AN ABSTRACT OF THE THESIS OF

KAMRAN MALIK for the degree of DOCTOR OF PHILOSOPHY

in ELECTRICAL AND COMPUTER ENGINEERING presented on 4 JUNE 1979

Title: DESIGNING A HIGH LEVEL MICROPROGRAMMING LANGUAGE

Redacted for privacy

Abstract approved:

TED LEWIS

The purpose of this research is to design a high level language (HLL) suitable for microprogramming. A top down design technique has been adopted which makes the language design process simple and accurate.

The primitive operations of a high level language for producing emulators is shown to include special purpose features specific to virtual machine implementations. A hierarchy of data types, short reliable language constructs, and control structures that minimize emulator complexity are suggested by a goal-directed, structural design methodology. In addition, structural clues generated by the language compiler assist in producing portable yet efficient horizontal microcode for partially encoded host architectures.

Furthermore, software tools in the form of a simulator-compiler combination are presented which provide features for design, development, test and eventual certification of microprograms.

DESIGNING A HIGH LEVEL MICROPROGRAMMING LANGUAGE

by

KAMRAN MALIK

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Completed 4 June 1979

Commencement June 1980

APPROVED:

Redacted for privacy

Associate Professor of Computer Science
in charge of major

Redacted for privacy

Head of Department - Electrical and Computer Engineering

Redacted for privacy

Dean of Graduate School

Date thesis is presented 4 June 1979

Typed by Cheryl DeHart for KAMRAN MALIK

ACKNOWLEDGEMENT

I would like first of all, to thank my supervisor, Dr. Ted Lewis, for his help, guidance and constant encouragement during the preparation of this dissertation. The innumerable conversations, which I have had with him over the years, have so vastly influenced my philosophical views on computers and computation, that it is an intellectual debt I shall always carry with me.

I also thank the members of my committee and, in particular, Dr. V. M. Powers for his careful reading and critical comments on this dissertation.

I cannot express the thanks I owe my parents through whom I have received everything.

The computer programs were made possible by the facilities and staff of the O.S.U. Computing Center. In addition, the work reported in this dissertation was supported by N.S.F. grant MCS 76-20710.

Last but not least, I would like to thank Cheryl DeHart for her relentless typing of this thesis.

TABLE OF CONTENTS

		PAGE
	ACKNOWLEDGEMENTS	i
	TABLE OF CONTENTS	ii
	LIST OF FIGURES	v
	LIST OF TABLES	vii
1.	INTRODUCTION	1
	1.1 EVOLUTION OF MICROPROGRAMMING	1
	1.1.1 ECONOMIC ADVANTAGE	2
	1.1.2 ARCHITECTURAL COMPATIBILITY	2
	1.1.3 DELAYED BINDING	2
	1.2 MOTIVATION	4
	1.2.1 HARDWARE DESCRIPTION LANGUAGES	6
	1.2.2 TAILORED LANGUAGES	6
	1.2.3 MACHINE INDEPENDENT LANGUAGES	7
	1.3 OBJECTIVES	12
2.	A LANGUAGE FOR MICROPROGRAMMING	13
	2.1 INTRODUCTION	13
	2.2 BASIS OF LANGUAGE	14
	2.3 LANGUAGE DESIGN	15
	2.4 DEFINITION OF VMPL	34
	2.4.1 DECLARATIONS	34
	2.4.1.1 VARIABLE DECLARATIONS	34
	2.4.1.2 INFORMATION DECLARATIONS	37
	2.4.2 ASSIGNMENT STATEMENTS, OPERATORS & EXPRESSIONS	47
	2.4.3 CONTROL CONSTRUCTS	54
	2.4.4 MISCELLANEOUS STATEMENTS	58
	2.4.5 PROGRAM STRUCTURE	58
	2.5 CONCLUSIONS	60

3. MACHINE CONSIDERATIONS 62

3.1 INTRODUCTION 62

3.2 m-COMPUTER ARCHITECTURAL FEATURES 64

3.2.1 STORAGE FACILITIES 64

3.2.2 FUNCTIONAL UNITS 68

3.2.3 DATA WIDTHS 70

3.2.4 BUSSES 72

3.2.5 MICROINSTRUCTIONS 72

3.2.6 INPUT, OUTPUT & INTERRUPT STRUCTURE 73

3.3 m-COMPUTER PRIMITIVES 76

3.4 m-COMPUTER CLASSES 80

3.4.1 CLASS 1. 81

3.4.2 CLASS 2. 81

3.4.3 CLASS 3. 82

3.4.4 CLASS 4. 83

3.4.5 CLASS 5. 84

3.4.6 CLASS 6. 85

3.5 CONCLUSIONS 85

4. PORTABILITY 86

4.1 INTRODUCTION 86

4.2 CLASSIFICATION OF TECHNIQUES 86

4.2.1 CONVERSION TECHNIQUES 86

4.2.2 INHERENTLY PORTABLE TECHNIQUES 87

4.3 APPLICATION TO MICROPROGRAM PORTABILITY 90

4.3.1 THE MALLETT-LEWIS MODEL 91

4.4 CONCLUSIONS 91

5. INTERMEDIATE MACHINE LANGUAGE 94

5.1 INTRODUCTION 94

5.2 IML FORMATS & ATTRIBUTES 94

5.3 INTERMEDIATE LANGUAGE 107

5.3.1 IISG 108

5.3.2 IESG 110

5.3.1.1 IESG FORMATS 110

5.3.1.2 IESG STATEMENTS 116

5.4 CONCLUSIONS 118

6. RESULTS 123

6.1 SIMULATOR 123

6.2 VMPL COMPILER 125

6.3 VMPL-IML ANALYSIS 126

6.3.1 COUNTING EXPERIMENTS 127

6.3.2 VARIABLE ASSIGNMENT EXPERIMENTS 132

6.4 VMPL EXTENSIONS 136

6.5 CONCLUSIONS & FUTURE WORK 138

REFERENCES 139

APPENDIX A 144

APPENDIX B 147

APPENDIX C 150

GLOSSARY 172

LIST OF FIGURES

1.1 COST OF MICROPROGRAMMED CONTROL VS. CONVENTIONAL CONTROL. 3

1.2 AN EXAMPLE PROGRAM WRITTEN IN MPL. 8

1.3 AN EXAMPLE DATA TYPE WRITTEN IN EMPL. 10

2.1 CARRY FLAG SETUP SCHEME. 19

2.1a CARRY FLAG SETUP AS A SUBROUTINE 20

2.2 OVERFLOW FLAG SETUP SCHEME. 21

2.3 DEFINITION OF D'STRUCTURES 27/28

2.4 VARIABLE ALLOCATION PROBLEM. 32

2.5 VMPL GLOBAL AND LOCAL VARIABLE DECLARATIONS. 38

2.6 VMPL GLOBAL INFORMATION DECLARATIONS. 40

2.7 WORDSIZE DECLARATION EQUIVALENTS. 41

2.8 VMPL LOCAL INFORMATION DECLARATIONS. 43

2.9 EXPECT & RETURN DECLARATION EXAMPLES. 45/46

2.10 EXAMPLES OF VMPL FIELD FACILITY. 51

2.11 PRECEDENCE RULES FOR VMPL OPERATORS. 53

2.12 ENABLING VMPL FLAGS. 55

2.13 VMPL CONTROL CONSTRUCTS. 57

2.14 MISCELLANEOUS VMPL STATEMENTS. 59

3.1 MICROPROGRAMMED IMPLEMENTATION OF HIGH LEVEL LANGUAGES. 65

3.2 OVERVIEW OF m-COMPUTER ARCHITECTURAL FEATURES. 66

3.3 HIERARCHY OF INSTRUCTIONS. 69

3.4 SOME ALU STRUCTURES. 71

3.5 NO ENCODING (a) vs. SINGLE LEVEL ENCODING (b). 74

3.6 MICROINSTRUCTION ENCODINGS. 75

3.7 A GENERAL ALU STRUCTURE. 77

4.1 HIGH LEVEL LANGUAGES AND COMPLICATION. 89

4.2 THE MALLETT-LEWIS MODEL OF A TRANSLATION SYSTEM FOR PRODUCING INHERENTLY PORTABLE MICROCODE. 92

5.1a	A SMALL PART OF A VMPL PROGRAM.	100
5.1b	THREE ADDRESS CODE FOR SOURCE PROGRAM OF Fig. 5.1a.	101
5.1c	SINGLE ADDRESS CODE FOR SOURCE PROGRAM OF Fig. 5.1a.	102
5.2	OTHERINFORMATION VALUES FOR IISG.	111
5.3	VMPL-IML EQUIVALENTS.	112
5.4	CLASS 6 IESG STATEMENTS.	120
5.5	CLASS 7 IESG OPERATIONS.	121
6.1	MICROCODE CORRECTNESS MODEL.	124
6.2	RESULTS OF COUNTING EXPERIMENT CE(2).	129
6.3a	RESULTS OF VARIABLE ASSIGNMENT EXPERIMENT ON VM1.	133
6.3b	RESULTS OF VARIABLE ASSIGNMENT EXPERIMENT ON VM2.	134
6.3c	RESULTS OF VARIABLE ASSIGNMENT EXPERIMENT ON VM3.	135

