      JT=TAB(I+1)-1
      IF(MINT(I)+1).GT.0)  WRITE(6,1002) MINT(I),(TABLE(J),
     1    J=IT,JT)
   70 CONTINUE
C
C      FORMATS FOR SUBROUTINE MINTE
C
  999 FORMAT(21X,'*** BOOLEAN MINIMIZATION PROGRAM ***'
     1  /// 16X,'THIS FUNCTION CONTAINS',I9,' VARIABLES'///
     2  16X,'A LISTING OF THE INPUT DATA FOLLOWS',/16X,'IF',
     3  ' THE NUMBERS = 1',/ 16X,'FALSE MINTERMS = 0',// 16X,
     4  'PROGRAMMER (UNIVERSITY OF CAPE) = -I',/)
  700 FORMAT(5(1X,I5,3H = ,I2,7H,  ))
 1000 FORMAT(6X,'THE FOLLOWING IS A LIST OF THE PRIME ',
     1  'IMPLICANTS OF THE',/ 6X,'MINIMIZED FUNCTION.',//
     2  10X,'ESSENTIAL PRIME IMPLICANTS ARE SO LABELED, AND',
     3  /,10X,'PRIME IMPLICANTS SELECTED FROM A CYCLIC CHA',
     4  'RT ARE',/ 10X,'LABELED AS CHOSEN.',/// 6X'   NO.',
     5  2X,'COST',7X,'PRIME IMPLICANTS',/
     6  22X,10(4H----),/)
 1003 FORMAT(1H ,5X,I4,2X,I4,6X,13A4)
 2001 FORMAT(1H ,/ 6X,'X INDICATES A MISSING VARIABLE,0 IN',
     1  'DICATES A COMPLEMENTED',/ 6X,'VARIABLE AND 1 INDIC',
     2  'ATES A TRUE VARIABLE.',/ 6X,'THE FUNCTION IS REPRE',
     3  'SENTED BY THE SUM OF BOTH THE',/ 6X,'ESSENTIAL AND',
     4  ' THE CHOSEN PRIME IMPLICANTS.')
 1004 FORMAT(10X,'CONSTRAINT TABLE',/// 8X,'COV',
     1  'ERED BY',6X,'COVERING PRIMES',/ 8X,'MINTERMS',6X,'IMPLI',
     2  'CANTS')
 1002 FORMAT(1H ,I9,I5,I16,15I6,9(/,7X,15I6))
 4000 FORMAT(1X,///)
```

```
      FUNCTION IAND(I1,I2)
      IAND=I1.AND.I2
      RETURN
      END
```

Appendix 3 :

A NOTE ON THE BACKUS-NAUER FORM (BNF) EQUATION FOR SYNTAX DESCRIPTION

A Note on the Backus-Nauer Form (BNF) Equation for Syntax Description

The Backus-Nauer Form (BNF) is a metalanguage due to Backus & Nauer whom pioneered the practical description of Computer languages. The symbols & conventions of BNF may be summarized as follows (see below for examples):

1) Certain symbols have special meanings in the metalanguage:
   a. → means "is defined as."
   b. <> is used to delimit a string that is the name of another string.
   c. | means "or else," i.e., separates alternatives.

2) All symbols except those above denote themselves.

3) Symbols or strings or names in any combination that follow in succession implicitly specify the operation of concatenation.

4) The order of application of the operations in the metalanguage is first concatenations, then alternatives, then the definition arrow.

5) Unless otherwise stated, the syntax equations shall be applied to strings in left-to-right order.

Example 1:

(1) <DIG> → 0|1|2|3|4|5|6|7|8|9

May be read "the object named DIG is defined as the string 0 or else 1 or else 2 ..... or else 9."

Example 2: Consider Eq. (1) along with the following equations for the syntax of integers, fractions, and numbers:

(2) <INT> → <DIG> | <DIG><DIG> | <DIG><DIG><DIG>

(3) <FR> → .<INT>

(4) <NUM> → <INT> | <FR> | <INT><FR>

Using the five rules of the metalanguage, we can interpret these four BNF equations as a grammar for recognizing 1 - or 2-digit integers as INT or up to 2 digits following a period as FR, or any instance of INT or of FR or of INT followed by FR as (numbers) NUM.

Each alternative term of a syntax equation is called a production. The problem of recognition can then be described as a sequence of searches of the productions for matches to the string to be translated, both directly and by means of intermediate named classifications that are matched to productions and then used to form later matches. We shall see examples of this presently.

It often happens that strings of indefinite length are of interest. For this purpose, the metalanguage permits recursive statements as illustrated by

(5) <INT> → <DIG> | <INT> <DIG>

This equation is recursive because it defines INT and INT also appears on the right side of the arrow. To see the power of this kind of equation consider the string

23473658

The first symbol, 2, is found to match an alternative in Eq.(1) so 2 is classified as DIG. Using DIG as an argument in Eq.(5), it is found to be an INT. Consider now the second symbol, 3, which is also first found to be a DIG. The string 23 is at this point then classified as <INT><DIG>. This matches the second alternative of Eq.(5), making the string an INT. This process of using Eqs.(1) and (5) continues and clearly may be used to recognize a string of any number of digits. Also, Eqs.(1),(5),(3), and (4) are then able to recognize numbers of any length.

With the specification of the property of recursion, we have completed the informal definition and illustration of the BNF metalanguage. It is worth noting that although strings of arbitrary length can be handled, there is no provision for simple specification of strings of a definite or of maximum length like "names can be up to eight characters long." There are

	Syntax Equation	Meaning
1	<AOP> → + | -	Add-operator symbol is + or -
2	<MOP> → * | /	Multiply-operator symbol is * or /
3	<V> → A | B | C	A variable symbol is A or B or C
4	<F> → <V> | (<EX>)	A factor is <V> or left paren then EX then right paren
5	<T> → <F> | <T><MOP><F>	A term is <F> or term then <MOP> then <F>
6	<EX> → <T> | <EX><AOP><T>	An expression is <T> or <EX> then <AOP><T>

(a) An arithmetic-expression grammar.

	Successive Classifications*	Equations Used	Compiled Subexpression
1	A+B*(C+A*C)+B	Original expression	-
2	V+V*(V+V*V)+V	3	-
3	V+V*(V+T*F)+V	4,5	-
4	V+V*(V+T)+V	3	T1=A*C
5	V+V*(EX+T)+V	4,5,6	-
6	V+V*(EX)+V	6	T2=C+T1
7	V+V*F +V	4	-
8	V+T*F +V	4,5	-
9	V+T +V	5	T3=B*T2
10	EX+T +V	4,5,6	-
11	EX +V	6	T4=A+T3
12	EX +T	4,5	-
13	EX	6	T5=T4+B

NOTES

1) In this example, AOP is + and MOP is *
2) Enq.(3) of (a) is used to replace all variable symbols by V. for clarity on this chart, angle brackets are omitted.

(b) Example of use of syntax equations

versions of BNF that can conveniently specify properties such as these[1].

Figure A3.1(a) shows a set of syntax equations that define a grammar typical of arithmetic expressions. The expressions may be of any length and include parenthesized subexpressions to any depth. Although names are here restricted to the single letters A, B, or C, it is easy to modify the syntax to permit multiple-letter names. One important property of this particular syntax to permit multiple-letter names. One important property of this particular syntax is not obvious; the usual precedence rule that multiplication (*) is done before addition (+) is built into the syntax. This occurs in the following way: consider the string A+B*C. In the left-to-right scan, the substring A+B will initially be classified, but this classification (as an EX) used with the following string *C will be found not to match any production. The scan must then abandon the tentative classification of A+B as an EX and seek another sequence of classifications which can be found as A is an EX, B*C a T, and finally, the entire expression is an EX followed by + followed by T which is found to match the last alternative of Eq. (6) of the figure. Thus in this expression, B*C is recognized as a legal operation while A+B is not. The chart of part (b) of the figure shows some key steps in the analysis of an example statement.

Using the technique described above, it is possible to determine whether any given string representing a program or a statement is a legal one of a given syntax. Since the string is recognized by a sequence of matches of its substrings with the syntax productions, this process can parse the string, i.e., identify the "parts of speech" such as operators, variables, product and sum terms, indexes, etc. Parsing is a major task of a language-translator program such as a compiler.

[1] see Lee, J.A.N. : "The Anatomy of a Compiler", New York : Reinhold Publishing Corp., 1967.

BNF is a valuable way to start the orderly design of a compiler. However, this first step must be followed by a host of other difficult judgments such as:

1. Does a given set of BNF equations consistently and unambiguously define a grammar?

2. If the grammar is unambiguous, precisely how are the equations to be scanned with the string to be translated? This depends on how the equations are represented.

3. The semantics of the language must somehow be specified so that the compiler can compile object instructions when it recognizes the productions. Fortunately, most semantic rules are rather simple, but the fact remains that there is no instrument as powerful and as widely accepted as BNF to specify semantics. There is also the issue whether the semantics, which usually take the form of a list of object instructions for each kind of production, should be applied just after each recognition or whether pointers to the substrings should be kept until the scan is complete and the object instructions assigned at the end.

4. Syntax errors are naturally detected during the syntax scan. However, in order to keep enough information about errors to best communicate the place where they occur, careful attention is required in the design of the compiler.

5. The many efficiency questions are of two classes: the efficiency of the compilation process itself and the efficiency of the object code produced by the compiler. By efficiency we mean storage space and time.

Appendix 4 :

A COLLECTION OF APL FUNCTIONS FOR DIGITAL FILTER ANALYSIS AND SIMULATION

Note

This suite of programs require plotting routines which are included in the appendix 5

APL functions for Digital Filter analysis and simulation

FIRN	A conversational program for the analysis of an Nth order FIR filter
FCONTINUE	A "house-keeping" function, calling appropriate functions to analysis an Nth order FIR filter
INFIR	Compute the time response of an Nth order FIR filter; called by FCONTINUE
OUTFIRN	Tabulates outputs; function called by INFIR
ELPFIR	Compute the time response of an Even-order FIR Linear Phase digital filter; data input functi
ECON	A "continue funcion" for ELPFIR
EVFIRLP	Compute the time response, called by ECON
OUTFIRN	Tabulates outputs for the EVFIRLP; called by ECON
FILT	Compute the frequency response of an Nth order digital filter; calling functions — DBS, PHS
DBS	Compute the amplitude response; called by FILT and calls function — FUN1, F2
FUN1	Compute the real part of the ratios of two polynominals; called by DBS
F2	Compute the imaginary part of the ratios of polynominals; called by DBS
AR	Auxilary function called by FUN1 and F2
AI	-do-
BR & BI	(comments as AR, AI; "duals" of these)
PHS	Computes the phase response; called by FILT; and calls FUN1 & F2
FILT1	Computes the frequency response of an Nth order filter; a more compact version of FILT. FILT1 calls functions: DBPHS, COSUM, and SINSUM
DBPHS	Computes the amplitude and phase characteristics; called by function FILT1; calling COSUM & SINSUM
COSUM	Computes the "cosine terms"; auxilary function and is called by DBPHS
SINSUM	Computes the "sine terms"; auxilary fucntion and is called by SINSUM

(This collection requires the plotting routines)

```
    ∇ FIRN
[1]   ' '
[2]   'ENTER ORDER OF FIR FILTER:'
[3]   N←□
[4]   'ENTER COEFFICIENTS OF FILTER AS AN ARRAY STARTING WITH A(0):'
[5]   A←□
[6]   'ENTER SIGNAL SAMPLES AS AN ARRAY.  AT END OF LINE, TYPE'
[7]   'CARRAGE RETURN (CR) THEN TYPE.... X← X, (REMAINING'
[8]   'SAMPLES) FOLLOWS BY (CR).  REPEAT THIS AS REQUIRED.'
[9]   'AFTER SIGNAL ENTRY, TYPE... FCONTINUE .. TO RESTART.'
[10]  X←□
[11]  ' '
    ∇
```

```
    ∇FCONTINUE[□]∇

    ∇ FCONTINUE
[1]   Y←INFIR X
[2]   OUTFIRN
    ∇
```

```
    ∇INFIR[□]∇

    ∇ Y←INFIR X;IX;I;J;M
[1]   K←⍴X
[2]   Y←K⍴0
[3]   M←N+1
[4]   IX←M⍴0
[5]   I←0
[6]  KLAB:IX←X[I],¯1↓IX
[7]   J←IY←0
[8]  NLAB:IY←IY+A[J]×IX[J]
[9]   J←J+1
[10]  →(J<M)/NLAB
[11]  Y[I]←IY
[12]  I←I+1
[13]  →(I<K)/KLAB
    ∇
```

```
    ∇OUTFIRN[□]∇

    ∇ OUTFIRN;I
[1]   ' '
[2]   ' '
[3]   'TIME RESPONSE OF';' ';N;'-TH ORDER FIR FILTER:'
[4]   ' '
[5]   '   SAMPLE';'        ';'INPUT SIGNAL';'       ';'OUTPUT SIGNAL'
[6]   ' '
[7]   I←⍳K
[8]   □←I AND X AND Y
    ∇
```

```
      )ORIGIN 0
WAS 1

      FIRN

ENTER ORDER OF FIR FILTER:
□:
      2
ENTER COEFFICIENTS OF FILTER AS AN ARRAY STARTING WITH A(0):
□:
      1 0.5 0.2
ENTER SIGNAL SAMPLES AS AN ARRAY.  AT END OF LINE, TYPE
CARRAGE RETURN (CR) THEN TYPE.... X← X, (REMAINING
SAMPLES) FOLLOWS BY (CR).  REPEAT THIS AS REQUIRED.
AFTER SIGNAL ENTRY, TYPE... FCONTINUE .. TO RESTART.
□:
      1 1 1 1 0 0 0 0

      FCONTINUE


TIME RESPONSE OF 2-TH ORDER FIR FILTER:

   SAMPLE       INPUT SIGNAL       OUTPUT SIGNAL

    0                 1                  1
    1                 1                  1.5
    2                 1                  1.7
    3                 1                  1.7
    4                 0                  0.7
    5                 0                  0.2
    6                 0                  0
    7                 0                  0
```

```
    ∇ ELPFIR
[1]   ' '
[2]   'ENTER ORDER(EVEN) OF ''FIR'' FILTER:'
[3]   N←□
[4]   'ENTER COEFFICIENTS OF_FILTER -- STARTING WITH A(0):'
[5]   A←□
[6]   'ENTER INPUT SIGNAL;END OF LINE, TYPE CARRIAGE'
[7]   'RETURN (CR), THEY TYPE ... X←X,<REMAINING INPUTS>'
[8]   'FOLLOWED BY (CR).  REPEAT AS REQUIRED.'
[9]   'AFTER SIGNAL ENTRY, TYPE ... ECON ... TO RESTART.'
[10]  X←□
    ∇

    ∇ ECON
[1]   Y←EVFIRLP X
[2]   OUTFIRN
    ∇


    ∇ Y←EVFIRLP X;IX;IY;I;J;M
[1]   K←⍴X
[2]   Y←K⍴0
[3]   IX←(N+1)⍴0
[4]   M←N÷2
[5]   I←0
[6]  KLAB:IX←X[I],¯1+IX
[7]   IY←J←0
[8]  NLAB:IY←IY+A[J]×(IX[J]+IX[N-J])
[9]   J←J+1
[10]  →(J<M)/NLAB
[11]  Y[I]←IY+A[M]×IX[M]
[12]  I←I+1
[13]  →(I<K)/KLAB
    ∇


    ∇ OUTFIRN;I
[1]   ' '
[2]   ' '
[3]   'TIME RESPONSE OF';' ';N;'-TH ORDER FIR FILTER:'
[4]   ' '
[5]   '  SAMPLE';'       ';'INPUT SIGNAL';'      ';'OUTPUT SIGNAL'
[6]   ' '
[7]   I←⍳K
[8]   □←I AND X AND Y
    ∇
```

```
      ELPFIR

ENTER ORDER(EVEN) OF 'FIR' FILTER:
⎕:
      4
ENTER COEFFICIENTS OF FILTER -- STARTING WITH A(0):
⎕:
      1 0.5 0.2
ENTER INPUT SIGNAL;END OF LINE, TYPE CARRIAGE
RETURN (CR), THEY TYPE ... X←X,<REMAINING INPUTS>
FOLLOWED BY (CR).  REPEAT AS REQUIRED.
AFTER SIGNAL ENTRY, TYPE ... ECON ... TO RESTART.
⎕:
      1 1 1 1 1 1 1 1 0 0 0 0 0
      ECON


TIME RESPONSE OF 4-TH ORDER FIR FILTER:

  SAMPLE        INPUT SIGNAL       OUTPUT SIGNAL

      0                1                 1
      1                1                 1.5
      2                1                 1.7
      3                1                 2.2
      4                1                 3.2
      5                1                 3.2
      6                1                 3.2
      7                1                 3.2
      8                0                 2.2
      9                0                 1.7
     10                0                 1.5
     11                0                 1
     12                0                 0
```

Note : origin zero is assumed

```
      )GRP FREQGP
FILT    DRS     PHS     FUN1    F2      AR      BR      AI      BI

      ∇FILT[⎕]∇
    ∇ FILT
[1]   'CHECK ORDER OF COEFFICS.'
[2]   'FILTER ORDER = COEF. ORDER - 1'
[3]   ⎕←XX←⍴A
[4]   ⎕←YY←⍴B
[5]   'SPECIFY FILTER ORDER'
[6]   LL←⎕
[7]   L←LL+1
[8]   'NO. OF POINTS IN HALF SAMPLING INTERVAL'
[9]   NN←⎕
[10]  N←2×NN
[11]  'SPECIFY POINTS TO BE EVALUATED'
[12]  I←⎕
[13]  ' '
[14]  ' '
[15]  OUT←⍉((2,(⍴I))⍴(DRS,PHS))
[16]  '  GAIN       PHASE'
[17]  ⎕←OUT
    ∇

      ∇DRS[⎕]∇
    ∇ ZZ←DRS
[1]   ZZ←⍳0
[2]   ZZ←20×10⍟(((FUN1*2)+(F2*2),)*0.5)
    ∇

      ∇PHS[⎕]∇
    ∇ Z←PHS
[1]   Z←⍳0
[2]   Z←(180÷○1)×(¯3○(F2÷FUN1))
    ∇

      ∇FUN1[⎕]∇
    ∇ ZR←FUN1
[1]   ZR←⍳0
[2]   ZR←((AR×BR)+(AI×BI))÷((BR*2)+(BI*2))
    ∇

      ∇F2[⎕]∇
    ∇ ZI←F2
[1]   ZI←⍳0
[2]   ZI←((AI×BR)-(AR×BI))÷((BR*2)+(BI*2))
    ∇

      ∇AR[⎕]∇
    ∇ NR←AR;K
[1]   K←0
[2]   NR←0
[3]   NR←NR+(2○(K×2×○1×I÷N))×A[K+1]
[4]   →3×LL≥K←K+1
    ∇

      ∇BR[⎕]∇
    ∇ DR←BR;K
[1]   K←0
[2]   DR←0
[3]   DR←DR+(2○(K×2×○1×I÷N))×B[K+1]
```

```
      ∇ NI←AI;K
[1]     K←0
[2]     NI←0
[3]     NI←NI+¯1×(10(K×2×01×I+N))×A[K+1]
[4]     →3×LL≥K←K+1
      ∇


      ∇ BI←BI;K
[1]     K←0
[2]     BI←0
[3]     BZ←BI+¯1×(10(K×2×01×I+N))×B[K+1]
[4]     →3×LL≥K←K+1
      ∇

      )GRP PREPLOT
A       B       GAIN    PHASE

      A   THESE ARE VARIABLES

      A
1  5  15  20  15  6  1
      B
33.7972  0  26.284  0  3.8596  0  0.0592

      GAIN   AND   PHASE

  0.000000000E0    0.000000000E0
  2.290839616E¯7   ¯1.087859871E1
  8.641330341E¯7   ¯2.182932490E1
  1.751120504E¯6   ¯3.292671603E1
  2.630158485E¯6   ¯4.425035533E1
  2.969568322E¯6   ¯5.588800910E1
  5.295166529E¯7   ¯6.793966280E1
 ¯1.699963104E¯5   ¯8.052310572E1
 ¯1.108613062E¯4    8.621777764E1
 ¯5.474544472E¯4    7.209982800E1
 ¯2.368222237E¯3    5.687834432E1
 ¯9.335828011E¯3    4.020786381E1
 ¯3.426702472E¯2    2.157329133E1
 ¯1.186365836E¯1    1.830322472E¯1
 ¯3.875334487E¯1   ¯2.510824463E1
 ¯1.163624230E0    ¯5.541414757E1
 ¯3.010216658E0    ¯9.000000000E1
 ¯6.288274322E0     5.541414757E1
 ¯1.068674926E1     2.510824463E1
 ¯1.569471807E1    ¯1.830322472E¯1
 ¯2.104715660E1    ¯2.157329133E1
 ¯2.668533275E1    ¯4.020786381E1
 ¯3.264755730E1    ¯5.687834432E1
 ¯3.902109172E1    ¯7.209982800E1
 ¯4.593319310E1    ¯8.621777764E1
 ¯5.356234332E1     8.052310572E1
 ¯6.216728931E1     6.793966280E1
 ¯7.214578235E1     5.588800910E1
 ¯8.416058302E1     4.425035533E1
 ¯9.945041498E1     3.292671603E1
 ¯1.207922001E2     2.182932491E1
```

```
      )GRP FREQGP1
FILT1   DRPHS   COSUM   SINSUM

      ∇FILT1[□]∇
      ∇ FILT1;XX;YY;FILORDA;N;I
[1]     'CHECK ORDER OF COEFFICIENTS'
[2]     'FILTER ORDER = COEF. ORDER - 1'
[3]     □←XX←⍴A
[4]     □←YY←⍴B
[5]     'SPECIFY FILTER ORDER'
[6]     FILORDA←□
[7]     'NO. OF POINTS IN HALF SAMPLING INTERVAL'
[8]     N←□
[9]     N←2×N
[10]    'SPECIFY POINTS TO BE EVALUATED'
[11]    I←□
[12]    I←⍳I
[13]    DRPHS
      ∇

      ∇DRPHS[□]∇
      ∇ DRPHS;AR;BR;AI;BI;AR1;AI1;DEN;F1;F2;OUT
[1]     ⍝ PRELIMINARY COMPUTATION: (AR×BR),(AI×BI).
[2]     ⍝                          (BR*2), AND (BI*2),ETC.
[3]     AR←COSUM A
[4]     BR←COSUM B
[5]     AI←SINSUM A
[6]     BI←SINSUM B
[7]     AR1←(AR×BR)+(AI×BI)
[8]     AI1←(AI×BR)-(AR×BI)
[9]     DEN←(BR*2)+(BI*2)
[10]    F1←AR1÷DEN
[11]    F2←AI1÷DEN
[12]    ⍝ COMPUTE GAIN,PHASE AND OUTPUT RESULTS
[13]    GAIN←20×10⍟(((F1*2)+(F2*2))*0.5)
[14]    PHASE←(180÷01)×(¯3○(F2÷F1))
[15]    ' '
[16]    '       GAIN                    PHASE'
[17]    ' '
[18]    □←OUT←⍉((2,(⍴GAIN))⍴(GAIN,PHASE))
      ∇


      ∇COSUM[□]∇
      ∇ Z←COSUM A;K;WT
[1]     ⍝ COMPUTE THE SUM OF COSINE TERMS (0-ORIGIN)
[2]     K←Z←0
[3]     WT←2×01×I÷N
[4]     LAB:Z←Z+(2○(WT×K))×A[K]
[5]     →(FILORDA≥K←K+1)/LAB
      ∇


      ∇SINSUM[□]∇
      ∇ Z←SINSUM B;K;WT
[1]     K←Z←0
[2]     WT←2×01×I÷N
[3]     LAB:Z←Z-(1○(WT×K))×B[K]
[4]     →(FILORDA≥K←K+1)/LAB
      ∇
```

```
      )VARS
A       B
      A
1  6  15  20  15  6  1
      B
33.7972  0  26.284  0  3.8596  0  0.0592


      FILT
CHECK ORDER OF COEFFICS.
FILTER ORDER = COEF. ORDER - 1
7
7
SPECIFY FILTER ORDER
□:
      6
NO. OF POINTS IN HALF SAMPLING INTERVAL
□:
      32
SPECIFY POINTS TO BE EVALUATED
□:
      0,¡31

  GAIN       PHASE
  0.000000000E0    0.000000000E0
  2.290839616E¯7   ¯1.087859871E1
  8.641330341E¯7   ¯2.182932490E1
  1.751120504E¯6   ¯3.292671603E1
  2.630158485E¯6   ¯4.425035533E1
  2.969568322E¯6   ¯5.588800910E1
  5.295166529E¯7   ¯6.793966280E1
 ¯1.699963104E¯5   ¯8.052310572E1
 ¯1.108613062E¯4    8.621777764E1
 ¯5.474544472E¯4    7.209982800E1
 ¯2.368222237E¯3    5.687834432E1
 ¯9.335828011E¯3    4.020786381E1
 ¯3.426702472E¯2    2.157329133E1
 ¯1.186365836E¯1    1.830322472E¯1
 ¯3.875334487E¯1   ¯2.510824463E1
 ¯1.163624230E0    ¯5.541414757E1
 ¯3.010216658E0    ¯9.000000000E1
 ¯6.288274322E0     5.541414757E1
 ¯1.068574826E1     2.510824463E1
 ¯1.569471807E1    ¯1.830322472E¯1
 ¯2.104715660E1    ¯2.157329133E1
 ¯2.668533275E1    ¯4.020786381E1
 ¯3.264755730E1    ¯5.687834432E1
 ¯3.902109172E1    ¯7.209982800E1
 ¯4.593319310E1    ¯8.621777764E1
 ¯5.350234332E1     8.052310572E1
 ¯6.216728931E1     6.793966280E1
 ¯7.214578235E1     5.588800910E1
 ¯8.416058302E1     4.425035533E1
 ¯9.945041498E1     3.292671603E1
 ¯1.207922001E2     2.182932491E1
 ¯1.570417299E2     1.087859913E1
```

Appendix 5 :

APL PLOTTING ROUTINES [1]

---

[1] Extracted from : P.K.M.Lai, "A Set of Graph Plotting Functions for the CYBER*APL at Imperial College"; Internal Report, Electrical Engineering Department, Imperial College; September, 1975.

APL Plotting Routines

Two versions of plot routines are available; one these these is specially written for the CYBER*APL.

(A) Multiple plot, dyadic function without explicit result

PLOT

Plots one ore more graphs simultaneously; it permits automatic scaling so that the graphs will fit approximately within the scale dimensions specified by the user. Origin 1 is assumed.

Syntax : scalesize PLOT Numbers

scalesize : may be one ore two numbers. The first number specifies the approximate size of the vertical axis and the second number does the same for the horizontal axis. If only one number is supplied, it is applied to both axis.

AND

Essentially a column catenator; i.e. permits two matrices or a matrix and a vector to be joined.

VS

Essentially a column catenator, similar to AND except that the right hand argument must be of rank smaller or equal to unity.

(AND & VS, may now be superceded by the formating funct ⎕FMT)

```
∇A PLOT B;C;D;F;G;H;I;J;L;T;Y;HZ;NB;VT;PT;ST;ISV;U;HS;PC;SM;SZ
[1]  PC←'o*•∇Δ□'
[2]  HS←0
[3]  SZ←1 2 5
[4]  SM←5 10
[5]  →((0=×/(2ρA),ρB),3 2 1<ρρB)/0,PL7,PL1,PL2
[6]  →PL2,ρB←⍉(2,D)ρ(ιD←ρ,B),B
[7]  PL1:B←B[1;;]
[8]  PL2:Y←1+ι(ρB)[2]-1
[9]  C←(((⌈/⌈/B[;Y])-⌊/⌊/B[;Y]),(⌈/B[;1])-⌊/B[;1]
[10] F←|(2ρA)↑C+(C=0)×B[1;2 1]+B[1;2 1]=0
[11] F←(SZ[+/(⌊0.0001+F×10*-G)∘.≥SZ])×10*G←⌊10⊕F
[12] G←(SM↑F)×⌊((⌊/⌊/B[;Y]),⌊/B[;1])×F↑SM←16⌊SM⌈1 4
[13] B[;1]←⌊0.5+F[2]×B[;1]-G[2]
[14] B[;Y]←⌊0.5+F[1]×B[;Y]-G[1]
[15] H←SM×⌈(((⌈/⌈/B[;Y]),⌈/B[;1])↑SM
[16] NB←G[1]+(SM[1]↑F[1])×0,ιH[1]↑SM[1]
[17] HZ←G[2]+(SM[2]↑F[2])×0,ιH[2]↑SM[2]
[18] 0ρST←6ρ~ISV←1≠U←9
[19] PL3:VT←∨/0>NB←NB×10*U-ST[6-ISV]←I←1+⌈/⌊10⊕|(NB≠0)/NB
[20] PT←⌊1+10|PT-1|PT←1E¯5+(|NB)∘.↑10*¯1+⍉ιU
[21] L←U+1-(⍉((C←ρNB)ρ1)∧.=PT)ι0
[22] →((U>T←VT+⌈/I,(L+L≠I),(I≤0)×2+L-I),ST[2-ISV]←ST[2-ISV]∨L≥U-VT+L>I)
     /3 2+ι26
[23] →(¯1+ι26),ρST[4-ISV]←I←1
[24] →PL3,ρNB←SM[1+~ISV]×¯1+ιC
[25] PT←(-VT+0⌈1-I)⌽PT
[26] PT←(,PT)×J←,⍉(⌽ρPT)ρ(,⍉(1≠PT)∨.∧(ιU)∘.≤ιVT⌈I-1),
     (C×U+1-I←VT+I⌈I≤0)ρ1
[27] →(~VT)/2+ι26
[28] PT[(U+-/(C,U)ρJ)+U×¯1+ιC]←11×NB<0
[29] PT←(~(ιU+J)∈(I+J),ι¯1+J+U-T)\(1 0+C,U)ρPT,Uρ0
[30] PT[ιC;I+J]←12
[31] PT←' 0123456789¯.'[1+PT[;ιU-1]]
[32] →(~ISV)/PL6
[33] L←1,H[2]ρ0×C←H[1]
[34] PL4:L←(L×HS×C≠0)⌈1,H[2]ρ0
[35] L[1+(D≠0)/B[;1]]←(D≠0)/D←(C=B[;Y])⌈.×Y
[36] →(C≠0)/PL5
[37] L←L⌈0=(SM[2]↑2)|0,ιH[2]
[38] PL5:PT[((ρPT)[1],1+C↑SM[1])[1+0=SM[1]|C];],(' |',(ρY)ρPC)[1+L]
[39] →(0≤C←C-1)/PL4
[40] →(U=U↑SM[2]-~ISV←~ρρNB←.HZ)/PL3
[41] PL6:(SM[2]-9)⌽(,(0 0,(U-1)ρ1)\PT),' '
[42] →(ST[1 3 2 4],1)/1 3 5 7 10+ι26
[43] 'ORIGIN AND SCALE FACTOR FOR ORDINATE:';G[1],↑F[1]
[44] →(0=ST[3])/2+ι26
[45] 'SCALE FACTOR FOR ORDINATE:';10*ST[5]-1
[46] →(0=ST[2])/2+ι26
[47] 'ORIGIN AND SCALE FACTOR FOR ABSCISSA:';G[2],↑F[2]
[48] →(0=ST[4])/0
[49] 'SCALE FACTOR FOR ABSCISSA';10*ST[6]-1
[50] →0
[51] PL7:'THE RIGHT ARGUMENT OF PLOT MUST BE OF RANK ≤3.'
∇
```

```
    ∇ L←A AND B;C;D
[1]   →(((2<ρρA)∨3<ρρB),0≠ρρB)/ 17 3
[2]   B←,B
[3]   →(((3=ρρB)∧1≠1ρρB),2=ρρA)/ 17 7
[4]   A←,A
[5]   →(∧/((ρA)≠1,D),1≠D←1ρ¯2ϕρB)/16
[6]   A←(((D×ρA)⌊D⌈ρA),1)ρA
[7]   →(1≠ρρB)/9
[8]   B←(((ρB)⌈(1=ρB)×1ρρA),1)ρB
[9]   →((∧/D≠1,1ρρA),1≠D←1ρ¯2ϕρB)/ 16 11
[10]  B←(((3=ρρB)ρ1),(1ρρA),1ρϕρB)ρB
[11]  →(3=ρρB)/14
[12]  L←(((C←1ρϕρA)ρ0),(1ρϕρB)ρ1)\B
[13]  →0×ρρL[;ιC]←A
[14]  L←(1,((C←1ρϕρA)ρ0),(¯1+1ρϕρB)ρ1)\B
[15]  →0×ρρL[;1+ιC]←A
[16]  →0=ρ⍬←'ARGUMENTS OF  AND  ARE NOT CONFORMABLE.'
[17]  'AN ARGUMENT OF  AND  IS OF IMPROPER RANK.'
    ∇
```

```
    ∇ M←A VS B;C;D
[1]   →(((ρρB←,B)<ρρB),2 1 0<ρρA)/8 8 4 3
[2]   A←((ρB),1)ρA
[3]   A←((×/ρA),1)ρA
[4]   →(∧/(ρB)≠1,1ρρA)/9
[5]   M←(0,(1ρϕρA)ρ1)\A
[6]   M[;ι1]←B
[7]   →0×ρρM←(1,ρM)ρM
[8]   →0=ρ⍬←'AN ARGUMANT OF VS IS OF IMPROPED RANK.'
[9]   'ARGUMENTS OF VS ARE NOT CONFORMABLE.'
    ∇
```

(B) Single plot; dyadic function with explicit result (i.e. the graph may be saved as a variable)

The plot function PLOT in section (A) was unable to be executed on the CYBER*APL (Ver.1, level 6). The reason being the expand and encode primitives for this implementation of APL appeared to be incompatible with APL/360. The plot rountines described in this section were written to enable this author to continue with his research. (see documentation : P.K.M. Lai, "A Set of Graph Plotting Functions for the CYBER*APL at Imperial College", Internal report, Electrical Engineering Department, Imperial College, 25th Sept.,1975).

The routines are grouped under PLOTGP and GRAPH. The general syntax is:

curve scalings Plot function name depending variable
VS independent varaible

e.g. Z 30 50 SPLOT Y VS X. An error message of *** DATA ERROR (ARRAYS NOT CONFORMATBLE) *** is issued when X and Y are of different lengths.

1) PLOTGP group

QPLOT	Enables a plot without labelling of vertical and horizontal axes. The ranges (i.e. the maximum and minimum values) of these axes are printed. A title heading of the graph may be printed as an option. The title heading is predefined as a blank line in the workspace and is stored as a global variable called TITLE. This option permits a quick plot of the graph computed.
PLOT	Enables a plot with axis-labelling; title option is also available. Plotting time is about 1/3 slower than QPLOT.
SPLOT	Enables a plot with axis labelling and saves (optionally) as a variable. Title option is not available. SPLOT is the slowest among the three plot functions.

NOTE: QPLOT, PLOT, SPLOT display negative numbers as ten's complements; i.e.
-0.65 will appear as 9.35 (i,e. 10-0.65)
Corresponding functions in group GRAPH overcome this difficulties.
Auxilary functions called are: DIGIT,RESHAPE, & VS

```
      )LOAD PLOTGP
PLOTGP  SAVED - 75/08/12. 12.45.52.

      )CBS
      FNS
DESCRIBE        DIGIT   PLOT    QPLOT   RESHAPE SPLOT   VS
      VARS
T       TITLE
      GRPS
PLOG    QPLOG   SPLOG

 DESCRIBE

THIS WORKSPACE CONTAINS FUNCTIONS FOR PLOTTING ONE-VARIABLE GRAPH

  THE MAIN FUNCTIONS ARE:
    **QPLOT**    <->  QUICK PLOT    (NO SCALING ON AXES)
    ** PLOT**    <->  NORMAL PLOT    (SCALING ON AXES; BUT SLOWER)
    **SPLOT**    <->  SAVE PLOT      (SAME AS **PLOT'S; TITLE NOT
                                      AVAILABLE, PLOT IS SAVED)

  ALL FUNCTIONS REQUIRE:  RESHAPE, DIGIT,  AND  VS
  FUNCTION  **VS**  IS USED TO PREPARE DATA FOR **PLOT'S**

  ONE-INDEX ORIGIN IS ASSUMED

  TITLE (OPTIONAL, GLOBAL VARIABLE):
   HEADING OF PLOT IS STORED AS CHARACTER VARIABLE;
   CALLED  **TITLE**  AND IS DEFINED AS BLANK,
   UNTIL ALTERED BY USER.

       TITLE ← ' <HEADING OF PLOT> '

  SYNTAX OF **PLOT'S**:
    A  QPLOT  (B  VS  ⍳⍴B)
    A   PLOT  (B  VS  ⍳⍴B)  :  (B VS ⍳⍴B) = D
  Z←A  SPLOT  (B  VS  ⍳⍴B)
   ( '⍳⍴B' MAY ALSO BE ANOTHER VARIABLE 'C' )

   WHERE
       A <->  TWO-ELEMENT VECTOR;
              THE FIRST ELEMENT SPECIFIES SCALE OF
                    THE VERTICAL AXIS
              THE SECOND ELEMENT SPECIFIES SCALE OF
                    THE HORIZONTAL AXIS
       D <->  TWO-BY-N MATRIX
              THE FIRST COLUMN BEING THE DEPENDENT
                    VARIABLE (VERTICAL)
              THE SECOND COLUMN BEING THE INDEPENDENT
                    VARIABLE (HORIZONTAL)

  NEGATIVE NUMBERS ARE DISPLAYED AS **10'S COMPLEMENT**
  NUMBERS; I.E.
        -0.65  APPEARS AS  (10-0.65)=9.35

  FUNCTIONS ARE GROUPED AS:
     **QPLOG**  : QPLOT RESHAPE DIGIT VS TITLE
     ** PLOG**  :  PLOT RESHAPE DIGIT VS TITLE
     **SPLOG**  : SPLOT RESHAPE DIGIT VS TITLE

  TO CONSERVE ACTIVE WORKSPACE, ERASE DESCRIBE
  BEFORE PLOTTING;
        OR
```



```
      )GRP QPLOG
DIGIT   QPLOT   RESHAPE TITLE   VS

    ∇ Y QPLOT X;A;C;D;MAX;MAXX;MAXY;MINX;MINY;XD;X1;X4;YD;Y1;Y4;T
[1]   X4←X[(⍒X[;1])[1,(⍴X)[1]];1]
[2]   MINX←X4[2]
[3]   MAXX←X4[1]
[4]   Y4←X[(⍒X[;2])[1,(⍴X)[1]];2]
[5]   MINY←Y4[2]
[6]   MAXY←Y4[1]
[7]   C←RESHAPE((MAXY-MINY)+Y[1]-1)
[8]   YD←(C[1]+1)×10*T
[9]   D←RESHAPE((MAXX-MINX)+Y[2]-1)
[10]  XD←(D[1]+1)×10*T
[11]  Y1←1+L((X[;2]-MINY)+YD)
[12]  X1←1+L((X[;1]-MINX)+XD)
[13]  A←(Y[1]×Y[2])⍴' '
[14]  A[⍳(L(Y[2]+5)×5)]←(L(Y[2]+5)×5)⍴'     1'
[15]  A[1,1+(Y[2]×⍳(Y[1]-1))]←'1'
[16]  A[((Y1-1)×Y[2])+X1]←'o'
[17]  A←⊖(Y[1],Y[2])⍴A
[18]  ' '
[19]  TITLE
[20]  ' VERT.:';'    ';MAXY:';'  ';'(MAX.)';'    ';MINY:';'  ';'(MIN.)'
[21]  ' HORZ.:';'    ';MAXX:';'  ';'(MAX.)';'    ';MINX:';'  ';'(MIN.)'
[22]  ' '
[23]  A
    ∇


    ∇ A←RESHAPE X;N;I
[1]   X←,X
[2]   MAX←X[(⍒X)[1]]
[3]   T←⌊10⊛MAX
[4]   X←⌈X×10*(3-T)
[5]   →(1<N←⍴X)/L1
[6]   A←(10 10 10 10⊤X)
[7]   →0
[8]  L1:A←(10 10 10 10⊤X[I←1])
[9]  L2:A←A,(10 10 10 10⊤X[I←I+1])
[10]  →(N>I)/L2
[11]  A←(((⍴A)+4),4)⍴A
    ∇


    ∇ Z←N DIGIT X
[1]   Z←'0123456789'[1+X]
[2]   →((⍴⍴Z)=1)/L1
[3]   Z←Z[;1],'.',Z[;1+⍳N-1]
[4]   →0
[5]  L1:Z←Z[1],'.',Z[1+⍳N-1]
    ∇


    ∇ A1←X VS Y;A
[1]   →L1×⍳((⍴X)≠(⍴Y))
[2]   A←(2,(⍴X))⍴,Y,X
[3]   A1←⍉A
[4]   →0
[5]  L1:' *** DATA ERROR (ARRAYS NOT CONFORMABLE) ***'
    ∇
```

```
      )GRP SPLOG
DIGIT   RESHAPE SPLOT   TITLE   VS
```

```
    ∇ Z←Y SPLOT X;N4;Y2;H;K;T;C;D;X4;Y4;Y3;Q;Y1;X1;N;Q1;YD;XD;MAXX;MIN
      X;MAXY;MINY;A;MAX
[1]   X4←X[(⍳X[;1])[1,(⍴X)[1]];1]
[2]   MINX←X4[2]
[3]   MAXX←X4[1]
[4]   Y4←X[(⍳X[;2])[1,(⍴X)[1]];2]
[5]   MINY←Y4[2]
[6]   MAXY←Y4[1]
[7]   Y[1]←Y[1]+1
[8]   C←RESHAPE((MAXY-MINY)+Y[1]-1)
[9]   YD←(C[1]+1)×10*T
[10]  D←RESHAPE((MAXX-MINX)+Y[2]-1)
[11]  XD←(D[1]+1)×10*T
[12]  Y1←1+L((X[;2]-MINY)+YD)
[13]  X1←1+L((X[;1]-MINX)+XD)
[14]  A←(Y[1]×Y[2])⍴' '
[15]  A[\(L(Y[2]+5)×5)]←(L(Y[2]+5)×5)⍴'     I'
[16]  A[1,1+(Y[2]×\(Y[1]-1))]←'I'
[17]  A[((Y1-1)×Y[2])+X1]←'o'
[18]  A←⌽(Y[1],Y[2])⍴A
[19]  ' '
[20]  N←N4←0
[21]  Y2←MINY+(YD×5×(0,\(LY[1]+5)))
[22]  Y2←4 DIGIT RESHAPE Y2
[23]  H←K←0⍴' '
[24] L1:N←N+1
[25]  K←K,Y2[N;],' E',('0123456789'[1+10⊤10TT]),32⍴' '
[26]  →L1×\(N≤L(⍴A)[1]+5)
[27]  Z←(⌽((⍴A)[1],8)⍴K),A
[28]  Y3←1+L((⍴A)[2])+10
[29]  X←MINX+10×XD×(0,\Y3)
[30]  Q←4 DIGIT RESHAPE X
[31] L4:N4←N4+1
[32]  H←H,Q[N4;],' E',('0123456789'[1+10⊤10TT]),(2⍴' ')
[33]  →L4×\(N4<Y3)
[34]  Z←(Z,((⍴Z)[1],(4+(⍴H)-(⍴Z)[2]))⍴' '),[1](4⍴' '),H
    ∇
```

```
      )GRP PLOG
DIGIT   PLOT    RESHAPE TITLE   VS
```

```
    ∇ Y PLOT X;N4;Y2;H;K;T;C;D;X4;Y4;Y3;Q;Y1;X1;N;Q1;YD;XD;MAXX;MINX;M
      AXY;MINY;A;MAX
[1]   X4←X[(⍳X[;1])[1,(⍴X)[1]];1]
[2]   MINX←X4[2]
[3]   MAXX←X4[1]
[4]   Y4←X[(⍳X[;2])[1,(⍴X)[1]];2]
[5]   MINY←Y4[2]
[6]   MAXY←Y4[1]
[7]   Y[1]←Y[1]+1
[8]   C←RESHAPE((MAXY-MINY)+Y[1]-1)
[9]   YD←(C[1]+1)×10*T
[10]  D←RESHAPE((MAXX-MINX)+Y[2]-1)
[11]  XD←(D[1]+1)×10*T
[12]  Y1←1+L((X[;2]-MINY)+YD)
[13]  X1←1+L((X[;1]-MINX)+XD)
[14]  A←(Y[1]×Y[2])⍴' '
[15]  A[\(L(Y[2]+5)×5)]←(L(Y[2]+5)×5)⍴'     I'
[16]  A[1,1+(Y[2]×\(Y[1]-1))]←'I'
[17]  A[((Y1-1)×Y[2])+X1]←'o'
[18]  A←⌽(Y[1],Y[2])⍴A
[19]  ' '
[20]  TITLE
[21]  ' '
[22]  N←N4←0
[23]  Y2←MINY+(YD×5×(0,\(LY[1]+5)))
[24]  Y2←4 DIGIT RESHAPE Y2
[25]  H←K←0⍴' '
[26] L1:N←N+1
[27]  K←K,Y2[N;],' E',('0123456789'[1+10⊤10TT]),32⍴' '
[28]  →L1×\(N≤L(⍴A)[1]+5)
[29]  (⌽((⍴A)[1],8)⍴K),A
[30]  Y3←1+L((⍴A)[2])+10
[31]  X←MINX+10×XD×(0,\Y3)
[32]  Q←4 DIGIT RESHAPE X
[33] L4:N4←N4+1
[34]  H←H,Q[N4;],' E',('0123456789'[1+10⊤10TT]),(2⍴' ')
[35]  →L4×\(N4<Y3)
[36]  4⍴' ':H
    ∇
```

```
      )LOAD GRAPH
GRAPH  SAVED - 75/09/10. 16.56.35.


      )WSID
GRAPH
      )CLR
      FNS
DESCRIBE        DIGIT   PLOT1   RESHAPE1        ROUNDOFF
TEST    TEST1   VS
      VARS
W
      GRPS
PLOG1
      )SAVE PLOG1
DIGIT   PLOT1   RESHAPE1        ROUNDOFF        TEST    TEST1
VS


      DESCRIBE

THIS WORKSPACE CONTAINS FUNCTIONS FOR PLOTTING ONE-VARIABLE GRAPH

  THE MAIN FUNCTION IS :  **PLOT1**

  SUPPORTING FUNCTIONS ARE:
   TEST, TEST1, RESHAPE1, DIGIT,  AND  VS
  FUNCTION  **VS**  PREPARES INPUT DATA FOR **PLOT1**.

  ONE-INDEX ORIGIN IS ASSUMED

  SYNTAX OF FUNCTION:
     Z ←  A  PLOT1  (B  VS  ⍳⍴B)

  WHERE,
    A ←→  TWO-ELEMENT VECTOR;
          THE FIRST ELEMENT SPECIFIES SCALE OF
              THE VERTICAL AXIS
          THE SECOND ELEMENT SPECIFIES SCALE OF
              THE HORIZONTAL AXIS
    B ←→ (B  VS  ⍳⍴B); TWO-BY-N MATRIX
          THE FIRST COLUMN BEING THE DEPENDENT
              VARIABLE (VERTICAL)
          THE SECOND COLUMN BEING THE INDEPENDENT
              VERIABLE (HORIZONTAL)

    [ '⍳⍴B' MAY ALSO BE ANOTHER VARIABLE 'C' ]

NOTE:
    **PLOT1**  IS IDENTICAL TO  **SPLOT** IN WORKSPACE
    **PLOTSP**; EXCEPT THAT NEGATIVE NUMBERS ARE DISPLAYED
    INSTEAD OF '10'S COMPLEMENT' NUMBERS AS  BY **SPLOT**

  TO CONSERVE ACTIVE WORKSPACE, ERASE DESCRIBE BEFORE PLOT
```

A LISTING OF ALL FUNCTIONS IN **GRAPH** WS

```
     V Z←Y PLOT1 X;MAXX;MINX;MAXY;MINY;C;YD;T;D;XD;Y1;X1;A;Q;N;Y2;N4;H;
       K;Y3;X3
[1]    MAXX←X[(⍒X[;1])[1];1]
[2]    MINX←X[(⍋X[;1])[1];1]
[3]    MAXY←X[(⍒X[;2])[1];2]
[4]    MINY←X[(⍋X[;2])[1];2]
[5]    Y[1]←Y[1]+1
[6]    C←RESHAPE1(MAXY-MINY)+Y[1]
[7]    YD←(C[1]+1)×10*T
[8]    D←RESHAPE1((MAXX-MINX)+Y[2])
[9]    XD←(D[1]+1)×10*T
[10]   Y1←1+⌊((X[;2]-MINY)+YD)
[11]   X1←1+⌊((X[;1]-MINX)+XD)
[12]   A←((Y[1]+1)×Y[2])⍴' '
[13]   A[⍳(⌊(Y[2]+5)×5)]←(⌊(Y[2]+5)×5)⍴'     I'
[14]   A[1,1+(Y[2]×⍳(Y[1]))]←'I'
[15]   Q←((Y1-1)×Y[2])+X1
[16]   A[Q]←'o'
[17]   A←⊖Y⍴A
[18]   ' '
[19]   N←N4←0
[20]   Y2←MINY+(YD×5×(0,⍳(⌊(Y[1]+1)+5)))
[21]   H←K←3⍴''
[22]  L1:N←N+1
[23]   H←3⍴''
[24]   K←K,(TEST Y2[N]),44⍴' '
[25]   →L1×⍳(N≤⌊(⍴A)[1]÷5)
[26]   Z←(⊖((⍴A)[1],11)⍴K),A
[27]   Y3←1+⌊((⍴A)[2])+10
[28]   X3←MINX+10×XD×(⍳,⍳Y3)
[29]  L4:N4←N4+1
[30]   H←H,(TEST1 X3[N4]),' '
[31]   →L4×⍳(N4<Y3)
[32]   Z←(Z,((⍴Z)[1],(4+(⍴H)-(⍴Z)[2]))⍴' '),[1](4⍴' '),H
     ∇


     ∇ K←TEST X;N;Y
[1]    X←,X
[2]    K←''
[3]    N←0
[4]   L6:N←N+1
[5]    →0×⍳(N>(⍴X))
[6]    →((X[N]>0),(X[N]≤0))/L7,L8
[7]   L7:→((X[N]>1),(X[N]≤1))/L1,L2
[8]   L8:→(((⌊X[N])>1),((⌊X[N])≤1))/L4,L5
[9]   L1:Y←4 DIGIT RESHAPE1 ⌊X[N]
[10]   K←K,Y,' E',('0123456789'[1+10⊥10⍴TT]),2⍴' '
[11]   →L6
[12]  L2:Y←4 DIGIT RESHAPE1 ⌊X[N]
[13]   K←K,Y,' E¯',('0123456789'[1+10⊥10⍴TT]),' '
[14]   →L6
[15]  L5:Y←4 DIGIT RESHAPE1 ⌊X[N]
[16]   K←K,'-',Y,' E¯',('0123456789'[1+10⊥10⍴TT])
[17]   →L6
[18]  L4:Y←4 DIGIT RESHAPE1 ⌊X[N]
[19]   K←K,'-',(Y),' E',('0123456789'[1+10⊥10⍴TT]),' '
[20]   →L6
     ∇
```

```
    ∇ K←TEST1 X;N;Y
[1]   X←,X
[2]   K←''
[3]   N←0
[4]  L6:N←N+1
[5]   →0×⍳(N>(⍴X))
[6]   →((X[N]>0),(X[N]≤0))/L7,L8
[7]  L7:→((X[N]>1),(X[N]≤1))/L1,L2
[8]  L8:→(((⌊X[N])>1),((⌊X[N])≤1))/L4,L5
[9]  L1:Y←2 DIGIT RESHAPE1 ⌊X[N]
[10]  K←K,Y,' E',('0123456789'[1+1⍴10TT]),2⍴' '
[11]  →L6
[12] L2:Y←2 DIGIT RESHAPE1 ⌊X[N]
[13]  K←K,Y,' E¯',('0123456789'[1+1⍴10TIT]);' '
[14]  →L6
[15] L5:Y←2 DIGIT RESHAPE1 ⌊X[N]
[16]  K←K,'-',Y,' E¯',('0123456789'[1+1⍴10TIT])
[17]  →L6
[18] L4:Y←2 DIGIT RESHAPE1 ⌊X[N]
[19]  K←K,'-',(Y),' E',('0123456789'[1+1⍴10TT]),' '
[20]  →L6
    ∇
```

```
    ∇ A←RESHAPE1 X;N;MAX;I
[1]   X←,X
[2]   MAX←X[(⍒X)[1]]
[3]   T←L 10⊛MAX
[4]   X←ROUNDOFF X×10*(3-T)
[5]   →(1<N←⍴X)/L1
[6]   A←(10 10 10 10TX)
[7]   →0
[8]  L1:A←(10 10 10 10TX[I←1])
[9]  L2:A←A,(10 10 10 10TX[I←I+1])
[10]  →(N>I)/L2
[11]  A←(((⍴A)+4),4)⍴A
    ∇
```

```
    ∇ Y←ROUNDOFF X
[1]   Y←⌊X+0.5
    ∇
```

```
    ∇ Z←N DIGIT X
[1]   Z←'0123456789'[1+X]
[2]   →((⍴⍴Z)=1)/L1
[3]   Z←Z[;1],'.',Z[;1+⍳N-1]
[4]   →0
[5]  L1:Z←Z[1],'.',Z[1+⍳N-1]
    ∇
```

```
    ∇ A1←X VS Y;A
[1]   →L1×⍳((⍴X)≠(⍴Y))
[2]   A←(2,(⍴X))⍴Y,X
[3]   A1←⍉A
[4]   →0
[5]  L1:' *** DATA ERROR (ARRAYS NOT CONFORMABLE) ***'
    ∇
```

```
⍝ ILLUSTRATIONS OF PLOT-FUNCTIONS

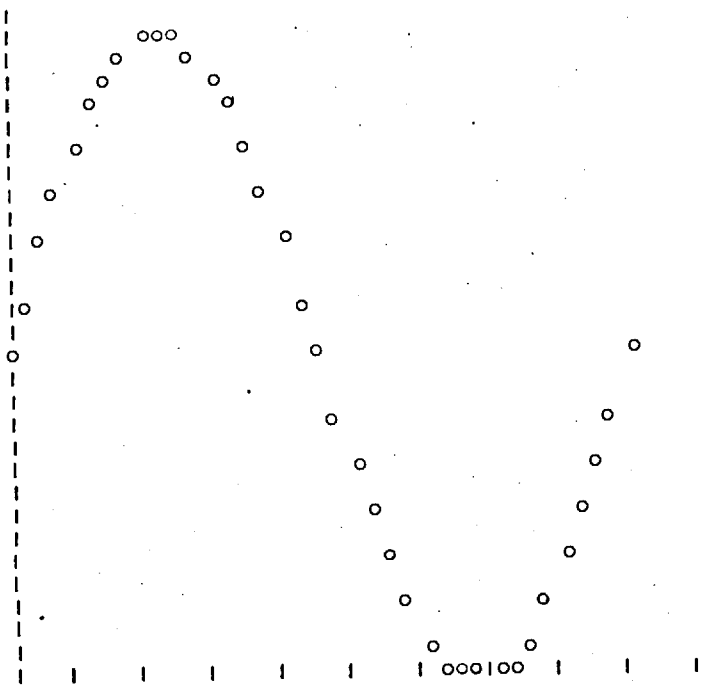
TITLE←' ** SINE-WAVE PLOT **'



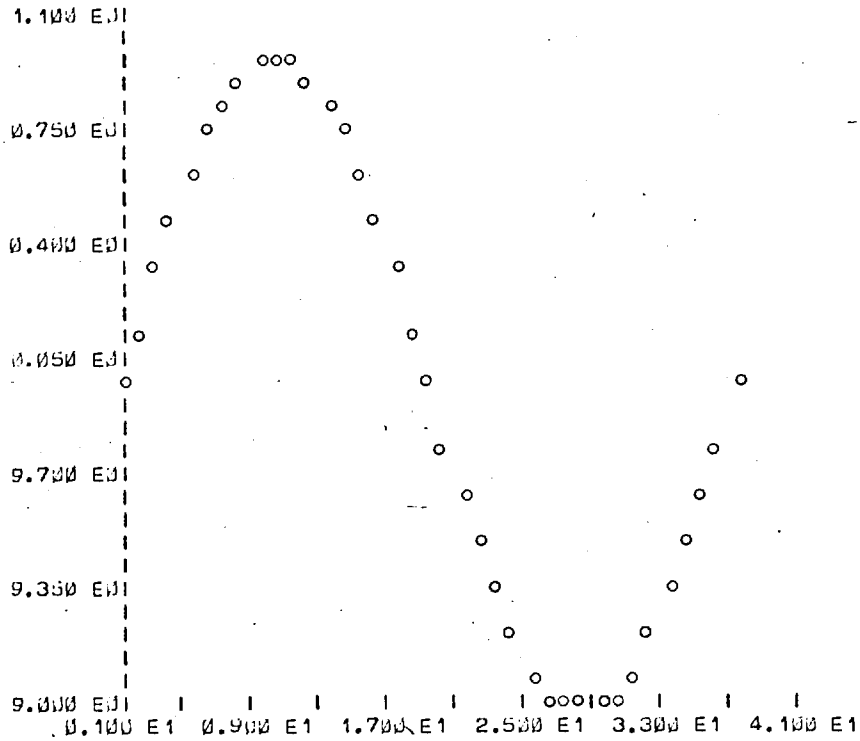
⍝ *** GROUP QPLOG ***

30 50 QPLOT T VS ⍳⍴T
```

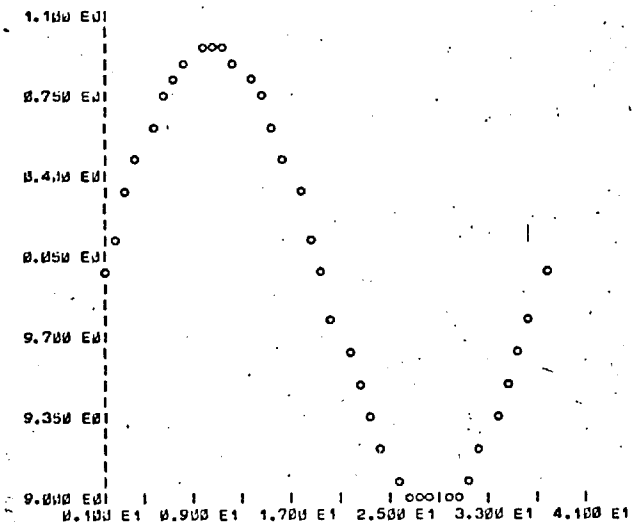
```
** SINE-WAVE PLOT **
VERT.:     1  (MAX.)   ¯1  (MIN.)
HORZ.:    37  (MAX.)    1  (MIN.)
```



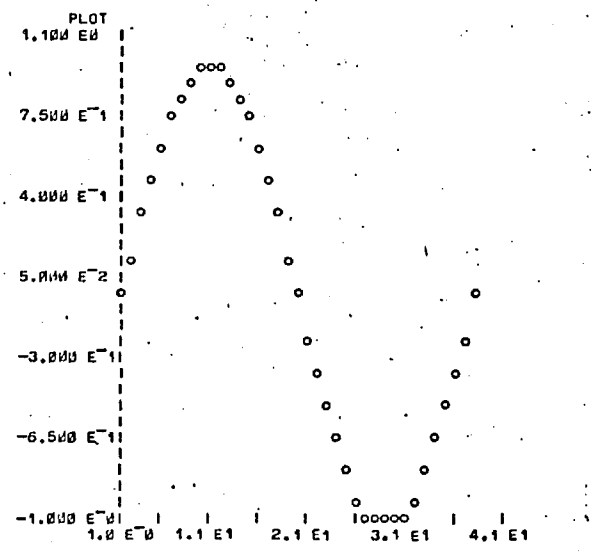
```
A *** GROUP PLOG ***
3.0 5.0 PLOT T VS \PT
** SINE-WAVE PLOT **
```




```
A *** GROUP SPLOR ***
A    THE SINE-WAVE PLOT IS TO SAVED AS  **GRAPH**
GRAPH=30 50 SPLOT T VS \PT
A  **GRAPH** IS DISPLAYED AT A LATER DATE
GRAPH
```




```
PLOT= 30 40 PLOT1 W VS \PW
```



Appendix 6 :

A COLLECTION OF APL FUNCTIONS FOR SIGNAL ANALYSIS

APL functions for signal analysis

SAMPLE — Generate a normal distribution ; calling function RAN (origin 0 is assumed)

RAN — Generates a pseudorandom sequence with a uniform distribution; function may be used individually.

MEAN — Computes the arithmetic mean of a vector

FAST — Computes the fast fourier transform by the method developed by Cooley-Tukey. FAST can performs transformations from the "time to frequency domain" and the Inverse Transform or analysis from the "frequency to time domain". FAST calls the function FFT

- A = 1 : transform (time to frequency)
  = 2 : inverse transform (frequency to time)
- B : an N by 2 array of the data points, where N must be a power of 2 and not exceeding 256. B[;1] represents the real components and B[;2] represents the imaginary components.
- Z : an N by 2 array of the result, where Z[;1] represents the real components, and Z[;2] represents the imaginary components

FFT — Performs the fast fourier transforms; and the environment is set up by FAST (functions FAST and FFT assume origin 1)

```
      )GRP SIGNALGP
RAN     SAMPLE  MEAN

      ∇RAN[□]∇
    ∇ R←RAN N;I;TRIAL
[1]   '  THIS GENERATES A PSEUDORANDOM SEQUENCE WITH UNIFORM DIST.'
[2]   ' '
[3]   'ENTER ''TRIAL'' '
[4]   TRIAL←□
[5]   R←TRIAL⍴0
[6]   ' '
[7]   I←1
[8] LAB:R[I]←?N
[9]   I;'    ';R
[10]  I←I+1
[11]  →(TRIAL>I)/LAB
    ∇

      ∇SAMPLE[□]∇
    ∇ R←SAMPLE X;SIGMA;N;MAX;C1;C2;R1;R2;I
[1]   ⍝  GENERATION OF NORMAL DISTRIBUTION (0-ORIGIN INDEX)
[2]   ⍝
[3]   ⍝    X IS A VECTOR WITH ELEMENTS:-
[4]   ⍝       X[0] ↔ SIGMA (STANDARD DEVIATION)
[5]   ⍝       X[1] ↔   N   (NO. OF SAMPLE TO BE COMPUTED)
[6]   ⍝       X[2] ↔  MAX  (INTERVAL -- RANDOM NO. CHOSEN)
[7]   SIGMA←X[0]
[8]   N←X[1]
[9]   MAX←X[2]
[10]  R←N⍴0
[11]  C1←⍟MAX
[12]  C2←○2×MAX
[13]  I←0
[14] LAB:R1←SIGMA×(¯2×(⍟?MAX)-C1)*0.5
[15]  R2←C2×?MAX
[16]  ⍝  TWO RANDOM VARIATES FROM A STANDARD NORMAL DISTRIBUTION
[17]  R[I]←R1×2○R2
[18]  I←I+1
[19]  R[I]←R1×1○R2
[20]  I←I+1
[21]  →(N>I)/LAB
    ∇

      ∇MEAN[□]∇
    ∇ Z←MEAN X;TOTAL
[1]   TOTAL←⍴X
[2]   Z←(+/X×X)÷TOTAL
    ∇
```

Fast Fourier Transform

```
      ∇R←EXP FAST SIG
[1]     ⍝ CHECK IF SIG IS A (2×⍴SIG)x2 MATRIX
[2]     →(2=¯1↑⍴SIG)/OK
[3]     ⍝ FORM (2×⍴SIG) x2 MATRIX
[4]     SIG←((⍴SIG),1)⍴SIG
[5]     SIG←SIG,0
[6]   OK:FFT EXP
[7]     R←SIG
      ∇


      ∇FFT EXP;B;D;E;L;M;N;P;T
[1]     →L0⌈1024<N←2×P←⌈2⍟1↑⍴SIG
[2]     →0×⍴□←'1024 POINT FFT MAXIMUM'
[3]   L0:SIG←(N,2)↑SIG
[4]     SIG←(2,D←P⍴2)⍴SIG
[5]     E←M,T←0,Φ-1↓M←2○(1-⍳N÷4)×○2÷N
[6]     T←D⍴E,[1.5]T,-M
[7]     D←P-0.5,0↑E←LE,L1
[8]   L1:M←-⌿SIG
[9]     →L2⌈1↑L←0 1=P←P-1,0⍴SIG←+⌿SIG
[10]    B←-/T×M
[11]    B←0⍴M←B,[D]+/T×ΦM
[12]  L2:SIG←SIG,[P+0.5]M
[13]    →L/E
[14]    →L1,⍴T←T,[P]T←1 0/[P]T
[15]  LE:M←T←÷EXP↑1,N
[16]    →0×⍳EXP=¯1,0⍴SIG←M×(N,2)⍴SIG
[17]    SIG←⊖SIG
[18]    SIG←¯1⊖SIG
      ∇
```

Appendix 7 :

A LIST OF APL FUNCTIONS FOR BINARY-ARITHMETIC ("ROUNDING")[1]

---

[1] This collection was developed for the study of "rounding" in the analysis & simulation of digital filters.

List of APL functions for binary-arithmetic ("rounding")

Function Name	Usage
SINFA	Separates the sign, integer and fractional parts of a signed-real decimal number
INBIN	Converts an integer part of a decimal number to binary equivalent
FDB	Converts fractional part of a real decimal number to binary
DEBINSM	Decimal to binary (signed-magnitude) conversion; decimal can be real or integer
SMONES	Converts signed-magnitude to one's complement representation
SMTWOS	Converts signed-magnitude to two's complement
ONESM	Converts one's complement to signed-magnitude
TWOSM	Converts two's complement to signed-magnitude
FSMDEC	Converts a fractional part of a signed-magnitude binary into decimal
SMBINDEC	Converts a binary signed-magnitude to decimal
ONESDEC	Converts one's complement to decimal
TWOSDEC	Converts two's complement to decimal
DECONES	Converts decimal to one's complement
DECTWOS	Converts decimal to two's complement

(origin zero is assumed;

also see "listing for Algorithmic Representation of Computer Arithmetic"

```
    ∇ SINFA X
[1]   ABX←|X
[2]   SGNX←0≠X-ABX
[3]   INX←⌊ABX
[4]   FRX←ABX-INX
    ∇
```

```
      SINFA 7.56
      ABX;'   ';SGNX;'   ';INX;'   ';FRX
7.56   0   7   0.56

      SINFA ¯7.56
      ABX;'   ';SGNX;'   ';INX;'   ';FRX
7.56   1   7   0.56
```

```
    ∇ BINX←INBIN INX
[1]   BINX←(N⍴2)⊤INX
    ∇
```

```
      N←5
      ⎕←ANSWER←INBIN 7.56
0  0  1  1  1.56
```

This shows that the fractional part is not converted.

```
      ⎕←ANSWER←INBIN 7
0  0  1  1  1

      ⎕←ANSWER←INBIN ¯7
1  1  0  0  1
```

This shows that when -ve; 2's complement representation is used.

```
    ∇ Z←FDB FRX
[1]   BFPX←I←0
[2]   MLAB:RB←FRX×2
[3]   IB←⌊RB
[4]   FRX←RB-IB
[5]   BFPX←BFPX,IB
[6]   I←I+1
[7]   →(I<M)/MLAB
[8]   BFPX←1↓BFPX
[9]   Z←BFPX
    ∇
```

RB can be replaced by FRX hence eliminate one variable

← this statement can be eliminated if function header is modified to BFPX ← FDB FRX

```
      M←5
      ⎕←ANSWER←FDB 0.5
1  0  0  0  0

      ⎕←ANSWER←FDB 0.25
0  1  0  0  0

      ⎕←ANSWER←FDB 0.125
0  0  1  0  0

      ⎕←ANSWER←FDB 0.1
0  0  0  1  1
      ⎕←ANSWER←FDB 0.01
0  0  0  0  0
```

This shows the effect of underflow.

```
    ∇ Z←DEBINSM X;ABX;SGNX;INX;FRX
[1]   ⍝ SEPARATE -- SIGN, INTEGER AND FRACTIONAL PARTS
[2]   ABX←|X
[3]   SGNX←0≠X-ABX
[4]   INX←⌊ABX
[5]   FRX←ABX-INX
[6]   ⍝ CONVERT INTEGER PART INTO BINARY REPRESENTATION
[7]   BINX←(N⍴2)⊤INX
[8]   ⍝ CONVERT FRACTIONAL PART TO BINARY REPRESENTATION
[9]   BFRX←I←0
[10] MLAB:BB←FRX×2
[11]  IB←⌊BB
[12]  FRX←BB-IB
[13]  BFRX←BFRX,IB
[14]  I←I+1
[15]  →(I<M)/MLAB
[16]  BFRX←1↓BFRX
[17]  ⍝ ASSEMBLE BINARY NUMBER
[18]  Z←SGNX,BINX,BFRX
[19]  SGNX;' ';BINX;' ';BFRX
    ∇
```

```
      N←5
      M←5

      □←ANSWER←DEBINSM 7.55
0 0 0 1 1 1 1 0 0 0 1
0 0 0 1 1 1 1 1 0 0 0 1

      □←ANSWER←DEBINSM ¯7.56
1 0 0 1 1 1 1 0 0 0 1
1 0 0 1 1 1 1 1 0 0 0 1
```

An updated version of DEBINSM : (with illustration)

```
    ∇ Z←DEBINSM X;ABX;SGNX;INX;FRX;BINX;BFRX;IB;I
[1]   ABX←|X                          ⎫
[2]   SGNX←0≠X-ABX                    ⎪ Separate SIGN, INTEGER and FRACTIONAL parts
[3]   INX←⌊ABX                        ⎪
[4]   FRX←ABX-INX                     ⎭
[5]   BINX←(N⍴2)⊤INX                  ⎫ Convert INTEGER part into binary
[6]   →(M≠0)/REAL                     ⎬ Test if M≠0 (i.e. if X is real); otherwise X is integer
[7]   Z←SGNX,BINX                     ⎪ Assemble integer X and ends program.
[8]   →0                              ⎭
[9]  REAL:BFRX←I←0                    ⎫
[10] MLAB:FRX←FRX×2                   ⎪
[11]  IB←⌊FRX                         ⎪
[12]  FRX←FRX-IB                      ⎪ Convert FRACTIONAL part into binary
[13]  BFRX←BFRX,IB                    ⎪
[14]  I←I+1                           ⎪
[15]  →(I<M)/MLAB                     ⎭
[16]  Z←SGNX,BINX,1↓BFRX              ⎫ Assemble REAL X.
    ∇                                 ⎭
```

```
      DEBINSM 7.65
DEBINSM[1] 7.65
DEBINSM[2] 0
DEBINSM[3] 7
DEBINSM[4] 0.65
DEBINSM[5] 0 0 1 1 1
DEBINSM[6]
DEBINSM[7] 0 0 0 1 1 1
DEBINSM[8] 0
0 0 0 1 1 1
```

This program illustrates the truncation of the fractional part of a real decimal number.

```
      N←M←5
      DEBINSM 7.65
DEBINSM[1] 7.65
DEBINSM[2] 0
DEBINSM[3] 7
DEBINSM[4] 0.65
DEBINSM[5] 0 0 1 1 1
DEBINSM[6] 9 ← go to statement 9.
DEBINSM[9] 0
DEBINSM[10] 1.3
DEBINSM[11] ①
DEBINSM[12] 0.3
DEBINSM[13] 0 1
DEBINSM[14] 1
DEBINSM[15] 10 ← go to statement 10
DEBINSM[10] 0.6
DEBINSM[11] ⓪
DEBINSM[12] 0.6
DEBINSM[13] 0 1 0
DEBINSM[14] 2
DEBINSM[15] 10
DEBINSM[10] 1.2
DEBINSM[11] ①
DEBINSM[12] 0.2
DEBINSM[13] 0 1 0 1
DEBINSM[14] 3
DEBINSM[15] 10
DEBINSM[10] 0.4
DEBINSM[11] ⓪
DEBINSM[12] 0.4
DEBINSM[13] 0 1 0 1 0
DEBINSM[14] 4
DEBINSM[15] 10
DEBINSM[10] 0.8
DEBINSM[11] ⓪
DEBINSM[12] 0.8
DEBINSM[13] 0 1 0 1 0 0
DEBINSM[14] 5
DEBINSM[15]
DEBINSM[16] 0 0 0 1 1 1 1 0 1 0 0
```

Note

① When DEBINSM <number decimal> is used; explicit result is given. This is displayed after DEBINSM.

② When <argument> DEBINSM <decimal no.> is used; implicit result is given. Trace will stop after DEBINSM [16] and the final result is not displayed once more. (see page R.9.6)

③ This result show that conversion between BINARY/DECIMAL has inherent error. A type of "quantizing error" results.

```
0.1     ↔ 0.5
0.01    ↔ 0.25
0.001   ↔ 0.125
0.0001  ↔ 0.0625
0.00001 ↔ 0.03125
0.000001 ↔ 0.015625
```

0 | 0 0 1 1 1 | 1 0 1 0 0 ← sign-magnitude representation of 7.65

Sign bit | integer part ↔ 7 | fractional part ↔ 0.5+0.125 = 0.625

```
    ∇ Y←SMONES X;SGN
[1]   SGN←1↑X                ← Examine sign bit
[2]   →(SGN=0)/POS
[3]   Y←SGN,~(1↓X)           ← x negative, 1's complement; exit
[4]   →0
[5]  POS:Y←X                 ← x positive, no complement required; exit.
      ∇

    ∇ Y←SMTWOS X;SGN
[1]   SGN←1↑X
[2]   →(SGN=0)/POS
[3]   Y←~(1↓X)
[4]   Y←((⍴Y)⍴2)⊤1+(2⊥Y)     } These two statements can be combined, viz
[5]   Y←SGN,Y                    Y ← SGN, ((⍴Y)⍴2)⊤1 + 2⊥Y
[6]   →0
[7]  POS:Y←X
      ∇

    ∇ Y←ONESM X;SGN
[1]   SGN←1↑X          } Examine sign bit, jump to POS if x is positive
[2]   →(SGN=0)/POS     }
[3]   Y←~(1↓X)         } Take one's compl of x until is negative and exit
[4]   Y←SGN,Y          }
[5]   →0
[6]  POS:Y←X           } x is positive
      ∇

    ∇ Y←TWOSM X;SGN
[1]   SGN←1↑X          } Examine sign bit, jump to POS if x is positive
[2]   →(SGN=0)/POS     }
[3]   Y←~(1↓X)         ← Take 1's compl. of x giving Y
[4]   Y←((⍴Y)⍴2)⊤1+(2⊥Y) ← 2's compl. by adding 1 to Y
[5]   Y←SGN,Y          } ← Assemble number; exit.
[6]   →0               }
[7]  POS:Y←X           ← x is positive, no conversion necessary
      ∇
```

```
      'TEST RUNS OF ALL FUNCTIONS****'

TEST RUNS OF ALL FUNCTIONS****

      X←DEBINSM 13
      Y←DEBINSM ¯7
      X
0 1 1 0 1
      Y
1 0 1 1 1

      X1←SMONES X
      X2←SMTWOS X
      Y1←SMONES Y
      Y2←SMTWOS Y

      X1X←ONESM X1
```




```
      X2X←TWOSM X2
      Y1Y←ONESM Y1
      Y2Y←TWOSM Y2

      X
0 1 1 0 1   } (+13)
      X1X
0 1 1 0 1   } ✓
      X1
0 1 1 0 1   } One's complement

      X2
0 1 1 0 1   } Two's complement
      X2X   } ✓
0 1 1 0 1   }

      Y
1 0 1 1 1   } (-7)
      Y1
1 1 0 0 0   } ✓
      Y1Y
1 0 1 1 1   } One's complement

      Y2
1 1 0 0 1   } Two's complement
      Y2Y
1 0 1 1 1   } ✓

      M←N←5

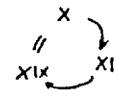
      ⎕←X←DEBINSM 6.75
0 0 0 1 1 0 1 1 0 0 0 ✓ ← SM as y
      ⎕←Y←DEBINSM ¯6.75
1 0 0 1 1 0 1 1 0 0 0   ← -SM

      ⎕←X1←SMONES X
0 0 0 1 1 0 1 1 0 0 0 ✓ same as SM
      ⎕←X2←SMTWOS X
0 0 0 1 1 0 1 1 0 0 0 ✓  " "

      ⎕←X1X←ONESM X1
0 0 0 1 1 0 1 1 0 0 0 (+6·75) ✓
      ⎕←X2X←TWOSM X2
0 0 0 1 1 0 1 1 0 0 0 (+6·75) ✓

      ⎕←Y1←SMONES Y
1 1 1 0 0 1 0 0 1 1 1 1's compl. of (-SM) ✓
      ⎕←Y2←SMTWOS Y
1 1 1 0 0 1 0 1 0 0 0 2's compl of (-SM) ✓

      ⎕←Y1Y←ONESM Y1
1 0 0 1 1 0 1 1 0 0 0
      ⎕←Y2Y←SMTWOS Y2
1 0 0 1 1 0 1 1 0 0 0
```



```
    ∇ FRX←FSMDEC X;I
[1]   X←,X
[2]   I←(⍴X)-1
[3]   FRX←X[I]×0.5
[4]   →(I=0)/0
[5]  FINLAB:I←I-1
[6]   FRX←(FRX+X[I])×0.5
[7]   →(I≠0)/FINLAB
    ∇


      )ORIGIN 0
WAS 0

      TΔFSMDEC←19

      FSMDEC 1
FSMDEC[1] 1
FSMDEC[2] 0
FSMDEC[3] 0.5
FSMDEC[4] 0
0.5

      FSMDEC 0 1
FSMDEC[1] 0  1
FSMDEC[2] 1
FSMDEC[3] 0.5
FSMDEC[4]
FSMDEC[5] 0
FSMDEC[6] 0.25
FSMDEC[7]
0.25

      FSMDEC 0 0 1
FSMDEC[1] 0  0  1
FSMDEC[2] 2
FSMDEC[3] 0.5
FSMDEC[4]
FSMDEC[5] 1
FSMDEC[6] 0.25
FSMDEC[7] 5
FSMDEC[5] 0
FSMDEC[6] 0.125
FSMDEC[7]
0.125

      FSMDEC 1 0 1
FSMDEC[1] 1  0  1
FSMDEC[2] 2
FSMDEC[3] 0.5
FSMDEC[4]
FSMDEC[5] 1
FSMDEC[6] 0.25
FSMDEC[7] 5
FSMDEC[5] 0
FSMDEC[6] 0.625
FSMDEC[7]
0.625
```

```
    ∇ Z←SMBINDEC X;BSGN;BIN;BFR;DBFR;DEBFR
[1]   BSGN←X[0]
[2]   BIN←1↓((N+1)↑X)
[3]   →((N+1)=⍴X)/XINT
[4]   BFR←(N+1)↓X
[5]   →OK
[6]  XINT:Z←2⊥BIN
[7]   →ANS
[8]  OK:Z←2⊥BIN
[9]   DEBFR←FSMDEC BFR
[10]  Z←Z+DEBFR
[11] ANS:→(BSGN=0)/0
[12]  Z←-Z
    ∇

      X←0,1 0 1,0 0 1
      N←M←3
      SMBINDEC X
SMBINDEC[1] 0
SMBINDEC[2] 1  0  1
SMBINDEC[3]
SMBINDEC[4] 0  0  1
SMBINDEC[5] 8
SMBINDEC[8] 5
SMBINDEC[9] 0.125
SMBINDEC[10] 5.125
SMBINDEC[11] 0
5.125

      X←1,1 0 1, 0 0 1
      SMBINDEC X
SMBINDEC[1] 1
SMBINDEC[2] 1  0  1
SMBINDEC[3]
SMBINDEC[4] 0  0  1
SMBINDEC[5] 8
SMBINDEC[8] 5
SMBINDEC[9] 0.125
SMBINDEC[10] 5.125
SMBINDEC[11]
SMBINDEC[12] ¯5.125
¯5.125
```

```
    ∇ Z←TWOSDEC X
[1]   Z←SMBINDEC(□←TWOSM X)
    ∇

    ∇ Z←ONESDEC X
[1]   Z←SMBINDEC(□←ONESM X)
    ∇

    ∇ Z←DECTWOS X
[1]   Z←SMTWOS(□←DEBINSM X)
    ∇

    ∇ Z←DECONES X
[1]   Z←SMONES(□←DEBINSM X)
    ∇
```

□ is inserted for observations during the developments of the functions; □ should be be removed for final versions

```
      N←M←5

      DECONES 6.75
0  0  0  1  1  0  1  1  0  0  0
0  0  0  1  1  0  1  1  0  0  0
      DECONES ¯6.75
1  0  0  1  1  0  1  1  0  0  0
1  1  1  0  0  1  0  0  1  1  1
```

a real decimal number

```
      DECTWOS 6.75
0  0  0  1  1  0  1  1  0  0  0
0  0  0  1  1  0  1  1  0  0  0
      DECTWOS ¯6.75
1  0  0  1  1  0  1  1  0  0  0
1  1  1  0  0  1  0  1  0  0  0
```

a real decimal number

```
      X1←DECONES ¯6.75
1  0  0  1  1  0  1  1  0  0  0
      X2←DECTWOS ¯6.75
1  0  0  1  1  0  1  1  0  0  0
      X1
1  1  1  0  0  1  0  0  1  1  1
      X2
1  1  1  0  0  1  0  1  0  0  0

      ONESDEC X1
1  0  0  1  1  0  1  1  0  0  0
¯6.75
      TWOSDEC X
1  0  0  1  1  0  1  1  0  0  0
¯6.75
```

← intermediate value; sign-magnitude of -6.75

← the intermediate value (sign-mg.) of ¯6.75 is converted as 1's compl. — X1.

← X1 is now first converted into sign-magnitude
← then converted to decimal-equivalent.

Appendix 8 :

APL FUNCTIONS FOR THE ALGORITHMIC REPRESENTATION OF COMPUTER ARITHMETIC

APL functions for the Algorithmic Representation of Computer Arithmetic

(origin zero is assumed)

CONV	Converts an integer N in base R[0] to its representation in base R[1]
FCONV	Converts N the representation of a fraction in base R[0] to it representation in base R[1]
MCONV	Converts N the representation of a mixed number form base R[0] to its representation in base R[1] A mixed number is a real number, consisting of an integer part and a fractional part. These integer & fractional parts may be of different bases.
IMCONV	As MCONV, except that the left argument is such that R[0 1] are the bases associated with the integral part and R[2 3] with the fractional part
DECBCD	Converts the representation N of an integer in base 10 to its binary coded decimal (BCD) representation
BCDEC	Converts the BCD representation N of an integer to its representation in base 10
BADD	Serially add binary streams A & B with a carry vector K ; but overflow, when occur, of the last carry is lost. It is also assumed that there is no carry into the lowest order position. BADD may be interpreted as a serial adder.
BPLUS	Adds two binary numbers A & B in parallel; and may be interpreted as a parallel adder. Overflow is again lost; as for the BADD
MBPLUS	As BPLUS, except that overflow is retained.
PLUS	PLUS is similar to MBPLUS but performs the addition operation in any base R. The argument AB is a matrix of two rows, the first represents the left operand and the second represents the right operand of the addition operation.
TC	Converts a one's complement (as the right argument) to a two's complement representation
COMP	Computes the R complement of N by additing 1 to the R-1 complement of N.
RCOMP	Defines the R complement of N.

BMINUS	Binary substraction with arguments A & B being expressed in "sign and magnitude" representation; calling auxilary functions BSUB & MBPLUS
BSUB	An auxilary function of BMINUS
TCBMINUS	Performs "radix complement" substraction; calling auxilary function TCBSUB and MBPLUS
TCBSUB	An auxilary function of TCBMINUS
MINUS	Performs substraction in any base; similar to the function TCBMINUS and calls auxilary function SUB and fuction PLUS
SUB	An auxilary function of MINUS
BTIMES	Performs multiplication for the signed and magnitude representations. Function calls MBPLUS and the multiplication speed is increased by skipping an addition when the multiplier bit is zero
TIMES	TIMES is similar to BTIMES but performs the multiplication in any base R. The argument AB is a 2 X N matrix, the first row represents the left operand and the second row represents the right operand. The operands and the results are expressed in the signed and magnitude representations. TIMES calles an auxilary function DTIMES
DTIMES	An auxilary function of TIMES, it multiplies a digit which is represented by the second row of the two rows matrix AB.
FLOAT	Normalized floating point Add/Substract in radix R
MULTFLOAT	Normalized floating-point multiplication in radix R
RNSPT	Generates a residue weight table; calling function RNS
RNS	An auxilary function of RNSPT

(see also section of "APL functions for Binary-arithmetic")

Algorithmic Representation of Computer Arithmetic

(assume origin zero)

Base (or radix) conversion

```
      ∇Z←R FCONV N;DEC
[1]    DEC←(÷R[0])⊥0,N
[2]    Z←10
[3]   L:Z←Z,⌊DEC×R[1]
[4]    DEC←(DEC×R[1])-¯1↑Z
[5]    →(0≠DEC)/L
       ∇
```

```
      ∇Z←R MCONV N;F;I
[1]    I←R CONV N[0;]
[2]    F←R FCONV N[1;]
[3]    Z←((-(⍴F)⌈⍴I)↑I),[-0.5]((⍴F)⌈⍴I)↑F
       ∇
```

```
      ∇Z←R IMCONV N;I;F
[1]    I←R[1 2] CONV N[0;]
[2]    F←R[2 3] FCONV N[1;]
[3]    Z←((-(⍴F)⌈⍴I)↑I),[-0.5]((⍴F)⌈⍴I)↑F
       ∇
```

```
      ∇Z←DECBCD N
[1]    Z←,⍉(4⍴2)⊤,N
       ∇
```

```
      ∇Z←BCDEC N
[1]    Z←2⊥⍉(((⍴N)÷4),4)⍴N
       ∇
```

```
      ∇Z←R CONV N;DEC
[1]    DEC←R[0]⊥N
[2]    Z←(((⌈R[1] ● 2⌈DEC)⍴R[1])⊤DEC
       ∇
```

```
      ∇Z←DECBCD N
[1]    Z←,⍉(4⍴2)⊤,N
       ∇
```

```
      ∇Z←BCDEC N
[1]    Z←2⊥⍉(((⍴N)(÷4),4)⍴N
       ∇
```

Addition

```
      ∇Z←A BADD B;I;K
[1]    Z←K←(I←⍴A)⍴0
[2]   L:→((I←I-1)≥0)/OUT
[3]    K[I-1]←(K[I]∧A[I])∨(K[I]∧B[I])∨A[I]∧B[I]
[4]    Z[I]←K[I]≠A[I]≠B[I]
[5]    →L
[6]   OUT:Z[0]←K[0]≠A[0]≠B[0]
       ∇
```

```
      ∇Z←A BPLUS B;K;TEMP
[1]    Z←A≠B
[2]    K←A∧B
[3]   L:→(∼∨/K)/0
[4]    Z←(TEMP←Z)≠K←1↓K,0
[5]    K←K∧TEMP
[6]    →L
       ∇
```

```
      ∇Z←A MBPLUS B;K;N;OV;TEMP
[1]    N←(⍴,A)⌈⍴,B
[2]    Z←(A←(-N)↑A)≠B←(-N)↑B
[3]    K←A∧B
[4]    OV←0
[5]   L:→(0=∨/K)/OUT
[6]    OV←OV∨1↑K
[7]    Z←(TEMP←Z)≠K←1↓K,0
[8]    K←K∧TEMP
[9]    →L
[10]  OUT:Z←(1+⍴OV),Z
       ∇
```

```
      ∇Z←R PLUS AB;K;OV
[1]    Z←+/[0]AB
[2]    OV←0
[3]   L:Z←Z-R×K←Z≥R-1
[4]    OV←OV∨1↑K
[5]    →(0=+/K←1↓K,0)/OUT
[6]    Z←K+Z
[7]    →L
[8]   OUT:Z←(1+⍴OV),Z
       ∇
```

Substraction

```
      ∇Z←TC A;I
[1]    I←¯1↑A/⍳⍴A
[2]    Z←(~A[⍳I]),A[I+⍳(⍴A)-I]
       ∇

      ∇Z←R COMP N
[1]    Z←((⍴N)⍴R-1)-N
       ∇

      ∇Z←R RCOMP N
[1]    Z← R PLUS (R COMP N),[-0.5](-⍴N)↑1
       ∇

      ∇Z←A BMINUS B
[1]    →((1↑A)≠1↑B)/OPP
[2]    Z←A BSUB B
[3]    Z[0]←Z[0]≠A[0]
[4]    →0
[5]   OPP:Z←(1↑A),(1↓A) MBPLUS 1↓B
       ∇

      ∇Z←A BSUB B;OV
[1]    Z←(1↓A) MBPLUS ~(1-(⍴B)⌈⍴A)↑1↓B
[2]    OV←(⍴Z)=(⍴A)⌈⍴B
[3]    Z←(~OV),OV=(OV↓Z) MBPLUS OV
       ∇

      ∇Z←A TCBMINUS B
[1]    →((1↑A)≠1↑B)/OPP
[2]    Z←A TCBSUB B
[3]    Z[0]←Z[0]≠A[0]
[4]    →0
[5]   OPP:Z←(1↑A),(1↓A) MBPLUS 1↓B
       ∇

      ∇Z←R MINUS AB
[1]    →(≠/⍴AB[;0])/OPP
[2]    Z←R SUB (0 0),[1]0 1↓AB[(∧/AB[;0]),∧/AB[;0];]
[3]    →0
[4]   OPP:Z←AB[0;0], R PLUS 0 1↓AB
       ∇
      ∇Z←A TCBSUB B;OV
[1]    Z←(1↓A) MBPLUS TC(1-(⍴B)⌈⍴A)↑1↓B
[2]    OV←(⍴Z)=(⍴A)⌈⍴B
[3]    →OV/L
[4]    Z←1,TC Z
[5]    →0
[6]   L:Z←0,1↓Z
       ∇
      ∇Z←R SUB AB;OV
[1]    Z←R PLUS (1↓AB[0⍳;]),[-0.5] R RCOMP 1 ↓AB[1⍳;]
[2]    OV←(1↓⍴AB)=⍴Z
[3]    →OV/L
[4]    Z←1,R RCOMP Z
[5]    →0
[6]   L:Z←0,1↓Z
       ∇
```

Multiplication

```
      ∇Z←R TIMES AB;T;COUNT;T1;MD;M
[1]    T←⍬T1⍴1),((1↓⍴M)↓T1←0≠,M←0 1↓AB)⍴1
[2]    MD←(T[0]⊥T[1])↓M[T[0]⊤T[1];]
[3]    Z←((1+⍴MD)⍴0),(T[0]⌈T[1])↓M[T[0]<T[1];]
[4]    COUNT←(1↓⍴AB)-1+T[0]⌈T[1]
[5]   L:→(0>COUNT←COUNT-1)/OUT
[6]    →(0=¯1↑Z)/SHIFT
[7]    Z[⍳1+⍴MD]←R PLUS Z[⍳1+⍴MD],[-0.5](-1+⍴MD)
        ↑R BTIMES MD,[-0.5](-⍴MD)↑¯1↑Z
[8]    SHIFT;Z←0,¯1↓Z
[9]    →L
[10]  OUT;Z←((1↑AB[0;])≠1↑AB[1;]),((Z≠0)⍴1)↓Z
       ∇

      ∇PD←R BTIMES AB;I;MD;TP;SUM;K
[1]    PD←⍳0
[2]    K←0
[3]    I←⍴MD←,AB[0;]
[4]   L:→(0)I←I-1)/OUT
[5]    TP←(10,R) CONV (R⊥¯1↑AB[1;])×R⊥¯1↑MD
[6]    TP←R PLUS TP,[-0.5] (-⍴TP)↑K
[7]    PD←(¯1↑TP),PD
[8]    K←(2=⍴TP)↑TP
[9]    MD←¯1↓MD
[10]   →L
[11]  OUT;PD←K ,PD
       ∇

      ∇AQ←A BTIMES B;COUNT
[1]    AQ←((⍴A)⍴0),1↓B
[2]    COUNT←⍴B
[3]    L;→(0=COUNT←COUNT-1)/OUT
[4]    →((¯1↑AQ)=0)/SHIFT
[5]    AQ[⍳⍴A]←AQ[⍳⍴A] MBPLUS 1↓A
[6]    SHIFT;AQ←0,¯1↓AQ
[7]    →L
[8]    OUT;AQ←((1↑A)≠1↑B),(AQ⍴1)↓AQ
       ∇
```

```
     ∇ A FLOAT B
[1]    'ENTER RADIX OF SYSTEM: '
[2]    R←⍞
[3]    'ENTER MAXIMUM VALUE OF CHARACTERISTIC:'
[4]    K←⍞
[5]    C←F←3⍴0
[6]    C[0]←⌊A
[7]    V[1]←⌊B
[8]    F[0]←A-C[0]
[9]    F[1]←B-C[1]
[10]   D←C[1]-C[0]
[11]   X←D<0
[12]   F[X]←F[X]×R*-|D
[13]   'FOR ADDITION, ENTER ADD; FOR SUBSTRACTION, ENTER SUB'
[14]   →(∧/'ADD'=XXX←⍞)/AD
[15]   →(0=0×F[2]←F[0]-F[1])/AD1
[16] AD:F[2]←F[0]+F[1]
[17] AD1:C[2]←C[~X]
[18]   →(1≤|F[2])/LP
[19]   →(0≠|F[2])/GR
[20]   →(0=C[2]←0)/END
[21] GR:→((R*¯1)≤|F[2])/END
[22]   →(0>C[2]←C[2]-1)/UF
[23]   →(0=0×F[2]←F[2]×R)/GR
[24] LP:→(D<C[2]←C[2]+1)/OF
[25]   F[2]←F[2]×R*¯1
[26] END:'RESULT:    ';C[2]+F[2]
[27]   →0
[28] UF:'UNDERFLOW CONDITION HAS BEEN RAISED'
[29]   →0
[30] OF:'OVERFLOW CONDITION HAS BEEN RAISED'
     ∇


       67.4536  FLOAT  65.8796
ENTER RADIX OF SYSTEM:
⍞:
       10
ENTER MAXIMUM VALUE OF CHARACTERISTIC:
⍞:
       99
FOR ADDITION, ENTER ADD; FOR SUBSTRACTION, ENTER SUB
ADD
RESULT:    67.462396


       67.4536  FLOAT  65.8796
ENTER RADIX OF SYSTEM:
⍞:
       10
ENTER MAXIMUM VALUE OF CHARACTERISTIC:
⍞:
       99
FOR ADDITION, ENTER ADD; FOR SUBSTRACTION, ENTER SUB
SUB
RESULT:    67.444804
```

```
     ∇ A MULTFLOAT B
[1]    'ENTER RADIX OF SYSTEM:'
[2]    R←⍞
[3]    'ENTER MAXIMUM VALUE OF CHARACTERISTIC'
[4]    K←⍞
[5]    C←F←3⍴0
[6]    C[0]←⌊A
[7]    C[1]←⌊B
[8]    F[0]←A-C[0]
[9]    F[1]←B-C[1]
[10]   F[2]←F[0]×F[1]
[11]   →(0=F[2])/ZERO
[12]   C[2]←C[0]+C[1]
[13]   →(C[2]≥3×(K+1)÷2)/OF
[14]   →(C[2]<(K+1)÷2(/UF
[15]   C[2]←⌈C[2]-(K+1)÷2
[16] LP:→(F[2]>R*¯1)/END
[17]   →(C[2]=0)/UF
[18]   F[2]←R×F[2]
[19]   →(0=0×C[2]←C[2]-1)/LP
[20] ZERO:C[2]←0
[21] END:'RESULT:  ';C[2]+F[2]
[22]   →0
[23] UF:'UNDERFLOW CONDITION HAS BEEN RAISED'
[24]   →0
[25] OF:'OVERFLOW CONDITION HAS BEEN RAISED'
     ∇


       72.4526  MULTFLOAT  59.7831
ENTER RADIX OF SYSTEM:
⍞:
       10
ENTER MAXIMUM VALUE OF CHARACTERISTIC
⍞:
       99
RESULT:    81.35443106
```

```
      )GRP RNSGP[□]▽
RNS     RNSPT   ΔFMT

      ▽RNS[□]▽
    ▽ Y←RNS /K KI
[1]    ' '
[2]    'ENTER MODULI  (VECTOR  FORM)'
[3]    'MI←□'
[4]    M←⍴MI
[5]    WT←2*X
[6]    Y←(M,(⍴X))⍴0
[7]    I←0
[8]   LOOP:Y[I;]←MI[I]|WT
[9]    I←I+1
[10]   →(I<M)/LOOP
[11]   ' '
    ▽

      ▽RNSPT[□]▽
    ▽ RNSPT;MI;Y;TABLE;X
[1]   ⍝   GENERATES RESIDUE WEIGHT TABLE
[2]   ⍝       ** REQUIRES FUNCTIONS:
[3]   ⍝              RNS, ΔFMT
[4]   ⍝       ** ASSUMES 0-ORIGIN
[5]    ' '
[6]    ' ENTER WEIGHTS AS VECTOR, I.E. A←<NUMBER>;  '
[7]    X←□
[8]    TABLE←RNS X
[9]    ' '
[10]   '   ***  TABLE OF RESIDUE WEIGHTS  *** '
[11]   ' '
[12]   ' '
[13]   '   MOD.      RESIDUE WEITHTS (POWERS OF 2)  '
[14]   ' ------------------------------------------------------'
[15]   ' '
[16]   'X2,I3,A2,40I4' ΔFMT(MI;'  ';TABLE)
[17]   ' '
    ▽
```

```
      RNSPT

 ENTER WEIGHTS AS VECTOR, I.E. A←<NUMBER>:
□:
      ⍳16

ENTER MODULI  (VECTOR  FORM)
□:
      2+⍳31


   ***  TABLE OF RESIDUE WEIGHTS  ***

   MOD.      RESIDUE WEITHTS (POWERS OF 2)
 ------------------------------------------------------

  2     1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  3     1  2  1  2  1  2  1  2  1  2  1  2  1  2  1  2  1  2
  4     1  2  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  5     1  2  4  3  1  2  4  3  1  2  4  3  1  2  4  3  1  2
  6     1  2  4  2  4  2  4  2  4  2  4  2  4  2  4  2  4  2
  7     1  2  4  1  2  4  1  2  4  1  2  4  1  2  4  1  2  4
  8     1  2  4  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  9     1  2  4  8  7  5  1  2  4  8  7  5  1  2  4  8  7  5
 10     1  2  4  8  6  2  4  8  6  2  4  8  6  2  4  8  6  2
 11     1  2  4  8  5 10  9  7  3  6  1  2  4  8  5 10  9  7
 12     1  2  4  8  4  8  4  8  4  8  4  8  4  8  4  8  4  8
 13     1  2  4  8  3  6 12 11  9  5 10  7  1  2  4  8  3  6
 14     1  2  4  8  2  4  8  2  4  8  2  4  8  2  4  8  2  4
 15     1  2  4  8  1  2  4  8  1  2  4  8  1  2  4  8  1  2
 16     1  2  4  8  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 17     1  2  4  8 16 15 13  9  1  2  4  8 16 15 13  9  1  2
 18     1  2  4  8 16 14 10  2  4  8 16 14 10  2  4  8 16 14
 19     1  2  4  8 16 13  7 14  9 18 17 15 11  3  6 12  5 10
 20     1  2  4  8 16 12  4  8 16 12  4  8 16 12  4  8 16 12
 21     1  2  4  8 16 11  1  2  4  8 16 11  1  2  4  8 16 11
 22     1  2  4  8 16 10 20 18 14  6 12  2  4  8 16 10 20 18
 23     1  2  4  8 16  9 18 13  3  6 12  1  2  4  8 16  9 18
 24     1  2  4  8 16  8 16  8 16  8 16  8 16  8 16  8 16  8
 25     1  2  4  8 16  7 14  3  6 12 24 23 21 17  9 18 13  3
 26     1  2  4  8 16  6 12 24 22 18 10 20 14  2  4  8 16  6
 27     1  2  4  8 16  5 10 20 13 26 25 23 19 11 22 17  7 14
```

Appendix 9 :

SIMULATION OF A SIMPLE 8-BIT PROGRAMMABLE MACHINE — BCOMP8 USING APL[1]

---

[1] Extracted form : P.K.M.LAI,"Simulation of a Small 8-bit Digital Computer using APL/360", Internal report, Electrical Engineering Department, Imperial College, 25th September,1975.

Simulation of a Small 8-bit Digital Computer

An 8-bit digital computer called BCOMP8 is designed and simulated to illustrate and to investigate the techniques of simulation using APL. The topic to be discussed are:

1) Basic Specification of BCOMP8,
2) The Instruction format and Repertoire,
3) The Architecture of BCOMP8,

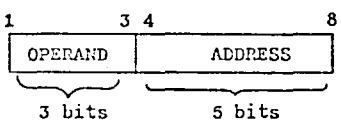
and 4) Simulation using APL.

1) Basic Specification of BCOMP8

Word length	=	8 bits,
Number of Instruction	=	8,
Memory capacity	=	32 words,
Number of registers	=	3 (viz. IC,IR,A)
Arithmetic unit	=	A simple binary adder

2) Instruction Format and Repertoire

i) Instruction format : single address



ii) Instruction Repertoire

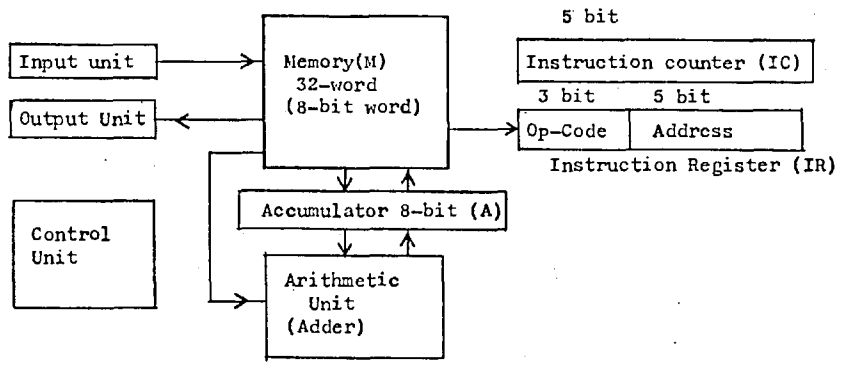
Mnemonic	Binary Code	Description
LDA	0 0 0	Load A from memory according to addess
STA	0 0 1	Store A to memory location .....
ETM	0 1 0	Enter from "input" to memory .....
OPM	0 1 1	Output from memory locaction .....
PMA	1 0 0	Add from memory (addressed) to A
CIA	1 0 1	Place constant in A from absolute value of the address
TEM	1 1 0	Terminate program (of the BCOMP8)
BRT	1 1 1	Branch to program according to the address part of the instruction if (A) ≠ M[31;]

Note :

(1) A designates the accumulator
(2) Instruction BRT assumes the loop-parameter being placed in location 31 of the memory (M). This also assumes that the memory locations are labelled from zero to 31. The program sequence or instructions for the computer BCOMP8 is also labelled from zero to 31.

3) Architecture of the computer BCOMP8

BCOMP8 consists of one arithmetic unit (a simple adder), a memory of 32 words (8-bits each), an accumulator (A), an instruction counter (IC), and an instruction register (IR). The organization is shown below.



Restrictions of this organization

1) Memory buffer is not shown or incooperated
2) Control unit although assumed; but is functionally simulated
3) Arithmetic unit consists of only an adder
4) In the simulation, hence the organization, branching only permitted to one point in the program (of the BCOMP8, not APL) while the test of branch is made with contents of the memory location 31.

The implication is that only one loop is permitted in the organization. The index variable to be tested against is required to be placed in location 31.

The memory is partitioned into program part and data part; with the former occupying the upper part of the memory and latter residing in the lower part of the memory.

4) Simulation using APL

The simulation of BCOMP8 is at the functional level; i.e. the transfer of data and the behaviour of the computer are of interest. The suite of APL programs written to simulate BCOMP8 may be grouped as follows:

i) BCOMP8 — the machine itself,
and ii) ASSEMBER — an assembler for BCOMP8.

The assembler accepts programs(written for the BCOMP8) written in mnemonic codes and translate them into binary codes so that they may be loaded into BCOMP8 for simulation. There are two versions for the assember; the newer version has the capability of reconverting the contents of BCOMP8 into mnemonic codes. This, however, requires information of the symbol table which is printed during the first time the BCOMP8 program is assembled. Other information is stored as variables (globals) in the APL workspace.

i) BCOMP8 suite

This consists of the function BCOMP8 and PLUS (which is called by BCOMP8). Both functions assumes orgin zero. BCOMP8 has the following syntax:

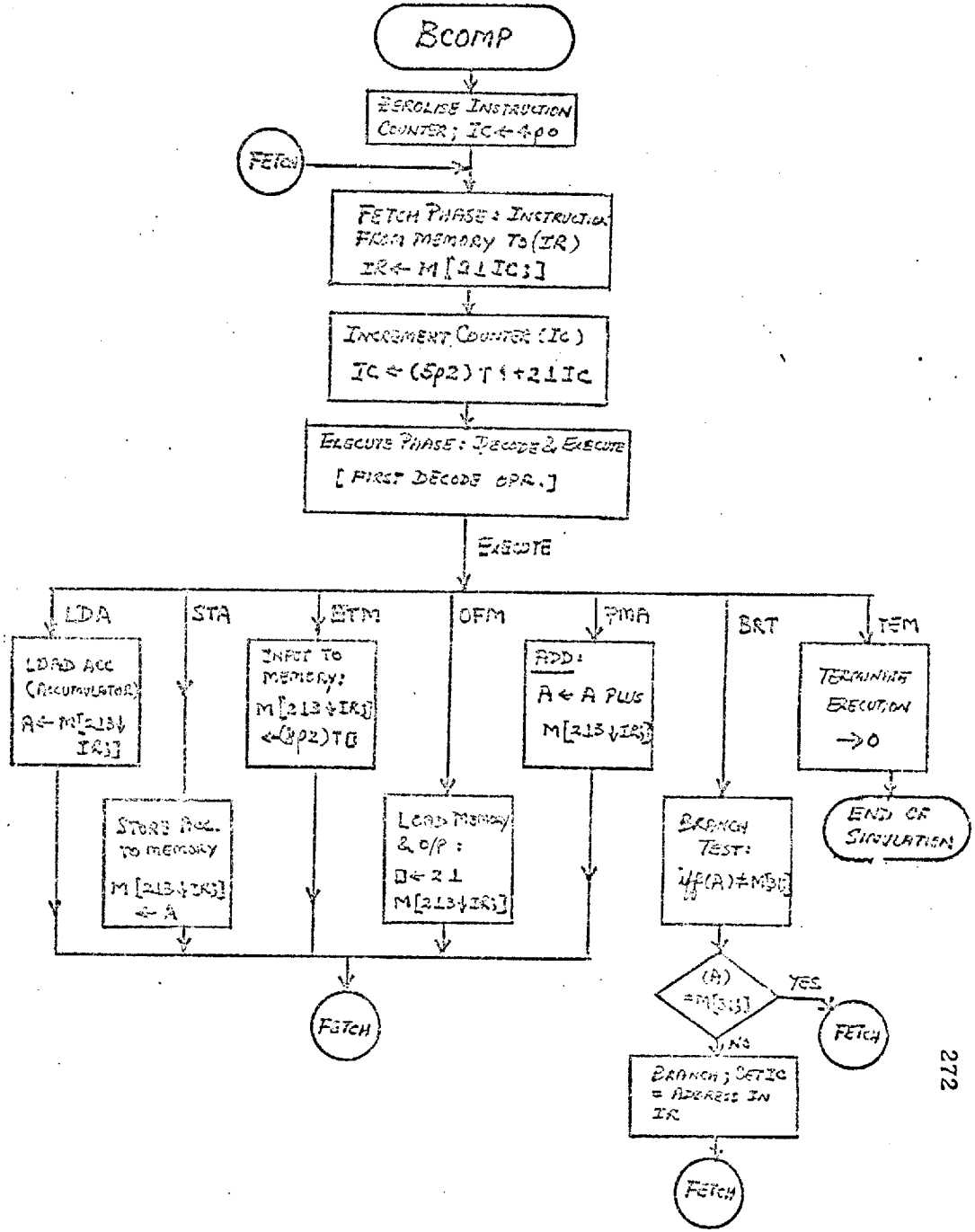
BCOMP8 M

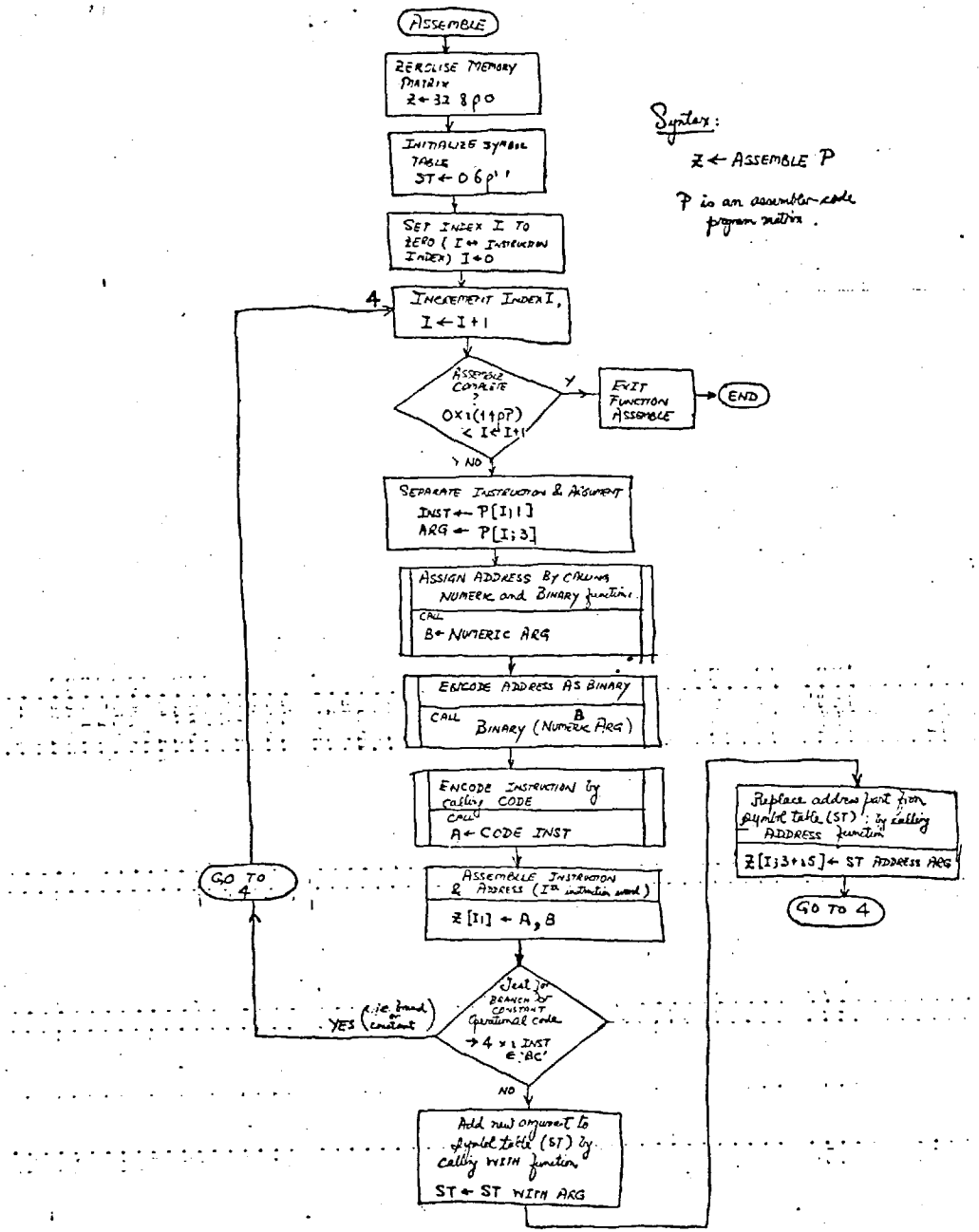
where M is any 32 by 8 matrix of 0 and 1 (i.e. boolean matrix) corresponding to the contents of the memory BCOMP8 computer. BCOMP8 is executed in two phases:

(a) FETCH phase : transferring data from memory M into the instruction register IR and advancing the instruction counter by one for the next fetch phase

(b) EXECUTION phase : the IR contents are decoded and executed, a test for "branching" out of the program sequence is made and executed. Program execution for BCOMP8 may be terminated by performing TEM in this execution phase.

Flow charts for the main function of BCOMP8 and ASSEMBLER are shown below.





Syntax:

Z ← ASSEMBLE P

P is an assembler-code program matrix.