LIST OF TABLES

2.1	VMPL STANDARD FLAGS	22
2.2	ELSHOFF'S ANALYSIS OF 120 COMMERCIAL PL/I PROGRAMS.	24
2.3	STACK OPERATIONS AND THEIR VMPL DESCRIPTOR REPRESENTATIONS.	36
2.4	VMPL DIADIC OPERATORS.	48
2.5	VMPL SHIFT OPERATORS.	48
2.6	VMPL ROTATE OPERATORS.	50
5.1	INSTRUCTION STREAM COMPARISON FOR THE ASSIGNMENT STATEMENT.	96
5.2	BASIC QUANTITIES TO BE MEASURED FOR DEVELOPING SOFTWARE SCIENCE CRITERIA FOR PROGRAMS.	103
5.3	BASIC QUANTITIES FOR THE PROGRAMS OF Fig. 5.1.	104
5.4a	CALCULATED MEASURES FOR PROGRAMS OF Fig. 5.1.	105
5.4b	CALCULATED MEASURES FOR VMPL PROGRAM AND ITS EQUIVALENT OF APPENDIX C(a).	105
5.5	DECLARATIONTAG VALUES FOR IISG.	109
5.6	IML COLUMN DESIGNATIONS.	114
5.7	OPERAND MODIFIERS.	115
5.8	CLASS 1 IESG OPERATIONS.	117
5.9	CLASS 2 IESG OPERATIONS.	117
5.10	CLASS 3 IESG OPERATIONS.	119
5.11	CLASS 4 IESG OPERATIONS.	119
5.12	CLASS 5 IESG OPERATIONS.	119
6.1	RESULTS OF COUNTING EXPERIMENT CE(1).	128
6.2	RESULTS OF COUNTING EXPERIMENT CE(3).	131

DESIGNING A HIGH LEVEL MICROPROGRAMMING LANGUAGE

1. INTRODUCTION

1.1 EVOLUTION OF MICROPROGRAMMING

Microprogramming was introduced by Wilkes (1) in 1951. His paper (1) introduces the fundamental ideas which lie behind the concept of microprogramming and can be summarized in his words as:

> "....consider the control proper, that is, the part of the machine which supplies the pulses for operating the gates associated with the arithmetical and control registers. The designer of this part of a machine usually proceeds in an ad hoc manner, drawing block diagrams until he sees an arrangement which satisfies his requirements and appears to be reasonably economical. I would like to suggest a way in which the control can be made systematic, and therefore less complex.
>
> Each operation called for by an order code of the machine involves a sequence of steps which may include transfers from the store to control or arithmetical registers, or vice versa, and transfers from one register to another. Each of these steps is achieved by pulsing certain of the wires associated with the control and arithmetical registers and I will refer to it as a 'micro-operation'. Each true machine operation is thus made up of a sequence or 'micro-program' or micro-operations."

In a microprogrammed control unit a memory is used to hold a program comprised of microinstructions. The execution of this program performs the function of the control unit--fetching and interpreting machine language instructions and then activating the appropriate control lines to execute the instructions. Each instruction in this program is termed a microinstruction since the level of control exercised by each is at the gate level.

Although microprogramming received some attention during the 1950's (2), (3), it wasn't until the early 1960's that technological development made it economically feasible to be used in computer design. IBM in the mid 1960's employed microprogrammed processors in most of the models of the system 360. The reason for microprogrammed processors given by IBM (4) was:

> "....it has been used to help design a fixed instruction set capable of reaching across a compatible line of machines

in a wide range of performances in the cheapest way possible." Some general reasons for building microprogrammed processors are:

1.1.1 ECONOMIC ADVANTAGE

There is a clear economic superiority of microprogrammed control over conventional logic. One of the most widely used graphs (3) to illustrate this fact is reproduced in Fig. 1.1. Here by complexity we mean the richness of the instruction repertoire. From this curve we see that after a heavy initial investment incurred for microprogram control, the cost of implementation by microprogram control increase slowly as compared to conventional logic. This curve and the obvious cost advantage which it indicates is the leading force behind the implementation of microprogrammed control units for commercially produced computers.

1.1.2 ARCHITECTURAL COMPATIBILITY

Looking at economic advantages from a software point of view, we see that by providing a basic set of machine instructions in a whole line of new computers we can run the same software with little modifications on all the 'different' computers. As a result we get architectural compatibility achieved through the use of microprogrammed control units. The IBM 360 series, as mentioned earlier, is a good example of this concept. Furthermore, because of the changing trend of software being costly or 'hard' and hardware being cheap or 'soft', we would rather duplicate computers with different performance levels, etc. but having the same machine instructions--this being achieved via microprogramming, so that we can execute the same software on all of them.

1.1.3 DELAYED BINDING

Architectural extensions and modifications are possible when we use microprogrammed control units. This results in what is known as the 'delayed binding' of the computer to its hardware. The hardware

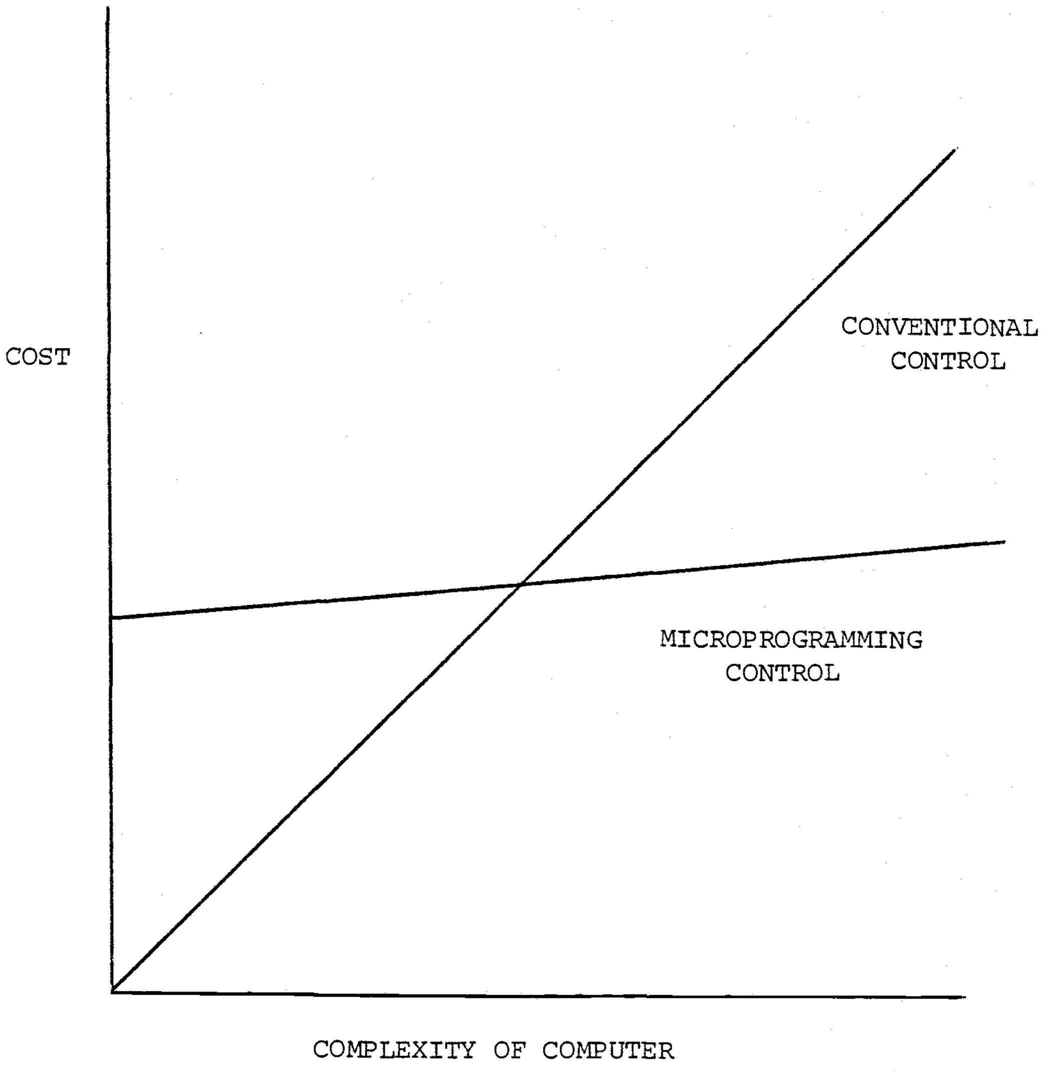


Fig. 1.1 COST OF MICROPROGRAMMED CONTROL VS CONVENTIONAL CONTROL

design primitives are known and are used to build the machine. The processing primitives may be defined at any later time and even changed or modified. This aspect of microprogramming also introduces the concept of 'amorphous' machines.

Continued technological improvements have facilitated the development of user microprogrammable computers. In these computers the control memory is of the read/write type. As such, new microprograms can be readily introduced into this memory by the user and thus user-microprogramming is possible. A large number of user-microprogrammable computers are available in the market today, with a variety of architectures (5).

1.2 MOTIVATION

The rapidly increasing acceptance of user-microprogramming because of the availability of a large number of user-microprogrammable computers, has focused increasing attention on the problems associated with writing microprograms. Traditionally, a user was provided a high level language translating system for programming. Some reasons which have been offered (6) for using high level languages are:

1) They do not require the user to be aware of such specific machine features as registers, internal representation of data, etc.
2) They offer the possibility of transferring programs from one machine to another. This means that they provide a degree of independence from a particular machine or system.
3) They allow programs to be written more easily than is possible in machine language. Furthermore, the whole software development cycle involves less time and effort and hence is efficient and cheap.
4) They allow programs to be written in problem-oriented terms. Examples of this include the ability to give symbolic names to data and the inclusion of mathematical operators and expressions in programming languages.

From the reasons enumerated above we see that a high level language translating system should also be provided for microprogramming. A microprogrammer would thus be free from machine register allocations, primitive I/O referencing, concurrency recognition, etc. In short, the microprogrammer would be able to concentrate more on the programming tasks of implementing his algorithm rather than the intricate features of the particular computer being used.

The earliest reference to a high level microprogramming language is made by Husson (3). The basic language is procedural in nature and is an adaption of the FORTRAN and PL/I languages with additional statements and declarations. The language is implemented via a multipass translator system. In the first pass the high level language statements are converted into a sequence of intermediate language statements. This intermediate language is a macro language. In the second pass the compiler 'expands' the macros to produce machine dependent microcode. Thus the compiler must have access to a library of macros which contain the microcode equivalent for each macro statement.