```
      ⍝ LISTING OF AN 8-BIT BINARY COMPUTER  **BCOMP8**
      ⍝
      ⍝    ORIGIN-0 INDEXING IS ASSUMED
      ⍝    FUNCTION  **PLUS** IS REQUIRED
      ⍝

      ∇BCOMP8[⎕]∇

    ∇ BCOMP8 M;IC;IR;A
[1]   IC←0 0 0 0 0
[2]   FETCH:IR←M[2⊥IC;]
[3]   IC←(5ρ2)⊤1+2⊥IC
[4]   ⍝   EXECUTE PHASE: DECODE + EXECUTE
[5]   →(2⊥3↑IR)⍎LDA,STA,ETM,OFM,PMA,CIA,TEM,BRT
[6]   LDA:→FETCH,A←M[2⊥3↑IR;]
[7]   STA:→FETCH,M[2⊥3↑IR;]←A
[8]   ETM:→FETCH,M[2⊥3↑IR;]←(8ρ2)⊤⎕
[9]   OFM:→FETCH,⎕←2⊥M[2⊥3↑IR;]
[10]  PMA:→FETCH,A←A PLUS M[2⊥3↑IR;]
[11]  CIA:→FETCH,A←0 0 0,3↓IR
[12]  TEM:→0
[13]  BRT:→(∧/A=M[31;])/FETCH
[14]  ⍝  SET COUNTER FOR BRANCHING
[15]  →FETCH,IC←3↑IR
    ∇


      ∇PLUS[⎕]∇


    ∇ Z←X PLUS Y
[1]   Z←(8ρ2)⊤(2⊥X)+2⊥Y
    ∇
```

```
⍝   ****  LISTING OF FUNCTIONS IN  **GROUP DIGITAL**   ****

                (ORIGIN-1 IS ASSUMED)

      ∇ASSEMBLE[⎕]∇
    ∇ Z←ASSEMBLE P;ARG;I;INST;ST
[1]   Z← 32 8 ⍴0
[2]   ST← 0 6 ⍴' '
[3]   I←0
[4]  NEXT:→((1↑⍴P)<I←I+1)/0
[5]   INST←P[I;1]
[6]   ARG←P[I;3]
[7]   Z[I;]←(CODE INST),BINARY NUMERIC ARG
[8]   →(INST∈'BC')/NEXT
[9]   ST←ST WITH ARG
[10]  Z[I;3+⍳5]←ST ADDRESS ARG
[11]  →NEXT
    ∇


      ∇CODE[⎕]∇
    ∇ Z←CODE X
[1]   Z← 2 2 2 ⊤¯1+'LSEPACTB'⍳X
    ∇


      ∇BINARY[⎕]∇
    ∇ Z←BINARY X
[1]   Z←(5⍴2)⊤X
    ∇

      ∇NUMERIC[⎕]∇
    ∇ Z←NUMERIC X
[1]   Z←¯1+'0123456789'⍳X
    ∇


      ∇WITH[⎕]∇
    ∇ Z←ST WITH NEW
[1]   Z←ST
[2]   →0×⍳∨/NEW=ST[;1]
[3]   Z←ST,[1] NEW,CHAR BINARY 31-(⍴ST)[1]
    ∇


      ∇ADDRESS[⎕]∇
    ∇ Z←ST ADDRESS X
[1]   Z←NUMERIC 1↓ST[ST[;1]⍳X;]
    ∇


      ∇CHAR[⎕]∇
    ∇ Z←CHAR X
[1]   Z←'0123456789'[1+X]
    ∇
```

```
⍝  ILLUSTRATION:  RUN OF PROGRAM P2 ON **BCOMP8**
⍝
⍝     NOTE:  SET ORIGIN TO ZERO INDEXING
⍝            ENTER THE VALUE OF (N) AT ⎕:


      )ORIGIN 0
WAS 0


      BCOMP8 MEM

⎕:
      6
1
1
2
3
5
8
```

Analysis Of Fibonacci Numbers Program

Program	X	Y	Z	A	V	Q	Comment
0 CO	–	–	–	0	–	–	A ← 0
1 S V				0	0		M<31> ← (A) where M<31> ≡ V
2 E Q						n_1	M<30> ← Q where Q is the number of Fno required
3 C 1				1			A ← 1
4 S X	1						M<29> ← (A) where M<29> ≡ X
5 S Y		1					M<28> ← (A) where M<28> ≡ Y
⑥ L X				1			A ← (M<29>)
7 A Y				2			A ← (A) + (Y) or M<28>
8 S Z			2				M<27> ← (A) where M<27> ≡ Z
9 P X							Print X which is F_i ≡ i-th Fibonacci Number
10 L Y				1			A ← (M<28>)
11 S X	1						M<29> ← (A) i.e. X ← Y by {L Y / S X}
12 L Z				2			A ← (M<27>)
13 S Y		2					M<28> ← (A) i.e. Y ← Z by {L Z / S Y}
14 C 1				1			A ← 1
15 A V				1			A ← (A) + V or M<31> (i.e. for index counter V is increment by one)
16 S V					1		M<31> ← (A)
17 L Q				n_1			i.e. is n_1 = V or is looping complete? yes → Exit, no return to ⑥
18 B 6				if n_1 = M<31>			Stop
19 T				Exit			Steps 17, 18 and 19 is a test for branching.

(A) of (M<31>)

If
(i) (A) = (M<31>) exist of program
(ii) (A) ≠ (M<31>) branch to program step 6

For this particular case we have
(A) = Q
(M<31>) = V

N.B.:
The computer has the structure that the test for jump is always against memory location 31. Hence the loop test is required to be stored first before storing any other variables.

New verison of BCOMP8 and ASSEMBLER

(a) ASSEMBLER : Prints the "symbol table" as a result of assembling a program of BCOMP8

(System function FMT is required)

(b) DUMPGP : The group contains the following functions. Note that BCOMP8 is modified as to permit a run-time program trace.

MDUMP — to convert the binary contents of BCOMP8 into mnonic codes (i.e. the reverse process of assembling is performed)

PDUMP — to dump the core and contents of other registers during the execution of BCOMP8T

BCOMP8T — to provide a trace of all contents of some registers and memory contents during the simulation run

DIGIT — an auxilary function called by MDUMP and PDUMP, to convert a numeric decimal number into a character vector

(some examples are included to illustrate the operations of the above functions)

```
⍝  FUNCTION **MDUMP**  TRANSLATES THE MEMORY CONTENTS
⍝           INTO ASSEMBLY LANGUAGE
⍝           DRAWBACK ⍞:  THE SYMBOLS USED FOR
⍝                        VARIABLES ARE LOST
⍝
⍝           FUNCTION REQUIRED ⍞:  DIGIT
⍝           ORIGIN-0 INDEX IS ASSUMED
```

```
∇MDUMP[⍞]∇

∇ MDUMP M;N;I;Z
[1]   ' '
[2]   N←1↑⍴M
[3]   I←0
[4]  LOOP:Z←'LSEPACTB'[2⊥3↑M[I;]],' ',2 DIGIT 2⊥3↓M[I;]
[5]   2 DIGIT I;'   ';M[I;];'   ';Z
[6]   →(N>I←I+1)↑LOOP
∇
```

```
∇DIGIT[⍞]∇

∇ Z←N DIGIT X
[1]   Z←'0123456789'[(N⍴10)⊤X]
∇
```

To re-translate or to examine the contents of the memory MEM which contains the translated assembled program P2 ; execute ⍎ program MDUMP.

```
      P2
      MDUMP MEM
```

```
C 0          00  1 0 1 0 0 0 0 0  C 00
S V          01  0 0 1 1 1 1 1 1  S 31
E Q          02  0 1 0 1 1 1 1 0  E 30
C 1          03  1 0 1 0 0 0 0 1  C 01
S X          04  0 0 1 1 1 1 0 1  S 29
S Y          05  0 0 1 1 1 1 0 0  S 28
L X          06  0 0 0 1 1 1 0 1  L 29
A Y          07  1 0 0 1 1 1 0 0  A 28
S Z          08  0 0 1 1 1 0 1 1  S 27
P X          09  0 1 1 1 1 1 0 1  P 29
L Y          10  0 0 0 1 1 1 0 0  L 28
S X          11  0 0 1 1 1 1 0 1  S 29
L Z          12  0 0 0 1 1 0 1 1  L 27
S Y          13  0 0 1 1 1 1 0 0  S 28
C 1          14  1 0 1 0 0 0 0 1  C 01
A V          15  1 0 0 1 1 1 1 1  A 31
S V          16  0 0 1 1 1 1 1 1  S 31
L Q          17  0 0 0 1 1 1 1 0  L 30
B 6          18  1 1 1 0 0 1 1 0  B 06
T            19  1 1 0 1 1 0 1 0  T 26
             20  0 0 0 0 0 0 0 0  L 00
             21  0 0 0 0 0 0 0 0  L 00
             22  0 0 0 0 0 0 0 0  L 00
             23  0 0 0 0 0 0 0 0  L 00
             24  0 0 0 0 0 0 0 0  L 00
             25  0 0 0 0 0 0 0 0  L 00
             26  0 0 0 0 0 0 0 0  L 00
             27  0 0 0 0 0 0 0 0  L 00
             28  0 0 0 0 0 0 0 0  L 00
             29  0 0 0 0 0 0 0 0  L 00
             30  0 0 0 0 0 0 0 0  L 00
             31  0 0 0 0 0 0 0 0  L 00
```

Assembler program which was assembled.

```
⍝     NEW VERSION OF  ** ASSEMBLE **  FUNCTION
⍝           ** SYMBOL TABLE - MEMORY ALLOCATION
⍝           ** ARE PRINTED.
```

```
∇ASSEMBLE[⍞]∇

∇ Z←ASSEMBLE P;ST;ARG;I;INST
[1]   Z← 32 8 ⍴0
[2]   ST← 0 6 ⍴' '
[3]   I←0
[4]  NEXT:→((1↑⍴P)<I←I+1)/PRINT
[5]   INST←P[I;1]
[6]   ARG←P[I;3]
[7]   Z[I;]←(CODE INST),BINARY NUMERIC ARG
[8]   →(INST∈'BC')/NEXT
[9]   ST←ST WITH ARG
[10]  Z[I;3+⍳5]←ST ADDRESS ARG
[11]  →NEXT
[12] PRINT:' ASSEMBLE COMPLETED, THE SYMBOL TABLE IS:'
[13]  ' '
[14]  'SYM.    LOC.'
[15]  ' '
[16]  'X2,A1,X4,I2' ⎕FMT(ST[;1];2↓⍉NUMERIC ST[;2 3 4 5 6])
∇
```

```
⍝ ILLUSTRACTION OF **ASSEMBLE** RUN
```

```
      P2
C 0
S V
E Q
C 1
S X
S Y
L X
A Y
S Z
P X
L Y
S X
L Z
S Y
C 1
A V
S V
L Q
B 6
T
```

Assembler Program P2 To Be "Assembled" Into Binary Codes For The BComp8 Computer.

Assembling P2 and Loaded Into Memory MEM

```
      MEM←ASSEMBLE P2
 ASSEMBLE COMPLETED, THE SYMBOL TABLE IS:

SYM.    LOC.

  V      31
  Q      30
  X      29
  Y      28
  Z      27
         26
```

```
      ⍝ GROUP DUMPGP --- ORIGIN-0 IS ASSUMED

      )GRP DUMPGP
MDUMP   PDUMP   BCOMP8TΔ        DIGIT


      ∇BCOMP8TΔ[⎕]∇

    ∇ BCOMP8TΔ M;IC;IR;A;ADDR;Z
[1]   A←IC← 0 0 0 0 0
[2]  FETCH:IR←M[2⊥IC;]
[3]   IC←(5⍴2)⊤1+2⊥IC
[4]   ADDR←2⊥3↑IR
[5]   ⎕←Z←PDUMP IR
[6]   ⍝ EXECUTE PHASE: DECODE + EXECUTE
[7]   →(2⊥3↑IR)⌽LDA,STA,ITM,OFM,AMA,CIA,TEM,BRT
[8]  LDA:→FETCH,A←M[ADDR;]
[9]  STA:→FETCH,M[ADDR;]←A
[10] ITM:→FETCH,M[ADDR;]←(8⍴2)⊤⎕
[11] OFM:→FETCH,⎕←2⊥M[ADDR;]
[12] AMA:→FETCH,A←A PLUS M[ADDR;]
[13] CIA:→FETCH,A← 0 0 0 ,3↑IR
[14] TEM:→0
[15] BRT:→(∧/A=M[31;])/FETCH
[16]  ⍝ SET COUNTER FOR BRANCHING
[17]  →FETCH,IC←3↓IR
    ∇


      ∇PDUMP[⎕]∇

    ∇ Z←PDUMP IR;OP
[1]   OP←2⊥3↑IR
[2]   Z←2 DIGIT ¯1+2⊥IC
[3]   Z←Z,'  ','LSEPACTB'[OP],' ',2 DIGIT 2⊥3↑IR
[4]   Z←Z,' ',3 DIGIT 2⊥M[ADDR;]
[5]   Z←Z,' ',3 DIGIT 2⊥A
    ∇


      ∇DIGIT[⎕]∇

    ∇ Z←N DIGIT X
[1]   Z←'0123456789'[(N⍴10)⊤X]
    ∇


      ∇PLUS[⎕]∇

    ∇ Z←X PLUS Y
[1]   Z←(8⍴2)⊤(2⊥X)+2⊥Y
    ∇
```

```
      ⍝ EXAMPLE SHOWING THE SIMULATION RUN OF COMPUTER **BCOMP8**
      ⍝
      ⍝  PROGRAM TO FIND THE FIRST N NUMBER OF THE
      ⍝          **FIBONACCI NUMBER SEQUENCE**
      ⍝


      ⍝ THE ASSEMBLER PROGRAM **P2** (FORMAT ENTERED AT KEYBOARD)


      P2
C 0S VE QC 1G XS YL XA YS ZP XL YS XL ZS YC 1A VS VL QB 6T


      ⍝ CONVERTING THIS INTO A MATRIX

      P2←20 3⍴P2

      P2
C 0
S V
E Q
C 1
S X
S Y
L X
A Y
S Z
P X
L Y
S X
L Z
S Y
C 1
A V
S V
L Q
B 6
T


      ⍝ ASSEMBLING P2 AND LOADING IT INTO MEMORY CALLED MEM
      ⍝       N.B.
      ⍝             )ORIGIN-1  IS SET

      )ORIGIN 1
WAS 1
      MEM←ASSEMBLE P2
 ASSEMBLE COMPLETED. THE SYMBOL TABLE IS:

 SYM.  LOC.

V   31
Q   30
X   29
Y   28
Z   27
    26
```

```
BCOMP8TA MEM
00  C 00 160 000
01  S 31 000 000
02  E 30 000 000
[]:
      3
03  C 01 063 000
04  S 29 000 001
05  S 28 000 001
06  L 29 001 001
07  A 28 001 001
08  S 27 000 002
09  P 29 001 002
1
10  L 28 001 002
11  S 29 001 001
12  L 27 002 001
13  S 28 001 002
14  C 01 063 002
15  A 31 000 001
16  S 31 000 001
17  L 30 003 001
18  B 06 029 003
06  L 29 001 003
07  A 28 002 001
08  S 27 002 003
09  P 29 001 003
1
10  L 28 002 003
11  S 29 001 002
12  L 27 003 002
13  S 28 002 003
14  C 01 063 003
15  A 31 001 001
16  S 31 001 002
17  L 30 003 002
18  B 06 029 003
06  L 29 002 003
07  A 28 003 002
08  S 27 003 005
09  P 29 002 005
2
10  L 28 003 005
11  S 29 002 003
12  L 27 005 003
13  S 28 003 005
14  C 01 063 005
15  A 31 002 001
16  S 31 002 003
17  L 30 003 003
18  B 06 029 003
19  T 26 000 003
```

BCOMPST providing a "trace" of computer BCOMP8 during execution of program P2

```
BCOMP8 MEM
[]:
      3
1
1
2
```

The execution of program "MEM" [P2 → compiled and located in memory MEM] without any "trace" output.

Appendix 10 :

COMPILING COMPOUND EXPRESSIONS WITH APL

Compiling a Compound Expression

This exercise demonstrates the capability of APL as a useful tool in examining the details of a compound expression. A compound expression is first decomposed into smaller units indicating the sequence of executions for the expression (i.e. the expression is parsed). This results a tree-like structrue representing the compound expression. A parenthesis-free expression or a polish string is then derived from this tree structure. Finally, the polish string is expressed as an ordered sequence of simple subexpression (dyadic form at most); while the execution of which will have the same effect as if the original compound expression was executed. Therefore, a compound expression has been compiled for a simple machine which has the capability of operating at the most simple dyadic expressions.

A set of APL functions capable of compiling a compond expression are included below with some illustrative examples. The functions are divided into three groups; viz. COMPILEGP, PARSEGP & POLISHGP. The functions of these groups are:

PARSEGP	Parses a given APL expression into sub-expressions which are represented by a tree-like structure or diagram
POLISHGP	Convert the tree-structure into a polish string
COMPILEGP	Order the polish string into simple dyadic expression and introducing intermediate variables when necessary. This results a compiled set of simple dyadic subfunctions which are equivalent to the original compound expression

⍝ *LISTING OF ALL FUNCTION GROUPS*

```
      LISTGRP
GRP NAME: COMPILEGP
------------------------------
CENTER
COMPILE
FUNCTIONS
LEFT
LOCCENTER
ON
RIGHT

    ∇ Z←CENTER E
[1]   Z←(LOCCENTER E)/E
    ∇


    ∇ Z←COMPILE E;CE
[1]   CE←CENTER E
[2]   Z←(((' '←'∈CE)⍱3≥⍴E)/NAMES[1],' '←'),CE[2 1 3]
[3]   NAMES←1↓NAMES
[4]   →0×⍳3≥⍴E
[5]   Z←Z ON COMPILE(LEFT E),Z[1],RIGHT E
    ∇


    ∇ Z←FUNCTIONS E
[1]   Z←E∈'←+-×÷<≤=≥>≠∨∧?∈⍴~↑↓⍳○*⍟⌈⌊⊥⊤|'
    ∇


    ∇ Z←LEFT E
[1]   Z←(~∨\LOCCENTER E)/E
    ∇


    ∇ Z←LOCCENTER E
[1]   Z←(⍳⍴E)∈0 1 2+(FUNCTIONS E)⌈.×⍳⍴E
    ∇


    ∇ Z←A ON B
[1]   A←(¯2+1 1,⍴A)⍴A
[2]   B←(¯2+1 1,⍴B)⍴B
[3]   Z←(((⍴A)⌈0 1×⍴B)↑A),[1]((⍴B)⌈0 1×⍴A)↑B
    ∇


    ∇ Z←RIGHT E
[1]   Z←LOCCENTER E
[2]   Z←(~Z∨∧\~Z)/E
    ∇
```

```
      LISTGRP

GRP NAME: PARSEGP

------------------------------

C
CENTRALFN
DEPTH
FUNCTIONS
L
ON
PARSE
R
STRIP
```

```
     ∇ Z←C E
[1]    Z←E[CENTRALFN E]
     ∇
```

```
     ∇ Z←CENTRALFN E
[1]    Z←((FUNCTIONS E)∧0=DEPTH E)⍳1
     ∇
```

```
     ∇ Z←DEPTH E
[1]    Z←+\(E='(')-0,¯1↓E=')'
     ∇
```

```
     ∇ Z←STRIP E
[1]    →0×⍳1≠⌊/DEPTH Z←E
[2]    Z←STRIP 1 ↓¯1↓E
     ∇
```

```
     ∇ Z←FUNCTIONS E
[1]    Z←E∈'←+-×÷<≤=≥>≠∨∧?∈⍴~↑↓⍳○*⍟⌈⌊⊥⊤|'
     ∇
```

```
     ∇ Z←L E
[1]    Z←(¯1↑CENTRALFN E)↑E
     ∇
```

```
     ∇ Z←A ON B
[1]    A←(¯2↑1 1,⍴A)⍴A
[2]    B←(¯2↑1 1,⍴B)⍴B
[3]    Z←(((⍴A)⌈0 1×⍴B)↑A),[1]((⍴B)⌈0 1×⍴A)↑B
     ∇
```

```
     ∇ Z←PARSE E
[1]    →0×⍳∧/~FUNCTIONS Z←STRIP E
[2]    Z←(' ',' ',PARSE L Z)ON(C Z)ON' ',' ',PARSE R Z
     ∇
```

```
     ∇ Z←R E
[1]    Z←(CENTRALFN E)↓E
     ∇
```

```
      LISTGRP

GRP NAME: POLISHGP

------------------------------

CT
FIRSTCOL
LT
POLISH
RT
```

```
     ∇ Z←CT M
[1]    →(1=⍴⍴Z←(' '≠FIRSTCOL M)/[1]M)/0
[2]    Z←,1 1↑Z
     ∇
```

```
     ∇ Z←FIRSTCOL M
[1]    Z←,((1↑⍴M),1)↑M
     ∇
```

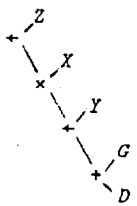
```
     ∇ Z←LT M
[1]    Z←(~∨\' '≠FIRSTCOL M)/[1]M
     ∇
```

```
     ∇ Z←POLISH M
[1]    Z←,CT M←(∨/[1]M≠' ')/M
[2]    →0×⍳1≥1↑⍴M
[3]    Z←Z,(POLISH LT M),POLISH RT M
     ∇
```

```
     ∇ Z←RT M
[1]    Z←(∨\¯1⌽' '≠FIRSTCOL M)/[1]M
     ∇
```

```
⍝ TEST FUNCTION
```

PARSE S4



POLISH PARSE S4

```
←Z×X←Y+GD
```

COMPILE POLISH PARSE S4

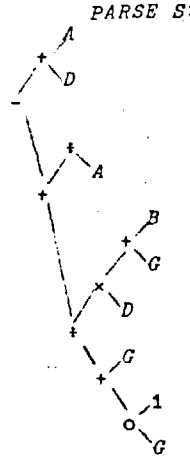
```
J←G+D
Y←J
L←X×Y
Z←L
```

```
⍝ ILLUSTRATE S3
```

S3

```
(A+D)-(⍎A)+(⍟B+G)×D)⍎G+1⍟G
```

PARSE S3



POLISH PARSE S3

```
-+AD+⍎ A⍎×+BGD+G⍟1G
```

COMPILE POLISH PARSE S3

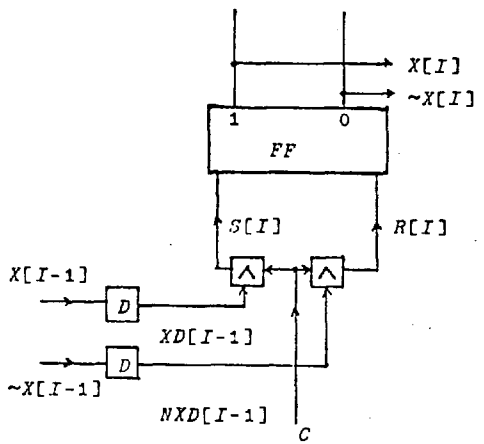
```
N←1⍟G
Q←G+N
P←B+G
Q←P×D
R←Q⍎Q
S←  ⍎A
T←S+R
U←A+D
U-T
```

Appendix 11 :

ARCHITECTURAL DESCRIPTION & SIMULATION

-- USE OF APL AT THE REGISTER-TRANSFER LEVEL

Shift Registers



One cell of shift register for right shift

Microprogram for right shift by K positions

```
     ∇ K RSHIFT X
[1]    P←X
[2]    T←∨/P
[3]   TST:→(~T)/0
[4]    RSHIFT1 X
[5]    P←(6ρ2)⊤¯1+2⊥P
[6]    →TST
       ∇

     ∇ RSHIFT1 X
[1]    XD←X
[2]    NXD←~X
[3]    S←C∧0,¯1↓XD
[4]    R←C∧1,¯1↓NXD
       ∇
```

Decoder circuits

```
     ∇ R←SDECN X
[1]   ⍝ SIMPLEX DECODER
[2]    R←X∧.=((⍴,X)ρ2)⊤⍳2*⍴,X
       ∇
```

(Simplex decoder; an architectural description)

```
     ∇OUR←SDEC2 IN
[1]   OUT←IN∧.=2 2⊤⍳4
       ∇
```

(A 2-input simplex decoder; an implementation description)

Matrix decoders :

```
     ∇ OUT←DEC IN;ADR
[1]   ⍝ DECODING MATRIX, USING: INNER PRODUCT, COMPRESSION
[2]   ⍝     FOR SIMULATION.  READ-ONLY-MEMORY(ROM) IS A
[3]   ⍝     GLOBAL VARIABLE
[4]   ⍝Δ ROM(N,M)
[5]    ADR←IN∧.=(⍴IN)ρ⊤⍳ρIN
[6]    OUT←ADR/[0]ROM
       ∇
```

(Matrix decoder, using read-only-memory; simulation using inner product & compression)

```
     ∇ OUT←DEC1 IN;ADR
[1]   ⍝ DECODING MATRIX, USING: DECODE OPERATOR FOR
[2]   ⍝     SIMULATION.  READ-ONLY-MEMORY(ROM) IS A
[3]   ⍝     GLOBAL VARIABLE
[4]   ⍝Δ ROM(N,M)
[5]    OUT←ROM[2⊥IN;]
       ∇
```

(Matrix decoder; simulation using indexing)

```
     ∇ OUT←MDEC8 IN
[1]   ⍝  MATRIX DECODER (8BIT)
[2]   ⍝ LEVEL 1 (SIMPLEX DECODER)
[3]    OUT10←SDEC2 IN[0 1]
[4]    OUT11←SDEC2 IN[2 3]
[5]    OUT12←SDEC2 IN[4 5]
[6]    OUT13←SDEC2 IN[6 7]
[7]   ⍝ LEVEL 2 (MATRIX AND-GATES)
[8]    OUT20←,OUT10 ∘.∧ OUT11
[9]    OUT21←,OUT12 ∘.∧ OUT13
[10]  ⍝ LEVEL 3 (MATRIX AND-GATES)
[11]   OUT←,OUT20 ∘.∧OUT21
       ∇
```

(An 8-input matrix decoder)

```
     ∇ RIPPLE4
[1]    ⍝ TRANSIMSSION AND GENERATION
[2]    T←A≠B
[3]    G←A∧B
[4]    ⍝ CARRY PROPAGATION
[5]    C←5⍴0
[6]    C[4]←CIN
[7]    C[3]←G[3]∨T[3]∧C[4]
[8]    C[2]←G[2]∨T[2]∧C[3]
[9]    C[1]←G[1]∨T[1]∧C[2]
[10]   C[0]←G[0]∨T[0]∧C[1]
[11]   ⍝ CARRY ASSIMILATION
[12]   S←T≠1↓C
[13]   COUT←1↑C
     ∇
```

(4-Bit Ripple Adder)

```
     ∇ RIPPLE;J
[1]    ⍝ TRANSMISSION AND GENERATION
[2]    T←A≠B
[3]    G←A∧B
[4]    ⍝ CARRY PROPAGATION
[5]    J←⍴T
[6]    C←(J+1)⍴0
[7]    C[J]←CIN
[8]    CONT:J←J-1
[9]    C[J]←G[J]∨T[J]∧C[J+1]
[10]   →(J≠0)/CONT
[11]   ⍝ CARRY ASSIMILATION
[12]   S←T≠1↓C
[13]   COUNT←1↑C
     ∇
```

(N-bit Ripple Adder)

```
     ∇ SKIP100;J
[1]    ⍝ TRANSMISSION AND GENERATION
[2]    T←A≠B
[3]    G←A∧B
[4]    ⍝ TRANSIMSSION FRO GROUPS OF 5
[5]    TG5←∧/20 5⍴T
[6]    ⍝ TRANSMISSION FOR GROUPS OF 25
[7]    TG25←∧/4 5⍴TG5
[8]    ⍝ CARRY PROPAGATION
[9]    J←⍴T
[10]   C←(J+1)⍴0
[11]   C[J]]CIN
[12]   CONT:J←J-1
[13]   →(0=5|J)/GROUP
[14]   C[J]←C[J]∨T[J]∧C[J+1]
[15]   →CONT
[16]   GROUP5:→(0=25|J)/GROUP5
[17]   C[J]←G[J]∨(T[J]∧C[J+1])∧TG5[J+5]∧C[J+5]
[18]   →CONT
[19]   GROUP25:C[J]←G[J]∨(T[J]∧C[J+1])∨(TG5[J+5]∧C[J+5])
         ∧TG25[J+25]∧C[J+25]
[20]   →(J≠0)/CONT
[21]   ⍝ CARRY ASSIMILATION
[22]   S←T≠1↓C
[23]   COUT←1↑C
     ∇
```

(Carry-skip Adder)

```
     ∇ ARCHSAVE;SUM
[1]    SUM←K+L+M
[2]    T←2⊤SUM
[3]    G←SUM≥2
     ∇
```

(Carry-save Adder)

```
     ∇ SAVEACC
[1]    K←A
[2]    L←T
[3]    M←1↓G,CIN
[4]    COUT←1↑G
[5]    ARCHSAVE
     ∇
```

(Carry-save Accumulator)

```
     ∇ SAVEADD
[1]    ⍝ DECLARATION
[2]    CB←CIN
[3]    COUT←0
[4]    ⍝ TRANSMISSION AND GENERATION
[5]    T←A
[6]    G←B
[7]    ⍝ CARRY PROPAGATION AND ASSIMILATION
[8]    CONT:P←T∧G
[9]    S←T≠G
[10]   COUT←COUT∨1↑P
[11]   T←S
[12]   G←1↓P,CB
[13]   CB←0
[14]   ⍝ TEST FOR ZERO GENERATION
[15]   NREADY←∨/G
[16]   →NREADY/CONT
     ∇
```

(Carry-save addition)

```
     ∇ RIPASYNC
[1]    ⍝ DECLARATION
[2]    NOA←~A
[3]    NOB←~B
[4]    NOCIN←~CIN
[5]    ⍝ TRANSMISSION AND GENERATION
[6]    G←A∧B
[7]    NOG←NOA∧NOB
[8]    NT←G∨NOG
[9]    T←~NT
[10]   ⍝ CARRY PROPAGATION
[11]   NOC←((⍴A)⍴0),NOCIN
[12]   C←((⍴A)⍴0),CIN
[13]   CONT:P←T∧1↓C
[14]   NOP←T∧1↓NOC
[15]   NOC←(NOG∨NOP),NOCIN
[16]   C←(G∨P),CIN
[17]   READY←∧/C∨NOC
[18]   →(~READY)/CONT
[19]   ⍝ CARRY ASSIMILATION
[20]   NS←P∨NT∧ 1↓NOC
[21]   S←~NS
[22]   NOCOUT←1↓NOC
[23]   COUT←1↑C
     ∇
```

(Asynchronous ripple adder)

Serial-Byte Adder

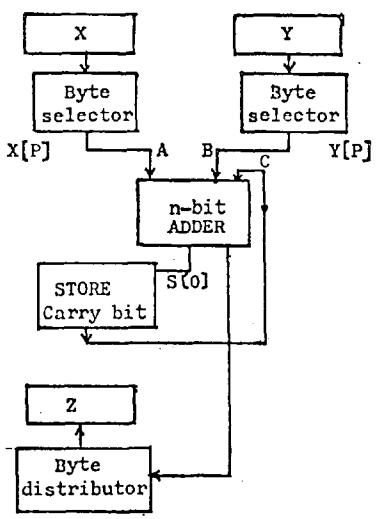
```
     ∇ A BINADD B
[1]    S←((N+1)ρ2)⊤C+(2⊥A)+2⊥B
     ∇

     ∇ Z←X BINADDER Y
[1]    'ENTER NO. OF BITS PER BYTE:'
[2]    N←□
[3]    J←ρY
[4]    Z←Jρ0
[5]    TC←C←0
[6]   LP:J←J-N
[7]    TC←TC+1
[8]    X[P] BINADD Y[P]
[10]   C←S[0]
[11]   Z[P]←(-N)↑S
[12]   →(J>0)/LP
[13]   'NO. OF ADDER USES=';TC
[14]   'SUM: '
     ∇

      1 1 1 0 0 0 BINADDER 1 1 1 0 0 1
ENTER NO. OF BITS PER BYTES:
□:
      2
NO. OF ADDER USES=3
SUM:
1  0  1  0  0  1
```

(Adder function; called by BINADDER)

(Addition of binary numbers by byte)



Multiplier with 1-bit shift

```
     ∇ ONEMPY
[1]    ⍝ DECLARATION
[2]    P←VL
[3]    Q←MR
[4]    R←MD
[5]    ⍝ MULTIPLICATION
[6]    W←0
[7]    K←ρQ
[8]    ⍝ MULTIPLE GENERATION
[9]   CONT: M0←R TWOFOLD ¯2↑Q,W
[10]   ⍝ ADDITION
[11]   A←1 EXTEND P
[12]   B←1 EXTEND M0
[13]   CIN←CX
[14]   ARCHADD
[15]   W←¯1↑Q
[16]   P←¯1↑S
[17]   Q←(¯1↑S),¯1↓Q
[18]   ⍝ COUNT
[19]   J←J-1
[20]   →(J≠0)/CONT
[21]   ⍝ RESULT
[22]   PD←P,Q
     ∇

     ∇ ARCHADD;SUM
[1]    SUM←(2⊥A)+(2⊥B)+2⊥CIN
[2]    S←((ρA)ρ2)⊤SUM
[3]    COUT←SUM≥2*ρA
     ∇

     ∇ RX←N EXTEND R
[1]    RX←R[Nρ0],R
     ∇

     ∇ MX←R TWOFOLD F
[1]    ⍝ DECISION
[2]    ONE←F[0]≠F[1]
[3]    ⍝ COMPLEMENT
[4]    CON←F[0]≠R
[5]    ⍝ SELECTION
[6]    MX←ONE∧COM
[7]    ⍝ CARRY-IN
[8]    CX←F[0]∧ONE
     ∇
```

Variable Shift Multiplier

```
     ∇ SHIFTMPY
[1]   ⍝ DECLARATION
[2]    P←VL
[3]    Q←1 EXTEND MR
[4]    R←MD
[5]   ⍝ MULTIPLICATION
[6]    W←0
[7]    J←¯1+⍴Q
[8]   ⍝ MULTIPLE GENERATION
[9]   CONT:M0←R TWOFOLDX ¯3↑Q,W
[10]  ⍝ ADDITION
[11]   A←1 EXTEND P
[12]   B←1 EXTEND M0
[13]   CIN←CX
[14]   ARCHADD
[15]   W←WX
[16]  ⍝ DISPLACEMENT
[17]   DP←DISPLACE (¯1↓¯4↑Q),WX
[18]P← P←(DP[0]∧¯4↑S[0 0 0],S)∨(DP[1]∧¯3↑S[0 0],S)∨(DP[2]∧¯2↑S[0],S)
       ∨DP[3]∧¯1↑S
[19]   Q←(DP[0]∧(¯4↑S),¯4↑Q)∨(DP[1]∧(¯3↑S),¯3↑Q)∨(DP[2]∧(¯2↑S),¯2↑Q)
       ∨DP[3]∧(¯1↑S),¯1↑Q
[20]  ⍝ COUNT
[21]   J←J-(DP[0]×3)+(DP[1]×2)+DP[2]+1
[22]   →(J≠0)/CONT
[23]  ⍝ RESULT
[24]   DP←¯1↑P,Q
     ∇

     ∇ DP←DISPLACE F
[1]   ⍝ DISPLACEMENT DECISION
[2]    DP←4⍴0
[3]    DP[3]←(J=1)∨F[2]≠F[3]
[4]    DP[2]←(~DP[3])∧(J=2)∨F[1]≠F[2]
[5]    DP[1]←(~∨/DP[2 3])∧(J=3)∨F[0]≠F[1]
[6]    DP[0]←~∨/DP[1 2 3]
     ∇
```

Note

SHIFTMPY requires auxilary functions : *ARCHADD*, *EXTEND*

```
     ∇ MX←R TWOFOLDX F
[1]   ⍝ DECISION
[2]    ONE←F[1]≠F[2]
[3]   ⍝ COMPLEMENT
[4]    COM←F[0]≠R
[5]   ⍝ SELECTION
[6]    MX←ONE∧COM
[7]   ⍝ CARRY-IN
[8]    CX←F[0]∧ONE
[9]   ⍝ NEW BALANCE
[10]   WX←(F[0]∧F[1])∨)F[0]∧F[2])
       ∨F[1]∧F[2]
     ∇
```

Multiplier with 8-bit shift

```
     ∇ EIGHTMPY
[1]   ⍝ DECLARATION
[2]    P←VL
[3]    Q←MR
[4]    DR←MD
[5]   ⍝ MULTIPLICATION
[6]    W←0
[7]    J←(⍴Q)÷8
[8]   ⍝ MULTPILE GENERATION
[9]   CONT: M7←R FOURFOLD ¯3↑Q,W
[10]   C7←CX
[11]   M5←R FOURFOLD 3↑¯4↑Q
[12]   C5←CX
[13]   M3←R FOURFOLD 3↑¯6↑Q
[14]   C3←CX
[15]   M1←R FOURFOLD 3↑¯8↑Q
[16]   C1←CX
[17]  ⍝ CARRY-SAVE ADDITION
[18]   K←3 EXTEND ¯2↑P
[19]   L←2 EXTEND ¯2↑M7
[20]   M←M5
[21]   ARCHSAVE
[22]   T5←T
[23]   G4←G
[24]   K←2 EXTEND ¯2↑T5
[25]   L←1 EXTEND ¯1↑G4
[26]   M←M3
[27]   ARCHSAVE
[28]   T3←T
[29]   G2←G
[30]   K←2 EXTEND ¯2↑T3
[31]   L←1 EXTEND ¯1↑G2
[32]   M←M1
[33]   ARCHSAVE
[34]   T1←T
[35]   G0←G
[36]  ⍝ CARRY-PROPAGATE ADDITION
[37]   A←1 EXTEND T1,(¯2↑T3),(¯2↑T5),¯2↑P
[38]   B←G0,C1,(¯1↑G2),C3,(¯1↑G4),C5,¯2↑M7
[39]   CIN←C7
[40]   ARCHADD
[41]   W←1↓¯8↑Q
[42]   P←¯8↑S
[43]   Q←(¯8↑S),¯8↑Q
[44]  ⍝ COUNT
[45]   J←J-1
[46]   →(J≠0)/CONT
[47]  ⍝ RESULTS
[48]   PD←P,Q
     ∇
```

```
     ∇ MX←R FOURFOLD F
[1]   ⍝ DECISION
[2]    ONE←F[1]≠F[2]
[3]    TWO←((~F[0])∧F[1]∧F[2])∨
       F[0]∧(~F[1])~F[2]
[4]   ⍝ COMPLEMENT
[5]    COM←F[0]≠R
[6]   ⍝ SELECTION
[7]    MX←(ONE∧COM[0],COM)∨TWO∧
       COM,F[0]
[8]   ⍝ CARRY-IN
[9]    CX←F[0]∧ONE∨TWO
     ∇
```

NOTE :

EIGHTMPY requires auxilary functions : *ARCHADD*, *ARCHSAVE*, *EXTEND*

Appendix 12 :

DESCRIPTION & SIMULATION OF MACHINE ARCHITECTURE USING APL

Description & Simulation of Machine Architecture

"Architecture" usually means the functional appearance of the system to its users. Architecture, which is user- and specification-oriented, should be constrasted with "implementation", which is concerned with the design of efficient real objects like circuits and programs that give acutal form to the architecure specifications. To an increasing extent, machine architecture reflects the results of careful study and observation of certain salient features of programs. For examples, it has been noticed that most programs exhibit the property of "locality of reference", i.e. over a time interval spanning several instruction executions, refernes to storage for program and data tend to cluster in localised areas of storage. This property suggests that highest speed of reference and access may be necessary only to a relatively small area of storage. The system, however, must be able to address larger storage also but not necessarily at the highest speeds.

In the following, we shall describe machine architecture features, such as indirect addressing, push-down stores and management of transmission between mainprocessor storage with the help of APL.

A) Addressing Modes (Immediate, Direct, Indirect)

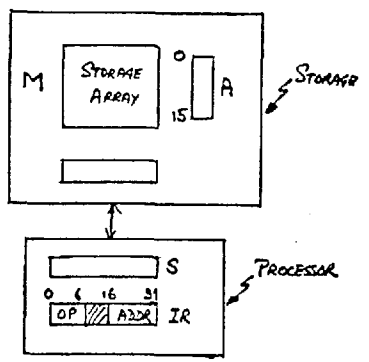
Assume for illustration the simple 32-bit machine and focuss attention on how a word is obtained. The instruction about to be executed is assumed to be available in the instruction register (IR). The addressing is discussed only for the case of obtaining an operand for the register called (S) which is 32-bit long.

i) Immediate (or sometimes literal) addressing : the operand is taken as the address part of the instructin itself. Access consists of routing the rightmost 16 bits of IR to the data register of the processor S. Since S is assumed in this example to be 32 bits long, the fits from IR are routed to the lower-order pos itions and the rest of S is filled with Os.

ii) Direct or normal addressing : The rightmost 16 bits of the instruction are sent to the memory address register A, and the word specified by A is fetched to the processor storage register S. Direct addressing is the most essential of the three types under discussion; many machines allow only direct addressing.

iii) Indirect Addressing : The address part of the instruction is first used to fetch a word from storage in the same way as in direct addressing. The rightmost 16 bits of the wrod thus obtained are routed to replace the address part of the instruction register. The system now behaves as if it had just obtained the isntruction; i.e. the address is used as for direct addressing. Although indirect addressing may appear at first glance to be a strange operation, it is a very convenient feature to the programmer.

Figure A12.1 illustrates the three addressing types with the aid of APL.



Notes :

0-origin indexing

S — Processor Storage register
IR — Instruction Register
M — Storage; M[I;] = word I
A — Storage Address register
B — Storage Data register

```
∇ IMMED
[1]  S←(16ρ0),¯16↑IR
∇
```

```
∇ DIRECT
[1]  A←¯16↑IR
[2]  B←M[2⊥A;]
[3]  S←B
∇
```

```
∇ INDIRECT
[1]  DIRECT
[2]  IR←(16↑IR),¯16↑S
[3]  DIRECT
∇
```

Fig. A12.1 : Description of three common addressing systems

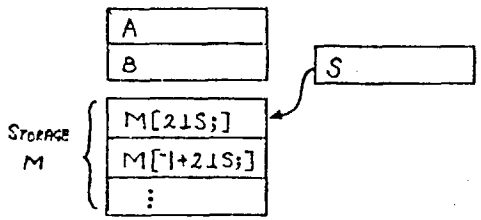
B) Push-down Storage, or Stack

An important illustration of implicit addressing is the control of a logical structure called a "push-down store", or LIFO (last in, first out) list, or stack. An address specifies a storage location as a source/destination of data to/from the top of a list of contiguous storage words. Operations implicitly reference the operand(s) at the top of the list, and the result operand replaces the operand(s) at the top of the list. The structure thus behaves as if an entered iten "pushed down" the other items. When an operation is performed, the result replaces one of the top two operands (at least for binary operators); the list is, as it were, pushed up one position.

Figure A12.2 refers to a sturcture consisting of two registers A & B which constitute the top two cells of the stack and a register S which contains the location in storage of the next cell in the stack. In other words, our logical push-down store has its top two cells implemented in registers (since these are the most active cells); the rest of the stack is in storage. This mechanization permits a size of push-down store that can be very large without many expensive registers. Furthermore, the iternal operations to be described can make this mixed nature of the implementation invisible to the users. This is done by automatically moving information between the cell whose address is given by S and the A and B registers. Information is moved between the A,B registers also; this movement is much faster than movement between registers and storage.

Tdo indicators E[1,2] are used to specify whether the A,B registers, respectively, contain information or not. These empty indicators can have a total of four states; these are listed for reference purpose (figure A12.2).

The operations described as microprograms specify how an entry is made to the stack and how operations obtain operands. Depending upon the state of the empty indicators and the type of operation (binary or unary), it is seen that an operation may require either zero, one, or two references to storage to perform an operation. This number may be reduced on the average by increasing the number of registers used to implement the top cells of the stack. This can lead to significant speed increase because most activity involves the top members of the stack.



(a) Logical stack

Notes

0-origin indexing

A,B — Top registers of stack
S — Pointer to stack extension in storage
M — Storage (bit matrix)
E — Empty indicators of A,B
BINOP — Binary operator
UNOP — Unary operator

```
E[0]  E[1]

  0     0      A,B  BOTH FULL
  0     1      ILLEGAL
  1     0      A EMPTY, B FULL
  1     1      A,B BOTH EMPTY
```

(b) Empty indicators

```
     ∇ XBINOP
[1]    →(B1,B2,B3,B4)[2⊥E]
[2]  B1:B←A BINOP B
[3]    E←1,0
[4]    →0
[5]  B3:A←B
[6]    STAKFETCH
[7]    →B1
[8]  B4:STAKFETCH
[9]    →B3
[10] B2:'ILLEGAL STATE'
     ∇
```

(c) Binary operator

```
     ∇ XUNOP
[1]    →(B1,B2,B3,B4)[2⊥E]
[2]   B1:A←UNOP A
[3]    →0
[4]   B3:B←UNOP B
[5]    →0
[6]   B4:STAKFETCH
[7]    →B3
[8]   B2:'ILLEGAL STATE'
      ∇
```

(d) Unary operator

```
     ∇STAKFETCH
[1]  B←M[2⊥S;]
[2]  S←((ρS)ρ2)⊤¯1+2⊥S
     ∇
```

```
     ∇ ENTER X
[1]    →(B1,B2,B3,B4)[2⊥E]
[2]   B1:S←((ρS)ρ2)⊤1+2⊥S
[3]    M[2⊥S;]←B
[4]    B←A
[5]   B3:A←X
[6]    →E[0]←0
[7]   B4:B←X
[8]    →E[1]←0
[9]   B2:'ILLEGAL STATE'
      ∇
```

(e) Stack operations

Fig. A12.2 : Control of a push-down stack

Appendix 13 :

SOURCE CODE LISTING OF "DISPIC's"

```
      ⍝  DOCUMENTATION OF WORKSPACE ** DISPIC1 **

      DESCRIBE

WORKSPACE CONTAINS  ** VERSION 1 OF DISPIC **

      ENVIRONMENT :
          WSID    -   DISPIC1
          WIDTH   -   130
          ORIGIN  -   0

      ** FNS **
ΔFINP  ΔFOUT  ΔMATRIX  ΔSPIN  BUILD  CLEARFILE  CKE   CKE1
CKI   CKS  CKZ   CNE   DESCRIBE  DROP  DUMP  EDIT  EXECUTE
GAD   GWA   INPFILE  INITIALIZE  LIST  LISTFILE  LOAD INFO
MEMORY  OPR   PATCH  QQP   QQS   QQS0  RENAME  RESET RESTART
RFIX  RLOAD  RUN  STATES  TCA   TCE   TCI   TMA   TNA   TNE
TNI  START

      ** VARS  **
ABS   ACC   AR  BS  CR  DG  ERR   FIBO   HDR   IC  IND  INS
LIB   LINKMAP   LOADMAP   LF  MEM   OFF  ON  QPC   RS  SORT
SQRT  TABLE  TIMER  TRACE  WA

      NOTE :  CR  ↔  CARRIAGE RETURN
              BS  ↔  BACKSPACE
              LF  ↔  LINE FEED
      ARE DERIVED FROM THE ATOMIC VECTOR (⎕AV)  AND THEREFORE
      SYSTEM DEPENDENT;  OTHER VARIABLES ARE THE RESULT OF
      EXECUTING FUNCTION 'INITIALIZE' WHICH SETS UP THE
      NECESSARY ENVIRONMENT FOR  ** DISPIC **

      ** GRPS  **
GRPAUXIL  GRPDISPIC

      WHERE,

** GROUP GRPAUXIL **
INITIALIZE  BUILD  RUN  LIST  EDIT  DUMP  DROP  RENAME  LOAD
EXECUTE  PATCH  INFO  RLOAD  RFIX  START  QQP  ΔSPIN  CLEARFILE
LISTFILE

** GROUP GRPDISPIC **
RUN  LOAD  EXECUTE  QPR  CKE   CKI   CKS   CKZ   GAD   GWA
QQS  TCA  TCE  TCI  TMA  TNA  TNE  TNI  ΔFINP  ΔFOUT
```

```
      ΔLISTGRP GRPAUXIL

** WORKSPACE : DISPIC1          ORIGIN : 0


 FUNCTIONS TO BE LISTED:


INITIALIZE
BUILD
RUN
LIST
EDIT
DUMP
DROP
RENAME
LOAD
EXECUTE
PATCH
INFO
RLOAD
RFIX
START
QQP
ΔSPIN
CLEARFILE
LISTFILE


=========================
```

```
      ∇INITIALIZE
[1]     ⍝ NEED BS (BACKSPACE) AND CR (CARRIAGE RETURN)
[2]     ⎕PW←130
[3]     ⎕IO←0
[4]     IND←10⍴AR←TIMER←ON←IC←0
[5]     MEM←LIB←0 11⍴WA←INS←ACC←11⍴' '
[6]     TRACE←OFF←1
[7]     BS←'0123456789'
[8]     HDR←'      +',BS,'_OPC***0000'
[9]     QPC←32 2⍴'LDSTADSUMPDIFIFLFAFSFMFDBLBRBMBCCSSEWWRWWBRBSIL
        IAXSXTITXCICXHLSH'
[10]    ⍝ DEFINE INPUT AND OUTPUT FILES
[11]    ΔQUIE←⍳0
[12]    ΔINFE←0 11⍴' '
[13]    ⍝ INPUT FILE COUNTER
[14]    IFCTR←0
[15]    MEMORY 500
      ∇
```