Several other efforts have been carried out in the direction of high level microprogramming languages since Husson's proposal. These languages can be classified into three basic groups (7) which are:

1) Hardware description.
2) Tailored.
3) Machine independent.

We review this classification scheme for two reasons:

1) This review will assist the reader in getting a better perspective of the various microprogramming languages which have been designed in the past. Thus he will see what can be done and what has been done to solve the problem of user microprogramming support and he will become aware of the richness of the attempts in several competing directions.
2) Furthermore, this review will indicate that no current microprogramming language is truly high level and machine independent. This will in part justify this research.

1.2.1 HARDWARE DESCRIPTION LANGUAGES

Hardware description languages are basically used in the design of computers but have been proposed as high level microprogramming languages (8). APL, CDL and ISP belong to this class of languages (8). Specific resource assignments are done by the microprogrammer who is also responsible for taking care of all machine details. The microprogrammer must have a complete knowledge of the host computer in order to program in these languages. Therefore, these languages are not high level.

1.2.2 TAILORED LANGUAGES

Most of the high level microprogramming languages belong to the tailored language group. The language lets the microprogrammer interact with the machine at the register transfer level. The syntax of these languages resembles the syntax of traditional high level languages like ALGOL or PL/I. Control constructs and a limited use of symbolic variables are allowed. This level of abstraction shields the microprogrammer from the trivial details of the computer's environment, but constraints him to the functions which are available on the computer since they are the only ones available in the language.

SIMPL is a tailored language proposed by Ramamoorthy (9). It is an ALGOL-like language with the conventional ALGOL reserved words. All variables are predefined and explicitly name all host machine resources.

Lloyd (10) describes a high level microprogramming language which was designed for the microprogrammed control unit (MCU) of the AN/UKY-17 Signal Processing Element. The syntax of the language resembles LSD (11) with several added built-in functions which correspond directly to testable conditions available on the MCU.

Tailored languages are designed to fit a particular machine. As such they are successful in producing microprograms for a spacific machine. This means a different high level language for each different machine. Besides being impractical, this approach also goes against the basic objectives underlying machine independent high level languages.

1.2.3 MACHINE INDEPENDENT LANGUAGES

Theoretically, any existing high level language like ALGOL, PL/I, etc. could be included in this group. However, these languages are often complex and require a large runtime support system which is difficult to maintain on most microprogrammed computers. This difficulty primarily arises from the high cost of control memory, resulting in the restriction of the amount of microcode which can be stored in it. A high level microprogramming language should be simple and not incorporate complex data structures or operations found in languages for general purpose use.

MPL, a microprogramming language designed by Eckhouse (12), was the first true attempt to define a machine independent microprogramming language. MPL is a procedure oriented language and is essentially a dialect of PL/I. There are basically six types of data items:

1) Machine registers and their parts--both real and virtual.
2) Memory--main and control.
3) Local and auxiliary storage.
4) Events which correspond to testable machine conditions.
5) Constants.
6) Variables which take on constant values.

There are three types of statements in MPL:

a) <u>DECLARATIVE STATEMENTS</u>: These statements are used to declare various data items along with their various attributes.
b) <u>ASSIGNMENT STATEMENTS</u>: These are very simple in nature, allowing at most two operands on the right hand side of the assignment operator. A provision has been made in the syntax of the language to allow concatenated registers as single operands.
c) <u>CONTROL STATEMENTS</u>: An IF and DO statement is provided. The IF statement permits branching on the value of some testable event which is previously declared.

An example of part of a program written in MPL and taken from (13) is given in Fig. 1.2. The following comments can be made about MPL.

1) Although Eckhouse claims that his language is machine

```
            INTERDATA3: PROCEDURE OPTIONS(MAIN):

        DECLARE  (R0,R1,R2,R3,R4,R5,R6,Ar,DFR,MDR) BIT (8)

                 MS  (0:32767) BIT (16)

                 MAR BIT (16)
                    MAH BIT (8) DEFINED MAR POSITION (1)
                    MAL BIT (8) DEFINED MAR POSITION (9)

                "LOCCNT" BIT (16)

                (CARRY, SINGL,CATN,TRUE,FALSE) EVENT:

        FETCH: PROCEDURE
               /*INSTRUCTION FETCH, LOC CNTR UPDATE & OP CODE DECODE*/
               MAR = R0//R1;          /*INSTRUCTION ADDRESS*/
               MDR =MS(MAR);
               R0//R1 = R0//R1+2      /*INCREMENT LOCATION COUNTER*/
               R4//R3 = MDR           /*GET OP CODE*/
               R5 = R3.RSH.3;         /*RIGHT JUSTIFY R1/X1*/
               AR = (R3.LSH.1)/1;     /*LEFT SHIFT REGISTERS R2/X2*/
                                      /*OF THE EMULATED 360 MACHINE*/
               R2,DFR = R4.RSH.4      /*INTO AR WITH LSB SET*/
               IF CARRY THEN GO TO RXFORM:
     RRFORM:   R6 = AR&1;             /*REG-REG FORMAT*/
               R4 = 0;
     DECODE:   IF SNGL/CATN THEN GO TO SUPORT:
     SUPRET:   R3 = R4&OFX:           /*MASK OP CODE*/
               AR = R3+(R3.LSH.1);    /*MULTIPLY BY 3*/
               DFR = R2;
               IF TRUE THEN GO TO ILLEG:
                           ELSE IF FALSE/CARRY THEN GO TO TROUBL:


               END/FETCH:

               END INTERDATA3:
```

Fig. 1.2 AN EXAMPLE PROGRAM WRITTEN IN MPL. (INCOMPLETE PROGRAM FRAGMENT)

independent, it is so in only a very limited sense. Since real machine registers show up as predefined variables, programs written in MPL are not transportable from machine to machine.

2) Semantic interpretation of some of the basic operations is drawn from the particular implementation. For example, shifts may or may not be circular. In other words, the actual interpretation is implementation dependent. This makes the language machine dependent.

3) MPL programs are compiled down into a simple machine language (SML). This language resembles the code of a single address computer. Arguments needed for operations must first be loaded into argument or A-registers. Results of operations are left in result or R-registers, and may be stored in other registers or memory locations. Temporary or T-registers are available for intermediate results. This process generates redundant loads and stores which produce more microcode than is necessary.

A recent addition to the family of machine independent languages is EMPL (14). This language is based on the core approach to language design (15) which allows the language to include only very basic features. Thus the user is permitted to extend the set of data structures and operations included in the core to customize the language.

Syntactically, EMPL is very much like MPL with the added statements which help define user data types. These statements are known as EXTENSION statements. An extension statement includes the new data structure, the operations on that data structure and the microperations which correspond to these operations. An example of a data type named STACK taken from (14) is shown in Fig. 1.3. Analyzing both these machine independent languages, we see that:

1) MPL specifically is machine dependent since real machine registers show up as predefined variables in the language. EMPL on the other hand becomes machine dependent because

```
TYPE STACK
    DECLARE STK (16) FIXED;
    DECLARE STKPTR FIXED;
    DECLARE VALUE FIXED;
    INITIALLY DO;
               STKPTR=0;
             END;
    PUSH:  OPERATION ACCEPTS (VALUE)
           MICROOP:  PUSH 3 0;
           IF STKPTR=16 THEN ERROR;/*OVERFLOW*/
           ELSE DO; /* PUSH VALUE */
               STKPTR=STKPTR+1;
               STK(STKPTR)=VALUE;
             END;
           END,
    POP:   OPERATION RETURNS(VALUE)
           MICROOP:  POP 3 0;
           IF STKPTR=0 THEN ERROR; /*underflow*/
           ELSE DO; /* POP VALUE */
                    VALUE=STK(STKPTR);
                    STKPTR=STKPTR-1;
               END;
           END,
    ENDTYPE:
```

Fig. 1.3 AN EXAMPLE DATA TYPE WRITTEN IN EMPL.

of the customizing feature which is primarily just a variation of the real resource associations found in MPL.

2) Furthermore, the languages are machine dependent due to the early binding to their host machine architecture.

3) The languages are designed without any consideration of the fact that code will eventually be produced for microprogrammable computers, i.e., no primitives were identified for writing code on microprogrammable computers. We believe that certain features of a special-purpose high level language for microprogramming must reflect the fact that efficient microcode is a design and implementation goal.

Because of these reasons the compilers for these languages fail to produce efficient code. In general the earlier tailored and machine independent languages can be viewed as just another set of languages. Some of them are tailored to certain machines and produce efficient code, while others are partially portable but incapable of being translated into efficient microcode for a variety of target machines. Furthermore, in both these cases we see that a general microprogramming support system has not been provided. A microprogramming support system consists of utilities that support the construction, debugging and testing of microprograms. When a microprogram has been produced from a program written in a high level language, then it should be tested for correctness. This can be achieved by providing a simulator for the microprogrammable machine. This would require a simulator for each microprogrammable machine for which microcode is to be produced. An alternate way is to compile the high level language program into an intermediate code and provide a simulator for this code. An even better situation would be to provide an interpretive system for the high level language so that the user could debug his programs in an interactive mode.

1.3 OBJECTIVES

The problem we have identified in the previous sections are:

PROBLEM 1: In order to produce low cost microprograms, a machine independent high level microprogramming language is needed, yet according to the analysis above, no such language exists.

PROBLEM 2: The languages which have been designed have not solved PROBLEM 1 effectively. This is because of two reasons:

1) The language designers have not identified the primitives for writing code on microprogrammable computers.
2) No design principles for designing high level microprogramming languages have been fabricated or adhered to. As such earlier language design is not justified in any manner.

PROBLEM 3: A general tool has not been provided. Specifically, the problem of portability has not been addressed or solved effectively.

PROBLEM 4: No testing mechanism has been provided; e.g. a topdown design cycle for firmware engineering which includes testing of microprograms etc. before implementation.

In Chapter 2 we present the formal definition of a new high level language for microprogramming. The design of this special purpose language includes constructs specifically suited for emulation, and simultaneously, the constructs that foster production of 'correct' emulators.