```
      ∇EDIT DS;LN;WD;X;M
[1]    M←DS,[0]11↑' '
[2]    'DO YOU WISH TO ADD LINES, DELETE LINES,',CR,'CHANGE A LINE, PRINT LINES, OR EXIT?'
[3]   ST:LN←30 QQS'ENTER: A,D,C,P, OR X      '
[4]    →(A,D,C,P,X,ST)['ADCPX'\1↑(' '≠LN)/LN]
[5]   A:→((↑M)[0]≥1+LN←¯1+QQF'BEFORE WHICH LINE (NUMBER) DO ADDITIONS COME?')/A1
[6]    LN←¯3+1↑↑M
[7]   A1:'ENTER A BLANK LINE AFTER ADDITIONS',CR,HDR
[8]   A2:LN←LN+1
[9]   A3:→(CKS WD←5 QQS(ICA LN),'0')/ST,A3,A4
[10]  A4:→A2,↑M←M[\LN+1;],[0](11↑WD),[0]M[1+LN+\(1↑↑M)-LN+1;]
[11]  D:LN←1+QQF'WHICH LINES(S) SHOULD BE DELETED?'
[12]   →ST,↑M←(~(\1↑↑M)∈LN)⌿M
[13]  C:→((1↑↑M)≥1+LN←1↑QQF'WHICH LINE IS TO BE CHANGED?')/P2
[14]   'THE LINE NOW READS:  ',(,ICA LN),'0',,M[LN+1;]
[15]   'ENTER THE CORRECTED LINE',3↓HDR
[16]  C1:→(CKS,WD←26 QQS(21↑' '),(,ICA LN),'0')/ST,C1,C2
[17]  C2:→ST,↑M[LN+1;]←11↑WD
[18]  P:LN←(LN(1↑↑M)/LN←1+QQF'WHICH LINE(S) SHOULD BE PRINTED?     '
[19]   HDR
[20]  P1:(,ICA ¯1+WD),'0',,M[WD←1↑LN;]
[21]   →(1 0⍳↑LN←1↓LN)/P1,ST
[22]  P2:→ST, ↑0←'LINE NUMBER TOO LARGE'
[23]  X:M←¯1 0↓M
[24]   A RIGHT JUSTIFY ADDRESS FIELD
[25]   M[;X]←ΔSPIN M[;X←¯4↑\¯1↑↑M]A  RIGHT JUSTIFY ADDRESS FIELD
[26]   ⍎DS[0;],'←M'
[27]   ' ** COMPLETED **'
[28]   ' '
      ∇
```

```
      ∇BUILD;LN;WD;X;M
[1]    M←1 11↑' '
[2]    'ENTER LINES OF DATA SET; PRESS RETURN AT END',CR,HDR
[3]    LN←0
[4]   A:→(CKS WD←5 QQS(ICA LN),'0')/C,A,B
[5]   B:M←M,[0]11↑WD
[6]     →A,↑LN←LN+1
[7]   C:LN←47 QQS'WHAT SHOULD THE NAME OF THIS DATA SET BE?
[8]     LIB←LIB,[0]M[0;]←11↑,LN
[9]     M[;X]←ΔSPIN M[;X←¯4↑\¯1↑↑M]A  RIGHT JUSTIFY ADDRESS FIELD
[10]    A DEFINE THE DATA SET AS SPECIFIED
[11]    ⍎LN,'←M'
[12]    '** DATA SET ': ',LN,' IS DEFINED ... '
      ∇

      ∇RUN DS
[1]    0 LOAD DS
[2]    EXECUTE 0
      ∇

      ∇LIST DS;L;K
       HDR
       K←0⍴L←¯1+1↑↑DS
[1]   A:(,ICA K),' ',,DS[K+1;]
[2]     →A⍴L←K←K+1
[3]    ∇

      ∇DUMP R;C
       A CONVERT R TO THE NEAREST MULTIPLES OF 10'S
[1]    R←10×⌈0.1×R
[2]    ' '
[3]    '** LAST LOCATION :  ';R
[4]    →(~C←0=R)/A0
[5]    R←(1↑↑MEM)-1+C←0
[6]   A0:'    ',120⍴8↓HDR,'| '
[7]   A:→(^/' '=,MEM[RLC+\10;])/B
[8]     (,ICA C),':',,MEM[RLC+\10;],10 1⍴'|'
[9]   B:→(R≥C←C+10)/A
[10]    ∇

      ∇DROP DS;Z
[1]    LIB←LIB[(LIB≠↑DS[0;])/\1↑↑LIB;]
[2]    '** DATA SET - ',DS[0;],'  ... DROPED'
[3]    Z←⍎EX DS[0;]
      ∇

      ∇N LOAD DS;K;DT
[1]    →((1↑↑MEM)<¯1+N+K←1↑↑DT←1 0↓DS)/A
[2]    ↑0,↑⎕←'MEMORY NOT LARGE ENOUGH'
[3]   A:MEM[N+1K;]←DT
[4]    'DATA SET LOADED INTO MEMORY',CR,'LOCATIONS ';N;' THRU';N+K-1
[5]    LOADMAP←LOADMAP,[CR+0](DS[0;]),(,ICA N),' TO ',,ICA N+K-1
      ∇
```

```
     ∇RENAME DS;NN;OD;ID
[1]    NN←(32+⎕OD)⍞'WHAT SHOULD NEW NAME OF ',(OD←(OD≠' ')
       /OD←DS[0;]),' BE?
[2]    →(0=⎕NC ID←(LIB∧.=DS[0;])/⍳1↑⍴LIB)/A
[3]    →B,⎕LIB[ID;]←11↑NN
[4]   A;LIB←LIB,[0]11↑NN
[5]   B;DS[0;]←11↑NN
[6]    M←DS
[7]    ⍝ REMOVE BLANKS IN NN; RE-ASSIGN NAME; DELETE OLD NAME
[8]    NN←(NN≠' ')/NN
[9]    ⍎NN,'←M'
[10]   OD←⎕EX OD
[11]   '** COMPLETED '
     ∇

     ∇EXECUTE IC;N;ER;IS;CA;T;K;SZ
[1]    T←⎕AI[1]+K←ER←IND[0]←0
[2]    SZ←(⎕MEM)[0]
[3]    →(2400<⎕WA[0])/A
[4]    →0,⎕←'NOT ENOUGH WS ROOM FOR EXECUTION'
[5]   A;INS←IS←MEM[IC←0=ER←SZ[IC←IC;]
[6]   B;K←K+1
[7]    →(0≠-/IS[4 5]='I*')/C
[8]    IS[4+⍳7]←4↓MEM[AR←0=ER←SZ[AR←GAD;]
[9]    →B
[10]  C;IC←IC+1
[11]   →(32)N←(OPC∧.=IS[1 2])⍳1)/D
[12]   N←1+9×ER←2
[13]  D;N←OPR N
[14]   →TIMER/E
[15]   →(0≠100|K)/E
[16]   'CPU TIME:  ';⌈0.001×⎕AI[1]-T ;' SECONDS'
[17]  E;→(TRACE∨N∧0=ER←⌈/ER)/A
[18]   (0≠ER)/ERR[ER;]
[19]   'IC ';IC;';   ',INS,'; AR ';AR;'; ACC ',ACC,'; WA ';WA;'; R1 '
       IND[1];'; R2 ';IND[2];'; R3 ';IND[3]
[20]   →(N∧0=ER)/A
[21]   'EXECUTION REQUIRED ';0.001×⎕AI[1]-T;' SECONDS OF CPU TIME'
     ∇

     ∇PATCH N;W
[1]    →(N<1↑⍴MEM)/A
[2]    →0,⎕←'MEMORY NOT THAT LARGE'
[3]   A;'MEMORY LOCATION ',(⍕IC N),'  □',(⍕MEM[N;]),'□'
[4]   B;→(0≠⍴W←⍞'ENTER NEW WORD          □')/0,B,C
[5]   C;MEM[N;]←11↑W
     ∇

     ∇INFO
[1]    'INSTRUCTION COUNTER:  ';IC
[2]    'OPERATION REGISTER :   ',4↑1↓INS
[3]    'ADDRESS REGISTER   :  ';AR
[4]    'ACCUMULATOR        :  ';ACC
[5]    'OPERAND REGISTER   :  ',WA
[6]    'INDEX REGISTERS    :  ';3↑1↓IND
     ∇

     ∇N RLOAD DS;V;ER;K;I;OD;AD;LM
[1]    ER←0
[2]    V←V/⍳⍴V←0,1↓'⍴'=DS[;7]
[3]    DS[V;7+⍳4]←IC∆ N+1 IMA DS[V;8+⍳3]
[4]    N LOAD DS
[5]    →(0=⎕NC I←((LM[;⍳5])∧.=DS[0;⍳5])/⍳1↑⍴LM←LINKMAP)/C
[6]    →('U'=LM[I;6])/A
[7]    →0,⎕←DS[0;⍳5],' ALREADY LOADED; CLEAR MEMORY'
[8]   A;AD←1-IMA,LM[I;8+⍳4]
[9]    LM[I;6,8+⍳4]←'D',V←,IC∆ N
[10]  B;OD←,MEM[AD;7+⍳4]
[11]   MEM[AD;4 5 7 8 9 10]←'  ',V
[12]   →('?'∈OD)/D
[13]   →B, AD←1 IMA OD
[14]  C;LM←LM,[0]DS[0;⍳5],' D ',,IC∆ N
[15]  D;→(∨/'ER'≠MEM[N;1 2])/H
[16]   AD←1 IMA MEM[K←N;7+⍳4]
[17]  E;→(AD[K←K+1)/H
[18]   →(0=⎕NC I←((LM[;⍳5]∧.=MEM[K;1+⍳5])/⍳1↑⍴LM)/G
[19]   →('U'=LM[I;6])/F
[20]   →E,⎕MEM[K;4 5 7 8 9 10]←'  ',,LM[I;8+⍳4]
[21]  F;MEM[K;4 5 7 8 9 10]←'*',,LM[I;8+⍳4]
[22]   →E,⎕LM[I;8+⍳4]←,IC∆ K
[23]  G;LM←LM,[0]MEM[K;1+⍳5],' U ',,IC∆ K
[24]   →E,⎕MEM[K;7+⍳4]←'????'
[25]  H;ERR[ER;]
[26]   □←LINKMAP←LM
[27]   ' '
     ∇

     ∇RFIX DS;A;B;V;W;AD
[1]    M←1 0↓DS
[2]    A←⍞'WITH WHICH LINE DO ADJUSTMENTS BEGIN?'
[3]    B←⍞'WHAT IS THE VALUE OF THE NET CHANGE?'
[4]    W←V/⍳⍴V←('⍴'=M[;7])∧A[AD←1 IMA M[;8 9 10]
[5]    AD[W]←AD[W]+B
[6]    M[W;8 9 10]←0 1↓V⌿IC∆ AD
[7]    'TYPE: ';((AD≠' ')/AD←DS[0;]),'←M'
[8]    M←DS[0;],[0] M
     ∇

     ∇START
[1]    EXECUTE IC
     ∇

     ∇V←⍞ S;T;X;N
[1]    X←(' '=1↑X)↓X←(T∨1⌽T←∧\X='' ')/X←(3+⍴S)⍞ S,'   ',V←⍳0
[2]   A;N←1+(X⍳' ')↓X⍳','
[3]    T←(1↑S←N↑X)∈'-¯'
[4]    →('→'∈S←(S∈DG,'→')/S)/C
[5]    V←V,1 ¯1[T]×10⊥DG⍳S
[6]   B;→A×0<⍴X←N↓X
[7]   C;T←10⊥DG⍳(1+S⍳'→')↓S
[8]    →(0>T←T-S←10⊥DG⍳(S⍳'→')↑S)/0
[9]    V←V,S+⍳T+1
[10]   →B
     ∇

     ∇R←⍙SPIN X
[1]    ⍝ RIGHT JUSTIFY X AND RETURN RESULT AS R
[2]    R←(-+/∧\⌽' '=X)⌽X
     ∇
```

```
      ∇LISTFILE;S;SI
[1]   L0:' '
[2]    □←S←'LIST INPUT/OUTPUT FILES? (CR TO EXIT) '
[3]    SI←1↑(⍴,S)↓□
[4]    →(L1,L2,L3)['IO'⍳SI]
[5]    →L3
[6]   L1:⍝ LIST INPUT FILE
[7]    2⍴⎕LE
[8]    '** INPUT FILE (ΔINPF) :      SAMPLES = ';1↑⍴ΔINPF
[9]    ' '
[10]   ΔINPF
[11]   ' '
[12]   →L0
[13]  L2:⍝ LIST OUTPUF FILE
[14]   2⍴⎕LE
[15]   '** OUTPUT FILE (ΔOUTF) :      SAMPLES = ';⍴ΔOUTF
[16]   ' '
[17]   ΔOUTF
[18]   ' '
[19]   →L0
[20]  L3:⎕LE
[21]   '** COMPLETED ';⎕LE
      ∇


      ∇CLEARFILE;S;SI
[1]   L0:□←S←'CLEAR INPUT/OUTPUT FILES? '
[2]    SI←1↑(⍴,S)↓□
[3]    →(L1,L2,L3)['IO'⍳SI]
[4]    →L3
[5]   L1:⍝ CLEAR INPUT FILE
[6]    ΔINPF←0 11⍴' '
[7]    IFCTR←,0
[8]    '** INPUT FILE (ΔINPF) IS CLEARED'
[9]    →L0
[10]  L2:⍝ CLEAR OUTPUT FILE
[11]   ΔOUTF←⍳0
[12]   '** OUTPUT FILE (ΔOUTF) IS CLEARED'
[13]   →L0
[14]  L3:'** COMPLETED'
      ∇
```

```
** WORKSPACE : DISPIC1          ORIGIN : 0


 FUNCTIONS TO BE LISTED:


RUN
LOAD
EXECUTE
OPR
CKF
CKI
CKS
CKZ
GAD
GWA
GOS
ICA
ICF
ICI
IMA
INA
INF
INI
ΔFINP
ΔFOUT


=======================


      ∇RUN DS
[1]    0 LOAD DS
[2]    EXECUTE 0
      ∇

      ∇N LOAD DS;K;DT
[1]    →((1↑⍴MEM)>¯1+N+K←1↑⍴DT←1 0↓DS)/A
[2]    →0,⍴□←'MEMORY NOT LARGE ENOUGH'
[3]   A:MEM[N+⍳K;]←DT
[4]    'DATA SET LOADED INTO MEMORY',CR,'LOCATIONS ';N;' THRU';N+K-1
[5]    LOADMAP←LOADMAP,[RS+0](DS[0;]),(,ICA N),' TO ',,ICA N+K-1
      ∇
```

```
     ∇EXECUTE IC;N;ER;IS;CA;T;K;SZ
[1]   T←⎕AI[1]+K←ER←INP[0]←0
[2]   SZ←(⍎MEM)[0]
[3]   →(2400<⎕WA[0])/A
[4]   →0,⍞←'NOT ENOUGH WS ROOM FOR EXECUTION'
[5]  A:INS←IS←MEM[IC×0=ER←SZ[IC←IC;]]
[6]  B:K←K+1
[7]   →(0∊-/IS[4 5]='I⍟')/C
[8]   IS[4+⍳7]←4↓MEM[AR×0=ER←SZ[AR←GAD;]]
[9]   →B
[10] C:IC←IC+1
[11]  →(32)⍴←(OPC^.=IS[1 2])⍳1)/D
[12]  N←1+9×ER←2
[13] D:N←OPR N
[14]  →TIMER/E
[15]  →(0≠100|K)/E
[16]  'CPU TIME:  ';⌈0.001×⎕AI[1]-T;' SECONDS'
[17] E:→(TRACE∧N×0=ER←⌈/ER)/A
[18]  (0≠ER)/ERR[ER;]
[19]  'IC ';IC;';  ',INS,';  AR ';AR;';  ACC ',ACC,';  WA ';WA;';  R1 ';
      IND[1];';  R2 ';IND[2];';  R3 ';IND[3]
[20]  →(N∧0=ER)/A
[21]  'EXECUTION REQUIRED ';0.001×⎕AI[1]-T;' SECONDS OF CPU TIME'
     ∇

     ∇Z←CKI C
[1]   ER←ER⌈7×~Z←∧/(C[0]∈'+- '),(1↓C)∈DG,' '
     ∇

     ∇Z←CKF C
[1]   ER←ER⌈8×~Z←∧/(C[0]∈'+- '),(C[1]∈'+-'),C[2+⍳9]∈DG,' '
     ∇

     ∇R←CKS W
[1]   →2×(/2↑R←(('E'=1↑W)∨('⍟'∈W)∨0=⍎W),(∧/'+- '≠1↑W),1
[2]   ERR[3;]
     ∇

     ∇Z←CKZ C;T
[1]   Z←((∧/C[1+⍳10]∈'0 ')×~T)∨(∧/C[4+⍳7]∈'0 ')×T←C[1]∈'+-'
     ∇

     ∇A←GAD;N
[1]   A←1 INA CA←¯4↑IS
[2]   →(0=N←IND[1 INA,IS[6]])/0
[3]   CA←,ICA A←A+N
     ∇

     ∇W←GWA
[1]   AR←GAD
[2]   →('I'≠IS[4])/A
[3]   W←'      ',CA
[4]   →0,ER←ER×ER≠1
[5]  A:W←'
[6]   →(AR≤SZ)/B
[7]   →R←0×ER←1
[8]  B:W←MEM[AR;]
     ∇

     ∇R←OPR N;C;V
[1]   R←1
[2]   →((N∈8+⍳4),~N∈2+⍳4)/FL,OK
[3]   →OK×R←(CKI ACC)∧CKI WA←GWA
[4]  FL:→OK×R←(CKF ACC)∧CKF WA←GWA
[5]  OK:→(A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,
      X,X,X,X,X,X,X,Y,Z)[N]
[6]  A:→0,⍞ACC←WA←GWA
[7]  B:→0,⍞WA←MEM[AR×0=ER←SZ[AR←GAD;]]←((7×C)⍴' '),(7×C←IS[3]='A')↓ACC
[8]  C:→0,⍞ACC←ICI(INI ACC)+INI WA
[9]  D:→0,⍞ACC←ICI(INI ACC)-INI WA
[10] E:→0,⍞ACC←ICI(INI ACC)×INI WA
[11] F:→(0=N←INI WA)/L1
[12]   →0,⍞ACC←ICIL(INI ACC)÷N
[13] G:(~R←CKF ACC)/0
[14]   →0,⍞ACC←ICI INF ACC
[15] H:→(~R←CKI ACC)/0
[16]   →0,⍞ACC←ICF INI ACC
[17] I:→0,⍞ACC←ICF(INF ACC)+INF WA
[18] J:→0,⍞ACC←ICF(INF ACC)-INF WA
[19] K:→0,⍞ACC←ICF(INF ACC)×INF WA
[20] L:→(0=N←INF WA)/L1
[21]   →0,⍞ACC←ICF(INF ACC)÷N
[22] L1:→R←0×ER←9+2×ER=11
[23] M:IND[5 INA,IS[3]]←IC
[24] N:→0,IC←AR←GAD
[25] Q:→0,IC←(IC×~N)+(AR←GAD)×N←('-'=ACC[0])∧~CKZ ACC
[26] P:V←((ACC[0]∈'-¯')∧~N),N←CKZ ACC
[27]   →0,IC←(IC×~N)+(AR←GAD)×N←v/(2 2 2⊤6 INA,IS[3])×V,(ACC[0]∈'+ ')∧~N
[28] Q:→((((CKI ACC)∧CKI WA),(CKF ACC)∧CKF WA←GWA)/Q1,Q2
[29]   →R←0×ER←12
[30] Q1:→Q3+0×C←(INI ACC)-(INI WA)+ER←0
[31] Q2:C←(INF ACC)-(INF WA)+ER←0
[32] Q3:→0,IC←(IC×~N)+(IC+1)×N←v/(2 2 2⊤ 6 INA,IS[3])×¯1 0 1 =×C
[33] R:→0,IC←(IC×~N)+(IC+1)×N←∧/ACC=WA←GWA
[34] S:→0,⍞←'□',WA←GWA
[35] I:→(CKS WA←1 QQS0'?            □',12⍴BS)/Y1,I,T1
[36] T1:→0,⍞MEM[AR×0=ER←SZ[AR←GAD;]←WA←11↑WA
[37] U:→ΔFOUT A OUTPUT TO ΔOUTF
[38] V0:ERR[3;]
[39] V:→ΔFINP A     INPUT FROM    ΔINPF
[40]   A      V1:→(v/∧⍲'+- '∘.=⍳5 1↑V←5 12⊥60↑V)/V0
[41]   A         → 0,⍞WA←,1 11↑MEM[V×0=ER←SZ[V←(AR←GAD)+⍳5;]←0 1↓V
[42] W:→0,⍞MEM[AR×0=ER←SZ[AR;]←WA←(7↑WA←GWA),,ICA INP[5 INA,IS[3]]
[43] X:→(0=C←5 INA,IS[3])/0
[44]   →(X1,X2,X3,X4,X4,X5,X5)[N-23]
[45] X1:→0,IND[C]←5 INA ¯4↑WA←GWA
[46] X2:→0,IND[C]←IND[C]+5 INA ¯4↑WA←GWA
[47] X3:→0,IND[C]←10000|IND[C]-5 INA ¯4↑WA←GWA
[48] X4:→(0=IND[C])/X6
[49]   →0,IC←(IC×~N)+(AR←GAD)×N←v/(IS[2]='XI')×(~N),N←0=IND[C]←IND[C]-1
[50] X5:→(~CKI WA←GWA)/X7
[51]   →0,IC←(IC×~N)+(IC+1)×N←v/(IS[2])='XI')×(~N),N←IND[C]=INI WA
[52] X6:→R←0×ER←4
[53] X7:→R←0×ER←7
[54] Y:IC←AR←GAD
[55] Y1:→R←0
[56] Z:ACC[1+⍳10]←(C×10)↑((AR←GAD)×C←1+¯2×'R'=IS[3])↓1↓ACC
     ∇
```

```
      ∇R←ICA N
[1]     R←((⍴,N),4)⍴4 0↑10000|N
        ∇

      ∇R←ICF N;E
[1]     R←'++000000000'
[2]     →(N=0)/0
[3]     →(71≥|3-E←L10*|N)/A
[4]     →0⍴ER←10
[5]   A;E←1+L10*(|N)+10*⍣8+E
[6]     R←'+-'[0>N;],(2 0⍴|E),7 0⍴L0.5+|N×10*7-E
        ∇

      ∇R←ICI N
[1]     →(9999999999≥|N)/A
[2]     N←0⍴ER←10
[3]   A;R←'- +'[1+×N],10 0⍴|N
        ∇

      ∇N←K INA C
[1]     →(∧/,C∈DG,' ')/A
[2]     →N←0⍴ER←K⌈ER
[3]   A;K←⍴C
[4]     C[(' '=C)/⍳⍴C←,C]←'0'
[5]     N←10⊥DG⍳⍉K↑C
        ∇

      ∇N←K INA C
[1]     →(∧/C∈DG,' ')/A
[2]     →N←0⍴ER←K⌈ER
[3]   A;C[(' '=C)/⍳⍴,C]←'0'
[4]     N←⍎'0',C
        ∇

      ∇N←INF C;E
[1]     →(75≥E←INI C[1 2 3])/A
[2]     →N←0⍴ER←10
[3]   A;→((CKF C)∨0≠N←(INI C[0,4+⍳7])×10*E-7 )/0
[4]     ER←11
        ∇

      ∇N←INI C
[1]     C[⍳' '=1↑C]←'+'
[2]     C[(' '=C)/⍳⍴C]←'0'
[3]     N←⍎C
        ∇

      ∇R←N QQS S
[1]     ⍝ PRINTS STRING S AND ACCEPTS INPUT ON SAME LINE
[2]     ⎕←S
[3]     R←(⍴,S)↓⎕
        ∇

      ∇A←ΔFINP;V;N
[1]     A←0
[2]     V←QA←GAQ
[3]     →(V≠0)/L1
[4]     IFCTR[0]←V←IFCTR[0]+1
[5]   L1:⍝ CHECK END OF FILE REACHED
[6]     →(R←V[N←1↑⍴ΔINFF)/L2
[7]     →0,⍴,ER←13
[8]   L2:ACC←(⍴ACC)↑ΔINFF[V-1;]
[9]     →(V⟨N)/0
[10]    ⍝ RESET THE INPUT FILE COUNTER (IFCTR=0)
[11]    IFCTR←,0
```

```
      ∇A←ΔFOUT;V1;V
[1]     ⍝ OUTPUT CONTENTS OF THE ACCUMULATOR (ACC) TO THE ΔOUTF
[2]     A←0
[3]     ⍝ CHECK IF ACC_ IS FLOATING
[4]     →(0=CKE1 ACC)/L1
[5]     ⍝ CONVERT CHAR, FLOATING TO NUMERIC
[6]     V←CNF ACC
[7]     →L2
[8]   L1:V1←(ACC='-')/⍳⍴,ACC
[9]     V←ACC
[10]    V[V1]←'-'
[11]    V←⍕(⍴V)↑(V≠' ')/V
[12]  L2:ΔOUTF←ΔOUTF,V
[13]    'XX ONE SAMPLE IS WRITTEN TO THE OUTPUT FILE ... '
        ∇
```

```
      ⍝  DOCUMENTATION OF WORKSPACE ** DISPIC2 **

      DESCRIBE

WORKSPACE CONTAINS  ** VERSION 2 OF DISPIC **

      ENVIRONMENT :
          WSID    -   DISPIC2
          WIDTH   -   130
          ORIGIN  -   0

      ** FNS **
ΔBUILD  ΔDEFINE  ΔPARAMS  ΔSOURCE  ΔSYMTAB  ADDL  BRANCH
COUNTER  CPU1  DECODE DECODE4 DECODE5  DINIT1  DISPIC1
ENCODE  EXECUTE  FETCH  INTERRUPT  LOAD  MLTL  OPCODE
STATE  WITH

      ** VARS **
ΔIET  ΔIMOP  ΔMZ  ΔOPTAB  ΔOPT  ΔST  ACC  ADR  BUFFER  BST
CC  EST  GRPAUXIL  GRPDISPIC1  IC  IR  MP001  MA  MD MEMORY
OP  P001  P2  SMP001

      ** GRPS **
GRPAUXIL  GRPDISPIC1

      WHERE,

** GROUP GRPAUXIL **
ΔBUILD  ΔDEFINE  ΔPARAMS  ΔSOURCE  ΔSYMTAB  STATES

** GROUP GRPDISPIC1 **
DINIT1  ΔDEFINE  DISPIC1  LOAD  CPU1  BRANCH  INTERRUPT
EXECUTE  FETCH  COUNTER  OPCODE  DECODER4  DECODER5  ADDL
STORE  DECODE  ENCODE  MLTL


      NOTE :

      THE ASSEMBLER GROUP IS CONTAINED IN **WSID DISPIC3**
```

```
      ΔLISTGRP GRPAUXIL

** WORKSPACE : DISPIC2          ORIGIN : 0


FUNCTIONS TO BE LISTED;


ΔBUILD
ΔDEFINE
ΔPARAMS
ΔSOURCE
ΔSYMTAB
STATES


======================


      ∇R←ΔBUILD;IN;N;N1
[1]    ⎕IO←0
[2]    N1←(ΔIET[1]⍴1);0;ΔIET[3]⍴1
[3]    R←(0,¯1+N←1++/ΔIET)⍴' '
[4]    '** PLEASE ENTER; CODE,SPACE,ADDRESS ... (→ TO END)'
[5]    (ΔIET[1]⍴'C'),' ',ΔIET[3]⍴'⎕'
[6]   L1:→('→'=1↑IN←⍞)/END
[7]    R←R,[0]N1/N↑IN
[8]    →L1
[9]   END:' '
[10]   '** COMPLETED ... '
[11]   ' '
       ∇


       ∇ΔDEFINE
[1]    ⍝Δ  IC = INSTRUCTION COUNTER,  MA = MEMORY ADDRESS REGISTER
[2]    ⍝Δ   IR = INSTRUCTION REGISTER, MD =
[3]    ⍝Δ   MEMORY = MEMORY
[4]    IC←MA←ΔMZ[4]⍴0
[5]    IR←MD←ΔMZ[2]⍴0
[6]    MEMORY←ΔMZ[1 2]⍴0
[7]    ⍝  DECODER3 AND DECODER5  NEED BE DEFINED
[8]    '** MACHINE DEFINED ... READY TO RUN'
[9]    ⍝  DECLARE OPCODE AND ADDRESS FIELDS
[10]   OP←ΔIET[1]
[11]   ADR←ΔIET[3]
       ∇


       ∇R←ΔSYMTAB X;FMT;NX
[1]    ' '
[2]    NX←2⊥⍉'01'⍷X[;ΔSI[0]+⍳ΔSI[1]]
[3]    ((ΔSI[0]-3)⍴' '),'NAME      LOCATIONS'
[4]    FMT←'X2,',(⍕ΔSI[0]),'A1,X4,I',⍕ΔSI[1]
[5]    FMT ⎕FMT(X[;⍳ΔSI[0]];NX)
       ∇
```

```
     ∇ΔPARAMS;⎕IO;LE
[1]  .⎕IO←0
[2]   ⎕←LE←⎕AV[10]
[3]   '** PARAMETERS OF THE ASSEMBLER/MACHINE DATA FORMATS **'
[4]   ' '
[5]   '1.   INSTRUCTION FORMAT : LABEL, OPCODE, INDEX REG,, ADDRESS'
[6]   '   ΔIFI    ';ΔIFI;LE
[7]   '2.   SYMBOL TABLE :  VARIABLE NAME, MEMORY LOCATION'
[8]   '   ΔST     ';ΔST;LE
[9]   '3.   INSTRUCTION/OP-CODE LENGTH (ΔOPI) : ';ΔOPI;LE
[10]  '4.   MEMORY SIZE : NO, OF MEMORY, NO, OF WORDS, WORD LENGTH,
      OPCODE FIELD, ADDRESS FIELD'
[11]  '   ΔMZ      ';ΔMZ;LE
[12]  '5.   OP-CODE TABLE (ΔOPTAB) : '
[13]  ΔOPTAB;LE
[14]  '6.   IMMEDIATE OPERATION CODES (ΔIMOP) :'
[15]  ΔIMOP
[16]  ' '
      ∇

      ∇STATES
[1]   ' '
[2]   ' ** STATES OF THE MACHINE **'
[3]   ' '
[4]   'IC     : ';IC
[5]   'IR     : ';IR
[6]   'MA     : ';MA
[7]   'MD     : ';MD
[8]   'MSW    : ';MSW
[9]   ' '
      ∇

      ∇ΔSOURCE P;FMT
[1]   ' '
[2]   ' '
[3]   (ΔIFI[1]⍴'C'),' ',ΔIFI[3]⍴'D'
[4]   FMT←(⍕ΔIFI[1]),'A1,X1,',(⍕ΔIFI[3]),'A1'
[5]   FMT ⎕FRMT P
      ∇
```

```
      ΔLISTGRP GRPDISPIC1

** WORKSPACE : DISPIC2          ORIGIN : 0


 FUNCTIONS TO BE LISTED:


DINIT1
ΔDEFINE
DISPIC1
LOAD
CPU1
BRANCH
INTERRUPT
EXECUTE
FETCH
COUNTER
OPCODE
DECODER4
DECODER5
ADDL
STORE
DECODE
ENCODE
MLTL


 ========================

      ∇DINIT1
[1]    ΔIFI←0 4 0 5
[2]    ΔST←5 5
[3]    ΔOPI←4
[4]    ΔMZ←1 24 9 4 5
[5]    ΔOPTAB←9 4⍴'HLT LAD STO INP OUT ADD CST BRH MLT '
[6]    ΔIMOP←2 4⍴'BRH CST '
[7]    ⍝ DEFINE THE BRANCH ON CONDITIONS
[8]    BST←0
       ∇

      ∇ΔDEFINE
[1]    ⍝⍝  IC = INSTRUCTION COUNTER,  MA = MEMORY ADDRESS REGISTER
[2]    ⍝⍝   IR = INSTRUCTION REGISTER, MD =
[3]    ⍝⍝   MEMORY = MEMORY
[4]    IC←MA←ΔMZ[4]⍴0
[5]    IR←MD←ΔMZ[2]⍴0
[6]    MEMORY←ΔMZ[1 2]⍴0
[7]    ⍝  DECODER3 AND DECODER5  NEED BE DEFINED
[8]    '** MACHINE DEFINED ,,, READY TO RUN'
[9]    ⍝  DECLARE OPCODE AND ADDRESS FIELDS
[10]   OP←ΔIFI[1]
[11]   ADR←ΔIFI[3]
       ∇
```

```
      ∇R←BRANCH;BR
[1]    →(0=R←(BR←MSW⌊1)∈BST)/NEXT
[2]    →(BST∈BR)/CD1
[3]    CD1:' HALT CONDITION '
[4]    '      ** SET INTERRUPT FLAG **'
[5]    ESP←1
[6]    NEXT:
       ∇

      ∇R←INTERRUPT
[1]    ⍝ IF INTERRUPT FLAG (ESP)  SET , HALT
[2]    R←1∧ESP
       ∇

      ∇FETCH;SW
[1]    SW←∆MΣ[1]↑DECODER5 MA
[2]    MD←,SW/MEMORY
       ∇

      ∇R←N COUNTER IC
[1]    ⍝ N ←→ +VE UP COUNTER, -VE DOWN COUNTER
[2]    R←((⍴,IC)⍴2)⊤N+2⊥IC
       ∇

      ∇R←OPCODE OP
[1]    R←(1↑⍴∆OPTAB)↑DECODER4 OP
       ∇

      ∇R←DECODER4 ADR;SW
[1]    SW←ADR∘.=0 1
[2]    R←SW[0;]∘.∧SW[1;]∘.∧SW[2;]∘.∧SW[3;]
[3]    R←,R
       ∇

      ∇Z←DECODER5 ADR;SW
[1]    SW←ADR∘.=0 1
[2]    Z←SW[0;]∘.∧SW[1;]∘.∧SW[2;]∘.∧SW[3;]∘.∧SW[4;]
[3]    Z←,Z
       ∇

      ∇STORE OBJECT;SW
[1]    SW←DECODER5 MA
[2]    MEMORY[SW⌊1;]←MD←OBJECT
       ∇

      ∇Z←DECODE OUTPUT
[1]    Z←2⊥OUTPUT
       ∇

      ∇Z←ENCODE INPUT
[1]    Z←(9⍴2)⊤INPUT
       ∇

      ∇R←A MLTL B
[1]    R←((⍴A)⍴2)⊤(2⊥A)×(2⊥B)
       ∇
```

```
      ∇Z←A ADDL B;C
[1]    Z←A≠B
[2]    C←A∧B
[3]    →(C∧.=0)/0
[4]    A←Z
[5]    B←1↓C,0
[6]    →1
       ∇
```

```
      ∇DISPIC1 PROG
[1]    ⍝  LOAD PROGRAM INTO MEMORY
[2]    0 LOAD PROG
[3]    ⍝  RESET FLAGS (EST = BRANCH ON CONDITION)
[4]    ESP←0
[5]    ⍝ RUN PROG
[6]    CPU1
       ∇

      ∇ADR LOAD PROG
[1]    IC←(⍴IC)⍴0
[2]    MEMORY[ADR+⌊1↑⍴PROG;]←PROG
       ∇

      ∇CPU1;MSW
[1]    ⍝ SET MEMORY ACCESS REGISTER AS IC
[2]   NEXT:→INTERRUPT/HALT
[3]    MA←IC
[4]    FETCH
[5]    IR←MD
[6]    MA←OP↓IR
[7]    ⍝ INCREMENT COUNTER
[8]    IC←1 COUNTER IC
[9]    ⍝ DECODE OP-CODE
[10]   MSW←OPCODE OP↑IR
[11]   ⍝ BRANCH ON CONDITION SPECIFIED BY BST
[12]   →BRANCH/NEXT
[13]   ⍝ EXECUTE INSTRUCTION
[14]   →(1=EXECUTE MSW)/HALT
[15]   →NEXT
[16]  HALT:
       ∇

      ∇R←EXECUTE MSW
[1]    R←0
[2]    →MSW/(HA,LO,ST,IN,OU,AD,CO,BR,ML)
[3]   HA:⍝ SET FLAG TO HALT
[4]    →⍵R←1
[5]   LO:FETCH
[6]    ACC←MD
[7]    →NEXT
[8]   ST:STORE ACC
[9]    →NEXT
[10]  IN:ACC←BUFFER←ENCODE □
[11]   →NEXT
[12]  OU:FETCH
[13]   □←DECODE BUFFER←ACC←MD
[14]   →NEXT
[15]  AD:FETCH
[16]   ACC←ACC ADDL MD
[17]   →NEXT
[18]  CO:ACC←IR
[19]   ACC[0 1 2 3 4]←0
[20]   →NEXT
[21]  BR:MA←(∆MΣ[4]⍴2)⊤∆MΣ[1]-1
[22]   FETCH
[23]   CC←ACC∧.=MD
[24]   →(CC=1)/NEXT
[25]   IC←4↓IR
[26]   →NEXT
[27]  ML:FETCH
[28]   ACC←ACC MLTL MD
```

```
      ⍝  DOCUMENTATION OF WORKSPACE ** DISPIC3 **

      DESCRIBE

WORKSPACE CONTAINS ** VERSION 3 OF DISPIC3 **

      ENVIRONMENT :
          WSID    -   DISPIC3
          WIDTH   -   130
          ORIGIN  -   0

      ** FNS **
ΔBUILD  ΔDEFINE  ΔGRPS  ΔLISTGRP  ΔPARAMS  ΔSCOURCE  ΔSYMTAB
ADCODE  ADDCODE  ADDL  ADDRESS  AINIT  AINIT1  ASSEMBLE
BRANCH  COUNTER  CPU  CPU1  DECODE  DECODER3  DECODER4  DECODER5
DINIT1  DINIT2  DISPIC1  ENCODE  EXECUTE  FETCH  FETCH4  IMOPR
IN INTERRUPT  LOAD  MACHINE  MLTL  OPCODE  QPCODE  STATES
STORE WITH

      ** VARS **
ΔCOL  ΔFMT  ΔIET  ΔIMOP  ΔMZ  ΔMZ2  ΔOPTAB  ΔOPT  ΔST  ACC
ADR  BUFFER  BUFFER  EST  CC  EST  GRPAUXIL  GRPMC001
GRPASSEMBLE  GRPDISPIC1  IC  IR  MP001  MP011  MA  MD MEMORY
OP  P001  P2  SMP001  SMP011  XX1

      ** GRPS **
GRPAUXIL  GRPMC01  GRPUTIL  GRPASSEMBLE  GRPDISPIC2

      WHERE,

** GROUP GRPAUXIL **
ΔBUILD  ΔDEFINE  ΔPARAMS  ΔSOURCE  ΔSYMTAB  STATES

** GROUP GRPMC01 **
ΔDEFINE  MACHINE  LOAD  CPU  FETCH4  DECODER3  DECODER5  ADDL
STORE  DECODE  ENCODE

** GROUP GRPUTIL **
ΔGRPS  ΔLISTGRP

** GROUP GRPASSEMBLE **
ΔBUILD  ΔSOURCE  ΔPARAMS  ΔSYMTAB  ASSEMBLE  QPCODE  IN  ADCODE
IMOPR  WITH  ADDRESS  AINIT  AINIT1  DINIT2

** GROUP GRPDISPIC2 **
DINIT1  ΔDEFINE  DISPIC1  LOAD  CPU1  BRANCH  INTERRUPT  EXECUTE
FETCH  COUNTER  OPCODE  DECODER4  DECODER5  ADDL  STORE  DECODE
ENCODE  MLTL
```

(The assembler group is listed on the next page)

```
      ΔLISTGRP GRPASSEMBLE

** WORKSPACE : DISPIC3          ORIGIN : 0

FUNCTIONS TO BE LISTED;

ΔBUILD
ΔSOURCE
ΔPARAMS
ΔSYMTAB
ASSEMBLE
QPCODE
IN
ADCODE
IMOPR
WITH
ADDRESS
AINIT
AINIT1
DINIT2

=======================
```

```
      ∇R←ΔBUILD;IN;N;N1
[1]    ⎕IO←0
[2]    N1←(ΔIET[1]⍴1),0,ΔIET[3]⍴1
[3]    R←(0,¯1+N←1++/ΔIET)⍴' '
[4]    '** PLEASE ENTER; CODE,SPACE,ADDRESS ... (→ TO END)'
[5]    (ΔIET[1]⍴'C'),' ',ΔIET[3]⍴'⎕'
[6]   L1:→('→'=1↑IN←⎕)/END
[7]    R←R,[0]N1/N↑IN
[8]    →L1
[9]   END:' '
[10]   '** COMPLETED ... '
[11]   ' '
      ∇

      ∇ΔSOURCE P;FMT
[1]    ' '
[2]    ' '
[3]    (ΔIET[1]⍴'C'),' ',ΔIET[3]⍴'⎕'
[4]    FMT←(⍕ΔIET[1]),'A1,X1,',(⍕ΔIET[3]),'A1'
[5]    FMT ⎕FRMT P
      ∇
```

```
      ∇R←ΔSYMTAB X;FMT;NX
[1]    ' '
[2]    NX←2⊥⍉'01'∘.=X[;ΔSI[0]+⍳ΔSI[1]]
[3]    ((ΔSI[0]-3)⍴' '),'NAME      LOCATIONS'
[4]    FMT←'X2,',(⍕ΔSI[0]),'A1,X4,I',⍕ΔSI[1]
[5]    FMT ⎕FRMT(X[;⍳ΔSI[0]];NX)
      ∇
```

```
     ∇ΔPARAMS;⎕IO;LE
[1]    ⎕IO←0
[2]    ⎕←LE←⎕AV[10]
[3]    '** PARAMETERS OF THE ASSEMBLER/MACHINE DATA FORMATS **'
[4]    ' '
[5]    '1.   INSTRUCTION FORMAT : LABEL, OPCODE, INDEX REG., ADDRESS'
[6]    '   ΔIFI    ';ΔIFI;LE
[7]    '2.   SYMBOL TABLE :  VARIABLE NAME, MEMORY LOCATION'
[8]    '   ΔST     ';ΔST;LE
[9]    '3.   INSTRUCTION/OP-CODE LENGTH (ΔOPI) : ';ΔOPI;LE
[10]   '4.   MEMORY SIZE : NO. OF MEMORY, NO. OF WORDS, WORD LENGTH,
       OPCODE FIELD, ADDRESS FIELD'
[11]   '   ΔMZ      ';ΔMZ;LE
[12]   '4B.  DATA MEMORY : NO. OF WORDS, WORD LENGTH'
[13]   '   ΔMZ2     ';ΔMZ2;LE
[14]   '5.   OP-CODE TABLE (ΔOPTAB) : '
[15]   ΔOPTAB;LE
[16]   '6.   IMMEDIATE OPERATION CODES (ΔIMOP) :'
[17]   ΔIMOP
[18]   ' '
       ∇

       ∇R←ASSEMBLE P;ARG;I;INST;X1;X2;ST;S;WRD
[1]    R←⍳0
[2]    ⍝ CHECK IF MEMORY SPECIFICATIONS ARE CONSISTENT
[3]    ⍝ PROGRAM MEMORY CHECK
[4]    →(ΔMZ[2]=+/¯2↑ΔMZ)/OK1
[5]    '$$ INCONSISTENT WORD LENGTH FOR PROG. MEMORY ... ABORT'
[6]    →0
[7]    OK1:→(ΔMZ2[0]≥2*ΔMZ[4])/OK2
[8]    '$$ INSUFFICIENT ADDRESS TO REFERENCE DATA MEMORY ... ABORTED'
[9]    →0
[10]   ⍝ CHECK IF PROG. ⍴ MEMORY
[11]   OK2:→(ΔMZ[1]≥1↑⍴P)/OK3
[12]   '$$ PROGRAM EXCEEDS MEMORY CAPACITY ... ABORTED'
[13]   →0
[14]   ⍝ CHECK IF MEMORY ⍴ ADDRESS FIELD
[15]   OK3:→(ΔMZ[1]≥2*ΔMZ[4])/OK4
[16]   '$$ ADDRESS FIELD INSUFFICIENT TO ADDRESS MEMORY ... ABORTED'
[17]   →0
[18]   OK4:X1←0+⍳ΔIFI[1]
[19]   X2←(+/3↑ΔIFI)+⍳ΔIFI[3]
[20]   WRD←⌈2⍟ΔMZ[1]
[21]   R←ΔMZ[1 2]⍴0
[22]   ST←(0,+/ΔST)⍴' '
[23]   I←0
[24]   L1:→((1↑⍴P)<I←I+1)/L2
[25]   INST←P[I-1;X1]
[26]   ARG←P[I-1;X2]
[27]   R[I-1;]←(OPCODE INST),ADCODE ARG
[28]   →((IMOPR INST)∨∧/ARG=' ')/L1
[29]   →(∧/ARG∈'0123456789 ')/L1
[30]   ST←ST WITH ARG
[31]   R[I-1;ΔMZ[3]+⍳ΔMZ[4]]←ST ADDRESS ARG
[32]   →L1
[33]   L2:⎕←S←'SYMBOL TABLE NAME?   '
[34]   S←(⍴,S)↓⍞
[35]   ⍎S,'←ST'
[36]   '** END OF ASSEMBLING ...'
       ∇

       ∇R←OPCODE X
[1]    R←,(ΔMZ[3]⍴2)⊤ΔOPTAB IN X
       ∇

       ∇R←TAB IN X;N
[1]    N←¯1↑⍴TAB
[2]    R←R/⍳⍴R←TAB∧.=N↑X
       ∇

       ∇R←ADCODE X
[1]    X←(' '≠X)/X ⍝ REMOVE SPACE
[2]    R←(ΔMZ[4]⍴2)⊤10⊥10|'0123456789'⍳X
       ∇

       ∇R←IMOPR X;N
[1]    N←¯1↑⍴ΔIMOP
[2]    R←+/ΔIMOP∧.=N↑X
       ∇

       ∇R←ST WITH NEW;ADR
[1]    R←ST
[2]    ⍝ RIGHT-JUSTIFY NAME
[3]    NEW←(' '≠NEW)/NEW
[4]    NEW←ΔST[0]↑NEW
[5]    →0×⍳∨/ST[;⍳ΔST[0]]∧.=NEW
[6]    ⍝ CHECK IF DATA MEMORY CAPACITY EXCEEDED
[7]    →(ΔMZ2[0]≥ADR←1+(⍴ST)[0])/OK1
[8]    '$$ DATA MEMORY CAPACITY EXCEEDED ... ABORTED AT : '
       ,(⍕I),' ',P[I-1;]
[9]    ⍝ RIGHT OUT
[10]   →
[11]   OK1:ADR←(WRD⍴2)⊤ADR
[12]   ADR←'01'[ADR]
[13]   R←ST,[0]NEW,ADR
       ∇

       ∇R←ST ADDRESS ARG;ADR
[1]    ⍝ RIGHT-JUSTIFY ARG
[2]    ARG←ΔST[0]↑(ARG≠' ')/ARG
[3]    ADR←ST[;⍳ΔST[0]]IN ARG
[4]    ADR←ST[ADR;ΔST[0]↓⍳+/ΔST]
[5]    R←'0123456789'⍳ADR
       ∇

       ∇AINIT
[1]    ΔIFI←0 1 0 2
[2]    ΔST←1 5
[3]    ΔOPI←1
[4]    ΔMZ←1 32 8 3 5
[5]    ΔOPTAB←8 1⍴'HLSIDACB'
[6]    ΔIMOP←2 1⍴'BC'
       ∇

       ∇DINIT2
[1]    ΔIFI←0 4 0 5
[2]    ΔST←5 5
[3]    ΔOPI←4
[4]    ΔMZ←1 24 9 4 5
[5]    ⍝  PARAMS OF 2ND MEMORY (DATA)
[6]    ΔMZ2←20 9
[7]    ΔOPTAB←9 4⍴'HLT LAD STO INP OUT ADD CST BRH MLT '
[8]    ΔIMOP←2 4⍴'BRH CST '
[9]    ⍝ DEFINE THE BRANCH ON CONDITIONS
[10]   BST←0
```

```
       ∇AINIT1
[1]    ΔIFI←0 3 0 3
[2]    ΔST←3 5
[3]    ΔOPI←3
[4]    ΔMZ←1 32 8 3 5
[5]    ΔOPTAB←8 3⍴'HLTLADSTOINPOUTADDCSTBRH'
[6]    ΔIMOP←2 3⍴'BRHCST'
       ∇
```

Appendix 14 :

SOURCE CODE LISTING OF "SIMLOGS" [1]

---

[1] This is the basic version of SIMLOGS; i.e. this version may be executed in any installation which supports "basic APL".