Chapter 3 considers the characteristics of current m-computers. The architectural features of m-computers are studied in detail and a formal classification scheme for m-computers is presented.

Chapter 4 discusses the various problems associated with producing portable software. Since our goal is to produce microcode for a variety of m-computers, we discuss the portability model which we have adopted to attain this goal.

In Chapter 5 we present a variety of intermediate language formats and their attributes. A number of experiments are done to find out the best possible format. We select the quadruple format and formally present all the intermediate language constructs.

In Chapter 6 we discuss the language translator and the intermediate language simulator. Based on well known software engineering techniques we present the results of a number of experiments which were done on the high level language and the intermediate language. A summary of the overall significance and contribution of this research is presented, along with suggestions for further study. Finally, there are several appendices which give in-depth presentations of the material discussed in Chapters 2 through 6.

A list of references and a glossary of terms follows the appendices.

2. A LANGUAGE FOR MICROPROGRAMMING

2.1 INTRODUCTION

Before we study the various m-computer (we abbreviate: m-computers for microprogrammable computers) hardware features that have such an influence on microprogramming, we have to ask ourselves:

What is the language going
to be used for

i.e., what kinds of programs are we going to write in the language. By knowing the kind of problems we are going to program in the language, we can put constructs in the language which will make this task, i.e., of programming these problems in the language, simpler. If the problems are numerical in nature, then maybe the language should be similar to FORTRAN (40). If it is business programming we have to do, then a COBOL (41) like syntax is more appropriate. Then again, maybe we want to do all these things, i.e., make a universal language more on the lines of UNCOL (42) or PL/I (43). But is all of this really practical? The answer is no, and there are two reasons for this.

1) All the languages pointed to above require a large run time support system. This requires a lot of memory. Control store in m-computers is very limited (2-8 K words) and as such not enough to support a large runtime support system.
2) Scientific, business programming, etc., are not the most common applications of microprogramming.

Historically, microprogramming has been viewed as an emulation tool or a means of extending hardware instruction sets (44). According to (5) the most common application of microprogramming is emulation. This leads us to conclude that the most frequent program written in our language for microprogrammable computers will be an emulator, i.e. a program which realizes the functional characteristices of a computer. The usual term for such a system is a virtual machine. The computer, i.e., the m-computer on which the virtual machine is realized, is known as the host and the virtual machine itself is known as the target machine. We have thus constrained the problem to writing emulators.

Problems 1 and 2 (chapter 1) can now be restated in terms of the following six objectives and constraints referred to as the basis on which the language is to be designed.

2.2 BASIS OF LANGUAGE

PREMISE 1: The language should be 'high level' and facilitate the writing of programs which realize virtual machines.

PREMISE 2: The language should have simple, reliable constructs.

PREMISE 3: The language should be compilable into compact microcode for a variety of microprogrammable computers.

PREMISE 4: The microprogrammable computers taken into consideration are those with horizontal microinstructions.

PREMISE 5: The class of target machines emulated are register oriented, word addressable machines.

The first premise is a general statement of the problem as outlined in PROBLEM 1 Chapter 1. Premise 2 is an objective of any good high level language design experiment. The reliability data available about syntactical entities of computer languages should be used in designing the language. Simplicity, of course, is a key to good design. Premise 3 is motivated by the need for fast execution of target machine emulators. Efficient microcode is defined as horizontal (parallel) microcode that takes maximal advantage of host machine concurrency. Efficiency is contradictory to generality, however, and the goal poses a problem for portability (PROBLEM 3 Chapter 1). Premise 4 is used to make the problem more general. The problem of producing code for m-computers with vertical microinstructions is a degenerate case of the more general problem of producing code for m-computers with horizontal microinstructions. Premise 5 is included to solve the following problem:

Although we have narrowed the scope of problem 1, by constraining the universality of the language, we are still covering a very wide range of targets, i.e., the sub problem is still nearly universal. Should the language have the capability of writing emulators of targets as distinct and far apart as the IBM 360/50 and the INTEL 8080 (45)? No. IBM 360/50 is a very complex machine and a very difficult one to emulate. Furthermore, things like the I/O channels and interrupt structure of IBM 360 like machines (28) will be very difficult to emulate. INTEL 8080 on the other hand is a simple machine to emulate. This leads us to modify our constraint of virtual machine realization to one of word addressable, register oriented second generation machines as the targets which we will use in writing emulators. The qualifiers 'Word addressable, register oriented' have been included only to define what we mean by second generation machines.

In the rest of this chapter we refer to the language and programs written in the language interchangeably. The language is called VMPL which is an acronym for Virtual MicroProgramming Language. A reference to machine instructions of a target machine refer to the instruction set processor (46) representation of the target machine. The premises are reference by numbers P1 (Premise 1), P2 (Premise 2), etc.

2.3 LANGUAGE DESIGN

We are now at the stage where the problem has been completely defined and are ready to proceed with the design of the language.

In the previous section we have specified a set of premises which form the basis of the language. Next, we specify eight implications or design features of the language. Each feature is justified by reference to the premises and what is known about programming language in general. In following this approach we 'derive' the language.

IMPLICATION 1: VMPL must have facilities for declaring variables, performing basic operations on the declared variables, and executing a basic set of control structures.

JUSTIFICATION 1: Since VMPL is going to be used for writing virtual machines (P1), it should be able to describe and represent storage and transformation/control entities which comprise a virtual machine. What this means is that:

1) VMPL must be able to describe all the storage resources of the target machine like registers, memory, etc.
2) VMPL should be able to describe (perform) the various functional processes of a computer (target), e.g., instruction fetch, instruction decode, operations on storage resources, etc.

This can be achieved in VMPL in the following way:

1) Declare variable types in the language which can be associated with the various storage resources of the target.
2) Provide a set of operators which can operate on the variables and a set of control structures which can be used to control the flow of the program.

IMPLICATION 2: The variables declared in VMPL should have various attributes associated with them.

JUSTIFICATION 2: Although some programming languages allow the programmer to associate sizes with the declared variables, these sizes are a multiple of a fixed size, e.g., FORTRAN's (40) single and double precision words. Furthermore, this size is implicit and dependent upon the computer on which FORTRAN is being executed, i.e., the size of a double precision FORTRAN variable on the CDC 6600 (47) is 120 bits whereas on a PDP 11/40 (48) it is only 32 bits.

In VMPL since the variables declared are used to represent registers, etc., they should have an explicit SIZE attribute associated with them. This attribute indicates the width of a register or in other words the precision of a variable explicity. This is required because of P1 and P2 which require high level machine independence. Since the language is used to write emulators for different targets (P5), each with (possibly) different word widths, we cannot have an implicit (built in) precision associated with all variables of a given program. Therefore, each individual declaration of a variable should carry the information about its size. But in case it doesn't, for programmer convenience, a default size for a variable is taken from a globally defined parameter.

To declare storage entities like memory, stack, etc., a DIMENSION attribute has to be associated with some variables. The dimension attribute then gives the number of elements contained in the array.

Target machine registers often serve both general and special purposes. As an example, a stack pointer register is incremented or decremented automatically before or after a push or a pull operation. Thus, a variable which represents the stack pointer register should have this information associated with it as a DESCRIPTOR attribute.

Finally, we describe the CLASS attribute through the following illustration. Suppose a particular host machine and target machine both have a stack. Further, suppose the target machine stack can be mapped down exactly into the host's stack. P1 and P2 forbid us to make this association, directly. In fact, as far as P2 is concerned, we are not even be aware that the host has a stack. How can the 'virtual' stack be mapped into the real stack? The only sensible or efficient way to do this is to retain the identity of the storage resource which a particular variable is representing. This is achieved by the CLASS attribute of a variable. Thus, the compiler retains the identity of the target resources and may (if possible) associate a certain variable with a certain resource on the host. The target memory is an obvious and trivial example of this attribute. Target memory should be (has to be) mapped into the host memory for a successful emulation of the target.

IMPLICATION 3: The language should have a provision to declare a standard set of condition code flags. Furthermore, there should be a mechanism to indicate directly which flags are to be affected in a particular operation. For example, an action like an assignment statement in the language must be able to affect the condition codes.

JUSTIFICATION 3: Condition code flags are an important part of a computer. The setting/clearing of these flags is a function of the various machine instructions of a computer. A number of approaches can be adopted to represent the target flags in the emulator program.

The flags can be declared as variables. Each time a flag is used, i.e., set or cleared, emulation code has to be written corresponding to the usage of the flag. Consider the example of a carry flag used during the emulation of an addition instruction Fig. 2.1. The addition operation is followed by code which emulates carry generation. A result of zero or one is placed in the carry flag. Adapting this approach for a flag like overflow becomes even more complex Fig. 2.2. This approach is thus complex and costly, producing a lot of microcode contradictory to P3.

An alternate procedure is to implement flag processing code as a sub-procedure which is called when needed Fig. 2.1a. This approach has the cost of subroutine linkage and parameter passing associated with it. Host machines usually lack subroutining facilities and as such trying to create this environment is costly and impractical. Parameter passing is specially costly because of the allocation/deallocation scheme used for assigning VMPL variables to host machine registers discussed later on.

Both approaches neglect the fact that the host may provide a set of target flags that can be enabled directly. All that is required then, is to declare the flags using some standard (reserved) names. Then when an operation happens, a way should be provided to indicate which flag is to be affected by the operation. The exact way in which the desired results (on the flags) are obtained, is left to the compiler until the point when the compiler knows the machine for which microcode is being produced.

Four standard flags have been provided for in VMPL Table 2.1. Lunde (49) suggests that a limited number of conditions are sufficient for an instruction set processor. Therefore, the four standard flags used in VMPL appear to be adequate.