```
      ⍝ DOCUMENTATION OF WORKSPACE ** SIMLOGS **

      DESCRIBE

WORKSPACE CONTAINS  ** SIMULATOR FOR LOGIC SYSTEMS (SIMLOGS) **

      ENVIRONMENT :
          WSID    -   SIMLOGS
          WIDTH   -   72
          ORIGIN  -   1

      ** FNS **
ADDER   AND ASSIGN  BIN  CIRCUIT  CLOCK  CNTR  CODE  COMPILE
CYCLING DEC  DELAY  ENC EXC EXOR  FILL FIND  FORM  FUNC   GE
GO  GRAPH  HAE  HOLD  INPUTDATA  INPUTS  JK1  JK2  LE LOGIC
ME  MONO  NAND  NOR  OR  OUTPUTS  PLOT  PRINT  PR8   RANDOM
READ  ROM  SELECT  SET  SETTIME SHE  SHOW  SHR  SH8   SN5420
SN5472  SN7401  SPACING   STACK  TCE  TIE  TIMINGS TRANSLATE
UPDATE  WIPE  WRITE

      ** VARS **
ADDRESS AV DATA D2 ELEM EXIT FP FROM NE NG NH NJ NM NS NT Q
QQ RN TYPE X  XXX  Z  ZZZ

      ** GRPS **
GRPICS  GRPSNPK  GRPSIMLOG

      WHERE,

** GROUP GRPICS **
ADDER  CNTR  EXC  EXOR  FNC  MONO  PR8  SH8

** GROUP GRPSNPK **
SN5420  SN5470  SN5472  SN7401

** GROUP GRPSIMLOG **
AND ASSIGN  BIN  CLOCK  CODE  COMPILE  CYCLING  DEC  DELAY
ENC  EXOR  FILL  FIND  FORM  GO GRAPH  GE  HAE  HOLD
INPUTDATA  INPUTS  JK1  JK2  LOGIC  LE  MONO  ME  NAND NOR
OR  OUTPUTS  PLOT  PRINT  PR8  RANDOM  READ  ROM  SELECT
SET SETTIME  SHOW  SHR  SHE  SPACING  STACK  TCE  TIE
TIMINGS  TRANSLATE  UPDATE  WIPE  WRITE


      N.B. :   THIS IS THE BASIC VERSION OF **SIMLOGS**
               I.E., THIS VERSION WILL RUN ON ALL
               INSTALLATION SUPPORTING  BASIC  APL
```

```
      ∇ Z ADDER X
[1]   INPUTS←17
[2]   (17↑16)HAE(1 2 10),,⍉(3 7)⍴(19+2×¯1+⍳7),(3+¯1+⍳7),11+¯1+⍳7
[3]   OUTPUTS(18+2×¯1+⍳8),33
      ∇

      ∇ Z AND X
[1]   Z GE 1,X
      ∇

      ∇ ASSIGN
[1]   TX←ELEM[(ADDRESS↑X+(10*2×RN)|AV)]
      ∇

      ∇ RR←BIN X
[1]   RR←X+.×2*(¯1+⍴X)-⍳(¯1+⍴X)
      ∇

      ∇ CIRCUIT
[1]   INPUTS 1 2
[2]   3 EXC 1 2
[3]   OUTPUTS 3
      ∇

      ∇ RR←A CLOCK B
[1]   PORMFLAG←1
[2]   PP←(A+B)|TIME
[3]   RR←(PP>A)+PP=0
[4]   INC←INC⌊/(((RR=0)×A-PP)+RR×(B+A)-PP)×(PP>1)×PP×A+1
[5]   →(STEADY=0)/0
[6]   INC←INC⌊SPACE-CYCLES
[7]   TIME←TIME+INC
[8]   CYCLES←CYCLES+INC
[9]   INC←0
[10]  RR←(A<(A+B)|TIME)+0=(A+B)|TIME
      ∇

      ∇ Z CNTR X
[1]   INPUTS↑3
[2]   4 5 6 7 JK1 12 1 12 3 2 4 1 4 3 2
[3]   8 9 10 11 AND 5 7 4 7 5 6 4 6,CODE 0
[4]   12 NAND 3
[5]   OUTPUTS 8 9 10 11
      ∇

      ∇ RR←CODE XX
[1]   DATA←XX
[2]   RR←⍳0
      ∇
```

```
    ∇ COMPILE
[1]   NT←NH←NS←NE←NJ←NM←AV←NG←RN←EXIT←0
[2]   ELEM←ADDRESS←FROM←⍳0
[3]   PP←1
[4]   CIRCUIT
[5]   →EXIT/L1
[6]   FILL
[7]   →EXIT/L1
[8]   GAA←(NG,5)⍴0
[9]   GAB←NG⍴0
[10]  GAC←NG⍴0
[11]  JAA←(NJ,5)⍴0
[12]  JAB←(NJ,2)⍴0
[13]  JAD←JAC←(NJ,1)⍴0
[14]  MAA←(NM,7)⍴0
[15]  MAB←(NM,8)⍴0
[16]  MAC←(NM,1)⍴0
[17]  SHA←(NS,7)⍴0
[18]  SHB←(NS,4)⍴0
[19]  SHD←(NS,2)⍴0
[20]  HAA←(NH,3)⍴0
[21]  HAB←(NH,2)⍴0
[22]  TCA←(NT,6)⍴0
[23]  TCB←(NT,4)⍴0
[24]  MEM←((1+NM×64),8)⍴0
[25]  PARAM←(NE,6)⍴0
[26]  DATA←NT←NH←NS←PP←NE←RN←NG←NJ←NM←AV←EXIT←0
[27]  CIRCUIT
[28]  TIE
[29]  WIPE
[30]  TIMEINGS NOM
[31]  FORM
[32]  →0
[33]  L1:'MAKE CORRECTIONS AND COMPILE AGAIN'
    ∇


    ∇ RR←CYCLING
[1]   →FORMFLAG/L1
[2]   CLOCK←0
[3]   FORMFLAG←1
[4]   L1:RR←CLOCK←(CLOCK∧~STEADY)∨STEADY∧~CLOCK
    ∇


    ∇ RR←DEC MA
[1]   RR←+/MA×10*2×⍳⍴MA
    ∇


    ∇ N DELAY T
[1]   TD[RO[RRO⍳N]]←PARAM[RO[RRO⍳N];T]
    ∇


    ∇ RR←ENC AV
[1]   RR←100|⌊100×AV÷10*2×⍳⌈0.5×10⍟AV
    ∇
    ∇ Z EXC X
[1]   INPUTS 1 2
[2]   4 5 SN5420 1 1 3 3,2 2 3 3
[3]   6 3 SN5420 4 4 5 5,1 1 2 2
[4]   OUTPUTS 6
    ∇


    ∇ Z EXOR X
[1]   Z GE 3,X
    ∇
    ∇ FILL
[1]   →(RN≠0)/ERR
[2]   L1:Q←(ELEM=0)/⍳⍴ELEM
[3]   →(0=⍴Q)/0
[4]   QQ←ADDRESS⍳FROM[Q]
[5]   →(∨/(⍴ELEM)<QQ)/L2
[6]   →(∧/ELEM[Q]=ELEM[QQ])/ER1
[7]   ELEM[Q]←ELEM[QQ]
[8]   →L1
[9]   L2:'IMPROPERLY DEFINED LEAD '
[10]  (FROM[Q])[(QQ>⍴ELEM)⍳1]
[11]  →0,EXIT←1
[12]  ERR:'DEFN,ERROR IN CIRCUIT OR MODULES'
[13]  →0,EXIT←1
[14]  ER1:'IMPROPERLY DEFINED LEAD  ';ENC ADDRESS[1+QQ]
[15]  EXIT←1
    ∇


    ∇ RR←FIND XX;Q
[1]   Q←((FROM=0)∧ELEM=XX)/ADDRESS
[2]   →(1≠⍴Q)/L1
[3]   RR←ENC Q
[4]   →0
[5]   L1:'NOT FOUND'
    ∇


    ∇ FORM
[1]   J←JAA[;1]
[2]   C←JAA[;2]
[3]   K←JAA[;3]
[4]   S←JAA[;4]
[5]   R←JAA[;5]
[6]   JAA←⍳0
[7]   Q1←JAB[;1]
[8]   Q2←JAB[;2]
[9]   JAB←⍳0
[10]  MEM←,0 6↑MAA
[11]  MAA←0 ¯1↓MAA
[12]  SC←SHA[;5 6]
[13]  SN←⍉(4,NS)⍴,SHA[;7]
[14]  SHA←0 ¯3↓SHA
[15]  TC1←,TCA[;5]
[16]  TC2←,TCA[;6]
[17]  TCA←0 ¯2↓TCA
    ∇
    ∇ Z GE X
[1]   TYPE←X[1]
[2]   X←1↓X
[3]   Z←,Z
[4]   →(~FP)/L1
[5]   TRANSLATE
[6]   →L2
[7]   L1:ASSIGN
[8]   STACK TYPE,1 5,DATA
[9]   →EXIT/0
[10]  GAA[NG+⍳⍴,Z;]←A
[11]  GAB[NG+⍳⍴,Z]←,B
[12]  GAC[NG+⍳⍴,Z]←,D
[13]  PARAM[NE+⍳⍴,Z;]←C
[14]  L2:NG←NG+⍴,Z
```

```
    ∇ GO
[1]   EN←NE
[2]   TIME←CYCLES←RF←STEADY←0
[3]   L1:TIME←TIME+1
[4]   INC←1000
[5]   CF←RF
[6]   CYCLES←CYCLES+1
[7]   INPUTDATA
[8]   LOGIC
[9]   UPDATE
[10]  →(RF×0=CF)/L2
[11]  →(0=((CYCLES≥PRESET)∧TIMEFLAG)∨(STEADY∧~FORMFLAG)
        ∨SPACEFLAG∧CYCLES≥SPACE)/L1
[12]  PRINTOUT
[13]  CYCLES←0
[14]  →(STEADY∧~FORMFLAG)/0
[15]  →L1
[16]  L2:→PF/L1
[17]  PRINTOUT
[18]  →L1
    ∇


    ∇ RR←GRAPH XX;K1;K2;KV
[1]   RR←,GM[1+GV+2×XX;]
[2]   GV←XX
    ∇


    ∇ Z HAE X
[1]   →(~FP)/L1
[2]   TRANSLATE
[3]   →L2
[4]   L1:ASSIGN
[5]   STACK 0 2 3 0
[6]   →EXIT/0
[7]   HAA[NH+⍳D1;]←A
[8]   HAB[NH+⍳D1;]←B
[9]   PARAM[NE+⍳⍴Z;]←C
[10]  L2:NH←NH+(⍴Z)÷2
[11]  NE←NE+⍴Z
    ∇


    ∇ N HOLD T
[1]   TH[RO[RRO⍳N]]←PARAM[RO[RRO⍳N];1+T]
    ∇


    ∇ INPUTDATA
[1]   X[3 4]←0 1
[2]   X[2]←20 CLOCK 20
    ∇


    ∇ INPUTS XX
[1]   →(RN≠0)/3
[2]   X←XXX←XX
[3]   AV←¯1
[4]   L3:RN←RN+1
[5]   AV←AV+10*2×RN-1
[6]   →(~FP)/L2
[7]   →((⍴,X)≠⍴,XX)/L4
[8]   ADDRESS←ADDRESS,AV+XX
[9]   FROM←FROM,X+(10*2×RN-1)|AV
[10]  ELEM←ELEM,(⍴,X)⍴0
[11]  →(RN≠1)/0
[12]  ELEM←⍳⍴,X
[13]  FROM←(⍴,X)⍴0
[14]  NE←NE+⍴,X
[15]  →0
[16]  L2:→(RN≠1)/0
[17]  NE←NE+⍴,X
[18]  →0
[19]  L4:'INPUT LENGTH ERROR IN MODULE ';1+NEC AV
[20]  EXIT←1
    ∇


    ∇ Z LE X
[1]   TYPE←X[1]
[2]   X←1↓X
[3]   Z←,Z
[4]   →(~FP)/L1
[5]   TRANSLATE
[6]   →L2
[7]   L1:ASSIGN
[8]   STACK TYPE,2 5 0
[9]   →EXIT/0
[10]  JAA[NJ+⍳D1;]←A
[11]  JAB[NJ+⍳D1;]←B
[12]  JAC[NJ+⍳D1;]←D
[13]  PARAM[NE+⍳⍴,Z;]←C
[14]  L2:NJ←NJ+(⍴Z)÷2
[15]  NE←NE+⍴,Z
    ∇


    ∇ Z ME X
[1]   →(~FP)/L1
[2]   TRANSLATE
[3]   →L2
[4]   L1:ASSIGN
[5]   STACK 0 8 7 0
[6]   →EXIT/0
[7]   MAA[NM+⍳D1;]←A
[8]   MAB[NM+⍳D1;]←B
[9]   MAC[NM+⍳D1;]←NM+¯1+⍳(⍴Z)÷8
[10]  PARAM[NE+⍳⍴Z;]←C
[11]  L2:NM←NM+(⍴Z)÷8
[12]  NE←NE+⍴Z
    ∇
```

```
    ∇ Z JK1 X
[1]   Z LE 1,X
    ∇

    ∇ Z JK2 X
[1]   Z LE 2,X
    ∇
```

```
    ∇ LOGIC;TRG;TRG1;TRG2;K1;K2
[1]   ⍝RELATE INPUTS
[2]   L[SSX]←V[SSX]←X
[3]   ⍝ GATE LOGIC
[4]   L[GAB]←((K1=GAC←1)∧GAC<3)∨((K2=GAC←4)∧GAC>3)∨(~K1←∧/V[GAA])∧(K2←∨/V[GAA])∧GAC←3
[5]   ⍝ JK LOGIC
[6]   TRG←V[C]∧~,JAD
[7]   JAD[;1]←V[C]
[8]   L[Q1]←(((V[J]∧TRG∧~V[Q1])∨V[Q1]∧~V[K]∧TRG)∧V[S]≠V[R])∨V[S]∧((~V[R])∨,JAC=2)
[9]   L[Q2]←((~L[Q1])∧V[S]≠V[R])∨V[R]∧((~V[S])∨,JAC=2)
[10]  ⍝ MEMORY LOGIC
[11]  L[MAB]←MEM[1+V[MEN]×1+(,MAC×64)+BIN V[MAA];]
[12]  ⍝ SHIFT LOGIC
[13]  TRG←SHD∧~V[SC]
[14]  TRG1←⍉(4,NS)ρTRG[;1]
[15]  TRG2←⍉(4,NS)ρTRG[;2]
[16]  SHD←V[SC]
[17]  L[SHB]←(V[SHA]∧TRG2∧~V[SM])∨(V[SM]∧TRG1∧¯1⌽V[SHB])∨V[SHB]∧TRG1≠TRG2
[18]  L[SHB[;1]]←(L[SHB[;1]]∧~TRG[;1])∨V[SHA[;1]]∧TRG[;1]
[19]  ⍝ HALF ADDER LOGIC
[20]  L[HAB[;1]]←=/V[HAA]
[21]  L[HAB[;2]]←2≤+/V[HAA]
[22]  ⍝ TRUTH/COMPLEMENT LOGIC
[23]  L[TCB]←(((V[TCA]∧⍉(4,NT)ρV[TC1])∨((~V[TCA])∧~⍉(4,NT)ρV[TC1]))∧~⍉(4,NT)ρV[TC2])∨⍉(4,NT)ρV[TC2
      ])∨⍉(4,NT)ρV[TC2]∧~V[TC1]
    ∇
```

```
    ∇ Z MONO Z
[1]   DATA←0,X[1]
[2]   X←1↓X
[3]   INPUTS 1 2
[4]   3 4 JK1 7 1 2 2 5
[5]   5 AND 3,CODE DATA
[6]   6 AND 4,CODE⌽DATA
[7]   7 NAND 2,CODE 0
[8]   OUTPUTS 5 6
    ∇

    ∇ Z NAND X
[1]   X GE 2,X
    ∇

    ∇ Z NOR Z
[1]   Z GE 5,X
    ∇

    ∇ Z OR X
[1]   Z GE 4,X
    ∇

    ∇ OUTPUTS ZZ
[1]   AV←(10*2×RN)⊥AV
[2]   →(RN≠1)/L3
[3]   ZZZ←Z←ZZ
[4]   L3:→((⍴,Z)≠⍴,ZZ)/L1
[5]   →((RN=1)∨~FP)/L2
[6]   ADDRESS←ADDRESS,Z+(10*2×RN-1)⊥AV
[7]   FROM←FROM,ZZ+AV
[8]   ELEM←ELEM,(⍴,ZZ)⍴0
[9]   L2:→0,RN←RN-1
[10]  L1:'OUTPUT LENGTH ERROR IN MODEULE ';1+ENC AV
[11]  RN←RN-1
[12]  EXIT←1
    ∇

    ∇ PRINTALL
[1]   PAE←1
    ∇

    ∇ PRINT XX
[1]   →(⍴V←+/XX)/L1
[2]   →PV←PP←PAE←0
[3]   L1:PV←RO,NE+1
[4]   PP←PV[RRO⍳XX]
    ∇

    ∇ PLOT XX
[1]   →(~PF←(1+XX)>0)/L1
[2]   GP←RO[RRO⍳XX]
[3]   GV←(⍴,XX)⍴0
[4]   GM←4 13⍴'  |  |---  ---|'
[5]   →0
[6]   L1:GP←GV←GM←0
    ∇
```

```
    ∇ PRINTOUT
[1]   →(PF)/L1
[2]   STATE←'0↑1+*'[((~L∨V)×CNT≤0)+(2×L∧~V)+(3×L∧V∧CNT≤0)
      +(4×V∧~L)+5×RF]
[3]   →PVF/L2
[4]   →PAE/L3
[5]   20ρ' ';STATE[SSX];'     ';STATE[SSS];'     ';STATE[SSZ];
      '     ';TIME;'     ';' *'[1+v/RF]
[6]   →0
[7]   L1:GRAPH V[GP];TIME;'    ';' *'[1+v/RF]
[8]   →0
[9]   L2:STATE←STATE,' '
[10]  20ρ' ';STATE[PP];5ρ' ';TIME;'     ';' *'[1+v/RF]
[11]  →0
[12]  L3:20ρ' ';STATE;'     ';TIME
    ∇

    ∇ Z PR8 X
[1]   INPUTS(ι8),17 18 19
[2]   (8+ι8)NAND(ι8),CODE 0
[3]   (19+ι16)JK1,⍪5 8ρ(ι8),(8ρ17),(8+ι8),(8ρ18),8ρ19
[4]   OUTPUTS 2×9+ι8
    ∇

    ∇ RR←RANDOM
[1]   RTD←,PARAM[;3 5]
[2]   RTH←,PARAM[;4 6]
[3]   TD←RTD[(NE×(NE?NE)≥NE↑2)+ιNE]
[4]   TH←RTH[(NE×(NE?NE)≥NE↑2)+ιNE]
    ∇

    ∇ READ X
[1]   →((64×(1↑A)-1)≥1↑ρMEM)/L2
[2]   →((ρ,X)=1)/L1
[3]   MEM[(64×X[1]-1)+2+1↑X;]
[4]   →0
[5]   L1:MEM[(64×X-1)+1+ι64;]
[6]   →0
[7]   L2:'DOES NOT EXIST'
    ∇

    ∇ SELECT XX
[1]   Q←(DEC XX)=100×⌊ADDRESS÷100
[2]   QQ←100|Q/ADDRESS
[3]   RO←(Q/ELEM)[QQ]
[4]   RRO←100|(Q/ADDRESS)[QQ]
[5]   →0×RO
[6]   'MODULE NOT FOUND'
    ∇

    ∇ A SET B
[1]   A←RO[RRO;A]
[2]   L[A]←V[A]←B>0
[3]   CNT[A]←PF[A]←0
    ∇

    ∇ Z ROM X
[1]   Z ME X
    ∇

    ∇ SETTIME XX
[1]   PRESET←XX
[2]   TIMEFLAG←XX>0
    ∇

    ∇ Z SHE X
[1]   →(~FP)/L1
[2]   TRANSLATE
[3]   →L2
[4]   L1:ASSIGN
[5]   STACK 0 4 7 0
[6]   →EXIT/0
[7]   SHA[NS+ιD1;]←A
[8]   SHB[NS+ιD1;]←B
[9]   PARAM[NE+ιρZ;]←C
[10]  L2:NS←NS+(ρZ)÷4
[11]  NE←NE+ρZ
    ∇

    ∇ SHOW
[1]   '
      ADDRESS              ELEM              FROM'
[2]   ⍉(3,ρELEM)ρADDRESS,ELEM,FROM
    ∇

    ∇ Z SHR X
[1]   Z SHE X
    ∇

    ∇ Z SH8 Z
[1]   INPUTS←ι10
[2]   11 NAND 10
[3]   12 13 AND 5 11 10 18
[4]   14 OR 12 13
[5]   (14+ι8)SHR(ι4),9 9 10,(14 6 7 8),9 9 10
[6]   OUTPUTS 14+ι8
    ∇

    ∇ Z SN5420 X
[1]   INPUTS←ι8
[2]   9 10 NAND(ι8),CODE 12 8 6 4 22 15
[3]   OUTPUTS 9 10
    ∇

    ∇ Z SN5470 X
[1]   INPUTS←ι9
[2]   12 14 AND 11 3 4 6 7 15,CODE 10 0 0 0 20 5
[3]   10 11 15 16 NAND 1 2 8 9,CODE 0
[4]   13 AND 5
[5]   19 20 AND 17 18,CODE 27 18 10 10 50 50
[6]   17 18 JK1 12 13 14 10 16
[7]   OUTPUTS 19 20
    ∇

    ∇ Z SN5472 X
[1]   INPUTS←ι9
[2]   10 12 14 NAND 1 5 9,CODE 0
[3]   11 13 AND 2 3 4 5 9 1 5 6 7 8,CODE 16 25 10 10 25 40
[4]   17 18 AND 15 16,CODE 16 25 10 10 25 40
[5]   15 16 JK2 11 12 13 10 14
[6]   OUTPUTS 17 18
    ∇

    ∇ Z SN7401 X
[1]   INPUTS 2 3 5 6 8 9 11 12 14 7
[2]   1 4 10 13 NAND 2 3 5 6 8 9 11 12,CODE 35 8 16 4 45 15
[3]   OUTPUTS 1 4 10 13
    ∇

    ∇ SPACING XX
[1]   SPACEFLAG←XX≠0
[2]   SPACE←XX+1000×XX=0
    ∇
```

```
    ∇ STACK KK
[1]   D1←(⍴,Z)↑KK[2]
[2]   →(D1≠⌊D1)/ERR
[3]   D2←(⍴,X)↑D1
[4]   →(D2≠⌊D2)/ERR
[5]   A←⍉(KK[3],D1)⍴,⍉(D1,D2)⍴TX
[6]   B←(D1,KK[2])⍴NE+⍳⍴,Z
[7]   C←((⍴,Z),6)⍴3↑KK
[8]   D←D1⍴KK[1]
[9]   →0
[10]  ERR:'INPUT LENGTH ERROR'
[11]  Z;'   FUNC  ';X;'  MODULE  ';1↑ENC AV
[12]  →0,EXIT←1
    ∇

    ∇ Z TCE X
[1]   →(~FP)/L1
[2]   TRANSLATE
[3]   →L2
[4]   L1:ASSIGN
[5]   STACK 0 4 6 0
[6]   →EXIT/0
[7]   TCA[NT+⍳D1;]←A
[8]   TCB[NT+⍳D1;]←B
[9]   PARAM[NE+⍳⍴Z;]←C
[10]  L2:NT←NT+(⍴Z)↑4
[11]  NE←NE+⍴Z
    ∇

    ∇ TIE
[1]   SELECT 0
[2]   SSX←RO[RRO⍳XXX]
[3]   SSZ←RO[RRO⍳ZZZ]
[4]   SSS←RO
[5]   X←(⍴SSX)⍴0
    ∇

    ∇ TIMINGS XX
[1]   TD←PARAM[;XX]
[2]   TH←PARAM[;XX+1]
    ∇

    ∇ TRANSLATE
[1]   ELEM←ELEM,NE+⍳⍴Z
[2]   D2←⍴Z
[3]   ADDRESS←ADDRESS,Z+(10*2×RN)|AV
[4]   FROM←FROM,D2⍴0
    ∇

    ∇ UPDATE
[1]   R←0<CNT×N←L=V
[2]   STEADY←∧/N∧~R
[3]   →(∧/(~R∨(~N))∨R∧~RF)/L1
[4]   INC←⌊INC⌊(SPACE-CYCLES)⌊⌊/(((L∧~V)×TD-CNT)+((V∧~L)×TH-CNT)+1000×N
[5]   CNT←((CNT+INC)×~N)+(CNT-INC)×R×CNT>INC
[6]   TIME←TIME+INC
[7]   CYCLES←CYCLES+INC
[8]   L1:V←((L∧~V)×CNT≥TD)+((V∧~L)×CNT<TH)+L∧V
[9]   CNT←((CNT+1)×(L≠V))+(CNT-1)×R
[10]  RF←R∧CNT>0
    ∇
```

```
    ∇ WIPE
[1]   EN←NE
[2]   L←CNT←V←FP←EN⍴0
[3]   STATE←EN⍴' '
[4]   SETTIME 1
[5]   SPACING 0
[6]   FORMFLAG←0
[7]   INC←1000
[8]   STEADY←0
[9]   GV←10⍴0
[10]  PLOT 0
[11]  PRINT 0
[12]  MAX←2+MIN←2+NOM←1
    ∇

    ∇ A WRITE X;Q
[1]   →((64×(1+A)-1)≥1+⍴MEM)/L2
[2]   →(1=⍴,A)/L1
[3]   Q←(1+A)-1
[4]   MEM[(64×Q)+2+1+A;]←(((⍴A)-1),8)⍴X
[5]   →0
[6]   L1:MEM[(64×A-1)+1+⍳64);]←64 8⍴,X
[7]   →0
[8]   L2:'DOES NOT EXIT'
    ∇
```

Appendix 15 :

FUNDAMENTALS & ALGORITHMS FOR RESIDUE NUMBER SYSTEMS[1]

Note

This appendix is subdivided into the following sub-sections :

I : A Review of the Fundamentals of the Residue Number System (RNS)

II : The Mixed-Radix Number System

III : Macrocoefficient and Macrocoefficient Extraction

IV : Base Extension and Alogrithms

---

[1] This appendix is extracted from a report of this author; see reference 16.

SECTION I

A REVIEW OF THE FUNDAMENTALS OF THE RESIDUE NUMBER SYSTEM (RNS)

(Notation, terminology,
identity and theorem)

I: A Review of the fundamentals of the RNS

In this section, we shall introduce and review most of the notations, terminologies, identities and theorems of the residue number system (RNS). More specialised concepts will be defined and discussed in other sections and the text of chapter 5 report. For conciseness, proofs of identities and theorems are omitted but their origins will be indicated. This section is compiled from references 54 through 102.

Theorem I.1: Division algorithm (77-80)

For every integer x and m, provided $m \neq 0$, there exists an integer q which will produce such a remainder r that

$$x = qm + r \qquad (I.1)$$

where $0 \leq r < |m|$. $|m|$ denotes the modulus or the absolute value of integer m.

Definition I.1 : Congruence (77-80)

In terms of the concept of congruence, theorem I.1 can be expressed as: If x, r and m are integers, m 0, then x is said to be congruent to "r modulo m", if and only if x and r have the same remainder when divided by m. Symbolically,

$$x \equiv r \bmod m \qquad (I.2)$$

Notation I.1 : (sometimes known as the Fundamental identity of residue arithmetic) Equation (I.1) is commonly written as,

$$\text{where} \quad \left.\begin{aligned} x &= m\left[\tfrac{x}{m}\right] + |x|_m \\ 0 &\leq |x|_m \leq m-1 \end{aligned}\right\} \qquad (I.3)$$

$\left[\frac{x}{m}\right] = q$ denotes the largest integer less than or equal to $\frac{x}{m}$, and $|x|_m = r$ is the least positive residue of x modulo m. $|x|_m$ is called a residue of x, with respect to m which is itself termed the modulus.

Notation I.2 : Residue codes

Instead of one modulus m, we can define an n-tuple of moduli $\mathcal{M} = (m_1, m_2, \cdots, m_n)$ with a corresponding ordered n-tuple of least positive residues $\{r_1, r_2, \cdots, r_n\}$ to represent integer x. The n-tuple of residues constitutes a residue code of x. Symbolically, we write,

$$\left.\begin{aligned} \mathcal{M} = (m_1, m_2, \cdots, m_n) &\leftrightarrow (m_i)_{i=1}^{n} \\ \{r_1, r_2, \cdots, r_n\} &\leftrightarrow (|x|_{m_i})_{i=1}^{n} \end{aligned}\right\} \qquad (I.4)$$

The number system corresponding to these residue codes is called the residue number system (RNS).

Notation I.3 : The square bracket function of an integer $Z \triangleq [Z]$ denotes the largest integer less than or equal to Z. Often during manipulations of numbers in RNS's, we require the residue of a square bracket function $|[Z]|_m$ which is more conveniently denoted as

$$|[Z]|_m = [Z]_m \tag{I.5}$$

Furthermore, the residue of the form $\left||x|_{m_i}\right|_{m_j}$ can be denoted by:

$$\left||x|_{m_i}\right|_{m_j} = |x|_{m_i,m_j} \tag{I.6}$$

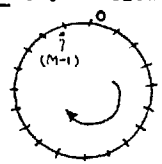
Theorme I.2: Unique representation theorem

Two integers x and x' have the same residue representation in a residue number system defined by the n-tuple of moduli $(m_1, m_2, \cdots, m_n)$ if and only if $(x-x')$ is an integer multiple of the least common multiple of the moduli, denoted by M.

This theorem is proved in references 81, 82, 87, 99. The interpretation of it is that the residue number system is periodic and the period is M. This means if representation is unambiguous in computation, we must restrict ourself to the use of a single period; hence

$$0 \leq x \leq M-1 \tag{I.7}$$

The periodicity of the number system can be pictured as a residue ring or numbers on a circle shown below,



Definition I.2 : Range

The range of a number system is defined as the interval over which every integer can be represented by the system without having two numbers with the same representation.

For an n-tuple moduli, $(m_i)_{i=1}^n$ where m_i are relatively prime, the range is given by

$$M = \prod_{i=1}^{n} m_i \tag{I.8}$$

The range M is also known as period or interval of definition.

Definition I.3 : Prime, relatively prime and composite

An integer p is a prime if p is not zero or ± 1 and if p is divisible only by ± 1 and $\pm p$.

Two integers a and b are relatively prime (or co-prime) if they have no common divisors except ± 1; that is $(a,b)=1$.

When $(a,b) \neq 1$, then they are said to be composite.

Some useful identities :

Some important identities are stated below without proves. They can be derived from equation (I.3) or their proves can be found in references 81, 85, 87, 97 - 99.

(a) Residue of multiple m (m being a modulus)

$$|Km|_m = 0 \qquad \text{for all integer K} \tag{I.9}$$

From this and equation (I.3) we can derive the following:

(i) $|x|_1 = 0$ (I.10)

(ii) $\left[\frac{Kx}{Km}\right] = \left[\frac{x}{m}\right]$ for all integer K (I.11)

(iii) $|Kx|_m = K|x|_m$ for all integer K (I.12)

(iv) $\left[\frac{a}{m}\right] = 0$ if and only if $0 \leq a < m$ (I.13)

i.e. m does not divide a except when a=0.

(v) $\left||x|_m\right|_m = |x|_m$ (I.14)

(b) Addition of multiples of m

$$|x \pm mK|_m = |x|_m \qquad \text{for integer K} \tag{I.15}$$

also

$$\left[\frac{x \pm mK}{m}\right] = \left[\frac{x}{m}\right] \pm K \tag{I.16}$$

(c) Additive inverse modulo m

$$|-x|_m = |(m-1)x|_m = |m-x|_m \tag{I.17}$$

(d) Addition and subtraction modulo m

$$|x \pm y|_m = \left||x|_m \pm |y|_m\right|_m = \left||x|_m \pm y\right|_m$$

in general,

$$\left|\sum_{i=1}^{n} x_i\right|_m = \left|\sum_{i=1}^{n} |x_i|_m\right|_m \tag{I.18}$$

(e) Multiplication modulo m

$$|x \cdot y|_m = \left||x|_m \cdot |y|_m\right|_m = \left||x|_m \cdot y\right|_m$$

in general,

$$\left|\prod_{i=1}^{n} x_i\right|_m = \left|\prod_{i=1}^{n} |x_i|_m\right|_m \tag{I.19}$$

(f) Multiplicative inverse

(i) Solution of linear equations

For integers a, b and x,

(a) $|x+a|_m = |b|_m$ (I.21)

has a unique solution given by $|x|_m = |b-a|_m$

(b) $|ax|_m = |b|_m$ (I.22)

has a unique solution iff (a,m)=1 and is given by (see Theorem I.3),

$$|x|_m = |b|_m \cdot \left|\frac{1}{a}\right|_m \quad (I.22.a)$$

(ii) Cancellation law of multiplication

For (K,m) = 1, if $|Ka|_m = |Kb|_m$ then

$$|a|_m = |b|_m . \quad (I.23)$$

(iii) Multiplicative inverse

If $0 \leq a < m$ and $|ab|_m = 1$, "a" is called the multiplicative inverse of "b mod m" and is denoted by

$a = \left|\frac{1}{b}\right|_m$ or sometimes simply written as $\left|\frac{1}{b}\right|_m$.

(iv) Operations with the multiplicative inverse

$$\left|\frac{1}{a}\cdot\frac{1}{b}\right|_m = \left|\frac{1}{ab}\right|_m = \left|\frac{1}{|ab|_m}\right|_m \quad (I.24)$$

Theorem I.3 : The Existence of multiplicative inverse

The quantity $\left|\frac{1}{b}\right|_m$ exists if and only if (b,m) = 1 and $|b|_m \neq 0$. In this case $\left|\frac{1}{b}\right|_m$ is unique.

(The proof can be found in references : 77-81, 85, 87, 97-99).

Theorem I.4 :

If the multiplicative inverse of b modulo m, $\left|\frac{1}{b}\right|_m$, is $|a|_m$ then $\left|\frac{1}{a}\right|_m = |b|_m$.

Theorem I.5 : Composite Modulus Theorem (82,89,90,100-102)

$$\left||x|_{m_i \cdot m_j}\right|_{m_i} = |x|_{m_i} \quad (I.25)$$

When one modulus is some multiple of the other, i.e. $m_j = K = k m_i$, when k is an integer, then

$$|x|_{K,m_i} = |x|_{|K|_{m_i}}$$

using the Composite Modulus Theorem, we have

$$|x|_{K,m_i} = \left||x|_K\right|_{m_i} = |x|_{m_i} \quad (I.26)$$

Theorem I.6 : The Chinese Remainder Theorem (77-82,97-99)

If the moduli $(m_1, m_2, \cdots, m_n)$ are pairwise relatively prime, i.e. $(m_i, m_j)=1$ for $i \neq j$, then the integer X represented by $\{\lambda_1, \lambda_2, \cdots, \lambda_n\}$ or $(|X|_{m_i})_{i=1}^{n}$ is given by,

$$|X|_M = \left| \sum_{i=1}^{n} \hat{m}_i \left|\frac{1}{\hat{m}_i}\right|_{m_i} \cdot \lambda_i \right|_M \quad (I.27)$$

where

$|X|_M$ denotes X being bounded by $0 \leq X \leq M-1$,

$M = \prod_{i=1}^{n} m_i$,

$\hat{m}_i = \frac{M}{m_i}$,

$\left|\frac{1}{\hat{m}_i}\right|_{m_i}$ is an integer Y, with $0 \leq Y \leq m_i - 1$ such that $|Y \cdot \hat{m}_i|_{m_i} = 1$.

The Chinese Remainder Theorem forms a basis for conversion between the RNS and the positional number system. For computational purposes the theorem is less useful because the residue computer is equipped to perform arithmetic operation modulo m_i and not modulo M . A better method is the use of the Mixed-radix conversion which will be discussed in section II.

Alternative forms of the Chinese Remainder Theorem:

(1) When x is known to lie between 0 and M-1, then $|X|_M = X$ and equation (I.27) can be written as,

$$X = \left| \sum_{i=1}^{n} \hat{m}_i \left|\frac{1}{\hat{m}_i}\right|_{m_i} \cdot n_i \right|_M \quad , \quad \text{for } 0 \leq X \leq M-1 \quad (I.28)$$

(2) Auxiliary functions are sometimes defined so that the Theorem appears in a form without the modulo-M operator, thus:

$$\left.\begin{aligned} X &= \sum_{i=1}^{n} \hat{m}_i \left|\frac{n_i}{\hat{m}_i}\right|_{m_i} - M A(x) \\ \text{where} \quad A(x) &= \frac{1}{M}\left(\sum_{i=1}^{n} \hat{m}_i \left|\frac{n_i}{\hat{m}_i}\right|_{m_i} - X \right) \end{aligned}\right\} \quad (I.29)$$

* * * * * * * *

This concludes the review of the fundamentals of residue number system. In subsequent sections we shall discuss other more specialised concepts.

SECTION II

THE MIXED-RADIX NUMBER SYSTEM

II.1 : The Mixed-Radix Number System

II.2 : Conversion of RNS to Mixed-radix Number System

- II.2.1 : Two useful identities
- II.2.2 : An alternate form for equation (II.12)
- II.2.3 : Interpretation of equations (II.12) & (II.16)
- II.2.4 : A numerical example (Conversion from Residue to Mixed-radix representation)

II : The Mixed-radix Number System [81,82,87,93-99]

The mixed-radix number system is a weighted number system; in which the radix changes from column to column but positional significance is maintained. The fixed-radix number system, such as the decimal and binary number systems, can be considered as a special case of the mixed-radix number system. The mixed-radix number is significant because:-

(1) The conversion from residue to mixed-radix is relatively fast in a residue computer;

(2) This conversion is a fundamental operation from which other important operations such as base-extension, scaling, relative magnitude comparision, sign determination, and fractional multiplication can be derived.

In this section, we shall discuss the mixed-radix number system and its relation (and conversion) to residue number systems. The derived important operations, numerated in (2) above, will be considered in other sections and the text of chapter 5.

II.1 : The mixed-radix number system

A number X may be represented in mixed-radix form:-

$$X = P_n\left(\prod_{i=1}^{n-1} R_i\right) + \cdots + P_3 R_2 R_1 + P_2 R_1 + P_1 \qquad (II.1)$$

where R_i are the radices and P_i are the mixed-radix digits and are bounded by $0 \leq P_i < R_i$. For a given set of radices, X can be represented in mixed-radix form by:

$$X \leftrightarrow \langle P_n, P_{n-1}, \cdots, P_1 \rangle \qquad (II.2)$$

where the digits are listed in order of decreasing significance. A positive integer X within interval $\left(0, \prod_{i=1}^{n} R_i - 1\right)$, is uniquely represented by equation (II.1) and (II.2). The constant multipliers for digits P_i are the mixed radix weights and for R_i the weight is

$$\text{and} \quad \begin{cases} \prod_{j=1}^{i-1} R_j & \text{for } i \neq 1 \\ 1 & \text{for } i = 1 \end{cases} \qquad (II.3)$$

Thus the decimal number system is a special case of the mixed-radix number system where $R_i = 10$ for all $i \neq 1$; hence the weights are powers of 10.

II.2 : Conversion of RNS to mixed-radix system

The n-tuple moduli, $(m_i)_{i=1}^{n}$, of a RNS can be chosen so that $m_i = R_i$ where R_i are radices of a mixed-radix system. Under this condition, the two systems are said to be associated and have the same range given by,

$$M = \prod_{i=1}^{n} m_i \quad \text{or} \quad \prod_{i=1}^{n} R_i \tag{II.4}$$

Equation (II.1) can now be written as,

$$X = P_n \prod_{i=1}^{n-1} m_i + \cdots + P_3 m_1 m_2 + P_2 m_1 + P_1 \tag{II.5}$$

where P_i are the mixed-radix digits and are bounded by,

$$0 \leq P_i \leq m_i - 1 \tag{II.6}$$

Taking the residue of equation (II.6) modulo m_i and solving for P_i (starting with i=1, then i=2 and so forth until i=n), we can express P_i in terms of the residues of X modulo m_1. Illustrating this process by a simple example, consider

$$X = P_3 m_1 m_2 + P_2 m_1 + P_1 \tag{II.7}$$

where $0 \leq P_i \leq m_i - 1$

Take residues of (II.7) modulo m_1, (i=1,2,3) and solve for P_i,

$$|X| = |P_3 m_1 m_2 + P_2 m_1 + P_1|_{m_1}$$
$$= \left| |P_3 m_1 m_2|_{m_1} + |P_2 m_1|_{m_1} + |P_1|_{m_1} \right|_{m_1}$$

From equation (I.9) of appendix I, i.e. $|Km|_m = 0$, therefore all terms on the right are equal to zero except $|P_1|_{m_1}$. Furthermore, by the definition of residue (I.3), we have

$$0 \leq |P_1|_{m_1} \leq m_1 - 1$$

and from equation (II.6),

$$\therefore \quad P_1 = |P_1|_{m_1} = |X|_{m_1} \tag{II.8}$$

Now, equation (II.7) can be expressed as,

$$\frac{X - P_1}{m_1} = P_3 m_2 + P_2 \tag{II.9}$$

With similar arguement as for P_1, take modulo m_2 of equation (II.9) and solve for P_2,

$$\therefore \quad P_2 = |P_2|_{m_2} = \left| \frac{|X|_{m_2} - P_1}{m_1} \right|_{m_2} \tag{II.10}$$

Similarly using equation (II.9) we can write

$$P_3 = |P_3|_{m_3} = \left| \frac{\left(\frac{X - P_1}{m_1}\right) - P_2}{m_2} \right|_{m_3}$$

$$\text{i.e.} \quad P_3 = \left| \frac{\frac{|X|_{m_3} - P_1}{m_1} - P_2}{m_2} \right|_{m_3} \tag{II.11}$$

We can generalise the results of this example for equation (II.5), thus,

$$P_1 = |X|_{m_1}$$

$$P_2 = \left| \frac{|X|_{m_2} - P_1}{m_1} \right|_{m_2}$$

$$P_3 = \left| \frac{\frac{|X|_{m_3} - P_1}{m_1} - P_2}{m_2} \right|_{m_3} \tag{II.12}$$

$$\vdots$$

$$P_n = \left| \frac{\frac{|X|_{m_n} - P_1}{m_1} - P_2}{m_2} \cdots \frac{\vdots \quad - P_{n-1}}{m_{n-1}} \right|_{m_n}$$

Equation (II.12) can be expressed in a more useful form by using equation (I.3) of section I and two other identities which we are first to derive.

II.2.1 : Two useful identities

(1) $$\left| \frac{|X|_{m_k} - |X|_{m_\ell}}{m_\ell} \right|_{m_k} = \left| \left[\frac{X}{m_\ell}\right] \right|_{m_k} \triangleq \left[\frac{X}{m_\ell}\right]_{m_k} \tag{II.13}$$

where k and ℓ are dummy subscripts. From (I.3) of section I, we have,

$$|X|_{m_k} = X - m_k \left[\frac{X}{m_k}\right] \quad \text{and} \quad |X|_{m_\ell} = X - m_\ell \left[\frac{X}{m_\ell}\right]$$

$$\left| \frac{|X|_{m_k} - |X|_{m_\ell}}{m_\ell} \right|_{m_k} = \left| \frac{X - m_k \left[\frac{X}{m_k}\right] - X + m_\ell \left[\frac{X}{m_\ell}\right]}{m_\ell} \right|_{m_k}$$

$$= \left| \left| \frac{1}{m_\ell} \right|_{m_k} \left\{ \left| -m_k \left[\frac{X}{m_k}\right] \right|_{m_k} + \left| m_\ell \left[\frac{X}{m_\ell}\right] \right|_{m_k} \right\} \right|_{m_k}$$

From (I.9), $\left| -m_k \left[\frac{x}{m_k}\right] \right|_{m_k} = 0$,

$$\therefore \left| \frac{|x|_{m_k} - |x|_{m_l}}{m_l} \right|_{m_k} = \left| \left|\frac{1}{m_l}\right|_{m_k} \cdot \left| m_l \left[\frac{x}{m_l}\right] \right|_m \right|_{m_k}$$

$$= \left| \frac{1}{m_l} \cdot m_l \left[\frac{x}{m_l}\right] \right|_{m_k} \quad \text{by (I.20)}$$

$$\therefore \left| \frac{|x|_{m_k} - |x|_{m_l}}{m_l} \right|_{m_k} = \left| \left[\frac{x}{m_l}\right] \right|_{m_k} \triangleq \left[\frac{x}{m_l}\right]_{m_k} \quad \text{by (I.5)}$$

(2) If "n" is a positive integer then,

$$\left[\frac{[x]}{n}\right] = \left[\frac{x}{n}\right] \qquad \text{(II.14)}$$

Proof: Let X be a number, then it may be represented by

$$X = [x] + \theta \qquad \text{where } 0 \leq \theta < 1$$

By the Division algorithm, equation (I.1), the integer $[x]$ may be expressed as, $[x] = nq + r$

for some integers n, q, r such that $0 \leq r \leq (n-1)$ or we can write,

$$\frac{[x]}{n} = q + \frac{r}{n} \;, \qquad 0 \leq r < 1$$

$$\therefore \left[\frac{[x]}{n}\right] = q \qquad \text{(II.15)}$$

But, $$\frac{x}{n} = \frac{[x]+\theta}{n} = \frac{nq + r + \theta}{n}$$

i.e. $$\frac{x}{n} = q + \frac{r+\theta}{n}$$

Since $0 \leq r \leq n-1$, hence

$$0 \leq \frac{r+\theta}{n} \leq \frac{(n+1)+\theta}{n} < 1$$

$$\therefore \left[\frac{x}{n}\right] = q$$

Comparing this with equation (II.15) thus,

$$\left[\frac{[x]}{n}\right] = \left[\frac{x}{n}\right] = q \quad \text{, as required.}$$

II.2.2 : An alternate form for equation (II.12)

Since $P_1 = |x|_{m_1}$, substituting this for P_1 in expression for P_2 in (II.12),

$$\therefore P_2 = \left| \frac{|x|_{m_2} - |x|_{m_1}}{m_1} \right|_{m_2} = \left| \frac{x - |x|_{m_1}}{m_1} \right|_{m_2}$$

$$= \left| \left[\frac{x}{m_1}\right] \right|_{m_2} \qquad \text{by equation (I.3)}$$

$$= \left[\frac{x}{m_1}\right]_{m_2} \qquad \text{by equation (I.5)}$$

Now substituting for P_1 and P_2 in the expression for P_3,

$$\therefore P_3 = \left| \frac{\frac{|x|_{m_3} - P_1}{m_1} - P_2}{m_2} \right|_{m_3} = \left| \frac{\frac{|x|_{m_3} - |x|_{m_1}}{m_1} - \left[\frac{x}{m_1}\right]_{m_2}}{m_2} \right|_{m_3}$$

$$= \left| \frac{\left[\frac{x}{m_1}\right]_{m_3} - \left[\frac{x}{m_1}\right]_{m_2}}{m_2} \right|_{m_3} \qquad \text{using equation (II.13)}$$

$$= \left| \left[\frac{\left[\frac{x}{m_1}\right]}{m_2}\right]_{m_3} \right|_{m_3} \qquad \text{using equation (II.13)}$$

$$= \left[\frac{x}{m_1 m_2}\right]_{m_3} \qquad \text{using equations (II.14) and (I.5)}$$

By the same procedure, the general term for the mixed-radix coefficient is, for $i > 1$,

$$P_i = \left[\frac{x}{m_1 m_2 \cdots m_{i-1}}\right]_{m_i} \qquad \text{(II.16.a)}$$

Hence equation (II.12) can be represented by

$$\left.\begin{aligned} P_1 &= |x|_{m_1} \\ P_2 &= \left[\frac{x}{m_1}\right]_{m_2} \\ P_3 &= \left[\frac{x}{m_1 m_2}\right]_{m_3} \\ &\vdots \\ P_n &= \left[\frac{x}{m_1 m_2 \cdots m_{n-1}}\right]_{m_n} \end{aligned}\right\} \qquad \text{(II.16)}$$

II.2.3 : Interpretation of equations (II.12) and (II.16)

The process by which equation (II.12) is transformed into (II.16) is the conversion process from a residue to a mixed-radix representation. The process is a sequential one; i.e. P_1 is found first and used to compute P_2. P_3 is then computed from P_1 and P_2; and so forth, until P_n is calculated. One can expressed the conversion process in tabular form (or a diagonal matrix) shown in equation (II.17).

Modular Step (i.e. row)	Modular Entries (i.e. column) 1	2	3	....	n
1	$P_1 = \lvert X \rvert_{m_1}$	$\lvert X \rvert_{m_2}$	$\lvert X \rvert_{m_3}$	....	$\lvert X \rvert_{m_n}$
2		$P_2 = \left[\frac{X}{m_1}\right]_{m_2}$	$\left[\frac{X}{m_1}\right]_{m_3}$	...	$\left[\frac{X}{m_1}\right]_{m_n}$
3			$P_3 = \left[\frac{X}{m_1 m_2}\right]_{m_3}$	...	$\left[\frac{X}{m_1 m_2}\right]_{m_n}$
⋮				⋮	
n					$P_n = \left[\frac{X}{m_1 m_2 \cdots m_n}\right]$

(Equation II.17)

Comparing the conversion of (II.12) to (II.16) with the tabulated form (II.17), the latter shows: except for modular step one, all entries for each step are obtained by subtracting the extreme left-hand entry of the preceding step from all entries of that step and dividing by its associated modulus. Interpreting the tabulation as a diagonal matrix A, each entry of the former is a_{ij}, where i is the row and j the column. The diagonal elements a_{ii} are the mixed coefficients P_i. The formula for calculating a_{ij} is:

$$a_{ij} = \left| \frac{a_{i-1,j} - a_{i-1,i-1}}{m_{i-1}} \right|_{m_j}$$

$$= \left| \left| \frac{1}{m_{i-1}} \right|_{m_j} \cdot \left| a_{i-1,j} - a_{i-1,i-1} \right|_{m_j} \right|_{m_j} \qquad \text{(II.18)}$$

where $\left|\frac{1}{m_{i-1}}\right|_{m_j}$ is the multiplicative inverse of m_{i-1} and it exists iff $(m_{i-1}, m_j) = 1$. (see section I).

Given all terms in one row, then all terms in the following row can be computed in one modular step if n adders are available. The complete conversion process requires (n-1) modular steps.

To conclude this section, a numerical example is used to illustrate the procedure.

II.2.4 : A numerical example (Conversion from Residue to mixed-radix representation)

For $m_1=8$, $m_2=5$, $m_3=7$, and $m_4=3$, find the associated mixed-radix digits of $\{3,4,2,1\}$, where the mixed-radix expression is

$$X = P_4(8 \times 5 \times 7) + P_3(8 \times 5) + P_2(8) + P_1$$

Modular Steps	Residue Operations	m_1 8	m_2 5	m_3 7	m_4 3	
1	X in Residue Code:	(3)	4	2	1	$P_1 = \lvert X \rvert_8 = 3$
	Subtract $P_1 = 3$	3	3	3	0	
		0	1	6	1	
	Multiply by $\left\lvert\frac{1}{8}\right\rvert_{m_2}$		2	1	2	
2			(2)	6	2	$P_2 = \left[\frac{X}{8}\right]_5 = 2$
	Subtract $P_2 = 2$		2	2	2	
			0	4	0	
	Multiply by $\left\lvert\frac{1}{5}\right\rvert_{m_3}$			3	2	
3				(5)	0	$P_3 = \left[\frac{X}{40}\right]_7 = 5$
	Subtract $P_3 = 5$			5	2	
				0	1	
	Multiply by $\left\lvert\frac{1}{7}\right\rvert_{m_4}$				1	
					(1)	$P_4 = \left[\frac{X}{280}\right]_3 = 1$

The mixed-radix representation of X is thus,

$$X \leftrightarrow \langle 1,5,2,3 \rangle$$

From this, X in the decimal system is found:

$$X = 1(8x5x7) + 5(8x5) + 2(8) + 3(1) = 499$$

Note:

(1) Mixed-radix conversion requires 2(n-1) residue arithmetic operations; split equally between subtraction and multiplication.

(2) The multiplicative inverses are predeterminable constants.

SECTION III

MACROCOEFFICIENT AND MACROCOEFFICIENT EXTRACTION

III.1 : Macrocoefficients and their associated RNS's

III.2 : Some special (K,K-1,K+a) RNS's

III.3 : Macrocoefficient Extraction

- III.3.1 : The Preliminaries
- III.3.2 : Correspondence Theorems
- III.3.3 : Two Residue Base-Extension (TRB-E) and Algorithm
- III.3.4 : Macrocoefficient extraction algorithm

III : Macrocoefficient and Macrocoefficient Extraction

(references: 84, 89, 90, 100-102)

The mixed-radix conversion, discussed in section II is useful for residue arithmetic and operations. Many advantages, however, can be gained by adopting its implicit form based on macrocoefficients computation, known as macrocoefficients extraction.[89,90,100-102] Macrocoefficients and macrocoefficient extraction were empirically discovered at Scope Inc. further investigated at Honeywell [89,90,100-102] We shall review the theory and algorithms of macrocoefficient extraction in this section.

III.1 : Macrocoefficients and their associated RNS's

From section I, given an n-tuple moduli, $(m_i)_{i=1}^{n}$ where the moduli are relatively prime, the range of an RNS is given by,

$$M = \prod_{i=1}^{n} m_i$$

Now, choose the first q moduli, with $q < n$ and define

$$K = \prod_{i=1}^{q} m_i \qquad K' = \prod_{j=q+1}^{n} m_j \qquad \text{(III.1)}$$

If we now define an integer "a" which can be positive or negative, such that

$$a \triangleq K' - K$$

then, we have

$$\left.\begin{aligned} K &= \prod_{i=1}^{q} m_i \\ K+a &= \prod_{j=q+1}^{n} m_j \end{aligned}\right\} \qquad \text{(III.1a)}$$

Now define,

$$\left.\begin{aligned} \gamma_x &= \left[\tfrac{X}{K}\right], & \delta_x &= |X|_K \\ \Gamma_x &= \left[\tfrac{X}{K+a}\right], & \varepsilon_x &= |X|_{K+a} \\ \beta_x &= \left[\tfrac{X}{K-1}\right], & \epsilon_x &= |X|_{K-1} \end{aligned}\right\} \qquad \text{(III.2)}$$

These are called macrocoefficients; the associated systems are the (K,K-1,K+a) residue number systems, the combinations of which form various residue number systems. For example, with X bounded by $0 \le X \le K(K-1)-1$ or $0 \le X \le M-1$ so that X is uniquely represented in the following number systems (K's being

defined as in equation III.5):

$$\left.\begin{array}{ll} \text{Number system A} & (K,K+a) \leftrightarrow (m_1,m_2,\ m_5,m_6) \\ \text{Number system B} & (K,K-1) \leftrightarrow (m_1,m_2,\ m_3,m_4) \\ \text{Number system C} & (K-1,K+a) \leftrightarrow (m_3,m_4,\ m_5,m_6) \end{array}\right\} \quad \text{(III.3)}$$

Now using the macrocoefficients, by equation (I.3), X can be represented by:

$$\left.\begin{array}{l} X = K\gamma_x + \delta_x \\ X = (K+a)\Gamma_x + \zeta_x \\ X = (K-1)\beta_x + \epsilon_x \end{array}\right\} \quad \text{(III.4)}$$

Therefore any one of the pairs, which are called <u>macrocoefficient pairs</u>: (γ_x,δ_x) , (Γ_x,ζ_x) , (β_x,ϵ_x)

uniquely represents X and the macrocoefficients themselves can unambiguiously be represented as residue numbers. Observe that the third equation of (III.4) can be derived from the second by writting a= -1. Hence the first two equations, i.e. the (K,K+a) RNS's, form a "general case" so that all results obtained can be applied to other RNS's (e.g. the (K,K-1)) by making appropriate substitutions.

<u>III.2 : Some Special (K,K-1,K+a) RNS's</u>

Macrocoefficients can be defined in any RNS; for reasons of system design economy [84,102], the number of moduli in an RNS is chosen with $n=2^N$ moduli where N is some positive integers: so that n = 4,8,16, etc. Further economics result if the relatively prime moduli are related as follows: [84,99-102]

$$\left.\begin{array}{l} m_1 \cdot m_2 = K \\ m_3 \cdot m_4 = K-1 \\ m_5 \cdot m_6 = K+a \end{array}\right\} \quad \text{(III.5)}$$

where K is an integer relatively prime to a and (a+1); $m_1 < m_2$, $m_3 < m_4$ and $m_5 < m_6$.

(1) <u>The K(K-1) N-RRNS</u>

A <u>non-redundant residue number system</u> (N-RRNS) known as K(K-1) RNS's is defined as:

$$\left.\begin{array}{l} K = m_1 \cdot m_2 \\ \text{and } K-1 = m_3 \cdot m_4 \end{array}\right\} \quad \text{(III.6)}$$

where moduli are relatively prime, also $m_1 < m_2$ and $m_3 < m_4$.

The four basic macrocoefficients are defined as:

$$\left.\begin{array}{ll} \gamma_x \equiv \gamma(x) \triangleq \left[\frac{x}{K}\right] , & \delta_x \equiv \delta(x) \triangleq |x|_K \\ \beta_x \equiv \beta(x) \triangleq \left[\frac{x}{K-1}\right] , & \epsilon_x \equiv \epsilon(x) \triangleq |x|_{K-1} \end{array}\right\} \quad \text{(III.6.a)}$$

The range of X is given by

$$M = K(K-1) = m_1 m_2 m_3 m_4 - 1 \quad \text{(III.6.b)}$$

so that X can be represented by,

$$\left.\begin{array}{l} X = K\gamma_x + \delta_x \\ X = (K-1)\beta_x + \epsilon_x \end{array}\right\} \quad \text{(III.6.c)}$$

(2) <u>The (K,K-1,K+a) RRNS's</u>

For error detection and correction applications, [100-102] a useful <u>Redundant Residue Number System (RRNS)</u> is defined by equation (III.5). The associated macrocoefficients are given by equation (III.3). Whereas the range of X is the same as that for the K(K-1) RNS's -- i.e. equation (III.6.b). Integer X is represented in the RRNS's by equation (III.4).

(3) <u>The general case : K(K+a) RNS's</u>

As stated in section III.1, the K(K+a) RNS's is the general case for the (K,K-1,K+a) RNS's and their combinations. For subsequent discussion, we shall consider the K(K+a) RNS's. The relevant equations are restated for later reference:

<u>The macrocoefficients:</u>

$$\left.\begin{array}{ll} \gamma_x \equiv \gamma(x) \triangleq \left[\frac{x}{K}\right] , & \delta_x \equiv \delta(x) \triangleq |x|_K \\ \Gamma_x \equiv \Gamma(x) \triangleq \left[\frac{x}{K+a}\right] , & \zeta_x \equiv \zeta(x) \triangleq |x|_{K+a} \end{array}\right\} \quad \text{(III.7)}$$

<u>Representation of X:</u>

$$\left.\begin{array}{l} X = K\left[\frac{x}{K}\right] + |x|_K = K\gamma_x + \delta_x \\ X = (K+a)\left[\frac{x}{K+a}\right] + |x|_{K+a} = (K+a)\Gamma_x + \zeta_x \end{array}\right\} \quad \text{(III.8)}$$

<u>Range of X:</u>

$$0 \leq X \leq M-1 \quad \text{(III.9)}$$

where

$$M = K(K+a) = m_1 m_2 m_5 m_6$$

Macrocoefficient pairs:

$$(\gamma_x, \delta_x) \quad \text{and} \quad (\Gamma_x, \mathcal{S}_x) \qquad \text{(III.10)}$$

The bounds of the macrocoefficients:

Since X is defined for interval $0 \leq X \leq M-1$ we can calculate the corresponding intervals for the macrocoefficients. From (III.8), and the definition of residue (i.e. equation I.3),

$$\left.\begin{aligned} 0 \leq \delta_x &= |x|_K \leq K-1 \\ \text{and} \quad 0 \leq \mathcal{S}_x &= |x|_{K+1} \leq (K+a)-1 \end{aligned}\right\} \qquad \text{(III.11)}$$

Substituting (III.8) into (III.9) and using the first equation of (III.11), hence $0 \leq K\gamma_x \leq M-K$

and from (III.9),

$$\therefore \quad 0 \leq K\gamma_x \leq K(K+a)-K$$

$$\text{i.e.} \quad 0 \leq \gamma_x \leq K+a-1$$

Similarly, $0 \leq \Gamma_x \leq (K-1)$

Summarising, the bounds are:

$$\left.\begin{aligned} 0 \leq \gamma_x &\leq (K+a)-1 \\ 0 \leq \delta_x &\leq K-1 \\ 0 \leq \Gamma_x &\leq K-1 \\ 0 \leq \mathcal{S}_x &\leq (K+a)-1 \end{aligned}\right\} \qquad \text{(III.12)}$$

Residue representation:

Within these intervals, the macrocoefficients are uniquely represented by residues:

$$\left.\begin{aligned} \gamma_x &\leftrightarrow (|\gamma_x|_{m_{q+1}}, |\gamma_x|_{m_{q+2}}, \cdots, |\gamma_x|_{m_n}) \\ \delta_x &\leftrightarrow (|\delta_x|_{m_1}, |\delta_x|_{m_2}, \cdots, |\delta_x|_{m_q}) \\ \Gamma_x &\leftrightarrow (|\Gamma_x|_{m_1}, |\Gamma_x|_{m_2}, \cdots, |\Gamma_x|_{m_q}) \\ \mathcal{S}_x &\leftrightarrow (|\mathcal{S}_x|_{m_{q+1}}, |\mathcal{S}_x|_{m_{q+2}}, \cdots, |\mathcal{S}_x|_{m_n}) \end{aligned}\right\} \qquad \text{(III.13)}$$

III.3 : Macrocoefficient Extraction

The process of computing unique residue representations of a particular pair of macorcoefficients from a sequence of residues $(|x|_{m_i})_{i=1}^{n}$, which uniquely represent X, is called macrocoefficient extraction. The heart of this process is the Two residue base-extension (TRB-E) which is based on the Correspondence Theorems. These theorems are easily proved if we first prove two lemmas and introduce the term identically phased. We shall discuss these concepts, in reverse order, in the following subsections. As macrocoefficient extraction and mixed-radix conversions are closely related theoretically, we shall discuss this relationship in section III.3.1 entitles the preliminariee.

III.3.1 : The Preliminaries

Lemma III.1

The quantity $_{j}{}_{a}\lambda_j(x) = \big||x|_{m_j} - |x|_{m_a}\big|_{m_j}$ has the same value for all X in the interval $Im_a \leq X \leq (I+1)m_a - 1$ where I is any integer.

Proof: Let $X_1 = Im_a$ and $X_2 = Im_a + R$ where $0 \leq R \leq m_a - 1$; satisfying interval $Im_a \leq X \leq (I+1)m_a - 1$.

Since $|Im_a|_{m_a} = 0$ from equation (I.9),

$$\therefore \quad \big||X_1|_{m_j} - |X_1|_{m_a}\big|_{m_j} = \big||Im_a|_{m_j} - |Im_a|_{m_a}\big|_{m_j} = |Im_a|_{m_j}$$

and

$$\big||X_2|_{m_j} - |X_2|_{m_a}\big|_{m_j} = \big||Im_a + R|_{m_j} - |R|_{m_a}\big|_{m_j}$$

As $0 \leq R \leq m_a - 1$ hence $|R|_{m_a} = R$, and by equation (I.19):

$$\big||X_2|_{m_j} - |X_2|_{m_a}\big|_{m_j} = \big|Im_a + R - R\big|_{m_j} = |Im_a|_{m_j}$$

Consequently,

$$\big||X_1|_{m_j} - |X_1|_{m_a}\big|_{m_j} = \big||X_2|_{m_j} - |X_2|_{m_a}\big|_{m_j} = |Im_a|_{m_j}$$

since X_2 can be chosen at will within interval $Im_a \leq X \leq (I+1)m_a - 1$, this proves the lemma; now define

$$_{j}{}_{a}\lambda_j(x) \triangleq \big||x|_{m_j} - |x|_{m_a}\big|_{m_j} \qquad \text{(III.14)}$$

Lemma III.2

The quantity ${}_j a\lambda_j(X) = \left| |X|_{m_j} - |X|_{m_a} \right|_{m_j}$ is unique for all $X = Nm_a$ where $N = 0, 1, 2, \ldots, (m_j - 1)$.

Proof : Let $X_1 = N_1 m_a$ and $X_2 = N_2 m_a$ with $N_1 \neq N_2$ and $0 \leq N_1, N_2 \leq m_j - 1$. (These intervals of N_1, N_2 ensure condition $N = 0, 1, \ldots, (m_j - 1)$ in the Lemma.) We shall prove the lemma by contradiction.

If the lemma was false, then there exist X_1, X_2 st.

$$\left| |X_1|_{m_j} - |X_1|_{m_a} \right|_{m_j} = \left| |X_2|_{m_j} - |X_2|_{m_a} \right|_{m_j}$$

i.e. $$\left| |N_1 m_a|_{m_j} - |N_1 m_a|_{m_a} \right|_{m_j} = \left| |N_2 m_a|_{m_j} - |N_2 m_a|_{m_a} \right|_{m_j}$$

by (I.9), $|N_1 m_a|_{m_a} = |N_2 m_a|_{m_a} = 0$,

$$\left| |N_1 m_a|_{m_j} - |N_2 m_a|_{m_j} \right|_{m_j} = 0 ,$$

i.e. $$\left| |N_1 - N_2|_{m_j} \cdot |m_a|_{m_j} \right|_{m_j} = 0 .$$

Since $(m_a, m_j) = 1$, i.e. relatively prime therefore $|m_a|_{m_j} \neq 0$; consequently, $|N_1 - N_2|_{m_j} = 0$. But we postulated that $N_1 \neq N_2$ and that $0 \leq N_1, N_2 \leq m_j - 1$; hence this contradiction proves the lemma.

Notice that lemma III.2 implies that for $0 \leq X < N m_a$ or $0 \leq X \leq m_a m_j - 1$, ${}_j a\lambda_j(X)$ will assume "j" distinct values. This follows from the definition of residus (see equation I.3) and that ${}_j a\lambda_j(X)$ is a residue modulo m_j.

Definition III.1 : Identically phrased

Two quantities $Q_1(X)$ and $Q_2(X)$ are said to be identically phrased if when tabulated side by side $Q_1(X) = c_1$ and $Q_2(X) = d_1$ for the first "n" values of X and $Q_1(X) = c_2$ and $Q_2(X) = d_2$ for the next "n" values of X, and so forth throughout the range of X; where c_i and d_i need not be equal.

Relationship between Macrocoefficient Extraction and Mixed-radix Conversion

In this particular discussion we shall assume the K(K-1) RNS's; but the results are perfectly general because m_3 and m_4 can be substituted for m_5, m_6 to give the K(K+a) RNS's. From equation (III.6a) we have:

$$X = K\left[\frac{X}{K}\right] + |X|_K = K\gamma_X + \delta_X \qquad \text{(III.15)}$$

where $K = m_1 m_2$ and $0 \leq |X|_K$, $\delta_X \leq m_1 m_2 - 1$. In terms of the mixed-radix number system (section II), using equations (II.5) and (II.16), X can be represented by:

$$\left.\begin{aligned} X &= P_1 + m_2 P_2 + m_1 m_2 P_3 + m_1 m_2 m_3 P_3 \\ \text{where} \quad P_1 &= |X|_{m_1} \quad , \quad P_2 = \left[\frac{X}{m_1}\right]_{m_2} \\ P_3 &= \left[\frac{X}{m_1 m_2}\right]_{m_3} \quad , \quad P_4 = \left[\frac{X}{m_1 m_2 m_3}\right]_{m_4} \end{aligned}\right\} \qquad \text{(III.16)}$$

From (III.15) and (III.16),

$$\therefore \quad P_1 = |X|_{m_1} = |\delta_X|_{m_1} \qquad \text{(III.17)}$$

$$P_3 = \left[\frac{X}{m_1 m_2}\right]_{m_3} = \left[\frac{X}{K}\right]_{m_3} = |\gamma_X|_{m_3} \qquad \text{(III.18)}$$

From (II.13) of section II, we have

$$\left[\frac{X}{m_i}\right]_{m_k} = \left|\frac{|X|_{m_k} - |X|_{m_i}}{m_i}\right|_{m_k}$$

$$\therefore \quad P_2 = \left[\frac{X}{m_1}\right]_{m_2} = \left|\frac{|X|_{m_2} - |X|_{m_1}}{m_1}\right|_{m_2}$$

From the Composite Modulus Theorem (I.25) in section I, and equation (III.15):

$$\therefore \quad |\delta_X|_{m_i} = \left||X|_K\right|_{m_i} = |X|_{m_i}$$

where i=1,2 (because $K = m_1 m_2$). Therefore,

$$P_2 = \left[\frac{X}{m_1}\right]_{m_2} = \left|\frac{|X|_{m_2} - |X|_{m_1}}{m_1}\right|_{m_2} = \left|\frac{|\delta_X|_{m_2} - |\delta_X|_{m_1}}{m_1}\right|_{m_2}$$

To find the relationship between macrocoefficients and P_4, consider equations (III.15) and (III.16),

$$\gamma_X \triangleq \left[\frac{X}{K}\right] = \left[\frac{X}{m_1 m_2}\right] = \left[\frac{P_1 + m_1 P_2 + m_1 m_2 P_3 + m_1 m_2 m_3 P_4}{m_1 m_2}\right]$$

i.e. $$\gamma_X = \left[\frac{P_1}{m_1 m_2} + \frac{P_2}{m_2} + P_3 + m_3 P_4\right]$$

Since P_1 and P_2 are residues of m_1 and m_2, i.e. P_1 and P_2 are not divisible by m_1, m_2 or $m_1 \nmid P_1$ and $m_2 \nmid P_2$, consequently the "square bracket function" (see notation I.3 of section I) give,

$$\gamma_X = P_3 + m_3 P_4$$

By the definition of mixed-radix, P_4 lies in range $0 \leq P_4 \leq (m_4-1)$ and by equation (I.3),

$$\therefore \quad \gamma_x = P_3 + m_3 P_4 = |\gamma_x|_{m_3} + m_3 \left[\frac{\gamma_x}{m_3}\right]_{m_4}$$

From this we have,

$$P_3 = |\gamma_x|_{m_3}$$

and $$P_4 = \left[\frac{\gamma_x}{m_3}\right]_{m_4} = \left[\frac{\left[\frac{X}{m_1 m_2}\right]}{m_3}\right]_{m_4} \quad \text{from eqn. (III.16)}$$

$$= \left[\frac{X}{m_1 m_2 m_3}\right]_{m_4} \quad " \quad " \quad (II.14)$$

Now using equation (II.13),

$$\therefore \quad P_4 = \left[\frac{X}{m_1 m_2 m_3}\right]_{m_4} = \left[\frac{\gamma_x}{m_3}\right]_{m_4} = \left|\frac{|\gamma_x|_{m_4} - |\gamma_x|_{m_3}}{m_3}\right|_{m_4}$$

Summarizing the results,

$$\left.\begin{aligned} P_1 &= |X|_{m_1} = |\delta_x|_{m_1} \\ P_2 &= \left|\frac{|X|_{m_2} - |X|_{m_1}}{m_1}\right|_{m_2} = \left|\frac{|\delta_x|_{m_2} - |\delta_x|_{m_1}}{m_1}\right|_{m_2} \\ P_3 &= |\gamma_x|_{m_3} \\ P_4 &= \left|\frac{|\gamma_x|_{m_4} - |\gamma_x|_{m_3}}{m_3}\right|_{m_4} \end{aligned}\right\} \quad (III.20)$$

Now, solving equations of (III.20) for P's, we can express the macrocoefficient pair, (γ_x, δ_x), in terms of the mixed-radix digits (or P's); e.g. for P_2,

$$|m_1 P_2|_{m_2} = \left| m_1 \left| \frac{|\delta_x|_{m_2} - |\delta_x|_{m_1}}{m_1} \right|_{m_2} \right|_{m_2} = \left| |\delta_x|_{m_2} - |\delta_x|_{m_1} \right|_{m_2}$$

by the definition of multiplicative inverse (see section I).

Further algebraic manipulation gives,

$$|\delta_x|_{m_2} = \left| |\delta_x|_{m_1} + m_1 P_2 \right|_{m_2} \quad \text{OR} \quad |P_1 + m_1 P_2|_{m_2}$$

Similarly, we have:

$$\left.\begin{aligned} |\delta_x|_{m_1} &= P_1 \\ |\delta_x|_{m_2} &= |P_1 + m_1 P_2|_{m_2} \\ |\delta_x|_{m_3} &= P_2 \\ |\delta_x|_{m_4} &= |P_3 + m_3 P_4|_{m_4} \end{aligned}\right\} \quad (III.21)$$

This concludes the preliminaries and we are in the position to discuss the concepts of macrocoefficients extraction.

III.3.2 : Correspondence Theorems

Correspondence Theorems are the bases of the residue base-extension algorithms; such as the TRB-E. There are two parts in the Theorems and we shall call them Correspondence Theorem One and Correspondence Theorem Two. Theorem one shows that the λ's, called transforms, can be calculated from "identically phased" λ's and a one-dimensional table which is computed empirically. While Theorem two, due to Hastings and Moore,[89,90,102] expresses the transforms in closed forms.

Correspondence Theorem One :

Given the moduli m_a, m_b and m_d where m_a and m_b are relatively prime and the residues of an integer X, $|X|_{m_a}$ and $|X|_{m_b}$ with $0 \leq X \leq (m_a m_b - 1)$; the quantity ${}_{da}\lambda_d(X) = \left||X|_{m_d} - |X|_{m_a}\right|_{m_d}$ can take on V values and these can be uniquely determined by the value ${}_{ba}\lambda_b(X) = \left||X|_{m_b} - |X|_{m_a}\right|_{m_b}$ and a one dimensional table, where V is the smaller of the integers m_d and m_a.

(equation III.22)

Proof: From lemma III.1,

$${}_{ba}\lambda_b(X) = \left||X|_{m_b} - |X|_{m_a}\right|_{m_b}$$ has the same value over the range $I m_a \leq X \leq (I+1)m_a - 1$ where I is an integer.

also,

$${}_{da}\lambda_d(X) = \left||X|_{m_d} - |X|_{m_a}\right|_{m_d}$$ -- do --

Since both expressions have the same form and contain the common term $|X|_{m_a}$, therefore ${}_{ba}\lambda_b(X)$ and ${}_{da}\lambda_d(X)$ are identically phrased. From lemma III.2, we observe:

${}_{ba}\lambda_b(X)$ assumes m_a distinct values for X in the interval $0 \leq X \leq m_a m_b - 1$

Furthermore,

${}_{da}\lambda_d(X)$ takes on at most m_d distinct values for X bounded by $0 \leq X \leq m_b m_d - 1$; iff $(m_b, m_d) = 1$.

If the range of X is restricted to $0 \leq X \leq m_a m_b - 1$, then ${}_{da}\lambda_d(X)$ will assumes at most m_a distinct values if $m_d \geq m_a$; or fewer

than m_a distinct values if $m_d < m_a$.

Now, because ${}_{da}\lambda_d(X) = \left|\,|X|_{m_a} - |X|_{m_a}\right|_{m_d}$ and ${}_{ba}\lambda_b(X) = \left|\,|X|_{m_b} - |X|_{m_a}\right|_{m_b}$ are identically phrased and because ${}_{da}\lambda_d(X)$ can take on at most m_a values, a one-dimensional table can be constructed matching the m_a values of ${}_{ba}\lambda_b(X)$ with the at most m_a values of ${}_{da}\lambda_d(X)$.

Consequently, the quantity ${}_{ba}\lambda_b(X) = \left|\,|X|_{m_b} - |X|_{m_a}\right|_{m_b}$ and the table uniquely determines

$$ {}_{da}\lambda_d(X) = \left|\,|X|_{m_d} - |X|_{m_a}\right|_{m_d} \quad \text{and} \quad 0 \leq X \leq m_a m_b - 1. $$

This proves the theorem.

Correspondence Theorem Two:

Given that m_a and m_b are relatively prime, $0 \leq X \leq m_a m_b - 1$ and the residues of X, $|X|_{m_a}$ and $|X|_{m_b}$ are available then,

$$ {}_{da}\lambda_d(X) = \left| m_a \left| \left|\tfrac{1}{m_a}\right|_{m_b} \cdot {}_{ba}\lambda_b(X) \right|_{m_b} \right|_{m_d} $$

where,

$$ {}_{da}\lambda_d(X) \triangleq \left|\,|X|_{m_d} - |X|_{m_a}\right|_{m_d} \quad \& \quad {}_{ba}\lambda_b(X) \triangleq \left|\,|X|_{m_b} - |X|_{m_a}\right|_{m_b} $$

(equation III.23)

Proof: In terms of the mixed-radix notations, a number X in interval $0 \leq X \leq m_a m_b - 1$ can be expressed as:

$$ X = P_1(X) + m_a P_2(X) \qquad \text{(III.24)} $$

where $0 \leq P_1(X) \leq m_a - 1$ and $0 \leq P_2(X) \leq m_b - 1$; assuming $m_a < m_b$. This equation has the same form as (I.3), hence

$$ P_1(X) = |X|_{m_a} $$
$$ P_2(X) = \left[\frac{X}{m_a}\right]_{m_b} $$

Invoking equation (III.20),

$$ \therefore \; P_2(X) = \left[\frac{X}{m_a}\right]_{m_b} = \left|\frac{|X|_{m_b} - |X|_{m_a}}{m_a}\right|_{m_b} $$
$$ = \left| \left|\tfrac{1}{m_a}\right|_{m_b} \cdot \left|\,|X|_{m_b} - |X|_{m_a}\right|_{m_b} \right|_{m_b} $$
$$ = \left| \left|\tfrac{1}{m_a}\right|_{m_b} \cdot {}_{ba}\lambda_b(X) \right|_{m_b} \quad \text{by definition (eqn. III.24).} $$

Hence equation (III.23) becomes,

$$ X = |X|_{m_a} + m_a \left| \left|\tfrac{1}{m_a}\right|_{m_b} \cdot {}_{ba}\lambda_b(X) \right|_{m_b} $$

$$ \text{i.e.} \quad \left|X - |X|_{m_a}\right|_{m_b} = \left| m_a \left|\tfrac{1}{m_a}\right|_{m_b} \cdot {}_{ba}\lambda_b(X) \right|_{m_b} $$

The left-hand side fo this expression is defined as ${}_{da}\lambda_d(X)$ and this proves the theorem.

Comments on the Correspondence Theorems:

A useful interpretation of the Correspondence Theorems is that: If an integer can be uniquely represented by two residues, then the Correspondence Theorems imply that it is possible to find the residue of that integer modulo any other modulus. This interpretation is the basis of the Two residue base-extension (TRB-E) algorithm which we are to discuss in the next section. The difference between the two theorems was stated at the beginning of this section.

III.3.3 : Two Residue Base-Extension (TRB-E)

Two residue base-extension (TRB-E) is the operation of computing residues of an integer X modulo m_d from the information contained in two residues of X, $|X|_{m_a}$ and $|X|_{m_b}$ which uniquely represent X; i.e. X is within range $0 \leq X \leq m_a m_b - 1$.

The validities of the Correspondence Theorems permit TRB-E operation; Theorem One shows that $|X|_{m_d}$ can be computed using the λ's transforms and a one-dimensional table which exists, while Theorem two gives the table in a closed form. Let us consider the Theorems and see how $|X|_{m_d}$ can be computed; from equation (III.23),

$$ {}_{da}\lambda_d(X) = \left| m_a \left| \left|\tfrac{1}{m_a}\right|_{m_b} \cdot {}_{ba}\lambda_b(X) \right|_{m_b} \right|_{m_d} \qquad \text{(III.25a)} $$

$$ {}_{da}\lambda_d(X) = \left|\,|X|_{m_d} - |X|_{m_a}\right|_{m_d} \qquad \text{(III.25b)} $$

and

$$ {}_{ba}\lambda_b(X) = \left|\,|X|_{m_b} - |X|_{m_a}\right|_{m_b} \qquad \text{(III.25c)} $$

The computational steps are:

1) Compute ${}_{ba}\lambda_b(X)$

Since $|X|_{m_a}$ and $|X|_{m_b}$ are given, hence ${}_{ab}\lambda_b(X)$ can be computed using equation (III.25c).

2) Compute ${}_{da}\lambda_d(X)$

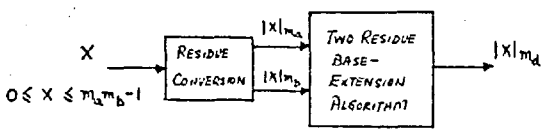
In equation (III.25a), m_a and $\left|\frac{1}{m_a}\right|_{m_b}$ are constants which can be pre-determined. ${}_{ba}\lambda_b(X)$ has just been calculated so that ${}_{da}\lambda_d(X)$ can be computed.

3) Compute $|X|_{m_d}$

Equation (III.25b) can be expressed more conveniently as, $$|X|_{m_d} = \left||X|_{m_a} + {}_{da}\lambda_d(X)\right|_{m_d}$$
Since $|X|_{m_a}$ and ${}_{da}\lambda_d(X)$ are given and calculated respectively, hence $|X|_{m_d}$ can be found.