ADDITION OF	:	SRC1, SRC2
RESULT TO BE STORED	:	DEST.
CARRY FLAG	:	CARRY.
CODE PRODUCED	:	

```
BEGIN
  DEST = SRC1 + SRC2  ;
  IF DEST < SRC1* THEN   CARRY = 1
  ELSE  CARRY = 0;
END
```

*NOTE: IN CASE DEST IS DISTINCT FROM BOTH SRC1 AND SRC2 THEN DEST CAN BE COMPARED AGAINST EITHER ONE, OTHERWISE DEST HAS TO BE COMPARED AGAINST THE 'SOURCE' WHICH IS DISTINCT FROM IT.

Fig. 2.1 CARRY FLAG SET-UP SCHEME

```
PROCEDURE   CARRY (SRC, DEST);
  BEGIN
    IF DEST < SRC1 THEN CARRY = 1
    ELSE CARRY = 0;
  END:


PROCEDURE  MAIN;
  BEGIN
  — —
  — —
  — —

  DEST = SRC1 + SRC2;
  CARRY (SRC1, DEST);          /*THIS INVOKES THE CARRY
                                         PROCEDURE*/
  — —

  END
```

Fig. 2.1a CARRY FLAG SET-UP AS A SUBROUTINE

ADDITION OF	:	SRC1, SRC2
RESULT TO BE STORED	:	DEST.
OVERFLOW FLAG	:	OVFLAG
FUNCTION AVAILABLE	:	MSB-GETS THE MOST SIGNIFICANT BIT OF ITS PARAMETER, i.e., THE SIGN BIT

CODE PRODUCED :

```
BEGIN
  SIGN1 = MSB(SRC1),
  SIGN2 = MST(SRC2);
  DEST = SRC1 + SRC2;
  IF ((SIGN1=SIGN2) ∧ (SIGN1=MSB(DEST)))
  THEN OVFLAG = 1
  ELSE OVFLAG = 0;
END
```

Fig. 2.2 OVERFLOW FLAG SET-UP SCHEME

TABLE 2.1 VMPL STANDARD FLAGS

FLAG	VMPL NAME
C	CARRY
O	OVERFLOW
N	NEGATIVE (SIGN)
Z	ZERO

IMPLICATION 4: The language should support very simple assignment statements using infix notation. It should allow only one variable on the left hand side and at most two operands and a diadic operator on the right hand side. However, the operands on the right hand side can have one of a number of monadic operators associated with each of them.

JUSTIFICATION 4: A study carried out by Elshoff (50) on 120 commercial PL/I programs found that assignment statements account for 41.2% of all program statements Table 2.2. In this same study it is revealed that 98% of all expressions (right hand side of assignment statements) have zero or one operator (excluding the assignment operator). Another study of 440 Fortran programs (51) produced similar results: 60% of all assignment statements contained no operator other than the assignment (=) operator.

Based on this data, VMPL is designed to support very simple assignment statements. VMPL allows only one variable on the left hand side, and at most two operands and one diadic operator on the right hand side of the assignment operator.

There can be two objections to using the above mentioned data as the basis of our justification. The first one is that the data was gathered from the pre-structured program era and as such the sample programs are 'bad'. Secondly, some of the data comes from business programs rather than emulators.

The first objection is overruled simply because no other data exists. Until data is collected for emulators, we are obliged to use what is known. The second objection is overruled on the basis that data collected for business and scientific applications seems to agree, even for diverse applications and languages such as PL/I and FORTRAN. This indicates that there is something consistent about the use of assignment statements in at least two samples. We assume that our results will be the same.

The next question deals with the number and type of operators that should go into an emulator-writing language. An instruction set processor has a large number of operators. Allowing a large number of operators

TABLE 2.2 ELSHOFF'S ANALYSIS OF 120 COMMERCIAL PL/I PROGRAMS

STATEMENT	PERCENTAGE
ASSIGNMENT	41.2
IF	17.8
GOTO	11.7
END	7.5
DO	7.2
DECLARE	6.3
WRITE	2.6
CALL	2.0
READ	0.5
PUT	0.4
NULL	0.5
PROCEDURE	0.2
PREPROCESSOR	0.1
OTHER	2.0

in the language will increase its complexity and affect reliability (P2). An alternate is to include a primitive set of operators only. There are two different considerations for selecting these operators. One is to use the data available on frequency of usage of various operators in programs and the effect they have on the reliability of the language. The other consideration is to determine which operators are best for the language based on their similarity with the instruction set processor of target machines to be emulated. The 'rotate' and 'shift' instructions, for example, are common target machine instructions.

Although relational operators are present in VMPL they are not used in assignment statement expressions but instead limited to boolean expressions. Gannon (52) shows that this makes expressions more reliable and reduces errors in assignment statements by 20%. Similarly, limiting the assignment operator to assignment statements and not allowing it in expressions reduces errors by 20% (52). It is also found (52) that the 'traditional' infix precedence expression evaluation technique is 5% more reliable than 'APL' (52) like expression evaluation technique and much more readable.

The model of an assignment statement based on Elshoff's and Gannon's data-single variable on the left and two operands and a diadic operator on the right of the assignment operator-is an extremely simple and reliable one and as indicated in the implication above is the basic one adopted by VMPL assignment statements. However, we augment this model based on the characteristics of host ALU's.

Since masking and shifting capabilities are available on host computers, VMPL expressions are modified to reflect these resources. We handle these 'side-effect' operators by a number of 'monadic' operators. These operators are 'shift', 'rotate', 'mask', 'field extraction', 'indirection' and 'register concatenate'. Each of the two operands on the right hand side can be modified or operated upon by one of these six operators. 'Shift' and 'rotate' make use of the shifter associated with the ALU. 'Mask' uses the masker and 'field extraction' uses both the shifter and the masker. 'Indirection' and 'register concatenate' makes use of the iterative capabilities of the host machine. 'Indirection' is the fetching of data from (host) memory via an indirect address and 'register-concatenate' operates on two more registers or data words as if they were a cascaded pair, i.e., a single long data word.

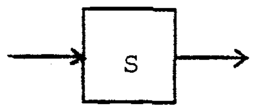
IMPLICATION 5: The control constructs of the language should be simple in nature and tailored to the needs of emulator writing.

JUSTIFICATION 5: The control constructs of the language should be of the D' type according to Kosaraju's hierarchy of control structures (53). The reason for choosing these structures is that they are found to be reliable, structured, simple (P2) and as pointed out by Ledgard (54) - 'the need for higher level (above D') control structures remains unproven'. The D' structures include the sequence, conditional, loop and case statements Fig. 2.3. However, in VMPL these structures have been modified for emulator writing.

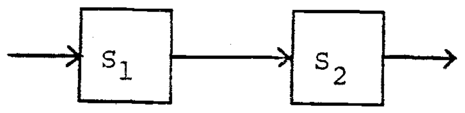
For specifying loop operations two constructs are provided. One allows the execution of the loop, a fixed number of times and can take advantage of the iterative mechanism available on the host. The other is a conditional looping mechanism.

Before any computer (target) executes any instruction, it has to decode it. This process is done by making a decision based on the opcode of the instruction (macro) and then executing it by enabling the appropriate circuit. The equivalent of this in VMPL is a special form of the case control structure. A jump to a multiple statement case statement is made based on the value of a variable. The statement to which the jump is made is another jump statement which mades a jump to a procedure (part of the program) which executes the instruction. This form of the case statement fits nicely into the N-way branch instruction available on hosts.

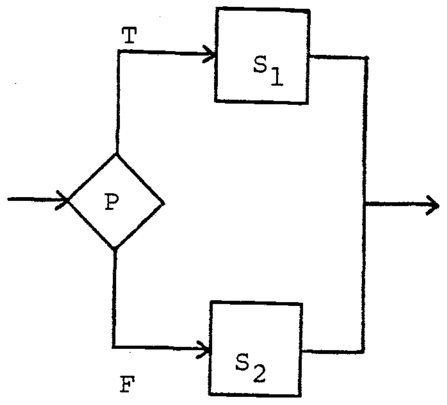
At times an equivalent operation is done wherein a number of conditions are checked one after the other and if any of them are true, to execute the code associated with them and then check the next condition. The case statement won't do this as it executes code corresponding to only one branch, i.e., one true condition. This is provided by an alternate version of the case statement.



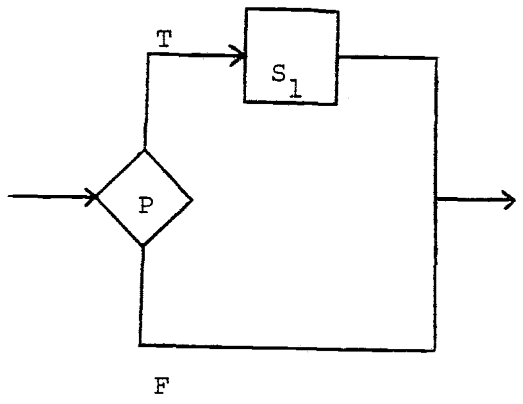
a, ACTIONS



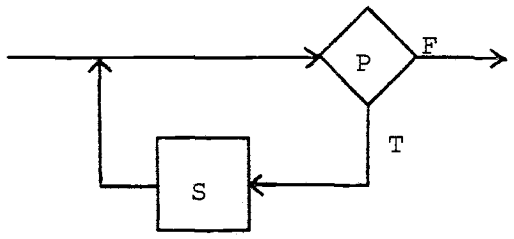
b, COMPOSITIONS



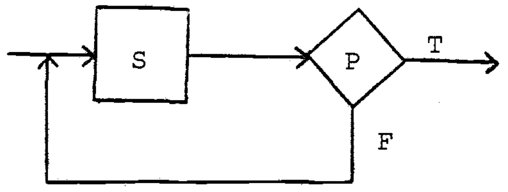
c, IF-THEN-ELSE



d, IF-THEN



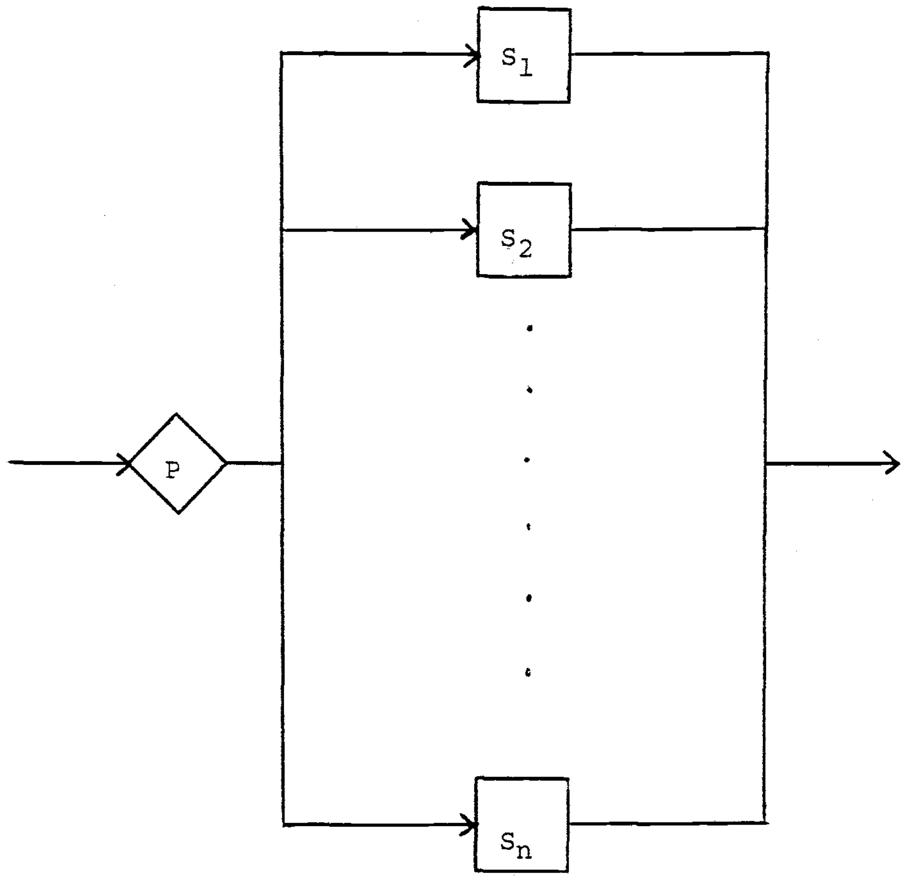
e, WHILE-DO



f, REPEAT-UNTIL

S, S_1 S_n	- STATEMENTS
P	- PREDICATE
T	- TRUE
F	- FALSE

Fig. 2.3 DEFINITION OF D'STRUCTURE



g, CASE

Fig. 2.3 DEFINITION OF D'STRUCTURES

In VMPL boolean expressions only the three relational operators which are found to be used 85% of the time by Elshoff (50) are used. Because of target machines capabilities of doing bit testing, which can be achieved on hosts by masking and shifting, we have explicitly provided for a way of directly indicating a certain bit of a variable as the one under test in a boolean expression (explicit feature). This can be considered as programmer convenience since the same effect can be obtained by using the 'shift' and 'mask' monadic operators discussed earlier (implicit feature). However, since some hosts do provide direct bit test capability, it is better (efficient) to keep this feature in the language explicitly and let the compiler decide on what to do.

IMPLICATION 6: The language should have the provision of declaring external procedures, variables, flags, etc. These are 'implemented' in runtime support microcode and linked to the rest of the emulator program.

JUSTIFICATION 6: There are two aspects of emulation which have not been discussed until now. These are Input/Output and interrupts. It is extremely difficult to match host I/O and interrupt structure to target machine I/O and interrupt structure without knowing the target machine in advance.

The true solution to this problem lies in the design of host machines. Host machines should provide a 'soft' (38) I/O and interrupt structure that can be mapped into primitive functions (in VMPL) which can in turn be used to describe the I/O and interrupt structures of various targets. Without soft I/O and interrupt structures, m-computers do not share a common set of properties that can be characterized for a class of machines. This aspect of emulation is further complicated by the wide variation of I/O and interrupt structures in various target machines. Although this wide variation is somewhat narrowed down because of P5, by limiting ourselves to targets which do not have I/O channels, I/O processors, direct memory access and other sophisticated I/O and interrupt structures (P5), we still have not found enough common primitives in hosts and target machines to have some kind of primitive language constructs which can be used to specify target I/O and interrupts.

This problem is temporarily 'solved' in VMPL by the use of externals. Each external name has associated with it the descriptor attribute which indicates whether the name is a procedure, a flag or a simple variable. Any peculiarity of the target which cannot be described in VMPL can thus be described in this way.

<u>IMPLICATION 7</u>: The overall program is broken into arbitrary sized blocks called procedures or subprocedures. All variables used within the block are declared at the start of the block. A priority status is associated with the variables used in a block by the programmer. The code is executed sequentially from one block to the next. There are two cases where control is nonsequential: 1) when a subprocedure is called, or 2) a 'case' statement is executed which jumps to another block.

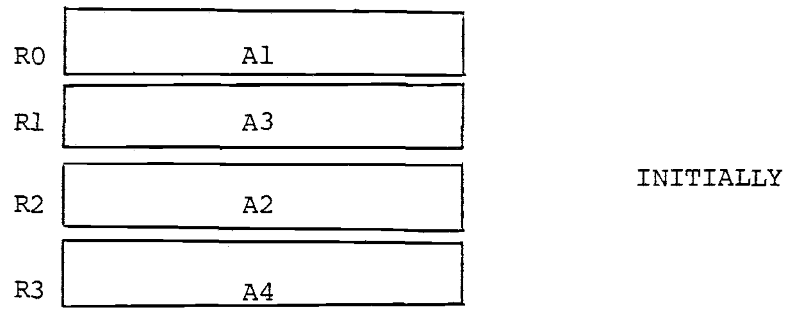
<u>JUSTIFICATION 7</u>: Register allocation/deallocation is a major problem in producing compact microcode (P3) for m-computers. Briefly, the problem is as follows. Suppose either the number, the size, or both of the target registers defined as variables in the VMPL program are greater than the local storage registers available on the host. Because of this disparity, the compiler is unable to permanently store these variables in local storage. The variables will have a copy somewhere in the main memory as the need arises. Now, when the compiler is binding (allocating) a variable to a register in the local storage, it may have to produce microcode to store the register (deallocate) in main memory. This may be necessary because the register may be associated with another variable and its copy in main memory is not fresh (active variable). Suppose at this point the compiler has the option of allocating/deallocating more than one register. Which one should the compiler choose? Similarly, the compiler has a very limited amount of look ahead and doesn't know what requirements for variables will be in the next statements. As such it might generate a large number of load-store instructions and hence a lot of microcode which goes against P3. This

situation is depicted in Fig. 2.4 and illustrated by the following example.

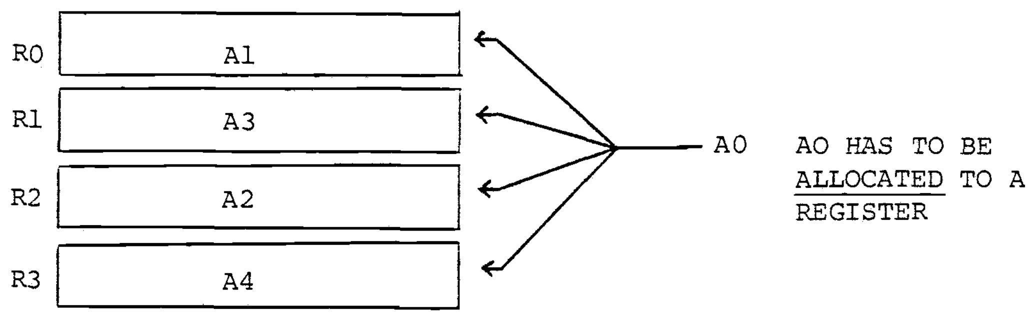
A four register (R0, R1, R2, R3) local storage host is assumed to have variables A1, A2, A3, A4, respectively, at some point within the compilation process. The compiler needs to put variable A0 in one of the four registers. However, if it stores the variable in any of the registers, it has to store the variable already present there back in main memory. Which register should the compiler choose? For example, if the compiler knew that variable A2 is not going to be used in the next statements, then it will store it back in main memory and allocate register R2 to variable A0. How is the compiler supposed to know this?

Sophisticated schemes have been developed to minimize register-variable swapping (14). One thing overlooked in these schemes is the fact that the programmer knows what variables are used in a certain 'block' of code. Why not take advantage of programmer 'look-ahead' with a compiler that uses this information when allocating/deallocating registers? This is achieved by making the program modular. All variables used throughout the program are declared at the start of the program. These are the global variables. At the start of each block the programmer specifies which of the globals are going to be used in a block of code and declares them as such. He also declares variables which are local to the particular block and will only be needed as long as the block is executing. The compiler uses this declarative information when allocating/deallocating variables. Since the variables which have not been declared will not be used in a block, they need not be kept in local storage.

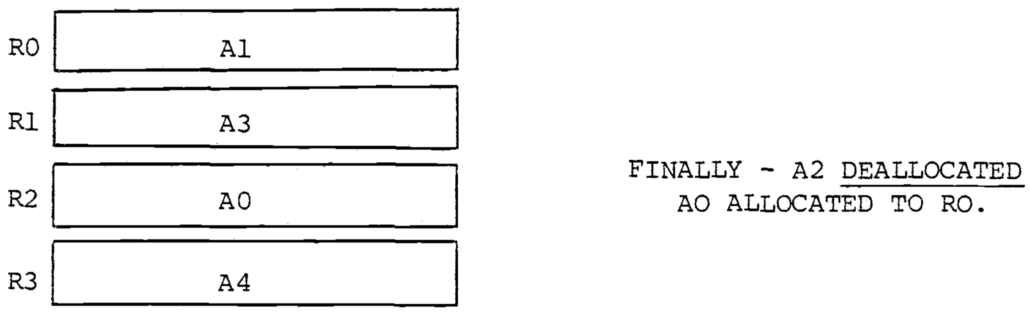
An implicit priority associated with the local-global classification of variables. Globals are given a higher priority than locals. By this, we mean that if the compiler has to allocate both a global and a local variable, it will first allocate the global variable. Similarly, if it has the similar options of deallocating variables, it will deallocate the local variable first. Further levels of priorities can be associated as tags with various variables. This provides the compiler with a hierarchy of look ahead. The compiler thus has finer control over which



(a)



(b)



(c)

Fig. 2.4 VARIABLE ALLOCATION PROBLEM

variables to allocate/deallocate based on the information provided by the programmer. Furthermore, the reliability of emulators is improved by introducing the redundant declarative information. The compiler can use the redundant declarative information to check for program consistency by seeing that only those variables which have been declared at the head of a block are used within the block.