As (TRB-E) algorithm is used extensively, let us summarize the procedure of it; but it must be emphasized that the procedure is valid only if the residue uniquely represent X; that is $0 \leq X \leq m_a m_b - 1$.

(1) The Two Residue Base-Extension (TRB-E) Algorithm



Input: Two residues of an integer X which uniquely represent X, $|X|_{m_a}$ and $|X|_{m_b}$.

Assuming: $(m_a, m_b) = 1$, $m_a < m_b$, and $0 \leq X \leq m_a m_b - 1$ for unique representation of X.

Output: The residue of X modulo another modulus; i.e.

Procedure:

1) Compute ${}_{ba}\lambda_b(X)$

from $$ {}_{ba}\lambda_b(X) = \left||X|_{m_b} - |X|_{m_a}\right|_{m_b} $$

2) Compute ${}_{da}\lambda_d(X)$

Use the ${}_{ba}\lambda_b(X)$ just computed as an address to a one-dimensional table to obtain,
$$ {}_{da}\lambda_d(X) = \left||X|_{m_d} - |X|_{m_a}\right|_{m_d}. $$
If higher speed of computation is not required, equation (III.25a) can be used to compute ${}_{da}\lambda_d(X)$.

3) Compute $|X|_{m_d}$

from $$ |X|_{m_d} = \left||X|_{m_a} + {}_{da}\lambda_d(X)\right|_{m_d}. $$

4) Exit.

(2) TRB-E as building blocks for other residue base-extensions

The results of section III.3.3 are applicable to the (K,K-1,K+a) RNS's where each has two moduli as factors. With similar argument we can construct "Three RB-E", or in general "q RB-E algorithm". In appendix IV, we shall discuss algorithms for thres residue base-extension (3 RB-E) and four residue base-extension (FRB-E). However, these algorithms can be constructed using TRB-E algorithm as a building block; i.e. arrange the TRB-E in cascades. The resulting base-extension algorithms have been shown[90,100] more economical than these based on explicit mixed-radix conversion.

With the knowledge of the foregoing subsections, we can now discuss macrocoefficient extraction operations.

III.3.4 : Macrocoefficient Extraction Algorithms

Macrocoefficient extraction is the process of computing the complete residue representations for the macrocoefficients for a given RNS; given a sequence of residues $(|X|_{m_i})_{i=1}^{n}$ which uniquely represent X.

In section III.2, we discussed the (K,K-1,K+a) RNS's and saw that (K,K+a) RNS was a general case for the (K,K-1,K+a) RNS's and other RNS's formed by combining the (K,K-1,K+a). We shall discuss procedure for calculating $\gamma_x, \delta_x, \Gamma_x$ & ζ_x in the

(K, K+a) RNS. The TRB-E algorithm is used for the macrocoefficient extraction. In the (K,K+a) RNS, we have:

$$\left.\begin{aligned} K &= m_1\, m_2 \\ K+a &= m_5\, m_6 \end{aligned}\right\}$$ moduli are relatively prime; $m_i < m_{i+1}$; $m_5 < m_6$.

and $0 \leqslant X \leqslant M-1$ where $M = K(K+a)$

also, (γ_x, δ_x) , $(\Gamma_x, \mathcal{S}_x)$ are the macrocoefficient pairs.

(1) To compute complete residue representations of (γ_x, δ_x)

From equation (III.8), integer X is represented by,

where
$$\left.\begin{aligned} X &= K\left[\frac{X}{K}\right] + |X|_K = K\gamma_x + \delta_x \\ 0 \leqslant \delta_x &\triangleq |X|_K \leqslant K-1 \\ 0 \leqslant \gamma_x &\triangleq \left[\frac{X}{K}\right] \leqslant (K+a)-1 \end{aligned}\right\} \qquad \text{(III.26)}$$

Invoking Composite Residue Theorem (I.25),

$$\therefore |\delta_x|_{m_i} = \left||X|_K\right|_{m_i} = |X|_{m_i} \quad , \; i = 1,2$$

and $|\delta_x|_{m_i}$ uniquely represent δ_x . Since $\left(|x|_{m_i}\right)_{i=1}^{2}$ are given, therefore, $|\delta_x|_{m_i}$ for i=1,2 are known. Because $|\delta_x|_{m_i}$, i=1,2 uniquely represent X and they are known, we can use the TRB-E algorithm to compute $|\delta_x|_{m_c}$ where c=5,6.

From equation (III.26) we can write,

$$\gamma_x = \frac{X - \delta_x}{K}$$

Since $0 \leqslant \gamma_x \leqslant (K+a)-1$, taking modulo m_c (c=5,6):

$$\therefore |\gamma_x|_{m_c} = \left|\frac{X-\delta_x}{K}\right|_{m_c} = \left|\left|\frac{1}{K}\right|_{m_c} \cdot \left||X|_{m_c} - |\delta_x|_{m_c}\right|_{m_c}\right|_{m_c} \qquad \text{(III.27)}$$

In this expression, $\left|\frac{1}{K}\right|_{m_c}$ are constants which can be pre-determined, $|X|_{m_c}$ are given; and $|\delta_x|_{m_c}$ have just been computed, therefore we can compute $|\gamma_x|_{m_c}$.

Now, $0 \leqslant \gamma_x \leqslant (K+a)-1$, therefore $|\gamma_x|_{m_c}$ (c=5,6) uniquely represent γ_x and we have just calculated $|\gamma_x|_{m_c}$. Hence we can use the TRB-E algorithm to compute $|\gamma_x|_{m_i}$ (i=1,2). With this computation, we have the complete residue representations of (γ_x, δ_x). Summarizing, the procedure is:-

Procedure to compute (γ_x, δ_x) :

1) Compute $|\delta_x|_{m_c}$ where c =5,6

Since $|\delta_x|_{m_c} = |x|_{m_i}$ where i=1,2 therefore use $\left(|\delta_x|_{m_i}\right)_{i=1}^{2}$ as inputs to TRB-E to obtain $|\delta_x|_{m_5}$, $|\delta_x|_{m_6}$.
(N.B. Furthermore we can compute $|\delta_x|_{m_k}$ where m_k is any other modulus as desired.)

2) Compute $|\gamma_x|_{m_c}$ where c = 5,6

With the knowledge of $|\delta_x|_{m_5}$ and $|\delta_x|_{m_6}$ and equation (III.27) compute $|\gamma_x|_{m_5}$, $|\gamma_x|_{m_6}$.

3) Compute $|\gamma_x|_{m_i}$ where i=1,2

Use $|\gamma_x|_{m_5}$ and $|\gamma_x|_{m_6}$ as inputs to TRB-E and calculate $|\gamma_x|_{m_1}$, $|\gamma_x|_{m_2}$ (and $|\gamma_x|_{m_k}$; k being any other modulus).

4) Complete residue representations obtained; exit.

(2) To compute complete residue representations of $(\Gamma_x, \mathcal{S}_x)$

The reasoning is identical in principle as for (γ_x, δ_x). From equation (III.8),

where
$$\left.\begin{aligned} X &= (K+a)\Gamma_x + \mathcal{S}_x \\ 0 \leqslant \mathcal{S}_x &\leqslant (K+a)-1 \\ 0 \leqslant \Gamma_x &\leqslant K-1 \end{aligned}\right\} \qquad \text{(III.27)}$$

Now, $|\mathcal{S}_x|_{m_c} = |X|_{m_c}$ where c=5,6 are known and they uniquely represent $\mathcal{S}_x$. Therefore $|\mathcal{S}_x|_{m_i}$ (i=1,2) can be found in using TRB-E algorithm.

From equation (III.27) we can write,

$$\Gamma_x = \frac{X - \mathcal{S}_x}{K+a}$$

taking modulo m_i where i=1,2 we obtain:

$$|\Gamma_x|_{m_i} = \left|\frac{X-\mathcal{S}_x}{K+a}\right|_{m_i} = \left|\left|\frac{1}{K}\right|_{m_i} \cdot \left||X|_{m_i} - |\mathcal{S}_x|_{m_i}\right|_{m_i}\right|_{m_i} \qquad \text{(III.28)}$$

$\left|\frac{1}{K}\right|_{m_i}$ and $|X|_{m_i}$ are predeterminable and given; also $|\mathcal{S}_x|_{m_i}$ have been found, therefore $|\Gamma_x|_{m_i}$ can be computed. With $|\Gamma_x|_{m_i}$, which uniquely represent Γ_x , we can use the TRB-E algorithm to obtain $|\Gamma_x|_{m_c}$ where c=5,6. Let outline the procedure.

Procedure to compute (Π_x, S_x) :

1) Compute $|S_x|_{m_i}$ where i=1,2

Since $|S_x|_{m_c} = |x|_{m_c}$ where c=5,6; therefore with them as inputs to the TRB-E algorithm to compute $|S_x|_{m_i}$ where i=1,2. (N.B. We can compute, if desired, $|S_x|_{m_R}$ where m_R is any other modulus.)

2) Compute $|\Pi_x|_{m_i}$ where i=1,2

With the knowledge of $|S_x|_{m_i}$ and equation (III.28) $|\Pi_x|_{m_c}$ where c=5,6 can be calculated.

3) Compute $|\Pi_x|_{m_c}$ where c=5,6

Use $|\Pi_x|_{m_i}$ (i=1,2) as inputs to the TRB-E algorithm to obtain $|\Pi_x|_{m_c}$ (also $|\Pi_x|_{m_R}$ where m_R is any other modulus).

4) Exit.

* * * * * *

This concludes the discussion of macrocoefficients and macrocoefficient extraction.

SECTION IV

BASE EXTENSION AND ALGORITHMS

IV.1 : Base-extension

IV.2 : Base-extension algorithms

IV.2.1 : Base-extension using mixed-radix conversion

IV.2.2 : Base-extension using macrocoefficient extraction

IV.2.2.1 : General Case

IV.2.2.2 : Three residue base-extension

IV.2.2.3 : Four residue base-extension

IV : Base - Extension and Algorithms (82,84,87,89,90,93-102)

IV.1 : Base-extension

Extension of bases or base-extension is the procedure of finding the residue digits for a new set of moduli ($m_{n+1}, m_{n+2}, \ldots, m_{n+k}$), when the residue digits $(|X|_{m_i})_{i=1}^{n}$ relative to another set of moudli $(m_i)_{i=1}^{n}$ are given. The new base (or set of moduli) is often an extension or enlargement of the orginal base.

Base-Extension is often required for fractional multiplication algorithms and error detection and correction algorithms. There are two approaches for base-extension: One uses ths mixed-base conversion and the other employs the macro-coefficient extraction techniques. We shall discuss these approaches in the next section and its subsections.

IV.2 : Base-Extension Algorithms

IV.2.1 : Base-extension algorithm using mixed-radix conversion

(references: 82,84,87,88,93-99)

Mixed radix number system was discussed in section II; from equation (II.5), an integer X can be expressed in the associated system by:

$$\left.\begin{aligned} X &= P_n \prod_{i=1}^{n-1} m_i + \cdots + P_3 m_1 m_2 + P_2 m_1 + P_1 \\ &\text{where} \\ &0 \leq X \leq M-1 \\ &M = \prod_{i=1}^{n} m_i \\ &(m_i)_{i=1}^{n} \\ &\text{and} \\ &0 \leq P_i \leq m_i - 1 \end{aligned}\right\} \quad (IV.1)$$

Now, required to add an extra mixed-radix coefficient P_c corresponding to a modulus m_c (c=n+1, n+2, ..., n+r), which is relatively prime to M, to equation (IV.1) where X is still bounded by $0 \leq X \leq M-1$, so that:

$$X = \underbrace{P_c \prod_{i=1}^{n} m_i}_{\text{Added Term}} + P_n \prod_{i=1}^{n-1} m_i + \cdots + P_3 m_1 m_2 + P_2 m_1 + P_1 \qquad (IV.2)$$

Since X is still restricted to range $0 \leq X \leq M-1$, the coefficient P_c must be equal to zero. This fact is the basis of the algorithm. From equation (II.12), P_c can be represented by:

$$P_c = \left| \begin{array}{c} \dfrac{|X|_{m_c} - P_1}{m_1} - P_2 \\ \hline m_2 \\ \vdots \\ \hline -P_n \\ \hline m_n \end{array} \right|_{m_c} \cdots$$

which can be expressed as,

$$\left.\begin{aligned} P_c &= \left| \frac{|X|_{m_c}}{M} + Q_c \right|_{m_c} \\ &\text{where} \\ &M = \prod_{i=1}^{n} m_i \\ &\text{and} \quad Q_c = -\left(\frac{P_1 + m_1 P_2 + \cdots + m_1 m_2 \cdots m_{n-1} P_n}{M} \right) = -\frac{X}{M} \end{aligned}\right\} \quad (IV.3)$$

Because $P_c = 0$, hence from equation (IV.3) we can solve for $|X|_{m_c}$ iff $(M, m_c)=1$. Since the moduli are relatively prime, therefore $(M, m_c)=1$ is satisfied.

$$\therefore \quad P_c = \left| \frac{|X|_{m_c}}{M} + Q_c \right|_{m_c} = 0$$

$$\left| |X|_{m_c} + M Q_c \right|_{m_c} = 0$$

$$|X|_{m_c} = \left| -M Q_c \right|_{m_c} = \left| |-M|_{m_c} \cdot |Q_c|_{m_c} \right|_{m_c} \qquad (IV.4)$$

$|-M|_{m_c}$ is a constant which can be predetermined, hence we are only required to calculate $|Q_c|_{m_c}$ to determine $|X|_{m_c}$.

Obeerve in equation (IV.3) that P_c is obtained at the end of a mixed-radix conversion process, therefore the quantity at the m_c digit position (or in the m_c adder) is

$$P_c = \left| \frac{|X|_{m_c}}{M} + Q_c \right|_{m_c}$$

Therefore, if we initially set m_c adder to zero and perform the mixed-radix conversion process as if P_c was to be computed; then

the contents of adder m_c at the end of the process is $|Q_c|_{m_c}$. Multiplying this by $|-M|_{m_c}$ we obtain $|X|_{m_c}$. Alternatively, we can use tables-lookup technique to obtain $|X|_{m_c}$ from $|Q_c|_{m_c}$.

We shall illustrate this algorithm with a numerical example and concludes this subsection with an outline of the algorithm.

A numerical example for base-extension using mixed-radix conversion

Given the residue representation $x \leftrightarrow \{1,1,3\}$ for the base with moduli 2, 3, and 5. Find $|x|_7$; i.e. extend base to include $m_4 = 7$.

Modular Steps	Residue Operations	m_1 2	m_2 3	m_3 5	m_4 7	
	Residue Representation of X	(1)	1	3	$\|X\|_7 + 0$	
1	Substract $P_1 = 1$	1	1	1	1	$P_1 = 1$
		0	0	2	$\|X\|_7 + 6$	
	Multiply By $\|\frac{1}{2}\|_{m_2}$		2	3	4	
			(0)	1	$4\|X\|_7 + 3$	$P_2 = 0$
2	Substract $P_2 = 0$		0	0	0	
				1	$4\|X\|_7 + 3$	
	Multiply By $\|\frac{1}{3}\|_{m_3}$			2	5	
				(2)	$20\|X\|_7 + 1$	$P_3 = 2$
3	Substract $P_3 = 2$			2	2	
				0	$20\|X\|_7 + 6$	
	Multiply By $\|\frac{1}{5}\|_{m_4}$				3	
					$60\|X\|_7 + 4$	

$$\therefore P_4 = \left|60|X|_7 + 4\right|_7 = \Big|\underbrace{\left|\tfrac{1}{2}\right|_7 \cdot \left|\tfrac{1}{3}\right|_7 \cdot \left|\tfrac{1}{5}\right|_7}_{\text{Predeterminable constant Equals } \left|\frac{1}{30}\right|_7} \cdot |X|_7 + 4\Big|_7 = 0 \qquad \text{(A)}$$

$$\therefore \left|\left|\tfrac{1}{30}\right|_7 \cdot |X|_7 + 4\right|_7 = 0$$

$$\text{or} \quad \left|\tfrac{1}{2} \cdot |X|_7 + 4\right|_7 = 0$$

$$\therefore \left|\tfrac{1}{2} \cdot |X|_7\right|_7 = |-4|_7 \text{ or } |3|_7 \qquad \text{(B)}$$

Multiply both sides by "2 modulo 7",

$$\therefore |X|_7 = |2 \cdot 3|_7 = 6 \qquad \text{(C)}$$

Observations:

1) In carrying out the "base-extension" process, the multipliers (viz. $\left|\frac{1}{2}\right|_7, \left|\frac{1}{3}\right|_7, \left|\frac{1}{5}\right|_7$) are independent of the residues of X, i.e. $\{1,1,3\}$. These multipliers are used in the last stage of the process (see line (A)). Hence they need not be carried along in the residue operations (i.e. the table).

This is why $|60|_7$ was left un-operated upon through out the conversion process to indicate that :

$$|60|_7 = \left|\left|\tfrac{1}{2}\right|_7 \cdot \left|\tfrac{1}{3}\right|_7 \cdot \left|\tfrac{1}{5}\right|_7\right|_7 \quad \text{or} \quad \left|\tfrac{1}{30}\right|_7 = \left|\tfrac{1}{2}\right|_7$$

Hence the conversion can be directed from step 3 immediately to line (C) , missing out steps (A) through (B),

$$\text{i.e.} \quad |X|_7 = \left|2 \times |-4|_7\right|_7 = |2 \times 3|_7 = \underline{6}$$

2) The last step of the base-extension process to obain P_4 is unnecessary (i.e. multiplication by $\left|\frac{1}{5}\right|_7$ to obtain P_4). Hence after subtraction at step 3, we can write:

$$\left|\left|\tfrac{1}{2}\right|_7 \cdot \left|\tfrac{1}{3}\right|_7 \cdot \left|\tfrac{1}{5}\right|_7 \cdot |X|_7 + \left|\tfrac{1}{5}\right|_7 \cdot 6\right|_7 = 0$$

Multiplying by 5,

$$\therefore \left|\left|\tfrac{1}{2}\right|_7 \cdot \left|\tfrac{1}{3}\right|_7 \cdot |X|_7 + 6\right|_7 = 0$$

$$\left|\left|\tfrac{1}{6}\right|_7 \cdot |X|_7\right|_7 = |-6|_7 \text{ or } |1|_7$$

$$\therefore |X|_7 = |6 \times 1|_7 = \underline{6}$$

which yields the correct result. This shows that $|x|_7$ can be obtained immediately after subtraction at step 3 by writing:

$$|X|_7 = \left|\left|2 \times 3\right|_7 \times |-6|_7\right|_7 = \left|6 \times 1\right|_7 = \underline{6}$$

Outline of the base-extension algorithm using mixed-radix conversion

GIVEN : "n" non-redundant moduli $m_1, m_2, \cdots, m_n$ and "r" redundant moduli $m_{n+1}, m_{n+2}, \cdots, m_{n+r}$; we wish to compute $\left(|X|_{m_c}\right)_{c=n+1}^{n+r}$ from the given residue $\left(|X|_{m_i}\right)_{i=1}^{n}$

Procedure :

1) (a) Enter residues $|X|_{m_i}$ in the corresponding moduli m_i adders.

(b) Set modulo m_c adders to zero.

2) (a) Subtract $P_1 = |x|_{m_1}$, modulo m_a, where $a = 1,2, \ldots, n+r$, from the contents of all the adders.

(b) Multiply the contents of the adders $(m_i)_{i=2}^{n+r}$ by $\left|\frac{1}{m_1}\right|_{m_j}$ modulo m_j where $j = 2,3, \ldots, n+r$.

3) (a) Subtract $P_2 = \left[\frac{x}{m_1}\right]_{m_2}$ modulo m_j where $j = 2,3, \ldots, n+r$ from contents of all the modulo m_j adders. P_2 was the content of modulo m_2 adder after the multiplication at step 2(b).

(b) Multiply the contents of the adder $(m_k)_{k=3}^{n+r}$ by $\left|\frac{1}{m_1 m_2}\right|_{m_k}$ modulo m_k where $k = 3,4, \ldots, n+r$.

$\ddots$

n) (a) Subtract $P_{n-1} = \left[\frac{x}{m_1 m_2 \cdots m_{n-2}}\right]_{m_{n-1}}$ modulo m_ℓ, where $\ell = n, n+1, n+2, \ldots, n+r$ from the contents of the modulo m_q adders. P_{n-1} was the contents of modulo m_{n-1} adder after multiplication at step (n-1)b.

(b) Multiply the contents of adders $(m_q)_{q=n}^{n+r}$ by $\left|\frac{1}{m_{n-1}}\right|_{m_q}$ modulo m_q where $q = n,n+1, \ldots, n+r$.

n+1) (a) Subtract $P_n = \left[\frac{x}{m_1 m_2 \cdots m_{n-1}}\right]_{m_n}$ modulo m_q, where $q = n,n+1, n+2, \ldots, n+r$ from the contents of the modulo m_q adders. P_n was the contents of modulo m_n adder after multiplication at step nb.

(b) Multiply the contents of addere $(m_{n+1}, m_{n+2}, \ldots, m_{n+r})$ by $\left|\frac{1}{m_n}\right|_{m_c}$ modulo m_c where $c = n+1, n+2, \ldots, n+r$.

Final step : compute sequence $(|x|_{m_c})_{c=n+1}^{n+r}$

$|Q_c|_{m_c}$ are the contents of modulo m_c adders after the multiplication at step (n+1)b. Multiply the contents of each modulo m_c adder by $|-M|_{m_c}$ modulo m_c results the sequence $(|x|_{m_c})_{c=n+1}^{n+r}$ being stored in the m_c adders.

IV.2.2 : Base-extension algorithms using Macrocoefficient Extraction (ref.: 89,90,100-102)

Algorithms presented in this section are less general than that of section IV.2.1 : base-extension algorithm using mixed-radix conversion is applicable to any number of moduli, whereas those using macrocoefficient extraction are limited to number systems with integers uniquely represented by only four moduli. We shall consider some algorithms of the latter type in this section.

The systems under discussion are the (K,K-1,K+a) RRNS, where $a > 0$ and X is bounded by $0 \leq X \leq K(K-1)$. From this RRNS, as discussed in section III.1 of appendix III, any of the following three RNS's uniquely represent X,

$$\left.\begin{array}{ll} \text{System A:} & (K,\ K+a) \\ \text{System B:} & (K,\ K-1) \\ \text{System C:} & (K-1,\ K+a) \end{array}\right\} \qquad \text{(IV.5)}$$

Using base-extension, for a given representation of X in one system we can compute the residue representation of X for the remaining two moduli in the (K,K-1,K+a) RNS's. For example, given the (K,K-1) residues of X we can find the (K+a) residues. As observed in section III.1, we need only consider the "general case"; that is,given an integer X uniquely represented in a (K,K+a) RNS, where

$$K = m_1 \cdot m_2 \ ,$$
$$K + a = m_5 \cdot m_6 \ ,$$

with moduli being relatively prime and X bounded by

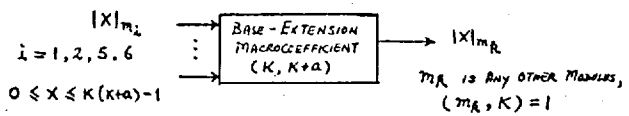
$$0 \leq X \leq K(K+a) - 1$$

we wish to compute the residue of X modulo m_k, where m_k is any other moduli relatively prime to m_1 and m_2. A comment on other base-extension algorithms is to follow.

Two residue base-extension (TRB-E) algorithm, which is the basis of macrocoefficient extraction, was discussed in section III.3.3 of section III. It was indicated therein the "q residue base-extension" (qRB-E), with "q" greater than two, could similarly be constructed as TRB-E. Alternatively, we could realise qRB-E using TRB-E as basic building block,

i.e. cascading TRB-E blocks. Herein, we shall discuss the "general case" of base-extension, described in the previous paragraph, and two other special cases; viz. three residue base-extension (3RB-E) and four residue base-extension (FRB-E). 3RB-E is useful for fractional multiplication and FRB-E is useful for error-correction and detection applications.

IV.2.2.1 : General case -- compute the m_k residue of X given the (K,K+a) residues of X



From equation (III.8), integer X is uniquely represented by,

$$X = K\gamma_x + \delta_x \qquad \text{(IV.6)}$$

Taking modulo m_k of (IV.6),

$$\therefore |X|_{m_R} = |K\gamma_x + \delta_x|_{m_R} = \left| |K|_{m_R} \cdot |\gamma_x|_{m_R} + |\delta_x|_{m_R} \right|_{m_R} \qquad \text{(IV.7)}$$

$|K|_{m_R}$ is a constant which is predeterminable; hence $|X|_{m_R}$ can be found if $|\gamma_x|_{m_R}$ and $|\delta_x|_{m_R}$ are computed using the TRB-E discussed in section III.3.3 of section III. Let us outline the procedure for this general case of base-extension.

Procedure : To compute m_k residue of X given the (K,K+a) residue of X

1) Compute $|\delta_x|_{m_i}$ where $i = 5,6,k$

Since $0 \leq \delta_x \leq K-1$, therefore by Composite Modulus Theorem (I.25), $|\delta_x|_{m_1} = |X|_{m_1}$ and $|\delta_x|_{m_2} = |X|_{m_2}$ and these residues uniquely represent X. Hence use $|\delta_x|_{m_1}$ and $|\delta_x|_{m_2}$ as inputs to a TRB-E to obtain.

$$|\delta_x|_{m_i} \quad \text{where} \quad i = 5,6,k$$

2) Compute $|\gamma_x|_{m_i}$ where $i = 5,6$

Rewrite equation (IV.7) as

$$|\gamma_x|_{m_i} = \left| \left|\frac{1}{K}\right|_{m_i} \cdot |X|_{m_i} - |\delta_x|_{m_i} \right|_{m_i}$$

$\left|\frac{1}{K}\right|_{m_i}$ is a constant and predeterminable, $|X|_{m_i}$ are given, and $|\delta_x|_{m_i}$ have just been found in step 1. Therefore, we can obtain,

$$|\gamma_x|_{m_i} \quad \text{where} \quad i = 5, 6.$$

3) Compute $|\gamma_x|_{m_R}$

Since $0 \leq \gamma_x \leq (K+a)-1$ therefore $|\gamma_x|_{m_i}$ $(i=5,6)$ uniquely represent γ_x ; thus can use TRB-E to obtain $|\gamma_x|_{m_R}$.

4) Compute $|X|_{m_R}$ and EXIT

$|\gamma_x|_{m_R}$ and $|\delta_x|_{m_R}$ are now known, therefore we can calculate $|X|_{m_R}$ using equation (IV.7).

IV.2.2.2 : Three residue base-extension algorithm

Three residue base-extension (3RB-E) is defined as follows: given an integer X bounded by range $0 \leq X \leq m_b m_d m_e - 1$ and uniquely represented by

$$|X|_{m_b}, \quad |X|_{m_d}, \quad |X|_{m_e}$$

where the moduli are relatively prime. Find the residue of X modulo m_k (where m_k is relatively prime to the moduli).

The number system is "similar" to the (K,K+a) RNS is we define,

$$\left.\begin{aligned} X &= K\left[\frac{X}{K}\right] + |X|_K = K\gamma_x + \delta_x \\ K &= m_b m_d \end{aligned}\right\} \qquad \text{(IV.8)}$$

Since $0 \leq X \leq m_b m_d m_e - 1$, and from (IV.8) therefore,

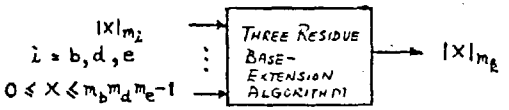
$$\left.\begin{aligned} & 0 \leq \delta_x \leq K-1 \text{ or } m_b m_d - 1 \\ \text{but} \quad & 0 \leq \gamma_x \leq m_e - 1 \\ \text{so that} \quad & |\gamma_x|_{m_e} = \gamma_x \end{aligned}\right\} \qquad \text{(IV.9)}$$

Now take residue modulo m_k of equation (IV.8) therefore,

$$\begin{aligned} |X|_{m_k} &= |K\gamma_x + \delta_x|_{m_k} \\ &= \left| |K|_{m_k} \cdot |\gamma_x|_{m_k} + |\delta_x|_{m_k} \right|_{m_k} \end{aligned} \qquad \text{(IV.10)}$$

Since $|K|_{m_k}$ is a constant and predeterminable, we only need to compute $|\gamma_x|_{m_k}$, $|\delta_x|_{m_k}$ to obtain $|X|_{m_k}$. Let us outline this process which is known as (3RB-E) algorithm.

Procedure of Three residue base-extension algorithm



1) Compute $_{db}\lambda_d(\delta_x)$

From equation (IV.9) we have $0 \leq \delta_x \leq m_b m_d - 1$ therefore δ_x is uniquely represented by $|\delta_x|_{m_b}$, $|\delta_x|_{m_d}$. By the Composite Modulus Theorem (I.25),

$$|\delta_x|_{m_b} = |X|_{m_b} \quad \text{and} \quad |\delta_x|_{m_d} = |X|_{m_d} \tag{IV.11}$$

The λ - transform, using equation (III.14), is given by,

$$\begin{aligned} {}_{db}\lambda_d(\delta_x) &= \left| |\delta_x|_{m_d} - |\delta_x|_{m_b} \right|_{m_d} \\ &= \left| |X|_{m_d} - |X|_{m_b} \right|_{m_d} \end{aligned} \tag{IV.12}$$

Since $|X|_{m_b}$ and $|X|_{m_d}$ are given therefore $_{db}\lambda_d(\delta_x)$ is computable.

2) Compute $_{kb}\lambda_k(\delta_x)$ and $_{eb}\lambda_e(\delta_x)$

Again from equation (III.14) we have,

$$ {}_{jb}\lambda_j(\delta_x) = \left| |\delta_x|_{m_j} - |\delta_x|_{m_b} \right|_{m_j} \quad , \; j = k, e \tag{IV.13}$$

Equation (IV.13) can be computed by using $_{db}\lambda_d(\delta_x)$ as an address to a one-dimension table as stated by Correspondence Theorem One (III.22). Sacrifying computation speed, we can alternatively use Correspondence Theorem Two (III.23) to obtain $_{jb}\lambda_j(\delta_x)$; so that write (IV.13) as:

$$ {}_{jb}\lambda_j(\delta_x) = \left| m_b \cdot \left| \frac{1}{m_b} \right|_{m_j} \cdot {}_{db}\lambda_d(\delta_x) \right|_{m_j} \quad , \; j = k, e \tag{IV.14}$$

3) Compute $|\delta_x|_{m_j}$, $j = k, e$

From equation (III.23) we can express $|\delta_x|_{m_j}$ as

$$ |\delta_x|_{m_j} = \left| {}_{jb}\lambda_j(\delta_x) + |\delta_x|_{m_b} \right|_{m_j} \tag{IV.15}$$

Since $_{jb}\lambda_j(\delta_x)$ and $|\delta_x|_{m_b}$ are known from equations (IV.14) and (IV.11), therefore we can find $|\delta_x|_{m_j}$.

4) Compute $|\gamma_x|_{m_e} = \gamma_x$

Taking modulo m_e of equation (IV.8),

$$\therefore \; |\gamma_x|_{m_e} = \left| \frac{x - \delta_x}{K} \right|_{m_e}$$

That is,

$$\gamma_x = |\gamma_x|_{m_e} = \left| \left| \frac{1}{K} \right|_{m_e} \cdot \left| |X|_{m_e} - |\delta_x|_{m_e} \right|_{m_e} \right|_{m_e} \tag{IV.16}$$

$\left| \frac{1}{K} \right|_{m_e}$ is a constant therefore predeterminable; $|X|_{m_e}$ and $|\delta_x|_{m_e}$ are respectively given and found in step 3. Hence we can calculate $\gamma_x = |\gamma_x|_{m_e}$.

5) Compute $|\gamma_x|_{m_k}$

Since γ_x is found from step 4, we can compute residue γ_x modulo m_k ; i.e. $|\gamma_x|_{m_k}$. This can be obtained using combinational logic.

6) Compute $|x|_{m_k}$ and EXIT

Take modulo m_k of equation (IV,8),

$$\therefore \; |X|_{m_k} = \left| |K|_{m_k} \cdot |\gamma_x|_{m_k} + |\delta_x|_{m_k} \right|_{m_k} \tag{IV.17}$$

Since $|K|_{m_k}$ is a predeterminable constant and $|\gamma_x|_{m_k}$, $|\delta_x|_{m_k}$ are respectively found in steps 6,3 hence $|X|_{m_k}$ is obtained as desired. EXIT.

IV.2.2.3 : Four residue base-extension algorithm

Four residue base-extension (FRB-E) is the process of computing the (K+a) residues of X given the (K,K-1) residues of X. Relevant equations are stated in section III.2 of appendix III. From equations (III.8) and (III.5) we have

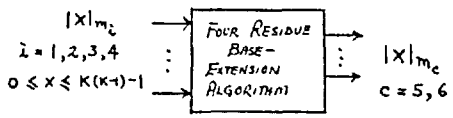
$$\begin{aligned} X &= K\left[\frac{x}{K}\right] + |x|_K = K\gamma_x + \delta_x \\ K &= m_1 \times m_2 \\ K+a &= m_5 \times m_6 \end{aligned} \tag{IV.11}$$

Taking modulo m_c , $c = 5,6$ of equation (IV.11),

$$\begin{aligned} |X|_{m_c} &= \left| K\gamma_x + \delta_x \right|_{m_c} \\ &= \left| |K|_{m_c} \cdot |\gamma_x|_{m_c} + |\delta_x|_{m_c} \right|_{m_c} \end{aligned} \tag{IV.12}$$

$|K|_{m_c}$ is a predeterminable constant, we only need to find $|\gamma_x|_{m_c}, |\delta_x|_{m_c}$ to obtain $|x|_{m_c}$. The process, i.e. FRB-E algorithm, is detailed below.

Procedure of Four residue base-extension algorithm



1) Compute $_{21}\lambda_2(\delta_x)$

Since $0 \le \delta_x \le K-1$, therefore δ_x is uniquely represented by $|\delta_x|_{m_i}$ where $i=1,2$. By the Composite Residue Theorem (I.25) we have,

$$|\delta_x|_{m_i} = |x|_{m_i} \quad ; \quad i = 1,2 \tag{IV.13}$$

The λ-transform defined by equation (III.14) is,

$$_{21}\lambda_2(\delta_x) = \left| |\delta_x|_{m_2} - |\delta_x|_{m_1} \right|_{m_2} = \left| |x|_{m_2} - |x|_{m_1} \right|_{m_2} \tag{IV.14}$$

Since $|x|_{m_1}$ and $|x|_{m_2}$ are given therefore $_{21}\lambda_2(\delta_x)$ can be found.

2) Compute $_{c1}\lambda_c(\delta_x)$ and $_{j1}\lambda_j(\delta_x)$; $c=5,6$ and $j=3,4$

From equation (III.14) we have,

$$\left.\begin{aligned} &_{c1}\lambda_c(\delta_x) = \left| |\delta_x|_{m_c} - |\delta_x|_{m_1} \right|_{m_c} \quad ; \quad c = 5,6 \\ \& \quad &_{j1}\lambda_j(\delta_x) = \left| |\delta_x|_{m_j} - |\delta_x|_{m_1} \right|_{m_j} \quad ; \quad j = 3,4 \end{aligned}\right\} \tag{IV.15}$$

By Correspondence Theorem One (III.22) we can obtain equation (IV.15) by using $_{21}\lambda_2(\delta_x)$, computed in step 1, as address to a one-dimension table. Sacrifying computation speed, we can alternatively use Correspondence Theorem Two (III.23) to obtain equation (IV.15) so that:

$$\left.\begin{aligned} &_{c1}\lambda_c(\delta_x) = \left| m_1 \left|\frac{1}{m_1}\right|_{m_c} \cdot {}_{21}\lambda_2(\delta_x) \right|_{m_c} \quad ; \quad c = 5,6 \\ \& \quad &_{j1}\lambda_j(\delta_x) = \left| m_1 \left|\frac{1}{m_1}\right|_{m_j} \cdot {}_{21}\lambda_2(\delta_x) \right|_{m_j} \quad ; \quad j = 3,4 \end{aligned}\right\} \tag{IV.16}$$

3) Compute $|\delta_x|_{m_c}, |\delta_x|_{m_j}$

From equation (III.23), we have

$$\left.\begin{aligned} &|\delta_x|_{m_c} = \left| c_1 \lambda_c(\delta_x) + |\delta_x|_{m_1} \right|_{m_c} \quad ; \quad c = 5,6 \\ \& \quad &|\delta_x|_{m_j} = \left| {}_{j1}\lambda_j(\delta_x) + |\delta_x|_{m_1} \right|_{m_j} \quad ; \quad j = 3,4 \end{aligned}\right\} \tag{IV.17}$$

Since $|\delta_x|_{m_1} = |x|_{m_1}$, equation (IV.13) and λ's are found in (IV.16) therefore we can compute $|\delta_x|_{m_c}$ and $|\delta_x|_{m_j}$.

4) Compute $|\gamma_x|_{m_j}$; $j=3,4$

Rewriting equation (IV.11) and taking it modulo m_j,

$$|\gamma_x|_{m_j} = \left| \frac{x-\delta_x}{K} \right|_{m_j} = \left| \left|\frac{1}{K}\right|_{m_j} \cdot \left| |x|_{m_j} - |\delta_x|_{m_j} \right|_{m_j} \right|_{m_j} \tag{IV.18}$$

$\left|\frac{1}{K}\right|_{m_j}$ is a predeterminable constant; $|x|_{m_j}$ and $|\delta_x|_{m_j}$ are respectively given and found in step 3, therefore $|\gamma_x|_{m_j}$ are computable.

5) Compute $_{43}\lambda_4(\gamma_x)$

Similar argument as step 1, we have

$$\begin{aligned} _{43}\lambda_4(\gamma_x) &= \left| |\gamma_x|_{m_4} - |\gamma_x|_{m_3} \right|_{m_4} \\ &= \left| |x|_{m_4} - |x|_{m_3} \right|_{m_4} \end{aligned} \tag{IV.19}$$

since $0 \le \gamma_x \le (K-1)-1$; hence $_{43}\lambda_4(\gamma_x)$.

6) Compute $_{c3}\lambda_c(\gamma_x)$; $c=5,6$

Corresponding to step 2, we can write

$$_{c3}\lambda_c(\gamma_x) = \left| |\gamma_x|_{m_c} - |\gamma_x|_{m_3} \right|_{m_c} \tag{IV.20}$$

therefore $_{c3}\lambda_c(\gamma_x)$ can be obtained using $_{43}\lambda_4(\gamma_x)$ and table-lookup. Alternatively, compute $_{c3}\lambda_c(\gamma_x)$ by:

$$_{c3}\lambda_c(\gamma_x) = \left| m_3 \left|\frac{1}{m_3}\right|_{m_c} \cdot {}_{43}\lambda_4(\gamma_x) \right|_{m_c} \tag{IV.21}$$

7) Compute $|\gamma_x|_{m_c}$; $c=5,6$

Similar to step 3, we have

$$|\gamma_x|_{m_c} = \left| {}_{c3}\lambda_c(\gamma_x) + |\gamma_x|_{m_3} \right|_{m_c} \tag{IV.22}$$

Since $0 \le \gamma_x \le (K-1)-1$, therefore $|\gamma_x|_{m_3}$ and $|\gamma_x|_{m_4}$ uniquely represent γ_x. By the Composite Residue Theorem (I.25), $|\gamma_x|_{m_3} = |x|_{m_3}$ and $|\gamma_x|_{m_4} = |x|_{m_4}$. $|x|_{m_3}$ and $|x|_{m_4}$ are given. $_{c3}\lambda_c(\gamma_x)$ were computed in step 6, therefore $|\gamma_x|_{m_c}$ can be found.

8) Compute $|x|_{m_c}$; c=5,6 and EXIT.

Take modulo m_c of equation (IV.11),

$$\therefore |x|_{m_c} = \left| K\gamma_x + \delta_x \right|_{m_c}$$

$$= \left| |K|_{m_c} \cdot |\gamma_x|_{m_c} + |\delta_x|_{m_c} \right|_{m_c} \qquad \text{(IV.23)}$$

$|K|_{m_c}$ is a predeterminable constant, $|\gamma_x|_{m_c}$ and $|\delta_x|_{m_c}$ are respectively found in steps 7 and 3. Consequently $|x|_{m_c}$ are computable. EXIT.

Observe that steps 1 through 3 and 5 through 7 parallel the TRB-E algorithm. This concludes the discussion of base-extension using macrocoefficient extraction and the appendix.

Appendix 16 :

A STUDY ON THE ARITHMETIC LOGIC FOR RESIDUE ADDITION & MULTIPLICATION

Arithmetic Logic for Residue Addition & Multiplication

In order to evaluate the feasibility of using residue number systems, it is of interest to determine how much logic (in terms of logical gates) is required to implement the required arithmetic operations. Addition & multiplication for moduli 2, 3, 5, 7, 11, 13, 17, 19, and 23 have been chosen for our study. CAD techniques developed have been employed for our study. These CAD tools are presented in appendices 2 & 8. Methods employed for Addition & Multiplication are discussed separately and is followed by the results obtained for each. Finally the findings are summarized in tables A15.4 & A15.5.

A) Residue Addition Logic

In order to develop the logic for this operation it is convenient to specify first the coding for the residue digits and for the flip-flops in which they will be stored. The residue digits will be coded in ordinary binary form so that if a number system consisting of residues of 2, 3, 5, 7 is assumed, then 9 bits of storage are required for each word. The two numbers which are to be added together will be stored in two flip-flop registers which will be called the addition register (AR) and the multiplication register (MR). The individual flip-flops will be coded for reference as AR1, AR2, AR3, etc. The logic which will be developed will permit the contents of the MR to be added to the contents of the AR with the sum appearing in the latter. To illustrate the procedure for determining the proper logic, consider the addition of residue of base 3. The addition table and the coding for this base is shown in table A15.1.

Mod 3 +		MR 0	MR 1	MR 2
AR	0	0	1	2
	1	1	2	0
	2	2	0	1

Digit	Code
0	0 0
1	0 1
2	1 0

Table A15.1 : Base 3 Addition Table and Binary Coding

The residue base 3 addition table is obtained using APL functions in appendix 6. The sum of the two base 3 digits will appear in AR2 and AR3, where the latter will be in the MSB (most significant bit) and the former the LSB (least significant bit). The logic to generate the two correct sum bits in the two flip-flop has a complementing input, may be complement logic. The complement input for the flip-flop AR2 will be coded as CAR2 and for AR3 as CAR3. The required logic for each flip-flop then must be true whenever its state must be changed. These conditions can be determined by referring to the addition table and bearing in mind the coding for each digit. LOGMIN has been used as an aid for logic minimization (appendix 2). Thus, the resulting equations for AR3 and AR2 are given below.

$$CAR3 = \overline{AR3}\ \overline{AR2}\ (MR3\ \overline{MR2}) + \overline{AR3}\ AR2\ (\overline{MR3}\ MR2) + AR3\ \overline{AR2}\ (\overline{MR3}\ MR2 + MR3\ \overline{MR2})$$

$$CAR2 = \overline{AR3}\ \overline{AR2}\ (\overline{MR3}\ MR2) + \overline{AR3}\ AR2\ (\overline{MR3}\ MR2 + MR3\ \overline{MR2}) + AR3\ \overline{AR2}\ (MR3\ \overline{MR2}).$$

Logic for the addition of residue digits of other bases may be developed in a similar manner. The logic for the addition of residue digits of bases up to 23 are given in section D. As might be suspected the logic becomes increasingly complex for each greater base so that as more residue digits are used the expense increases exponentially. It is thus desirable to use the smallest base number possible and only as many as is absolutely necessary.

B) Residue Multiplication Logic

The logic required for multiplication is developed in very much the same way as for addition. In this case the two registers which will be used will be called the multiplication register (MR) and the memory buffer register (MB). The flip-flops will be coded in a manner similar to that before. The logic which will be developed will permit the contents of MR to be multiplied by the contents of MB with the product appearing in the former. The multiplication of residues digits of base 3 will be developed here. The multiplication table and coding for base 3 digits are shown in table A15.2.

	Mod 3 X	MB 0	1	2
	0	0	0	0
MR	1	0	1	2
	2	0	2	1

Digit	Code
0	0 0
1	0 1
2	1 0

Table A15.2 : Base 3 Multiplication Table and Binary Coding

As for addition, complementing logic results in the least number of gates, so this form of logic will be used. Thus, referring to the table the logic for MR3 can be written as:

$$CMR3 = \overline{MR3}\ MR2\ (MB3\ \overline{MB2}) + MR3\ \overline{MR2}\ (\overline{MB3}\ \overline{MB2} + MB3\ \overline{MB2})$$

and the logic for MR2 would be:

$$CMR2 = \overline{MR3}\ MR2\ (\overline{MB3}\ \overline{MB2} + MB3\ \overline{MB2}) + MR3\ \overline{MR2}\ (MB3\ \overline{MB2})$$

These may further be simplified to:

$$CMR2 = MR3\ \overline{MB2} + MR2\ MB3$$

$$CMR3 = MR3\ MB3 + MR2\ \overline{MB2}$$

The logic for multiplication of residues digits of other bases is developed in a similar manner. The logic for multiplication of residue digits up to base 23 is given in section D.

C) Logic Equations for Addition Logic (Base 2,3,5,7,11,13,17,19,23)

The logic equations have been developed on the assumption that a complementary input is available on each flip-flop. The numbering of the flip-flops which will store the sum in AR is shown below. The numbers of the flip-flops in MR and MB are analogous. For brevity a letter code will be used in the logic equations.

Storage scheme used in addition register are depicted below:

DIGIT	2	3		5			. . .	23				
	AR	AR	AR	AR	AR	AR		AR	AR	AR	AR	AR
Flip-flop	1	2	3	4	5	6		28	29	30	31	32

MSB

1. Mod 2 Logic

AR1 = A MR1 = B CAR1 = B

2. Mod 3 Logic

AR3 = A MR3 = C CAR2 = BC + $\bar{A}$D + AC
AR2 = B MR2 = D CAR3 = AD + $\bar{B}$C + BD

3. Mod 5 Logic

AR6 = A MR6 = D CAR4 = $\bar{A}\bar{E}F$ + $\bar{A}\bar{B}\bar{F}$ + $B\bar{D}\bar{F}$ + $C\bar{D}\bar{E}$ + $A\bar{D}\bar{F}$ + $A\bar{E}\bar{F}$ + $BCE\bar{F}$
AR5 = B MR5 = E CAR5 = $\bar{B}C\bar{D}\bar{F}$ + $C\bar{E}\bar{F}$ + $B\bar{C}\bar{D}$ + AD + BE + $\bar{C}EF$
AR4 = C MR4 = F CAR6 = $\bar{B}\bar{D}$ + $B\bar{C}\bar{E}\bar{F}$ + AE + $BC\bar{E}\bar{F}$ + $\bar{B}CEF$ + AF

4. Mod 7 Logic

AR9 = A MR9 = D
AR8 = B MR8 = E
AR7 = C MR7 = F

CAR7 = $AC\bar{E}\bar{F}$ + $AB\bar{E}\bar{F}$ + $A\bar{D}\bar{F}$ + $BCD\bar{F}$ + CDE + $B\bar{D}E$
+ $\bar{A}DF$ + $\bar{A}BF$ + BDEF

CAR8 = $AC\bar{D}\bar{E}$ + $BC\bar{D}\bar{E}$ + $AB\bar{D}\bar{E}$ + $\bar{A}\bar{D}\bar{E}\bar{F}$ + $\bar{B}C\bar{E}\bar{F}$
+ $\bar{A}CE$ + $C\bar{D}EF$ + $C\bar{E}F$ + ADF + BDF + $AB\bar{E}F$

CAR9 = $\bar{B}\bar{C}D$ + $AC\bar{D}\bar{E}$ + $\bar{C}D\bar{E}\bar{F}$ + $AC\bar{D}E$ + BCF + $AB\bar{D}F$
+ $B\bar{D}E$ + BDEF + AEF + CEF

5. Mod 11 Logic

AR13 = A MR13 = E
AR12 = B MR12 = F
AR11 = C MR11 = G
AR10 = D MR10 = H

$$CAR10 = A\bar{C}\bar{E}\bar{F}\bar{G}H + AD\bar{E}G\bar{H} + AC\bar{E}\bar{G}H + BCFG\bar{H} + AE\bar{H} + AF\bar{H} + CEG + \bar{A}C\bar{E}\bar{G}H + BEF\bar{H} + BEFH + ABC\bar{H} + \bar{A}B\bar{E}H + BCDF\bar{H}$$

$$CAR11 = AD\bar{E}\bar{G} + D\bar{G}H + BCF\bar{G}H + AC\bar{G}H + \bar{C}DEG + BCDFG + ABDEG + \bar{A}B\bar{D}GH + AB\bar{D}\bar{G}H + BDE\bar{G} + CDE\bar{G} + AEH + BEH + AF\bar{G}H + BD\bar{F}G + ADF\bar{G} + \bar{A}E\bar{F}GH$$

$$CAR12 = \bar{A}\bar{E}\bar{C}DE + B\bar{C}EG + AE\bar{G}\bar{H} + BD\bar{E}G\bar{H} + AC\bar{E}\bar{G} + \bar{A}C\bar{E}\bar{F}G + ADFG + CD\bar{E}\bar{F}H + ACFG + BF + \bar{A}D\bar{F}GH + \bar{A}FG\bar{H} + C\bar{D}FGH + A\bar{C}F\bar{H} + \bar{A}CF\bar{G}$$

$$CAR13 = \bar{B}\bar{C}\bar{D}E + \bar{B}\bar{C}EG + \bar{B}\bar{D}EG\bar{H} + BCD\bar{E}\bar{F}H + BC\bar{E}\bar{F}G + AD\bar{E}G + B\bar{C}F\bar{G} + \bar{B}CFG + B\bar{D}F\bar{G}\bar{H} + \bar{B}DFGH + BD\bar{F}GH + AC\bar{E}G + AC\bar{E}H + AF + \bar{B}CDFH + B\bar{C}D\bar{F}H + AGH$$

6. Mod 13 Logic

AR17 = A MR17 = E
AR16 = B MR16 = F
AR15 = C MR15 = G
AR14 = D MR14 = H

$$CAR14 = ACDG\bar{H} + ABG\bar{H} + AE\bar{H} + \bar{A}G\bar{H} + ACF\bar{H} + ADF\bar{H} + \bar{B}CDEG + BCDF\bar{H} + AB\bar{D}F\bar{H} + \bar{A}B\bar{E}G\bar{H} + \bar{A}B\bar{C}H + \bar{A}B\bar{E}H + \bar{A}\bar{E}F\bar{H} + B\bar{E}\bar{F}GH + AB\bar{C}\bar{E}F\bar{H} + \bar{A}BEG\bar{H} + BCEG\bar{H} + BCE\bar{H} + BDE\bar{H} + \bar{A}\bar{C}E\bar{H} + \bar{A}CEF + DEF + AFG\bar{H} + BEF$$

$$CAR15 = ACD\bar{F}G\bar{H} + AD\bar{E}G\bar{H} + BC\bar{D}EG\bar{H} + ABF\bar{G}\bar{H} + B\bar{D}EF + C\bar{D}EF + D\bar{E}\bar{F}GH + \bar{A}DEG\bar{H} + \bar{A}B\bar{D}G\bar{H} + \bar{A}\bar{E}G\bar{H} + DG\bar{H} + \bar{A}B\bar{D}G + \bar{D}GH + B\bar{E}\bar{F}GH + AFGH + AEGH + BEGH + BCFG + CEGH + ACGH$$

$$CAR16 = C\bar{D}\bar{F}G\bar{H} + A\bar{C}FG + ACE\bar{F}\bar{G} + \bar{A}C\bar{E}\bar{F}G + CDF\bar{G}H + ADE\bar{F}\bar{G} + BDE\bar{F}G + B\bar{C}EG + AB\bar{F}H + BCE\bar{F}\bar{G} + A\bar{B}\bar{F}G + AEGH + BE\bar{G}H + \bar{C}DEF + \bar{A}D\bar{E}F\bar{G} + \bar{A}C\bar{D}FG + BC\bar{F}G\bar{H} + A\bar{C}\bar{D}F + ACFGH + BD\bar{E}F\bar{H} + \bar{C}DFGH + \bar{A}C\bar{E}F\bar{G} + \bar{A}\bar{E}FGH + A\bar{C}\bar{E}FH$$

$$CAR17 = ACD\bar{E}G + \bar{B}C\bar{E}F + \bar{B}\bar{C}\bar{D}E + \bar{B}E\bar{F}\bar{G} + \bar{B}C\bar{D}E\bar{F}H + BCD\bar{E}\bar{F}H + AC\bar{E}GH + BC\bar{E}FG + ABEG + \bar{C}DE\bar{F}GH + AB\bar{E}H + \bar{B}CFG + \bar{B}CDFH + AC\bar{E}F + B\bar{E}GF + B\bar{C}\bar{E}F + AFH + B\bar{D}\bar{E}F\bar{H} + AD\bar{E}F + \bar{B}DFGH + AFG + BD\bar{E}FGH$$

7. Mod 17 Logic

AR22 = A MR22 = F
AR21 = B MR21 = G
AR20 = C MR20 = H
AR19 = D MR19 = J
AR18 = E MR18 = K

$$CAR18 = BCDEJ\bar{K} + BCEH\bar{K} + BGJ\bar{K} + BCDH\bar{K} + BDEHJ\bar{K} + BEG\bar{K} + BDG\bar{K} + BCHJ\bar{K} + CDEGH\bar{K} + CGHJ\bar{K} + AJ\bar{K} + CDGH\bar{K} + BCG\bar{K} + BF + DF + CF + EF + AH\bar{K} + AF + AG\bar{K} + B\bar{C}\bar{D}G\bar{K} + \bar{A}B\bar{C}D\bar{K} + \bar{A}B\bar{C}H\bar{K} + \bar{A}B\bar{G}K + \bar{A}B\bar{D}GH\bar{K} + \bar{A}B\bar{C}J\bar{K} + \bar{A}\bar{C}G\bar{H}K + B\bar{D}G\bar{H}K + \bar{A}BH\bar{J}K + \bar{A}\bar{C}G\bar{J}K$$

7. Mod 17 Logic (Continued)

$$CAR19 = \bar{B}CD\bar{E}H\bar{J}\bar{K} + BC\bar{E}G\bar{J}\bar{K} + BD\bar{E}G\bar{J}\bar{K} + B\bar{E}GH\bar{J}\bar{K} + \bar{A}B\bar{C}J\bar{K} + \bar{A}B\bar{H}J\bar{K} + \bar{A}B\bar{G}J\bar{K} + \bar{A}G\bar{H}J\bar{K} + \bar{A}\bar{C}G\bar{J}\bar{K} + BCDEJ + BCHJK + BDHJK + CGHJK + CDGJK + BGJK + DGHJK + \bar{C}E\bar{G}\bar{J}\bar{K} + B\bar{C}E\bar{J}\bar{K} + \bar{B}EH\bar{J}\bar{K} + B\bar{G}H\bar{J}\bar{K} + \bar{B}E\bar{G}\bar{J}\bar{K} + D\bar{E}F\bar{H} + A\bar{E}F\bar{H} + A\bar{E}F\bar{H} + C\bar{E}F\bar{H} + EJ\bar{K} + \bar{E}JK$$

$$CAR20 = D\bar{F}\bar{H}JK + C\bar{D}GHJ + \bar{B}E\bar{H}JK + DE\bar{H}J\bar{K} + \bar{B}D\bar{H}J + D\bar{G}\bar{H}J + \bar{D}HJ\bar{K} + BC\bar{E}HJ\bar{K} + C\bar{E}GHJ\bar{K} + BC\bar{D}H\bar{K} + B\bar{E}GH\bar{J}\bar{K} + EGHJK + \bar{D}E\bar{H}K + B\bar{D}GHK + BCHJ\bar{K} + \bar{B}DE\bar{H}K + A\bar{E}F + DEG\bar{H}K + \bar{D}HJK + CGHJK + BGHJK + C\bar{D}E\bar{F} + B\bar{D}\bar{E}F + \bar{A}BHJ\bar{K} + \bar{A}DH\bar{J}\bar{K} + AG\bar{H}J\bar{K} + BC\bar{D}EG\bar{H}J\bar{K} + \bar{A}G\bar{H}J\bar{K} + EHJ\bar{K} + \bar{E}HJK$$

$$CAR21 = B\bar{C}\bar{D}G + C\bar{D}GJ + \bar{A}BC\bar{E}G\bar{K} + C\bar{G}H + \bar{D}G\bar{H}J\bar{K} + D\bar{G}HJ + C\bar{D}GH\bar{K} + \bar{C}GHJ + B\bar{E}GH\bar{K} + C\bar{D}\bar{E}GK + B\bar{D}G\bar{H}K + \bar{D}\bar{E}G\bar{H}K + CEGJK + E\bar{G}HJK + \bar{A}\bar{C}GJ\bar{K} + B\bar{C}GJ + CDE\bar{G}K + DEGHK + \bar{A}\bar{C}EGJ + \bar{D}GHJK + BGHJ + \bar{E}GHJK + B\bar{C}\bar{D}\bar{E}G + A\bar{E}F + B\bar{D}\bar{E}GJ\bar{K} + \bar{A}GHJK$$

$$CAR22 = \bar{B}C\bar{D}EGH\bar{J}K + B\bar{C}\bar{D}E\bar{G}HJK + \bar{B}\bar{C}\bar{D}EGHJK + BC\bar{D}\bar{E}G\bar{H}J\bar{K} + BCD\bar{E}\bar{G}H\bar{J}K + B\bar{C}\bar{D}E\bar{G}HJ\bar{K} + BC\bar{D}\bar{E}G\bar{H}J\bar{K} + B\bar{C}\bar{D}\bar{E}G + B\bar{C}\bar{D}E + GH\bar{J}\bar{K} + \bar{B}\bar{C}DEGH\bar{J}K + \bar{B}CDEG\bar{H}J\bar{K} + B\bar{C}\bar{D}EG\bar{H}J\bar{K} + BCDEG\bar{H}J\bar{K} + \bar{B}\bar{C}D\bar{E}GHJ\bar{K} + AH + AG + AJ + AK$$

8. Mod 19 Logic

AR27 = A MR27 = F
AR26 = B MR26 = G
AR25 = C MR25 = H
AR24 = D MR24 = J
AR23 = E MR23 = K

$$CAR23 = \bar{A}B\bar{C}\bar{D}K + \bar{A}\bar{B}\bar{C}FK + \bar{A}\bar{C}F\bar{G}K + \bar{A}B\bar{D}\bar{F}JK + \bar{A}B\bar{F}\bar{H}K + BCG\bar{K} + BCEHJ\bar{K} + CDGHJ\bar{K} + BCDHJ\bar{K} + BDGJ\bar{K} + \bar{B}CDEH\bar{J}K + \bar{A}BF\bar{G}K + \bar{A}FG\bar{H}K + \bar{D}FGHJK + B\bar{D}GHJK + \bar{A}CDF\bar{H}JK + BDEG\bar{K} + CDEGH\bar{K} + BEGJ\bar{K} + BGH\bar{K} + ADJ\bar{K} + AG\bar{K} + AH\bar{K} + CEGHJ\bar{K} + CF\bar{K} + BF\bar{K} + AF\bar{K} + DEF\bar{K} + DFJ\bar{K} + EFJ + AEJ\bar{K}$$

$$CAR24 = \bar{A}B\bar{E}\bar{H}J + \bar{A}B\bar{C}\bar{E}J + \bar{A}B\bar{C}\bar{F}J\bar{K} + \bar{A}B\bar{F}GJK + \bar{A}\bar{C}\bar{F}GJ\bar{K} + \bar{A}\bar{F}G\bar{H}J\bar{K} + \bar{A}B\bar{F}\bar{H}J\bar{K} + \bar{A}B\bar{E}\bar{G}J + \bar{A}\bar{C}\bar{E}GJ + \bar{A}\bar{E}G\bar{H}J + BCDH\bar{J}K + BDGJK + ADJ\bar{K} + BCGJ\bar{K} + CDGH\bar{J}K + AGJK + BGHJK + AGHJK + CFGK + DFK + BFK + AFK + AEG\bar{J} + BDEGJ + BCEG\bar{J} + CDEGH\bar{J} + AEH\bar{J} + BCDEH\bar{J} + AEF\bar{J} + BEF\bar{J} + CEF\bar{J} + DEF\bar{J} + \bar{E}J\bar{K} + EJK$$

$$CAR25 = \bar{A}B\bar{D}\bar{F}H\bar{J} + \bar{A}D\bar{F}\bar{G}HJ + BC\bar{D}\bar{E}G\bar{H}\bar{K} + \bar{A}B\bar{D}H\bar{K} + AD\bar{E}GH\bar{K} + ADHJ + BC\bar{G}HJ + \bar{A}B\bar{H}J\bar{K} + CDGHJ + \bar{D}EHJ\bar{K} + BDGHJ + BD\bar{E}\bar{G}H + \bar{A}GHJ\bar{K} + \bar{A}B\bar{E}H\bar{J}K + AHJK + CGHJK + DE\bar{F}G\bar{H}K + BC\bar{D}GH\bar{J} + B\bar{D}F\bar{G}J + \bar{A}E\bar{G}H\bar{J}K + \bar{A}B\bar{D}H\bar{J} + \bar{D}E\bar{H}\bar{J}K + \bar{B}DEF\bar{H}K + BGHJK + BD\bar{G}H\bar{J} + A\bar{E}\bar{D}G\bar{H}\bar{J} + BDEGH + CDGHK + C\bar{D}F\bar{J} + BC\bar{G}H\bar{J} + AD\bar{F}\bar{J} + \bar{A}G\bar{H}J\bar{K} + BC\bar{E}G\bar{H}J\bar{K} + B\bar{E}FJ\bar{K} + AD\bar{E}F + B\bar{D}\bar{E}F + C\bar{E}FJ\bar{K} + AFJ\bar{K} + C\bar{D}\bar{E}F$$

8. Mod 19 Logic (continued)

$$CAR26 = B\bar{C}\bar{D}F\bar{J} + A\bar{D}F\bar{J} + C\bar{G}H + AF\bar{J}\bar{K} + CDF\bar{G}J + \bar{A}D\bar{G}HJ + B\bar{C}GJ + \bar{A}D\bar{G}H\bar{K} + AGH + \bar{A}\bar{C}G\bar{H} + B\bar{G}H + ADGJ + \bar{C}\bar{D}GH\bar{K} + B\bar{D}\bar{E}G\bar{K} + CDEF\bar{G}K + B\bar{D}G\bar{J} + \bar{A}B\bar{C}F\bar{G}J + DE\bar{G}H\bar{K} + ADGK + \bar{A}D\bar{E}G\bar{H} + \bar{A}\bar{E}G\bar{H}\bar{J} + \bar{A}\bar{C}EG\bar{J} + \bar{A}G\bar{H}J\bar{K} + B\bar{E}GJ\bar{K} + B\bar{C}\bar{E}FJ\bar{K} + B\bar{C}\bar{D}\bar{E}F + AEGJ + \bar{C}D\bar{E}GJK + CE\bar{G}JK + \bar{A}EGHJK$$

$$CAR27 = BCD\bar{F}\bar{G}\bar{H}J + \bar{B}CDG\bar{H}J + B\bar{C}D\bar{G}HJ + B\bar{C}DG\bar{H}J + B\bar{C}\bar{D}\bar{E}G\bar{H}\bar{K} + B\bar{C}\bar{D}G\bar{H}\bar{J} + B\bar{C}\bar{E}G\bar{H}J\bar{K} + B\bar{C}\bar{E}G\bar{H}J\bar{K} + \bar{B}\bar{C}\bar{D}\bar{E}F + B\bar{C}\bar{D}FJ + BCD\bar{E}G\bar{H}\bar{K} + B\bar{C}\bar{D}\bar{E}G\bar{H}\bar{K} + BCDGHJK + BCDG\bar{H}\bar{J} + \bar{B}CDGHJ + B\bar{C}\bar{E}FJ\bar{K} + \bar{B}CDEG\bar{H}K + BCDE\bar{F}\bar{G}H\bar{K} + B\bar{C}D\bar{E}GHK + \bar{B}\bar{C}DEGHK + B\bar{C}E\bar{G}HJK + BCEG\bar{H}JK + \bar{B}CEGJ\bar{K} + AD\bar{F}K + AD\bar{F}J + AG + AH + AE\bar{F}J + AJK$$

9. Mod 23 Logic

AR23 = A MR32 = F
AR31 = B MR31 = G
AR30 = C MR30 = H
AR29 = D MR29 = J
AR28 = E MR28 = K

$$CAR28 = \bar{A}B\bar{F}K + \bar{A}\bar{C}F\bar{H}K + \bar{A}B\bar{C}\bar{D}K + \bar{A}B\bar{C}HK + \bar{A}F\bar{G}K + \bar{C}F\bar{G}\bar{H}K + \bar{A}\bar{C}\bar{D}FK\bar{J} + \bar{C}\bar{D}F\bar{G}JK + B\bar{D}F\bar{H}JK + \bar{D}F\bar{G}\bar{H}JK + \bar{A}B\bar{D}\bar{H}JK + AC\bar{H}K + ACDF\bar{J}K + BF\bar{K} + BCDGJ\bar{K} + ADHJ\bar{K} + BCEGJ\bar{K} + BDGHJ\bar{K} + BDEGH\bar{K} + BCGH\bar{K} + ADEH\bar{K} + AG\bar{K} + AF\bar{K} + BEGHJ\bar{K} + CDEFJ\bar{K} + CDFJ\bar{K} + CEFJ\bar{K} + CFH\bar{K} + DFHJ\bar{K} + DEFH\bar{K} + AEHJ\bar{K} + EFHJ$$

$$CAR29 = \bar{A}B\bar{C}\bar{E}J + \bar{A}B\bar{C}HJ\bar{K} + \bar{A}B\bar{E}\bar{F}J + \bar{C}\bar{E}F\bar{G}HJ + \bar{A}\bar{C}\bar{E}F\bar{H}J + \bar{A}\bar{E}GH\bar{K} + BDEGH\bar{J} + CDF\bar{J}K + ACD\bar{J}K + ACH\bar{J}K + BDGHJK + BCDGJ\bar{K} + BCGH\bar{J}K + BF\bar{J}K + CFHK + DFHK + AF\bar{J}K + ADH\bar{J}K + AG\bar{J}K + AEG\bar{J} + BCDEG\bar{J} + BCEGH\bar{J} + \bar{C}F\bar{G}HJ\bar{K} + \bar{A}\bar{C}FHJ\bar{K} + \bar{A}F\bar{G}HJ\bar{K} + \bar{A}E\bar{F}GJ + ACEH\bar{J} + ADEH\bar{J} + CDEF\bar{J} + AEF\bar{J} + BEF\bar{J} + CEFH\bar{J} + DEFH\bar{J} + EJ\bar{K} + EJ\bar{K}$$

$$CAR30 = D\bar{H}J + \bar{A}D\bar{F}GH\bar{K} + ACE\bar{H}J + BCEG\bar{H}J + \bar{A}B\bar{D}GH\bar{K} + \bar{D}\bar{E}H\bar{K} + \bar{A}B\bar{D}\bar{E}H + AEF\bar{H}J + CEF\bar{H}J + BEF\bar{H}J + BCGHJK + AC\bar{H}JK + B\bar{D}\bar{E}GH + DE\bar{H}K + \bar{D}HJ + ACD\bar{H}K + \bar{A}B\bar{E}F\bar{H}J + BCDG\bar{H}K + ADG\bar{H}K + \bar{A}\bar{F}G\bar{H}JK + ADEG\bar{H} + BCDEG\bar{H} + \bar{A}B\bar{F}HJ\bar{K} + BDEF\bar{H} + CDEF\bar{H} + ADEF\bar{H} + BDF\bar{H}K + CDF\bar{H}K + ADF\bar{H}K + CFKJ + AFJK + BFJK + EFJK + \bar{E}HJ\bar{K}$$

$$CAR31 = \bar{A}CD\bar{F}\bar{G}J + \bar{A}D\bar{F}\bar{G}HJ + \bar{A}C\bar{F}\bar{G}H + A\bar{C}\bar{D}\bar{E}F + A\bar{C}F\bar{G}\bar{H} + BDF\bar{G}HJ + ACEG + B\bar{C}\bar{D}\bar{E}F + \bar{A}\bar{D}G\bar{H}\bar{K} + \bar{A}B\bar{C}F\bar{H} + \bar{A}\bar{C}DGH\bar{K} + \bar{A}\bar{C}DGH\bar{J} + BG + AGHJK + ACGH + \bar{A}G\bar{H}J\bar{K} + \bar{A}\bar{C}GJK + \bar{A}\bar{C}E\bar{G}J + \bar{A}\bar{C}G\bar{H} + B\bar{C}\bar{D}F\bar{J} + AEGHJ + ACDGH + A\bar{C}\bar{D}FJ + ADGHJ + A\bar{D}G\bar{H}\bar{J} + ADFGH + ADGHK + \bar{A}DE\bar{F}G\bar{H}K + \bar{A}\bar{D}G\bar{H}\bar{J} + CD\bar{E}G\bar{H}\bar{J}K + B\bar{D}\bar{E}F\bar{H}\bar{K} + \bar{A}B\bar{E}G\bar{H}JK + B\bar{E}H\bar{J}K + \bar{A}\bar{C}DEGJ + A\bar{D}\bar{E}F\bar{H}\bar{K} + CD\bar{E}G\bar{H}\bar{J}K + \bar{A}CEF\bar{G}HJK + AF\bar{H}\bar{J}K + A\bar{C}\bar{E}FJ\bar{K} + B\bar{C}EFJ\bar{K} + B\bar{E}F\bar{H}\bar{J}K$$

9. Mod 23 Logic (continued)

$$CAR32 = AC\bar{E}\bar{F}J + BCDF\bar{G}J + B\bar{C}\bar{D}\bar{E}F + B\bar{C}G\bar{H} + BCF\bar{G}H + AC\bar{F}\bar{H} + \bar{B}CDGJ + \bar{B}C\bar{D}EGK + B\bar{C}FGH + BCDE\bar{F}\bar{G}K + \bar{B}C\bar{D}F\bar{J} + \bar{B}\bar{D}F\bar{H}\bar{J} + AC\bar{F}JK + ACDF\bar{J} + \bar{B}CGH + BDF\bar{G}HJ + AD\bar{E}F\bar{H} + A\bar{E}G\bar{H}\bar{J}K + \bar{B}DGHJ + B\bar{D}\bar{E}G\bar{H}\bar{K} + A\bar{E}\bar{F}\bar{H}J + \bar{B}CEGJK + B\bar{E}GHJK + \bar{B}DEGHK + B\bar{C}\bar{D}GJ + B\bar{C}\bar{E}\bar{F}JK + B\bar{C}\bar{E}GJ\bar{K} + B\bar{D}\bar{E}FH\bar{K} + ACD\bar{F}K + BDE\bar{F}GHK + ACD\bar{F}\bar{K} + BDE\bar{F}\bar{G}HK + AG + AD\bar{F}HK + A\bar{H}JK + BCE\bar{F}\bar{G}JK + B\bar{C}D\bar{E}GK + AD\bar{F}HJ + B\bar{D}G\bar{H}\bar{J} + B\bar{E}FHKJ$$

D) Logic Equations for Multiplication Logic

(Base 2,3,5,7,11,13,17,19,23)

The method is simlar to that for the Addition Logic; the numbers of flip-flops in register MR AND MB are analogous to those for the addition logic.

1. Mod. 2 Logic

MB1 = A MR1 = B

RMR1 = $\bar{A}$

2. Mod 3 Logic

MB3 = A MR3 = C
MB2 = B MR2 = D

CMR2 = AC + $\bar{B}$D

CMR3 = AD + $\bar{A}\bar{C}$

3. Mod 5 Logic

MB6 = A MR6 = D
MB5 = B MR5 = E
MB4 = C MR4 = F

CMR4 = $B\bar{C}D$ + BCE + AE + $\bar{C}\bar{E}F$ + $\bar{B}\bar{C}F$ + AD

CMR5 = $\bar{A}\bar{C}E$ + $BD\bar{F}$ + $AE\bar{F}$ + $B\bar{E}F$ + $BE\bar{F}$

CMR6 = AE + $\bar{C}D$ + BD + AF

6. Mod 13 Logic

MB17 = A MR17 = E
MB16 = B MR16 = F
MB15 = C MR15 = G
MB14 = D MR14 = H

$$
\begin{aligned}
CMR14 &= BCDG + AEGH + A\bar{E}\bar{F}\bar{G}\bar{H} + B\bar{D}\bar{F}\bar{G}\bar{H} + CDFG + \\
&\quad B\bar{C}\bar{D}\bar{E}FG + A\bar{C}\bar{D}FG + AD\bar{E}FJ + \bar{A}BC\bar{D}E\bar{H} + B\bar{C}D\bar{E}\bar{F}G + \\
&\quad A\bar{B}C\bar{D}EG + B\bar{C}D\bar{E}\bar{F}G + B\bar{C}EGH + ACEH + \bar{A}CD\bar{E}GH + \\
&\quad AC\bar{E}\bar{F}GH + B\bar{C}\bar{E}FGH + \bar{A}BCDFH + \bar{B}CD\bar{E}\bar{F}H + AD\bar{E}FGH + \\
&\quad ABP + A\bar{C}DFGH + BCD\bar{E}G + ABE + \bar{A}BCEFGH + \\
&\quad A\bar{C}DE\bar{F}G\bar{H} + \bar{A}BC\bar{D}H \\
CMR15 &= A\bar{C}DE\bar{F} + A\bar{B}C\bar{D}E\bar{F}G + B\bar{C}DFG + \bar{A}C\bar{D}FG + AC\bar{D}EH + \\
&\quad BC\bar{F}J + \bar{A}BE\bar{G}H + A\bar{C}D\bar{F}G + AB\bar{C}E\bar{G}H + \bar{A}BC\bar{D}E\bar{F} + \\
&\quad AC\bar{D}E\bar{H} + B\bar{C}E\bar{G}H + A\bar{B}D\bar{E}\bar{G}\bar{H} + ADF\bar{G}H + \bar{A}CDE\bar{G}H + \\
&\quad AD\bar{E}F\bar{G}H + \bar{B}DFG + \bar{A}CD\bar{E}F\bar{H} + ADEGH + BCE\bar{J} + B\bar{C}FH + \\
&\quad C\bar{E}F\bar{G}H + \bar{A}BE\bar{F}G\bar{H} + AB\bar{C}E\bar{F}\bar{G}\bar{H} + \bar{A}D\bar{E}GH + CEF \\
CMR16 &= B\bar{C}D\bar{E}GH + BCDE\bar{F}\bar{H} + \bar{A}BC\bar{F}G\bar{H} + \bar{A}B\bar{C}DJH + AC\bar{F}GH + \\
&\quad C\bar{D}E\bar{F}G + B\bar{C}D\bar{E}GH + ADEG\bar{H} + B\bar{C}DEGH + A\bar{C}DGH + \\
&\quad CDEGH + \bar{A}B\bar{C}DF + A\bar{C}DFG + \bar{B}CEF + AC\bar{E}F\bar{G}H + \\
&\quad AD\bar{E}GH + AD\bar{F}GH + \bar{B}C\bar{D}F\bar{G} + AC\bar{D}E\bar{F}H + BE\bar{F}GH + \\
&\quad CDFGH + BC\bar{E}P + AB\bar{F}GH + \bar{A}BFGH + BC\bar{E}\bar{F}H + \bar{A}CFGH + \\
&\quad A\bar{C}DF\bar{H} + AB\bar{D}F\bar{G} \\
CMR17 &= \bar{A}BCDE + ACDFGH + AC\bar{D}EGH + \bar{A}BDE\bar{F} + AB\bar{D}FGH + \\
&\quad A\bar{C}DE\bar{G} + BCDEG + A\bar{C}D\bar{E}\bar{F}H + C\bar{D}EGH + \bar{A}CDE\bar{G}H + \\
&\quad BC\bar{D}FH + BCDE\bar{H} + BC\bar{E}F\bar{G}H + BDFH + BCDEH + \bar{A}BD\bar{E}GH + \\
&\quad BC\bar{F}F\bar{G}H + \bar{A}BC\bar{E}FGH + ABF\bar{G} + \bar{A}CDEH + AC\bar{D}FGH + \\
&\quad B\bar{C}D\bar{F}GH + \bar{A}BCEH + A\bar{C}EGH + B\bar{C}D\bar{F}GH + C\bar{D}FG\bar{H} +
\end{aligned}
$$

7. Mod 17 Logic

MB22 = A MR22 = F
MB21 = B MR21 = G
MB20 = C MR20 = H
MB19 = D MR19 = J
MB18 = E MR18 = K

$$
\begin{aligned}
CMR18 &= B\bar{C}DEG\bar{H} + B\bar{C}D\bar{E}G\bar{H}J + BDG\bar{H}J\bar{K} + C\bar{D}EG\bar{H}\bar{J} + \\
&\quad BC\bar{E}GHJ + B\bar{D}E\bar{H}J\bar{K} + BC\bar{D}HJ\bar{K} + BC\bar{E}HJ\bar{K} + B\bar{E}EGJK + \\
&\quad B\bar{D}EHJK + CDGHJK + C\bar{D}\bar{E}JK + CDE\bar{G}HJ + B\bar{C}D\bar{E}GHJ\bar{K} + \\
&\quad B\bar{D}\bar{E}GHJ\bar{K} + B\bar{C}DE\bar{G}HJ + B\bar{C}D\bar{G}HJ\bar{K} + B\bar{C}DGHJ\bar{K} + \\
&\quad \emptyset DG\bar{H}J\bar{K} + B\bar{C}D\bar{E}GHJ + B\bar{C}D\bar{E}GH\bar{K} + BDE\bar{G}JK + \\
&\quad B\bar{D}EHJK + C\bar{D}EHJ + BD\bar{E}GHJK + DEGH\bar{J} + BC\bar{D}GH\bar{J}K + \\
&\quad BD\bar{E}\bar{G}JK + B\bar{C}DGHJK + B\bar{C}D\bar{E}H\bar{J} + BD\bar{E}GH\bar{J} + BCDEJK + \\
&\quad B\bar{C}\bar{E}GK + B\bar{C}D\bar{E}K + C\bar{D}\bar{E}GHJK + B\bar{D}\bar{E}GHJK + BDGHJK + \\
&\quad B\bar{C}DEG\bar{H}JK + D\bar{E}F + B\bar{E}P + AF + C\bar{E}P + \bar{E}GHJK + \\
&\quad AG + AH + AJ \\
CMR19 &= B\bar{C}DEGH + \bar{A}B\bar{C}DEJ + \bar{C}DEG\bar{H}K + CD\bar{E}GJK + C\bar{E}\bar{H}\bar{J} + \\
&\quad B\bar{D}EHJ\bar{K} + B\bar{C}\bar{D}F\bar{H}J\bar{K} + B\bar{C}D\bar{G}JK + B\bar{D}\bar{E}\bar{E}JK + C\bar{D}GH\bar{J} + \\
&\quad B\bar{C}DGHJ\bar{K} + BC\bar{E}GH + B\bar{C}\bar{D}FHJK + BC\bar{D}EGHJ + \\
&\quad C\bar{D}\bar{E}GHJK + B\bar{C}DGHJ\bar{K} + B\bar{C}GHJK + C\bar{D}E\bar{H}JK + B\bar{C}\bar{D}EJ + \\
&\quad B\bar{C}D\bar{E}GHJ + \bar{A}B\bar{E}GHJK + \bar{C}DE\bar{J}K + B\bar{C}\bar{E}GH\bar{J} + DF + \\
&\quad BC\bar{E}\bar{G}H\bar{K} + B\bar{C}DGH\bar{J}K + B\bar{C}\bar{D}\bar{E}GHJK + B\bar{C}\bar{E}GHJK + \\
&\quad C\bar{D}\bar{E}GJK + BCDGH\bar{J}K + B\bar{C}GH\bar{J}K + \bar{A}B\bar{C}\bar{E}GJK + BCDG\bar{J}K + \\
&\quad BC\bar{D}\bar{E}GH\bar{K} + BCDGHJ\bar{K} + CDEGHJK + BCD\bar{E}G\bar{H}J + \\
&\quad B\bar{C}DGHJK + \bar{A}CD\bar{E}\bar{G}JK + B\bar{C}D\bar{E}\bar{G}H\bar{K} + BE\bar{G}H\bar{J}K + \\
&\quad B\bar{D}\bar{E}GHJ\bar{K} + BD\bar{E}HJ\bar{K} + D\bar{G}H\bar{J}K + B\bar{D}\bar{E}GH\bar{J}K + B\bar{D}\bar{E}\bar{G}HJK + \\
&\quad BEGHJK + B\bar{D}\bar{E}\bar{G}H\bar{J}K
\end{aligned}
$$

4. Mod 7 Logic

MB9 = A MR9 = D
MB8 = B MR8 = E
MB7 = C MR7 = F

$$
\begin{aligned}
CMR7 &= \bar{C}\bar{D}\bar{E}P + A\bar{B}\bar{C}P + ADF + BD\bar{E}\bar{F} + A\bar{D}\bar{E}\bar{F} + B\bar{C}DF + B\bar{E}P + \\
&\quad A\bar{C}\bar{E}\bar{F} \\
CMR8 &= \bar{C}D\bar{E}\bar{F} + \bar{A}BC\bar{E} + BDE + AD\bar{E}\bar{F} + A\bar{C}D\bar{E} + B\bar{C}\bar{E}P + BD\bar{E}\bar{F} + \\
&\quad AEP \\
CMR9 &= \bar{A}CD\bar{E} + A\bar{C}DF + AD\bar{E}\bar{F} + BDF + AC\bar{D}F + B\bar{C}DE + ADE + \\
&\quad B\bar{C}DE + BD\bar{E}\bar{F}
\end{aligned}
$$

5. Mod 11 Logic

MB13 = A MR13 = E
MB12 = B MR12 = F
MB11 = C MR11 = G
MB10 = D MR10 = H

$$
\begin{aligned}
CMR10 &= \bar{A}B\bar{C}DH + BCFG\bar{H} + \bar{B}D\bar{E}\bar{F}H + BDF\bar{G}H + CFGH + \\
&\quad CFGH + BCDF + BC\bar{E}\bar{G}H + SCFGH + ADE + BCDEH + \\
&\quad DEFGH + ACF + AEH + BDFG + BDE\bar{G}H + A\bar{E}FGH + \\
&\quad BCEH + ADFGH + ADFG\bar{H} \\
CMR11 &= \bar{A}BCDG + BC\bar{E}\bar{G}H + BC\bar{F}\bar{G}H + BEFGH + \\
&\quad BDFGH + B\bar{C}D\bar{F}GH + BC\bar{F}GH + BCFGH + AEGH + \\
&\quad B\bar{C}DE\bar{H} + ADE\bar{G}H + D\bar{E}F\bar{G}H + A\bar{D}EH + A\bar{F}\bar{G}H + CEJ + \\
&\quad C\bar{E}\bar{F}GH + B\bar{D}F\bar{G}H + AD\bar{F} + ACE + ACG \\
CMR12 &= B\bar{C}DEH + AD\bar{E}\bar{G} + B\bar{C}DF + B\bar{C}D\bar{E}H + \bar{A}CD\bar{F} + \bar{A}C\bar{D}\bar{E}H + \\
&\quad \bar{A}CDE\bar{G}H + \bar{A}BC\bar{E}FGH + \bar{A}\bar{E}CH + B\bar{C}D\bar{F}GH + B\bar{E}GH + BFGH + \\
&\quad ADGH + ADE + B\bar{C}DFGH + B\bar{C}D\bar{F}GH + B\bar{E}GH + BFGH + \\
&\quad BEG + B\bar{C}DFGH + \bar{A}B\bar{D}\bar{E} + AC\bar{E}F\bar{G}\bar{H} \\
CMR13 &= BC\bar{D}F\bar{H} + AC\bar{F}J + \bar{B}CDEH + \bar{A}BC\bar{D}F\bar{G} + \bar{A}CDF\bar{G}H + \\
&\quad BDF\bar{G}H + BDFG\bar{H} + BCDFH + ADEH + AFGH + \\
&\quad CDFGH + B\bar{C}FG\bar{H} + BC\bar{E}H + BC\bar{F}\bar{G} + \bar{A}BCDE + BDEF + \\
&\quad ADE + BCEH + AEG
\end{aligned}
$$

7. Mod 17 Logic (continued)

CMR20 = $\bar{B}C\bar{D}\bar{E}\bar{G}HJ + \bar{B}DEGJ\bar{K} + \bar{B}CE\bar{G}HK + \bar{B}CDG\bar{K} + A\bar{B}C\bar{D}EH +$
$\bar{B}CGHJ\bar{K} + B\bar{D}E\bar{G}H\bar{K} + BCEGH\bar{J} + BCDE\bar{H}J\bar{K} + \bar{B}DEGHJK +$
$\bar{B}C\bar{E}GJ + CD\bar{E}HJK + \bar{B}D\bar{G}HJ\bar{K} + BCDEGHJ + \bar{B}CDEGK +$
$CDEGHJK + C\bar{D}E\bar{G}HJK + \bar{C}DP + CDP + B\bar{D}EGJ\bar{K} +$
$\bar{D}\bar{E}GHJK + \bar{C}DEHJK + BCE\bar{G}JK + BC\bar{D}G\bar{H}K + BDEGHJK +$
$B\bar{D}\bar{E}\bar{G}HJK + \bar{B}CD\bar{E}\bar{G}H\bar{K} + \bar{B}CDEGH\bar{K} + BCDGK + BDEGHJK +$
$BC\bar{E}\bar{G}JK + A\bar{B}C\bar{E}\bar{G}HJ + B\bar{C}E\bar{G}HJK + BCDEGHJ +$
$A\bar{B}DE\bar{G}HJ + BC\bar{D}\bar{E}\bar{G}HJK + \bar{C}DEGHJK + BCD\bar{E}HK +$
$A\bar{B}\bar{E}GHJK + \bar{B}D\bar{E}GHJ\bar{K} + BD\bar{E}GHJ + C\bar{D}\bar{E}GHJ + CEHJK +$
$CGHJK + B\bar{C}G\bar{H}JK + CDE\bar{G}HK + DE\bar{G}HJK + BEGHJK +$
$B\bar{E}GHJK + BC\bar{E}\bar{G}HJK + AJ$

CMR21 = $B\bar{C}D\bar{E}HJK + BCD\bar{H}JK + B\bar{C}DHJK + \bar{B}CD\bar{E}GH + BCDH\bar{J}K +$
$CDE\bar{G}JK + B\bar{C}DP + \bar{B}CDEHJK + BCD\bar{H}JK + BC\bar{D}EHJK +$
$B\bar{C}D\bar{E}GJK + B\bar{D}E\bar{G}H\bar{J}K + \bar{C}DEGHJK + CDGHJK + CDGHJK +$
$\bar{D}\bar{E}GHJK + BCE\bar{G}H\bar{K} + \bar{B}CE\bar{G}HJK + BCDEGJ + BCDEGHJ +$
$\bar{C}DEGHJK + \bar{B}D\bar{E}\bar{G}HK + \bar{B}DP + DEGH\bar{J}K + BC\bar{E}GHJK +$
$C\bar{D}\bar{E}GJK + B\bar{C}D\bar{E}G\bar{H}K + BCD\bar{E}GJK + \bar{B}CP + DBGHJK +$
$BCD\bar{E}JK + B\bar{C}H\bar{J}K + BDEGHJK + BCDEHJ + \bar{B}DEGH\bar{J}K +$
$CD\bar{E}\bar{G}HJK + A\bar{E}\bar{G}H\bar{J}K + BD\bar{E}\bar{G}HJK + BCDGHJ + CDEGHJK +$
$\bar{C}DEGHJK + \bar{B}D\bar{E}\bar{G}H\bar{J}K + BEGE\bar{J}K + B\bar{C}\bar{E}GHJK +$
$\bar{B}\bar{C}D\bar{E}\bar{J}K + BDE\bar{G}HK + BC\bar{E}GHJK + AJ + AH$

CMR22 = $B\bar{C}DEG\bar{H}J\bar{K} + B\bar{C}DE\bar{G}HJK + BCD\bar{E}GHJK + \bar{B}CDEG\bar{H}J\bar{K} +$
$BCD\bar{E}GH\bar{J}K + \bar{B}CDEGHJK + BCDEGH\bar{J}K + \bar{B}CD\bar{E}\bar{G}H\bar{J}K +$
$B\bar{C}DE\bar{G}\bar{H}J\bar{K} + BCDE\bar{G}H\bar{J}K + B\bar{C}D\bar{E}GHJK + BCDEGHJK +$
$B\bar{C}D\bar{E}\bar{G}HJK + BCDE\bar{F}\bar{G}JK + A\bar{G}\bar{H}JK + CP + BP + DP +$
$\bar{E}P$

8. Mod 19 Logic

MR27 = A MR27 = P
MR26 = B MR26 = G
MR25 = C MR25 = H
MR24 = D MR24 = J
MR23 = E MR23 = K

CMR23 = $B\bar{C}\bar{F}G\bar{H}J\bar{K} + B\bar{C}E\bar{G}\bar{H}JK + B\bar{C}\bar{D}\bar{E}F\bar{G}JK + B\bar{C}\bar{D}F\bar{G}\bar{H}JK +$
$B\bar{C}D\bar{E}\bar{H} + B\bar{C}\bar{D}E\bar{G}\bar{H} + \bar{B}CD\bar{E}\bar{H}JK + \bar{C}\bar{D}EG\bar{J}\bar{K} + B\bar{C}DG\bar{H}\bar{J}\bar{K} +$
$B\bar{C}D\bar{G}\bar{H}JK + B\bar{C}E\bar{G}\bar{H}JK + BCDE\bar{G}\bar{H}K + B\bar{C}\bar{D}E\bar{G}HJ +$
$BCD\bar{E}\bar{H}J + BC\bar{D}\bar{E}\bar{G}\bar{H}J + BCDG\bar{H}\bar{J}\bar{K} + \bar{B}CE\bar{G}\bar{H}J\bar{K} +$
$BCDF\bar{J}\bar{K} + BC\bar{D}EHJK + B\bar{C}D\bar{G}EHJK + A\bar{B}C\bar{D}E\bar{K} + B\bar{C}DGJ\bar{K} +$
$C\bar{D}\bar{E}GH\bar{J}\bar{K} + BCDEH\bar{J}\bar{K} + A\bar{D}\bar{E}\bar{G}H\bar{J}K + B\bar{C}\bar{D}E\bar{H}\bar{J}\bar{K} +$
$BC\bar{D}\bar{E}\bar{G}\bar{H}J + B\bar{C}EF\bar{J}\bar{K} + BCD\bar{E}\bar{G}H + BCFK + CDEGHJ +$
$BDFK + AF\bar{G}J\bar{K} + BDGH\bar{J}K + B\bar{D}\bar{G}\bar{H}J\bar{K} + B\bar{C}D\bar{E}\bar{G}H\bar{J} +$
$A\bar{G}H\bar{K} + A\bar{E}P + AGHK + ADG + A\bar{E}\bar{G}J\bar{K} + DGH\bar{J}\bar{K} +$
$AEGH\bar{J}K + DGH\bar{J}\bar{K} + B\bar{E}FG\bar{H}K + B\bar{C}E\bar{F}\bar{K} + B\bar{D}E\bar{F}\bar{K} +$
$A\bar{C}E\bar{F}GJK + AEGJK + CE\bar{G}HJK + CEGHJK + B\bar{E}FJ +$
$CD\bar{E}F\bar{K} + AFK + B\bar{D}\bar{E}GHJK + A\bar{E}\bar{J}JK + \bar{E}FGHJK$

Mod 19 Logic (continued — CMR24)

CMR24 = $B\bar{C}D\bar{G}HJ\bar{K} + B\bar{C}DE\bar{F}G\bar{H}K + B\bar{C}\bar{D}EHJ\bar{K} + B\bar{C}D\bar{E}G\bar{H}\bar{K} +$
$A\bar{B}C\bar{D}\bar{E}J + \bar{B}\bar{C}DGHJ\bar{K} + BCDEGJ + CD\bar{E}\bar{G}JK + BC\bar{D}HJ\bar{K} +$
$BCGHJ\bar{K} + BD\bar{E}GHJK + B\bar{C}\bar{D}EGK + \bar{B}DE\bar{H}J\bar{K} + BCD\bar{E}GJK +$
$B\bar{C}DGH\bar{J}K + BCDHJK + B\bar{C}EGHJK + B\bar{C}\bar{D}E\bar{F}GH +$
$\bar{B}CDGHJK + BCGEHJK + \bar{B}CE\bar{G}H + CDE\bar{H}JK + BCDGHJ\bar{K} +$
$B\bar{C}\bar{D}GHJK + \bar{B}CDE\bar{G}HK + BC\bar{D}EHJK + BCD\bar{E}\bar{G}HK +$
$B\bar{C}\bar{D}\bar{G}HJK + BCDEF\bar{G}JK + B\bar{E}\bar{E}FJ + CD\bar{E}\bar{E}EJK +$
$B\bar{C}\bar{E}GHJ + CD\bar{E}FJ + B\bar{D}\bar{E}HJK + DF\bar{G}HJK + ADJ +$
$CDEF\bar{G}H + DFJ + CEGHJK + CEGHJK + \bar{C}\bar{E}FHJK +$
$A\bar{B}HJK + A\bar{E}\bar{G}HJ + B\bar{C}\bar{E}GHJ + CD\bar{E}\bar{F}GHJ + BD\bar{E}HJK +$
$AEJK + A\bar{E}GK + ADJK + BCEFGHJ + DEFK + AEFK +$
$BC\bar{E}FJ + AD\bar{E}FJ + CD\bar{E}GHJK$

CMR25 = $B\bar{C}EGHJK + A\bar{B}DF\bar{G}HJK + BCDGHJK + AD\bar{E}G\bar{J}K +$
$B\bar{C}D\bar{E}JK + B\bar{C}DEGHJ + AD\bar{E}F\bar{G}JK + AE\bar{F}GHJ + B\bar{C}F\bar{G}H +$
$CD\bar{E}\bar{G}HJ\bar{K} + C\bar{D}HJK + C\bar{D}HJ\bar{K} + BCD\bar{G}JK + BC\bar{D}E\bar{G}HK +$
$\bar{B}CEGHJK + \bar{A}\bar{B}DFK + BCEGHJK + B\bar{E}GHJK + B\bar{E}GHJK +$
$\bar{A}BC\bar{D}EH + B\bar{C}D\bar{E}JJK + BDEHJK + A\bar{B}C\bar{D}E\bar{F}\bar{G}HJ + AD\bar{F}K +$
$CDEGHJ + \bar{C}DE\bar{G}HJ + CDEHJ\bar{K} + C\bar{D}EGJK + A\bar{B}\bar{E}GHJK +$
$BD\bar{E}GH\bar{K} + BCD\bar{G}JK + BC\bar{D}\bar{E}GJK + BDEGH\bar{K} + BD\bar{E}GH\bar{K} +$
$BC\bar{E}HJ\bar{K} + BCDEGHK + BC\bar{E}HJK + CD\bar{E}GHJK + A\bar{C}DEGHJ +$
$BCDEH\bar{J}K + BC\bar{D}EGHK + \bar{C}D\bar{E}GHJ + AD\bar{E}G\bar{H}JK + CDEGH\bar{J}K +$
$AEGHJ + CD\bar{E}GHJK + CFJ + AEGH\bar{J} + BDEGJ\bar{K} +$
$\bar{C}DEGH\bar{J} + \bar{C}D\bar{E}GH\bar{J} + A\bar{B}C\bar{E}GHJ + BCD\bar{E}GJK + BC\bar{E}GJK +$
$BD\bar{E}GH\bar{J}K + A\bar{B}\bar{E}GHJK + CF\bar{G}HJK + BD\bar{E}GHK + A\bar{B}CD\bar{E}F\bar{J} +$
$BCDF\bar{K} + BC\bar{D}\bar{E}FJK + B\bar{C}DFK + BDEFJK + AEFJ$

CMR26 = $B\bar{C}D\bar{E}GH + BCD\bar{E}\bar{G}JK + BCGHJ\bar{K} + BCDEHJ + BCD\bar{E}HK +$
$B\bar{C}DGHJ + BDEHJK + BD\bar{E}G\bar{H}K + CDFGHJK + BCGGHK +$
$BC\bar{E}FHK + B\bar{E}GJK + BC\bar{E}HJK + BC\bar{E}GHJ + ADH + CDEGHJK +$
$B\bar{C}DEHJ + A\bar{B}\bar{E}GHJK + BCGH\bar{J}K + A\bar{B}D\bar{E}GJ + BCEGJK +$
$BCDFJ + B\bar{C}DEG\bar{J}K + BCD\bar{E}F\bar{G}K + CDEGH\bar{J} + A\bar{C}\bar{E}G\bar{J}K +$
$AEGH + \bar{C}DGHJK + DEGHJK + AD\bar{F}GHJ + A\bar{B}C\bar{D}FJ +$
$B\bar{E}G\bar{H}K + BD\bar{E}GHJ + BCDGHJ + BD\bar{E}GHK + B\bar{C}DEGH +$
$CD\bar{E}\bar{G}HJK + AE\bar{G}HJ + A\bar{E}GHJ + B\bar{C}\bar{E}GHJ + AD\bar{G}HJK +$
$B\bar{C}D\bar{G}JK + C\bar{D}FK + A\bar{B}C\bar{E}\bar{G}H + C\bar{D}E\bar{G}HJK + BCDGHJK +$
$BCFJ + BCFJ + BD\bar{E}HJK + BC\bar{D}FK + BDEGHJK +$
$C\bar{D}\bar{E}\bar{G}HJ + BC\bar{E}FK + B\bar{C}\bar{E}GHJ + AFGHJK + DEGHJK +$
$AD\bar{E}FJK + BD\bar{E}GJK + BDE\bar{G}HJK + A\bar{E}GHJK$

CMR27 = $BCDEGHJK + B\bar{C}D\bar{E}GH\bar{K} + BCD\bar{E}\bar{G}JK + BCDEG\bar{H}JK +$
$\bar{C}DEGHJK + C\bar{D}E\bar{G}HJK + \bar{B}CDEGHJK + BCDE\bar{G}HK +$
$A\bar{B}CDEG\bar{H}\bar{J} + BC\bar{D}\bar{E}\bar{G}H\bar{J}K + BCDE\bar{G}HJK + AD\bar{G}HJ +$
$AD\bar{E}GH\bar{J}K + BC\bar{E}\bar{G}\bar{H}JK + B\bar{C}DEHJK + BCEG\bar{H}JK +$
$BDEGH\bar{J}K + B\bar{C}DEGJK + BCD\bar{E}H\bar{J}K + BCDEGHJK +$
$BD\bar{E}GH\bar{J}K + B\bar{C}D\bar{E}GH\bar{J}K + AEGHJ + CDE\bar{G}H\bar{J}K +$
$B\bar{C}DE\bar{G}HJK + AGH\bar{J}K + B\bar{C}D\bar{E}\bar{G}HJK + BCDEH\bar{J}K +$
$AD\bar{E}GHJK + BDFK + BDF\bar{K} + CFK + B\bar{C}D\bar{F} + C\bar{D}FJ\bar{K} +$
$BDFK + D\bar{E}P + B\bar{C}F\bar{K} + CFJ + AP + \bar{E}FJK$

9. Mod 23 Logic

MB32 = A MR32 = P
MB31 = B MR31 = G
MB30 = C MR30 = H
MB29 = D MR29 = J
MB28 = E MR28 = K

CMR28 = $\bar{A}\bar{E}\bar{C}\bar{D}E\bar{G}$ + $AC\bar{F}\bar{G}H\bar{K}$ + $\bar{A}\bar{B}C\bar{D}\bar{E}\bar{K}$ + $BC\bar{D}F\bar{J}\bar{K}$ + $AC\bar{F}K$ +
$C\bar{D}\bar{E}F\bar{J}\bar{K}$ + $B\bar{C}\bar{D}GJ\bar{K}$ + $B\bar{C}\bar{D}G\bar{J}\bar{K}$ + $\bar{A}\bar{B}C\bar{D}\bar{E}G\bar{H}\bar{J}$ +
$AC\bar{D}\bar{E}GHJ$ + $A\bar{D}\bar{E}G\bar{H}\bar{K}$ + $BC\bar{F}\bar{G}J\bar{K}$ + $AC\bar{D}\bar{E}H\bar{K}$ + $B\bar{C}\bar{F}\bar{G}H\bar{J}\bar{K}$ +
$\bar{A}\bar{B}C\bar{D}\bar{E}\bar{G}\bar{H}\bar{K}$ + $\bar{A}\bar{B}C\bar{D}\bar{E}F\bar{H}J$ + $B\bar{C}\bar{D}F\bar{H}\bar{J}\bar{K}$ + $B\bar{D}\bar{E}\bar{G}J\bar{K}$ +
$\bar{A}\bar{B}\bar{D}\bar{E}\bar{G}\bar{H}\bar{K}$ + $B\bar{D}\bar{G}\bar{H}J\bar{K}$ + $B\bar{C}\bar{F}JK$ + $B\bar{C}\bar{D}\bar{E}\bar{G}JK$ + $ACDGHJ$ +
$\bar{A}\bar{B}\bar{D}\bar{E}\bar{G}\bar{H}K$ + $BD\bar{E}\bar{G}J\bar{K}$ + $\bar{A}\bar{B}C\bar{D}\bar{E}\bar{F}\bar{K}$ + $AC\bar{E}\bar{G}K$ + $AC\bar{E}F\bar{K}$ +
$AD\bar{E}\bar{G}H\bar{J}$ + $B\bar{D}\bar{G}HJ\bar{K}$ + $BC\bar{D}\bar{E}\bar{F}\bar{G}\bar{J}$ + $ACEGHK$ + $BCD\bar{E}\bar{G}HJ$ +
$\bar{A}\bar{B}C\bar{D}\bar{F}\bar{G}HJK$ + $\bar{B}CDE\bar{F}\bar{H}\bar{J}$ + $BCD\bar{E}JK$ + $\bar{A}\bar{B}C\bar{E}\bar{G}K$ + $AD\bar{E}\bar{G}\bar{H}J$ +
$AD\bar{E}GH\bar{K}$ + $A\bar{C}\bar{D}\bar{E}G\bar{H}\bar{J}$ + $BCEG\bar{H}J\bar{K}$ + $B\bar{C}\bar{D}\bar{E}\bar{F}H\bar{J}$ + $B\bar{C}\bar{D}\bar{E}\bar{F}H\bar{J}\bar{K}$ +
$C\bar{D}\bar{E}\bar{G}HJK$ + $B\bar{D}\bar{E}GJ\bar{K}$ + $\bar{A}\bar{B}\bar{D}\bar{E}G\bar{H}$ + $BD\bar{E}GHJK$ +
$AC\bar{E}\bar{G}\bar{H}JK$ + $A\bar{F}\bar{G}H\bar{J}\bar{K}$ + $BC\bar{D}\bar{E}\bar{G}HJ\bar{K}$ + $AC\bar{D}E\bar{G}\bar{J}$ +
$B\bar{C}\bar{D}\bar{E}F\bar{H}\bar{J}\bar{K}$ + $\bar{A}\bar{C}\bar{D}\bar{E}F\bar{H}\bar{J}$ + $AC\bar{F}\bar{H}JK$ + $B\bar{E}\bar{F}\bar{G}\bar{H}\bar{J}K$ +
$C\bar{D}\bar{E}\bar{F}\bar{G}HJK$ + $ADE\bar{G}\bar{H}K$ + $BCD\bar{E}\bar{F}\bar{H}\bar{J}$ + $\bar{A}\bar{B}\bar{D}\bar{E}F\bar{J}\bar{K}$ +
$C\bar{E}\bar{G}HJ\bar{K}$ + $\bar{B}CDF\bar{H}\bar{K}$ + $A\bar{C}\bar{D}E\bar{F}\bar{K}$ + $BCPJ\bar{K}$ + $AD\bar{E}H\bar{J}\bar{K}$ +
$\bar{A}\bar{B}DE\bar{G}\bar{H}J$ + $BEFH\bar{J}\bar{K}$ + $B\bar{C}EGHJK$ + $ADPJK$ + $AD\bar{F}H\bar{J}\bar{K}$ +
$B\bar{E}\bar{F}\bar{G}\bar{H}K$ + $\bar{B}C\bar{D}\bar{E}\bar{F}\bar{G}\bar{H}\bar{J}$ + $DGHJK$ + $\bar{A}\bar{B}C\bar{D}F\bar{H}\bar{J}\bar{K}$ +
$CDEF\bar{H}J\bar{K}$ + $\bar{A}\bar{C}\bar{D}\bar{E}F\bar{J}\bar{K}$ + $A\bar{C}E\bar{F}\bar{H}\bar{J}\bar{K}$ + $B\bar{C}DEJ\bar{K}$ + $APHK$ +
$CF\bar{H}JK$ + $\bar{E}\bar{F}\bar{G}\bar{H}JK$ + $B\bar{C}\bar{E}F\bar{H}\bar{K}$ + $DGH\bar{J}\bar{K}$ + $A\bar{E}FHK$ +
$B\bar{E}\bar{F}HJ$ + $CF\bar{G}HJ\bar{K}$

CMR29 = $A\bar{C}\bar{D}E\bar{G}J\bar{K}$ + $\bar{A}\bar{B}C\bar{D}\bar{E}JK$ + $ACDK$ + $B\bar{C}GHJK$ + $B\bar{C}DHJK$ +
$AD\bar{E}\bar{G}\bar{H}K$ + $A\bar{D}E\bar{G}JK$ + $ACGJK$ + $\bar{A}\bar{B}CE\bar{G}H\bar{J}$ + $B\bar{D}\bar{F}\bar{G}\bar{H}JK$ +
$ACG\bar{J}\bar{K}$ + $\bar{A}B\bar{D}E\bar{F}\bar{G}\bar{H}K$ + $\bar{A}\bar{B}\bar{D}\bar{E}\bar{F}\bar{G}J$ + $BC\bar{D}F\bar{H}J$ +
$A\bar{C}\bar{D}\bar{E}\bar{G}\bar{H}K$ + $CDEGHJ$ + $\bar{A}\bar{B}C\bar{D}E\bar{F}\bar{H}K$ + $\bar{B}C\bar{G}\bar{H}JK$ +
$\bar{B}C\bar{D}\bar{E}\bar{G}HJ$ + $\bar{A}\bar{B}C\bar{D}\bar{E}\bar{F}\bar{G}\bar{H}$ + $\bar{A}\bar{C}\bar{D}\bar{E}\bar{F}\bar{G}\bar{H}\bar{J}$ +
$AD\bar{E}\bar{F}\bar{G}\bar{H}J\bar{K}$ + $B\bar{C}\bar{E}G\bar{H}\bar{K}$ + $A\bar{C}\bar{D}\bar{E}GJ\bar{K}$ + $\bar{B}C\bar{D}\bar{E}F\bar{K}$ +
$B\bar{D}\bar{E}F\bar{H}\bar{J}\bar{K}$ + $\bar{A}D\bar{E}\bar{F}\bar{H}\bar{J}K$ + $C\bar{D}\bar{E}\bar{F}GHJ\bar{K}$ + $A\bar{C}\bar{D}\bar{E}GHJ$ +
$\bar{B}C\bar{D}\bar{E}\bar{F}\bar{H}\bar{J}\bar{K}$ + $\bar{A}\bar{E}\bar{G}H\bar{J}\bar{K}$ + $A\bar{D}\bar{E}\bar{F}\bar{G}\bar{H}\bar{J}$ + $\bar{A}\bar{B}C\bar{D}E\bar{F}\bar{H}$ +
$AC\bar{D}GHJ$ + $\bar{A}B\bar{D}E\bar{F}\bar{H}JK$ + $\bar{A}\bar{E}G\bar{H}JK$ + $AD\bar{E}\bar{F}\bar{H}\bar{J}K$ +
$AD\bar{E}\bar{F}\bar{H}JK$ + $B\bar{D}\bar{E}\bar{F}\bar{H}\bar{J}K$ + $\bar{C}\bar{E}\bar{G}H\bar{J}K$ + $BFHJ\bar{K}$ + $B\bar{C}D\bar{E}\bar{G}\bar{H}\bar{K}$ +
$B\bar{C}\bar{D}E\bar{F}\bar{K}$ + $BE\bar{G}\bar{H}JK$ + $BFG\bar{H}JK$ + $\bar{A}\bar{B}\bar{C}\bar{E}\bar{G}$ + $BC\bar{D}GF\bar{K}$ +
$\bar{A}CD\bar{E}\bar{G}J\bar{K}$ + $BDEGJ\bar{K}$ + $CEG\bar{H}JK$ + $AD\bar{E}G\bar{H}\bar{J}$ +
$BD\bar{E}\bar{G}\bar{H}\bar{J}$ + $\bar{A}\bar{E}\bar{G}\bar{H}J\bar{K}$ + $B\bar{C}\bar{D}\bar{E}G\bar{H}\bar{J}$ + $AEGHJK$

9. Mod 23 Logic (continued — CMR30, etc.)

$B\bar{C}DGJK$ + $BCDG\bar{J}K$ + $\bar{A}B\bar{C}\bar{E}FJ\bar{K}$ + $A\bar{C}\bar{D}EG\bar{H}K$ + $C\bar{D}\bar{E}GHJ\bar{K}$ +
$BCD\bar{E}GH\bar{K}$ + $B\bar{C}\bar{D}EHJ$ + $AC\bar{E}GJ$ + $B\bar{C}\bar{D}E\bar{G}\bar{H}K$ + $A\bar{C}\bar{D}EGH\bar{K}$ +
$\bar{A}C\bar{D}\bar{E}\bar{F}HJ\bar{K}$ + $BC\bar{F}HJ\bar{K}$ + $\bar{A}B\bar{D}\bar{E}\bar{G}JK$ + $\bar{A}\bar{B}C\bar{D}E\bar{F}\bar{K}$ + $AD\bar{E}FJ$ +
$A\bar{C}\bar{D}\bar{E}FH$ + $AD\bar{G}HJK$ + $AD\bar{E}F\bar{G}JK$ + $\bar{A}EC\bar{D}\bar{E}FJ$ + $C\bar{D}\bar{E}\bar{F}HJ\bar{K}$ +
$A\bar{C}EGHJ\bar{K}$ + $B\bar{D}\bar{F}H\bar{J}\bar{K}$ + $ACDF$ + $\bar{A}\bar{E}C\bar{E}\bar{H}J\bar{K}$ + $B\bar{D}E\bar{F}H\bar{J}\bar{K}$ +
$BCPJK$ + $BCE\bar{F}\bar{H}J$ + $B\bar{C}\bar{D}\bar{F}\bar{H}\bar{J}\bar{K}$ + $BC\bar{D}\bar{E}H\bar{J}$ + $BC\bar{D}E\bar{H}\bar{J}K$ +
$AD\bar{E}\bar{G}\bar{H}K$ + $BDE\bar{F}\bar{H}J$ + $A\bar{D}\bar{E}FJ\bar{K}$ + $\bar{A}\bar{B}C\bar{D}\bar{E}\bar{F}G\bar{H}K$ +
$BC\bar{E}\bar{F}GH\bar{J}$ + $B\bar{C}\bar{D}E\bar{F}\bar{H}K$ + $AC\bar{D}\bar{E}\bar{G}\bar{H}\bar{J}K$ + $\bar{A}\bar{B}C\bar{D}E\bar{F}JK$ +
$A\bar{C}\bar{E}\bar{F}\bar{G}HJK$ + $\bar{A}\bar{C}\bar{D}\bar{E}\bar{F}\bar{G}HJK$ + $\bar{B}C\bar{D}E\bar{F}\bar{H}\bar{J}\bar{K}$ +
$\bar{B}\bar{C}\bar{D}\bar{E}\bar{F}\bar{G}\bar{H}\bar{J}K$ + $\bar{A}\bar{B}C\bar{D}\bar{E}\bar{F}\bar{G}\bar{H}$ + $\bar{A}\bar{B}\bar{C}\bar{D}\bar{E}\bar{F}\bar{G}HJK$ +
$\bar{B}C\bar{D}\bar{E}\bar{F}\bar{G}\bar{H}J$ + $\bar{B}\bar{C}\bar{D}\bar{E}\bar{F}\bar{G}\bar{H}\bar{J}K$ + $\bar{A}\bar{B}C\bar{D}\bar{E}\bar{F}\bar{G}\bar{H}\bar{J}$ +
$\bar{B}C\bar{D}\bar{E}\bar{F}\bar{G}HJ\bar{K}$ + $\bar{A}\bar{B}C\bar{D}\bar{E}\bar{F}\bar{G}H\bar{J}$ + $\bar{B}C\bar{D}\bar{E}\bar{F}\bar{G}\bar{H}\bar{J}K$ +
$\bar{A}\bar{B}C\bar{D}\bar{E}\bar{F}\bar{G}\bar{H}\bar{J}$ + $\bar{B}C\bar{D}\bar{E}\bar{F}\bar{G}\bar{H}J\bar{K}$ + $\bar{A}\bar{B}C\bar{D}\bar{E}\bar{F}\bar{G}\bar{H}J$ +
$\bar{A}\bar{B}C\bar{D}\bar{E}\bar{F}\bar{G}\bar{H}\bar{J}K$ + $\bar{A}\bar{B}C\bar{D}\bar{E}\bar{F}\bar{G}\bar{H}\bar{J}$

CMR30 = $A\bar{C}\bar{D}\bar{E}G$ + $B\bar{C}DE\bar{F}\bar{H}\bar{K}$ + $A\bar{C}\bar{D}EF\bar{H}\bar{J}\bar{K}$ + $A\bar{C}\bar{D}E\bar{F}\bar{G}HJK$ +
$\bar{A}\bar{B}C\bar{D}\bar{F}\bar{G}\bar{H}K$ + $\bar{A}\bar{B}C\bar{D}EGJ\bar{K}$ + $\bar{A}\bar{B}\bar{D}\bar{E}\bar{H}K$ + $AD\bar{E}\bar{F}H\bar{J}\bar{K}$ +
$A\bar{C}D\bar{E}F\bar{H}JK$ + $A\bar{C}D\bar{E}\bar{G}\bar{H}K$ + $A\bar{B}\bar{C}D\bar{F}\bar{H}\bar{K}$ + $AC\bar{H}JK$ + $B\bar{C}D\bar{G}\bar{H}JK$ +
$B\bar{C}\bar{G}HJK$ + $BCGRJK$ + $\bar{A}B\bar{C}D\bar{E}\bar{H}$ + $BC\bar{D}G\bar{H}\bar{K}$ + $AC\bar{E}F\bar{K}$ +
$A\bar{C}D\bar{E}F\bar{G}\bar{H}J$ + $B\bar{C}\bar{D}G\bar{H}\bar{J}K$ + $CD\bar{E}\bar{F}\bar{G}H\bar{K}$ + $\bar{A}\bar{E}D\bar{E}F\bar{J}\bar{K}$ +
$B\bar{C}D\bar{E}\bar{G}\bar{J}HK$ + $BCP\bar{H}\bar{J}\bar{K}$ + $B\bar{C}\bar{D}C\bar{H}\bar{J}K$ + $ACDJ$ + $\bar{A}\bar{B}\bar{D}\bar{E}\bar{F}\bar{G}J\bar{K}$ +
$CDEFH$ + $\bar{A}BCDE\bar{G}$ + $B\bar{C}\bar{D}\bar{E}F\bar{H}$ + $B\bar{C}\bar{D}E\bar{G}J\bar{K}$ + $ACDK$ +
$\bar{A}\bar{B}C\bar{D}F\bar{H}JK$ + $B\bar{D}\bar{F}\bar{G}H\bar{J}\bar{K}$ + $\bar{A}\bar{B}C\bar{D}\bar{E}\bar{E}\bar{H}J$ + $\bar{A}\bar{E}\bar{C}\bar{D}E\bar{G}J\bar{K}$ +
$BCDFH\bar{J}\bar{K}$ + $\bar{A}\bar{B}C\bar{D}E\bar{G}\bar{J}K$ + $AD\bar{E}\bar{H}\bar{J}\bar{K}$ + $BCD\bar{E}\bar{F}J$ + $A\bar{C}\bar{D}F\bar{H}\bar{J}\bar{K}$ +
$AD\bar{F}\bar{H}\bar{J}K$ + $C\bar{D}\bar{E}F\bar{H}\bar{K}$ + $B\bar{C}\bar{D}EF\bar{J}\bar{K}$ + $B\bar{D}\bar{E}\bar{F}\bar{G}\bar{H}JK$ + $BC\bar{D}\bar{G}H\bar{K}$ +
$BCD\bar{E}\bar{G}\bar{H}JK$ + $B\bar{C}\bar{D}E\bar{G}J\bar{K}$ + $A\bar{C}\bar{D}\bar{E}\bar{G}\bar{H}J\bar{K}$ + $AD\bar{E}\bar{H}J\bar{K}$ +
$B\bar{D}\bar{E}\bar{H}JK$ + $A\bar{E}D\bar{E}\bar{G}\bar{H}J$ + $BC\bar{D}\bar{E}\bar{H}\bar{J}K$ + $B\bar{D}\bar{E}\bar{G}\bar{H}JK$ + $A\bar{C}D\bar{E}F\bar{J}\bar{K}$ + $AGH\bar{J}\bar{K}$ +
$AD\bar{E}\bar{F}\bar{G}H\bar{J}$ + $BE\bar{G}\bar{H}JK$ + $B\bar{D}\bar{E}\bar{G}\bar{H}JK$ + $A\bar{C}D\bar{E}F\bar{J}\bar{K}$ + $AGH\bar{J}\bar{K}$ +

CMR32 = $BC\bar{E}\bar{H}J\bar{K}$ + $\bar{A}BCD\bar{E}FJK$ + $AC\bar{G}\bar{H}J\bar{K}$ + $AC\bar{D}\bar{E}G\bar{H}\bar{J}$ +
$A\bar{C}\bar{D}\bar{F}\bar{G}HJK$ + $\bar{A}\bar{B}C\bar{D}E\bar{G}H$ + $\bar{A}BCD\bar{G}H\bar{J}K$ + $BC\bar{D}\bar{E}\bar{G}K$ +
$\bar{A}\bar{B}CD\bar{F}\bar{H}$ + $\bar{A}C\bar{D}E\bar{F}H\bar{J}\bar{K}$ + $B\bar{C}\bar{D}\bar{E}\bar{G}H\bar{J}\bar{K}$ + $A\bar{C}\bar{E}GJ\bar{K}$ + $AC\bar{E}GH$ +
$\bar{A}BDE\bar{F}H\bar{J}\bar{K}$ + $AD\bar{G}H\bar{J}\bar{K}$ + $B\bar{C}\bar{D}\bar{G}HJK$ + $A\bar{C}D\bar{E}\bar{G}H\bar{J}K$ +
$B\bar{C}\bar{F}\bar{G}HJ\bar{K}$ + $\bar{A}\bar{E}C\bar{E}GJ\bar{K}$ + $\bar{A}\bar{B}C\bar{D}\bar{E}\bar{F}$ + $BCD\bar{E}\bar{F}H\bar{J}\bar{K}$ +
$\bar{A}C\bar{D}\bar{E}\bar{G}\bar{H}K$ + $AC\bar{D}E\bar{H}\bar{J}K$ + $\bar{A}\bar{C}\bar{D}\bar{E}\bar{G}\bar{H}\bar{J}$ + $C\bar{D}\bar{E}\bar{F}\bar{G}\bar{H}\bar{J}K$ +
$\bar{B}D\bar{E}\bar{F}\bar{H}J$ + $\bar{B}C\bar{E}F\bar{H}\bar{K}$ + $BFHJ\bar{K}$ + $AD\bar{F}J\bar{K}$ + $BE\bar{F}\bar{H}\bar{J}$ +
$BCGH\bar{J}\bar{K}$ + $DEFH\bar{J}\bar{K}$ + $AEGH\bar{J}\bar{K}$ + $DEF\bar{G}H\bar{J}\bar{K}$ + $ACFJK$

E) Summary of Logic Gates required for residue addition & multiplication

Findings for residue addition and multiplication (sections C & D) are summarized as tables A15.1 and A15.2 below.

Digit	Number of Gates	Flip-flop	Number of Gates
2	1	MSB LSB	3 3
3	6	MSB LSB	3 3
5	19	MSB LSB	6,6 7
7	30	MSB LSB	10,11 9
11	68	MSB LSB	17,17,18 16
13	89	MSB LSB	22,24,20 23
17	136	MSB LSB	20,26,30,27 33
19	166	MSB LSB	30,31,40,34 31
23	183	MSB LSB	38,42,35,35 33

Table A15.1 : Number of Gates required for Residue Addition

Digit	Number of Gates	Flip-flop	Number of Gates
2	1	MSB	1
3	4	MSB LSB	2 2
5	15	MSB LSB	4,5 6
7	25	MSB LSB	9,8 8
11	81	MSB LSB	19,21,21 20
13	111	MSB LSB	29,28,26 28
17	212	MSB LSB	19,48,51,48 46
19	276	MSB LSB	40,60,64,55 57
23	390	MSB LSB	70,77,78,80 85

Table A15.2 : Number of gates required for Residue Multiplication

From the tables, it is observed that the amojnt of logic required for arithmetic operations with residue digits becomes quite large as the base of the residue digit becomes larger. The numbers given in the tables indicate the number of AND gates which must be ORed together to implement the logic required for each digit and for each flip-flop required to store the digit.

Appendix 17 :

THE APPLICATION OF READ-ONLY-MEMORY (ROM) TO IMPLEMENT DIGITAL FILTERS

Contents :

A) Table lookup multiplication

B) Direct Form of Filter Implementation

C) Filter Function Implementation

D) Conventional ROM multiplier

E) Digital Filter based on Residue Number Systems

The Application of Read-Only-Memory (ROM) to Implement Digital Filters

In appendix 16, we saw that the amount of logic gates increased considerably with the increase of residue base. This has prompted our interest in examining the use of read-only-memory to perform multiplication in a digital filter. We shall first consider the application of ROM's [1] to implement a conventional digital filter and conclude this appendix with the indication of using ROM for "residue number system" digital filters.

A) Table lookup multiplication

To deomonstrate the replacement of a conventional multiplier circuit with a ROM lookup table, consider the generation of the product of the data word representation $\{D_m, D_{m-1}, \ldots, D_0\}$ and the filter coefficient $\{C_m, C_{m-1}, \ldots, C_0\}$. A conventional parallel multiplier logic equivalent for 8-bit data words is shown in figure A17.1(a). The data word and filter coefficients are applied to the data in terminals and through shifting and addition operations, the result is formed and appears at the output terminals. An equivalent ROM multiplier is shown in figure A17.1(b). It is programmed so that the product of data word and coefficient reside in the data word address. Thus when a data word appears at the input, the product appears at the output in the device address time. From this simple concept for lookup multiplication applied to digital filtering several factors are apparent:

1) The ROM multiplier has lost the capability for forming general products. As the filter coefficient is fixed and can be changed only by reprogramming the ROM (i.e. PROM) or by ROM replacement.

2) The filter system data structure is set by the dimension of the ROM. (e.g. If an 8 bit system is sufficient, inexpensive PROM's of 2K bits organized as 256x8 can be used. If a ten bit system is required, 1K x 10 ROM's can be utilized. For a 12 bit system, three 4K x 4 ROM's can be utilized to produced a 12 bit product. This concept can be extended till the limit of existing and availability of ROM's.)

3) For a parallel multiplier, such that shown in figure A17.1, the lookup table implementation can offer simplification in circuitry and usually in power dissipation depending upon the filter structure.

[1] In the discussion, details pertains to the 1702 or equivalent PROMs are assumed.

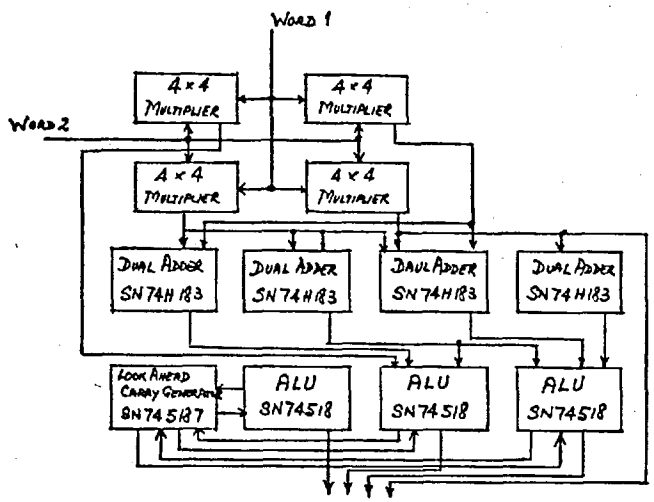


Fig. A17.1(a): Conventional 8 bit Multiplier

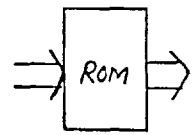


Fig. A17.1(b) : Table Lookup Multiplication

B) Direct Form of Filter Implementation

Consider the implementation of equation (1) in hardware. A block diagram of this system is shown in figure A17.2 and its direct hardware equivalent is shown in figure A17.3. Scaling is performed to minimize overflow and to produce coefficients whose magnitudes as less than unity. Although any number of bits up to 12 can can be accomodated in the filter by proper selection of ROM's or ROM arrangements. This system is a pararlel processing unit with one ROM for each filter coefficient hardwired to handle up to ten poles and ten zeros. The implementation of a transfer function can be performed in the programming time of the ROM. For this programming and transfer function scaling, a host minicomputer interfaced to the PROM programmer may be used to produce ROM's with the necessary lookup table.

$$H(z) = \frac{B_0 + B_1 Z^{-1} + B_2 Z^{-2} + \ldots + B_L Z^{-L}}{1 + C_1 Z^{-1} + C_2 Z^{-2} + \ldots : + C_m Z^{-m}}$$

$$= B_0 + \frac{Q_1 Z^{-1} + Q_2 Z^{-2} + \ldots Q_m Z^{-m}}{1 + C_1 Z^{-1} + C_2 Z^{-2} + \ldots C_m Z^{-m}} \qquad (1)$$

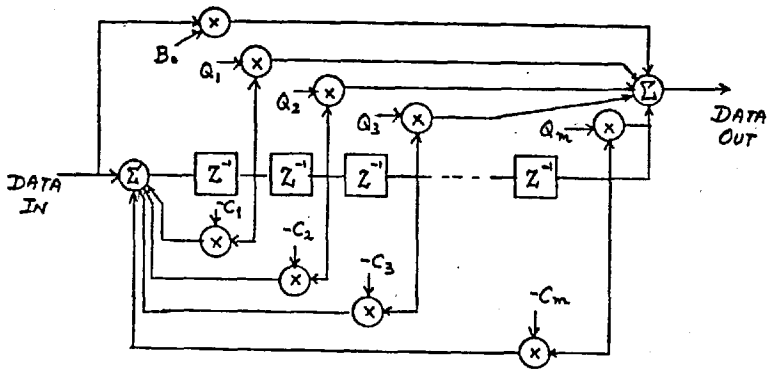


Fig. A17.2 : Direct Form Filter

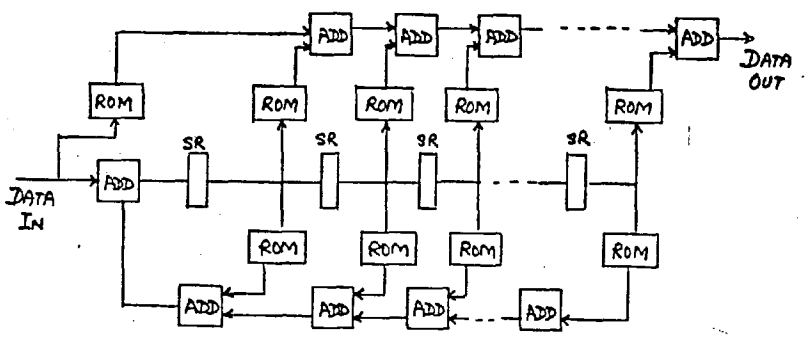


Fig. A17.3 : Direct Form Implementation

The processing speed is dependent upon the access time of the ROM and the length of the adder chain in either the upper or lower chains in Figure A17.3. TTL PROM's of 256 x 8 organization with 40ns access time are available and the add time of two 8 bit numbers is of the same order.

C) Filter Function Implementation

Peled and Liu[1] have presented a technique whereby a basic second - order section of a filter is implemented via a ROM, shift registers and acder as shown in figure A17.4. For this section the bit flow through the system is in series and can be represented as follows.

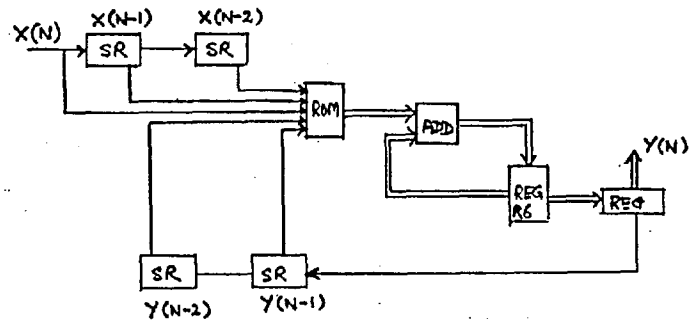


Fig. A17.4 : Filter Function Implementation

Consider the difference equation for the second order section as:

$$Y_j(N) = -C_1 Y_j(N-1) - C_2 Y_j(N-2) + B_0 X_j(N) + B_1 X_j(N-1) + B_2 X_j(N-2) + S_j[Y_B(N), Y_{B-1}(N), \ldots Y_{j+1}(N), Y_{j-1}(N), \ldots, Y_1(N)]$$

where, $j = B, B-1, \ldots, 1$

The contribution to bit j at the output is that from the filter coefficients weighted by 2^{-j} plus the spill over from all other bit's in Y and X into the jth position. This spill over is carried for B significant bits in the Register, R6, in Figure A17.4; and for each successive bit function through the ROM is aligned with the proper power of the ROM output for summing. After B bits are shifted through the system the result Y(N) is in R6 and the process on the next bit string is started.

[1] A.Peled and B. Liu, " A New Hardware Realization of Digital Filters",

This unit can also be utilized in B parallel sections producing all bit functions simultaneously with the ROM outputs being summed after the appropriate shift. Thus a parallel 12 bit system with a transfer function of 7 total poles and zeros would require 12 ROMs with 256 x 12 organization or 36 ROM's of 256 x 4 organization.

D) Conventional ROM Multiplier

While the table lookup multiplication implemented by a ROM for a filter coefficient requires N bits in and N bits outs or $N \times 2^N$ bits of ROM storage, a general lookup multiplier for 2 words of N bits in and N bits out appears to require $N \times 2^{2N}$ bits of ROM storage.

However , with appropriate partition, the actual memory needed is less because the product of certain bit positions produces a result whose contribution to the output does not appear in the first N significant bits.

Consider the multiplication of two 12 bit words represented in binary sign magnitude format. The product can be represented as

$$A \times B = A^H.B^H + A^H.B^L + B^H.A^L + A^L.B^L$$

where, H designates High portion
and L ..... Low .....

Clearly if the high and low portions are six bits each; then $A^L.B^L$ contr ibutes nothing to the first N bits in the result. Thus the ROM multiplier can be represented as in figure A17.5.

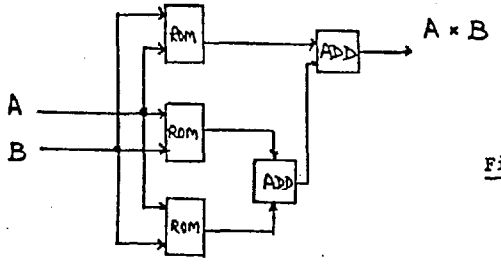


Fig. A17.5 :
General Multiplication

Thus two 12 bit numbers can be multiplied in less than 100 ns utilizing seven 2K x 4 ROM's. Two 8 bit numbers can be multiplied in 60 ns using three 512 x 8 ROM's. Although other circuits can multiply faster none are as simple as this modified table lookup multplier. This multiplier can be multiplexed in the conventional manner and used to perform all the required multiplication in a digital filter circuit.

In the next section we shall consider the application of ROM's to implement a digital filter based on residue number system.

E) Digital Filter based on Residue Number Systems

A complete discussion of the residue number system is contained in chapter 5 of this thesis. In the following discussion, we have arbitrarily chosen three moduli (viz. 4, 7, 15) for our discussion. The selection of moduli is an important consideration, as the range spanned by the filter, and therefore the accuracy of the filter, is determined by the moduli. The number of bits required internally is also determined by the moduli. Thus there are opposing considerations between precision and size, as would be expected. With the three moduli chosen, the range is M = 4 x 7 x 15 = 419 and zero. The number of binary bits required internally to represent a number is 2, 3, & 4 respectively for each moduli.

The addition operation ion RNS is very simple, and lends itself well to PROM implementation. A summation is implemented by adding two 2-bit, 3-b9t, or 4-bit numbers. The operation requiring the most bit (i.e. two 4-bit numbers) can be handled by an eight-input PROM, so a complete summation can be performed at each PROM level. Thee numbers from each modulus must be added togethr for the filter implementation, so at most 2 levels of PROM are necessary to implement the addition operation.

Multiplication by a fraction in RNS is generally a very complex operation, and is best avoided if possible. Since PROM's are being used, they can be operated in a lookup table format (as discussed in the previous sections), and the RNS multiplication operation can be avoided in the hardware. In programming the PROM's, a residue-to-decimal, a multiplication by a fraction, and a decimal-to-residue conversion may be performed; thus it is possible to avoid the need for an RNS multiplication operation.

As an illustration, let us consider implementing the filter structure depicted in figure A17.6. Adders, a multiplier, and a delay are needed to construct the desired filter. These components are implemented in two general types of sections, a multiplir and an adder. The delay is incorporated into the adder unit. The residue number system is used internally so a binary-to-residue and a residue-to-binary converter are also needed. The general adder sections is shown in figure A17.7.

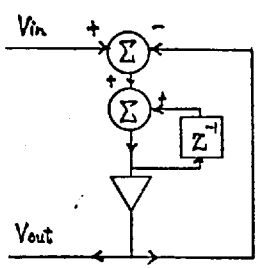


Fig. A17.6

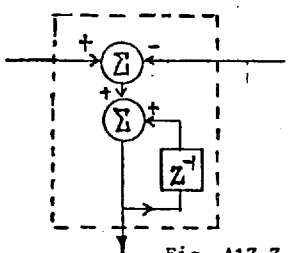


Fig. A17.7

Three such sections must be constructed, one for each modulus. The 1702 PROM is being adopted as a vehicle for our study. The PROM is a 256 x 8 memory unit, allowing an 8-bit input and an 8-bit output. The important specifications are : 1μ second address to output delay, 0.1μ second chip select delay, 0.9μ second output delay from chip select bar, 0.3μ second deselect, 1 MHz maximum repetition rate. These memories are erasable by ultra-violet light and are thus reusable.

The mod 4 addition, requiring only 6 inputs, can be performed on a single PROM. The mod 7 and Mod 15 adders require two 1702 PROMs. The delay in each adder unit is implemented using a delay flip-flop. Figures A17.8, A17.9, A17.10 show the Mod 4, Mod 7, and Mod 15 adders respectively.

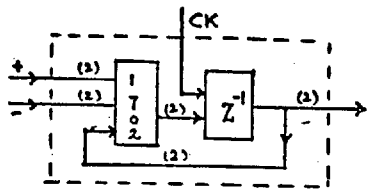


Fig. A17.8 : Mod 4 adder

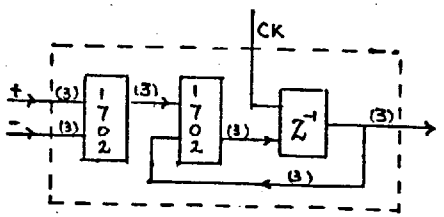


Fig. A17.9 : Mod 7 adder

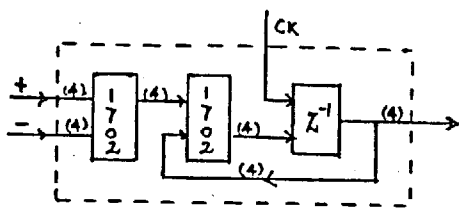


Fig. A17.10 : Mod 15 adder

CK represents a system clock with period T.

The multiplier is also implemented with 1702 PROM's. Each modulus is not considered individually in multiplication, as residue multiplication and division are complex operations. Instead of the modular operation, a direct 9-bit lookup system is used, in which all bits of all moduli an address at which the correct 9-bit result resides. The multiplier is shown in figure A17.11.

Logical ones appear at the outputs of an unselected chip. By NAND gating two chips outputs, the inverted output of the selected chip is routed to the delay flip-flop. By directing this signal into the $\overline{Q}$ input of the delay flip-flop, the outputs of a selected chip is then transmitted to the next section. The delay flip-flop also gates the signal to assure a settled input to the next section.

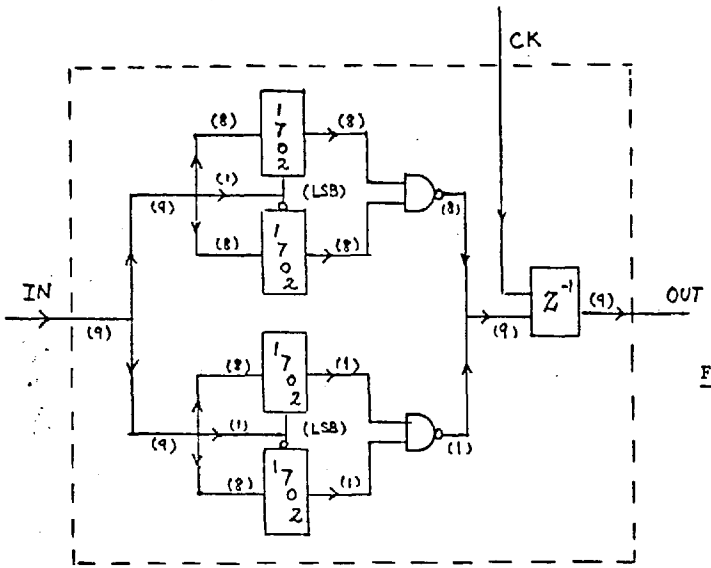


Fig. A17.11 :

A lookup multiplier for residue operation

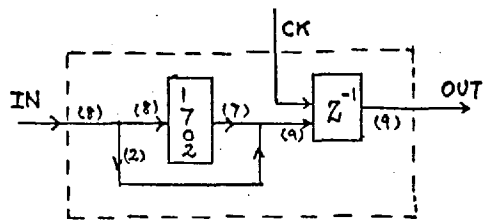


Fig. A17.12 :

A Binary-to-residue converter

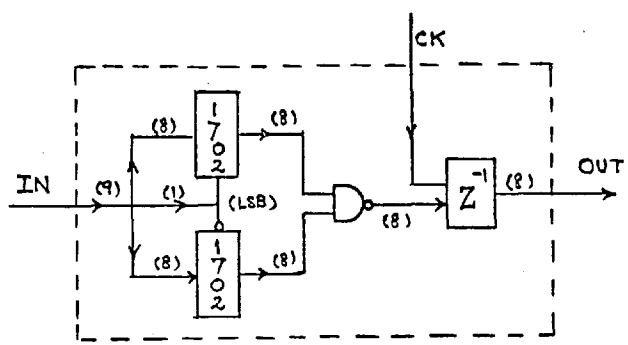


Fig. A17.13 :

A residue-to-binary converter

An eight to nine bit conversion is required of the binary-to-residue converter; this unit is also implemented by suing the 1702 PROM in lookup table format. Due to the nature of the residue number system, the mod 4 and two least significant bits of the binary number are common. Thus only seven outputs and not nine are required, and the implementation can be performed by a single PROM. This implementation of the binary-to-residue converter is shown in figure A17.12.

The final unit in this system is the residue-to-binary converter. This unit requires nine to eight bit conversion. The implementation for this is essentially the same as the multiplier; the unit is illustrated in figure A17.13.

The overall system realizing the filter structure (as depicted in figure A17.6) is shown in figure A17.14. To conclude, in this section, we have indicated a "hybrid method" of implementing a digital filter based on residue number systems.

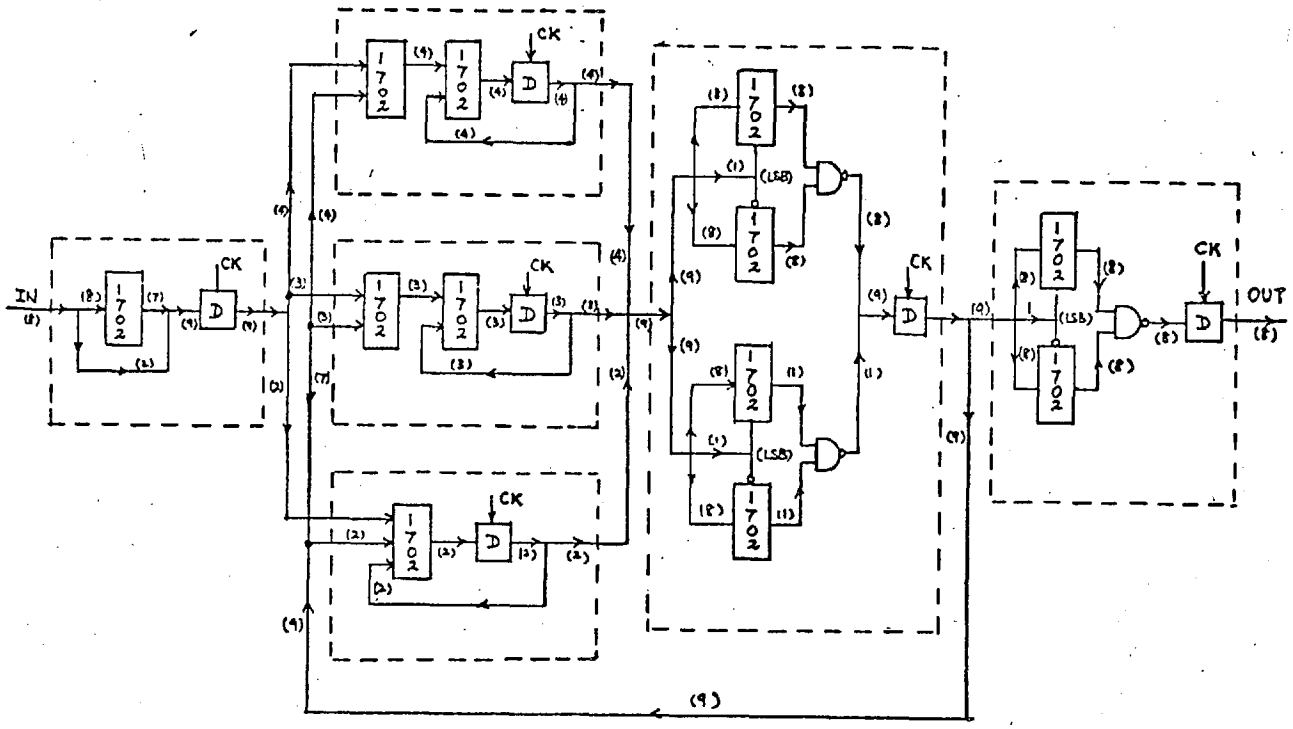


Fig. A17.14 : A "hybrid" implementation of a residue number system digital filter