Since the VMPL program is broken up into arbitrary sized blocks, program execution proceeds from one block to the next in a sequential manner. However, some blocks can be declared as subprocedures. These are the same as FORTRAN's subroutine (40). The sequential execution of a VMPL program is thus broken when a call to a subprocedure is made. Subprocedures return control to the place of their invocation. Another situation of non-sequential execution is the execution of a 'case' like statement. This will be discussed later on.

IMPLICATION 8: A subprocedure should be declared before it is used.

JUSTIFICATION 8: The reason for this is that the state of the machine (host), in terms of what is contained in its local storage at the entry and exit to a subprocedure, has to be known before a call is made to it from any other place in the program. Thus, the subprocedure is declared before it is used so that it gets compiled first. Then information about the state in which subprocedures expects and leaves the machine is known. The global variables that are expected by a subprocedure and those that are returned by it must be explicitly declared as such. The only variables that are passed as parameters are the local variables of the block making the call to the subprocedure. This is because the only variables not 'accessible' to the subprocedure are the locals of the calling block, since the globals are global to all the blocks.

The idea presented here is not in any way associated with the problems which arise with one pass compilers (55). As we will see later on, ours is not a one pass compiler. The only point of importance is that the subprocedure has to be compiled before it is used. 'Declaring' it before it is used also helps the programmer in structuring his whole program.

2.4 DEFINITION OF VMPL

In this section we specify the syntax of VMPL. The complete BNF representation of VMPL is given in Appendix A. The various syntactic entities of the language have been grouped together according to the implications presented in the previous section. Any syntactic feature of VMPL which is not covered by the implications of the previous section is justified as being in the language for programmer convenience.

All implications discussed in the previous section are referred to as I1, I2, I3, I4, I5, I6, I7 and I8.

2.4.1 DECLARATIONS

Declarations for the various data items provide the attributes of the items. Declarations can be grouped into the following four types:

1) Global variable declarations.
2) Global information declarations.
3) Local variable declarations.
4) Local information declarations.

2.4.1.1 VARIABLE DECLARATIONS

The syntactic description of VMPL variable declarations is:

```
    DCL   TYPE    PRIORITY    CLASS     NAME
        :  DIMENSION  :  DESCRIPTOR  :  SIZE
```

The TYPE of the variable is either LOCAL or GLOBAL. Variables which are used throughout the program are declared at the start of the program and are of the GLOBAL type. Variables used in any particular block (discussed later on) are of type LOCAL. Globals have a higher priority than locals (I7). In VMPL a further level of PRIORITY is associated with variables. The PRIORITY of a variable can be either PERMANENT or TEMPORARY. Permanents have a higher priority than temporaries. We will see in Chapter 6 how this hierarchy of priorities helps the compiler in producing efficient code.

There are six classes of data items or variables (I2 & I6). This is given by the CLASS entity of the variable declaration. The six classes are:

1) MEMORY
2) STACK
3) PSTACK
4) FLAG
5) SIMPLE
6) EXTERNAL

The MEMORY variable is used to define the target memory. It has associated with it the dimension attribute which is an integer defining the number of words (elements) which the memory has. In case the target has a stack, it can be described by the STACK variable. The stack pointer is described by the PSTACK class. In VMPL, we can only declare one stack and one stack pointer (target machines with one stack at most). The stack declaration requires the stack pointer declaration. However, the stack pointer declaration does not necessarily require the stack declaration. This covers those targets which have stack pointers but no explicit stack. In this case, the variable declared as MEMORY will be the default stack. In these machines the stack resides in the main memory. The question is why we need the PSTACK class, i.e., why treat the stack pointer as a special register. It could be covered by the same declaration which covers the other registers of the target. The reason is that the most frequent operations done on a stack are the PUSH & POP operations via the stack pointer. The PUSH & POP operations along with the knowledge about whether the stack grows upwards (from higher address towards low addresses) or downwards (from lower address towards higher addresses) completely determine the actions of the stack pointer. These actions are grouped together and shown in Table 2.3. The notation given in the last two columns of Table 2.3 is used in the DESCRIPTOR part of the PSTACK variable. Thus, by declaring the PSTACK variable, the compiler knows what code to generate to emulate the PUSH & POP operations which are a part of VMPL syntax. It may also match the stack pointer of the host (if one is present) to that of the target since it has all relevant information.

TABLE 2.3 STACK OPERATIONS AND THEIR VMPL DESCRIPTOR REPRESENTATION

FOR PUSHING DATA INTO STACK	FOR POPPING DATA FROM THE STACK	CORRESPONDING VMPL NOTATION PUSHING	POPPING
(PUSH, INC)	(DEC, POP)	$(\downarrow, +)$	$(-, \uparrow)$
(INC, PUSH)	(POP, DEC)	$(+, \downarrow)$	$(\uparrow, -)$
(PUSH, DEC)	(INC, POP)	$(\downarrow, -)$	$(+, \uparrow)$
(DEC, PUSH)	(POP, INC)	$(-, \downarrow)$	$(\uparrow, +)$

(a) (b)

Single bit flags are described in VMPL via the FLAG class. One of four descriptors may be associated with a flag variable. These descriptors are C, O, N, Z. The C descriptor makes the flag equivalent to the CARRY flag. For example, flag variable 'LINK' has associated with it the descriptor C. 'LINK' can now be used in conjunction with an assignment statement. Suppose the operation in the assignment statements is the add operation. Any carry generated would be stored in the flag variable 'LINK'. All the programmer has to do is to specify whether he wants LINK to be effected or not by a certain operation. The compiler then generates the code (I3) which actually affects the flag. The O, N and Z descriptors are associated with the overflow, negative and zero flags, respectively.

Target registers are described in VMPL via the SIMPLE class. The use of the reserved word SIMPLE is optional however. LOCAL variables can only be of the SIMPLE class or FLAG class.

The external problem as outlined in (I6) is covered by the EXTERNAL attribute in VMPL. The external data element can be a flag, a subprocedure or a simple variable. This qualification is done via the descriptors F, P or S, respectively.

The DIMENSION and SIZE entities of variable declaration are given by positive integers. The DIMENSION entity is enclosed in rectangular ([]) brackets. Examples of VMPL declarations which illustrate the various aspects discussed above are shown in Fig. 2.5.

2.4.1.2 INFORMATION DECLARATIONS

There are three global information declaration statements in VMPL and four local information declaration statements.

2.4.1.2.1 GLOBAL INFORMATION DECLARATIONS

Since the size attribute is associated with data items of different types and since it usually is the same for most of them in one program (the width of target stack registers memory words, etc. is usually the same), it seems better and convenient to make a global declaration of

```
             DCL GLOBAL PERMANENT MEMORY     MEM:[4096]:12      ;

             DCL GLOBAL PERMANENT STACK      STK:[32]:12        ;

             DCL GLOBAL PERMANENT PSTACK     PSTK:(↓,+):5       ;

             DCL GLOBAL TEMPORARY FLAG       LINK:C, OV:O       ;

            ┌DCL GLOBAL PERMANENT SIMPLE     ACC: 8,T1:12       ;
EQUIVALENT  ┤
            └DCL GLOBAL PERMANENT            B1:8, B2:12        ;

             DCL EXTERNAL PERMANENT          REG1:S:4,IOHAND:P;

             DCL LOCAL PERMANENT SIMPLE      REG2:8             ;

             DCL LOCAL TEMPORARY FLAG        FLG                ;
```

Fig. 2.5 VMPL GLOBAL AND LOCAL VARIABLE DECLARATIONS

this fact. This is done via the WORDSIZE declaration as shown in Fig. 2.6a.

Each individual declaration of a data item may carry the information about its size, but in case it doesn't, and the information is required for that type, then the default value of the size is taken from the globally provided information. Fig. 2.7 illustrates this wherein both (a) and (b) are equivalent declarations for an 8-bit register R1.

The two basic ALU arithmetic modes for computers (targets & hosts) are one's-complement and two's complement arithmetic. The host ALU may perform arithmetic differently than the target ALU. In order to keep VMPL machine independent, possible conflicts must be handled at compile time. The VMPL programmer declares the mode of arithmetic to execute under, (the mode of arithmetic done on the target) and the compiler then handles any mismatches which may occur. The declarations are shown in Fig. 2.6b and c.

In VMPL bits of a n bit data item are numbered from 0 to n-1 starting at the right or the least significant bit. The 'field extract' operation is frequently done in the instruction decode part of a virtual machine. A field is meant to be the result of the logical AND operation between a data item and a mask followed by a shift operation. The shift may be left or right. Fields are declared and given a name in VMPL. They have two or three integer parameters associated with them. The first two (equivalent to the mask) indicate the range in bit position which has to be masked out. The third optional integer indicates the shift amount. Its absence indicates no shift. A left shift is indicated by a negative number and a right shift by a positive number. The field OPCODE in Fig. 2.6d refers to bits 9, 10 & 11 to be extracted and shifted right by 9 positions. ADDRES on the other hand is a reference to the extracted bits 0 to 8 but doesn't shift them. The way fields are used is discussed later on.

2.4.1.2.2 LOCAL INFORMATION DECLARATIONS

As discussed in I7, we expect to improve the efficiency of the microcode produced by indicating to the compiler the global variables

```
 DCL        WORDSIZE             12  ;

               (a)

DCL         ARITHMETIC            1  ;

               (b)

DCL         ARITHMETIC            2  ;

               (c)

DCL         FIELD         OPCODE (9,11,9)  ;

                          ADDRES (0,8)     ;
               (d)
```

Fig. 2.6 VMPL GLOBAL INFORMATION DECLARATIONS

```
DCL GLOBAL PERMANENT  R1:8;
```

(a)

```
DCL WORDSIZE  8;

DCL GLOBAL PERMANENT  R1;
```

(b)

Fig. 2.7 WORDSIZE DECLARATION EQUIVALENTS

which will be used in a certain block. This information is provided by importing the global variables used in a certain block, see Fig. 2.8a (global USE variables).

The next two information declarations are associated only with subprocedure blocks. The only variables passed as parameters to a subprocedure are the variables which will be local to the block from which the subprocedure is called. Global variables (like FORTRAN's (40) COMMON) needn't be passed as parameters since they can be declared as global USE variables as discussed above. Thus, within the subprocedure we have three kinds of variables being used; the subprocedure parameters, the global variables imported with the USE statement, and any variables LOCAL to the subprocedure block. We have already discussed the reasons why a subprocedure has to be compiled before a call to it is made (I8). The non-local variables (i.e., variables not declared by the LOCAL statement) can be used in three ways within the subprocedure.

a) Their value is used somewhere within the subprocedure block, i.e., the subprocedure <u>expects</u> their value when a call is made to it or it doesn't modify the variables value.

b) They are assigned a value somewhere within the subprocedure block, i.e., the subprocedure <u>returns</u> their value when a call is made to it or it modifies the variables value.

c) Both a & b happen.

We will discuss these three cases by the VMPL code shown in Fig. 2.9. The subprocedure block is named SUB. It has one parameter PAR. The procedure block which 'calls' SUB is block MAIN. It has a local variable LOCL which it passes to SUB, i.e., PAR is the formal parameter and LOCL is the actual argument. We assume that SUB doesn't use any global variables. The m-machine for which code is being produced has two registers R1 & R2 in its local storage.

<u>CASE a</u>

SUB expects the value of PAR, i.e., LOCL when it is called from MAIN.

Since SUB is compiled before MAIN, we associate a certain register,

```
DCL GLOBAL USE  MEM, ACC, T1;
```

(a)

```
DCL GLOBAL EXPECT  REG2;
DCL LOCAL EXPECT   TMP1;
DCL LOCAL RETURN   TMP1;
DCL LOCAL PERMANENT VAL1, VAL2:3;
```

(b)

```
DCL SPROC USE  SUB;
```

(c)

Fig. 2.8 VMPL LOCAL INFORMATION DECLARATIONS

say R1, with the parameter PAR. This means that when we compile SUB and we want the value of PAR (statement 9), we assume it to be in R1 and pick it up from there. Thus, when we call SUB from MAIN, we first store the value of LOCL in R1 and then do the call operation. It also means that when we compile SUB, we do not use R1 for any other variable being used in SUB until statement 9, i.e., until the time when we first use PAR (R1), we want to preserve it since it has the value which was passed from MAIN to SUB. From there onwards all registers are handled by the compiler in the same manner.

All this can be done if within SUB we make a declaration of the fact that the value of the variable PAR is EXPECTED from the block which makes the call. Thus, all global variables and parameters which are expected by a subprocedure are declared as having this property.

CASE b

SUB returns the value of PAR, i.e., LOCL when it is called from MAIN.

After having compiled SUB, we know that PAR is in register R2. Within MAIN after producing the code for making the call to SUB (statement 24), we have to produce code for statement 25. For this statement and the following statements, we know LOCL is in R2 since that is where SUB leaves it or returns it. This information is available if we declare PAR within SUB to have the RETURN attribute. Now when the compiler compiles SUB, it will preserve R2 from the point where it last stores the value of PAR in it to the point where it leaves and returns to MAIN.

```
- -
- -   ────→ SUBPROCEDURE (SPROC)
            DECLARATION
- -

 PROC:MAIN;
DCL LOCAL PERMANENT  LOCL:8;

 - -
#

 EXECUTE  SUB(LOCL);      [*STATEMENT NO. 24*]
 - -
 - -    #
```

(a)

```
SPROC:SUB(PAR:8);

 DCL LOCAL EXPECT PAR;
 DCL LOCAL PERMANENT T1,T2;
  #
   - -
   - -
   T1 = PAR;   [*STMT.No. 9*]
   - -
   - -    #
```

CASE a

```
SPROC:SUB(PAR:8);

 DCL LOCAL RETURN PAR;
 DCL LOCAL PERMANENT T1,T2;
  #
   - -

   PAR = T1;   [*STMT. No.
                    24*]
   - -
   - -   #
```

CASE b

Fig. 2.9 EXPECT & RETURN DECLARATION EXAMPLES

```
SPROC:  SUB(PAR:8);
  DCL LOCAL EXPECT PAR;
  DCL LOCAL RETURN PAR;
  DCL LOCAL PERMANENT T1,T2;
        #
         - -

         PAR = T1;
         - -
         - -

         T2 = PAR;
         - -
         - -    #
```

CASE c

Fig. 2.9

EXPECT & RETURN DECLARATION EXAMPLES

CASE c

This is just a combination of cases a and b. For this case a variable has to be declared both as an EXPECT & RETURN variable. Examples for these declarations are shown in Fig. 2.9b.

Using the same arguments, we can say that upon entry to a block if the compiler knows which, if any, subprocedure will be called from within that block, then it can produce more efficient code. This is because while producing code for the block, the compiler knows at entry which variables are going to be expected and returned by the subprocedure which is going to be called from within the block. It knows the registers which will hold those values when the call is made to the subprocedure since the subprocedure has been compiled first. Thus, for the statements before the call statement in the block, it tries to utilize the machines registers in a way which will make the change to the subprcedure's environment efficient. As in our previous example, case a, if the compiler knows that SUB is going to be called from within MAIN, then while producing code for statements 1 through 23, it will try to keep LOCL in R1 so that when the call is made to SUB, the value of LOCL needn't be loaded into R1 (extra code).

2.4.2 ASSIGNMENT STATEMENTS, OPERATORS & EXPRESSIONS

A very simple model for the assignment statement is discussed in I4. Expressions in assignment statements are limited to diadic arithmetic operators and special monadic operators. Logical and relational operators are limited to boolean expressions. The basic arithmetic, logical and relational operators available in VMPL are shown in Table 2.4. Each operand of an assignment statement can have a number of monadic operators associated with it. These monadic operators are:

TABLE 2.4 VMPL DIADIC OPERATORS

ARITHMETIC	LOGICAL	RELATIONAL
+	.AND.	.GT.
-	.OR.	.LT.
*	.XOR.	.EQ.
/		

TABLE 2.5 VMPL SHIFT OPERATORS

SHIFT DIRECTION	FILL UP BIT	VMPL NOTATION
RIGHT	Ø	.SHTRØ.
RIGHT	1	.SHTR1.
LEFT	Ø	.SHTLØ.
LEFT	1	.SHTL1.

(a)

```
A = B.SHTRØ.6     ;

X = Y.SHTL1.2  +  3    ;
```

(b)

a) SHIFT AND ROTATE: There are four monadic shift operators available in VMPL. A data item can be shifted left or right and the bit positions which become available can be filled by either a one (1) or a zero (0) Table 2.5a. In reality, a shift operator is diadic because it requires another operand which specifies the shift amount. In VMPL this operand has to be an integer, i.e., the shift amount is fixed at compile time. Table 2.5b illustrates the use of the 'monadic' shift operator. In the first example, variable B is right shifted and zero filled by six positions and the result stored in A. In the second example Y is shifted as indicated by the shift operator and the shifted result is added to 3. The result is then stored in X.

Similarly, two other operators which correspond to rotation are available. These are used for left and right rotation of data items. ROTL is used for left rotation and ROTR is used for right rotation. Like shift, the rotate operator also requires an integer specifying the amount of rotation Table 2.6.

b) FIELD EXTRACTION: As mentioned in the section on global information declarations, in VMPL a programmer can define field operators, which effectively mask and shift various bits out of a data item, i.e., perform the extract operation. The statements in Fig. 2.10b use the fields defined in Fig. 2.10a. The first statement extracts the OPCODE field, i.e., (bits 9, 10 and 11 shifted 9 places to the right) from the variable IR and stores the result in OPCD. The second statement extracts the ADDRES field from IR, adds the result to 20 and stores this result in PAGE.

c) INDIRECTION: One of the most common addressing modes found on various target machines is the indirect memory mode. In VMPL we define an indirection operator by the @ sign. It can only be used on the MEMORY class of data item. If M is a memory variable and ADDR & IR are simple variables, then

```
IR = @M [ADDR]
```

is equivalent to

```
IR = M [M[ADDR]]
```

which by itself is not allowed in VMPL. The square brackets ([]) are

TABLE 2.6 VMPL ROTATE OPERATORS

ROTATE DIRECTION	VMPL NOTATION
LEFT	.ROTL.
RIGHT	.ROTR.

```
A = B.ROTL.7   ;

R1 = REG1.ROTR.3;
```

(c)

```
DCL FIELD                      OPCODE (9,11,9) ,
                               ADDRES (0,2)    ;
```

(a)

```
OPCD = OPCODE (IR)   ;

PAGE = 20+ ADDRES (IR)   ;
```

(b)

Fig. 2.10 EXAMPLES OF VMPL FIELD FACILITY

themselves a monadic operator which are used to index into dimensioned variables, i.e., variables which have the DIMENSION attribute. In the example above, once the data at the ADDR index of M is obtained, it is used to index again (indirection) into M to fetch new data which is then assigned to IR.

d) CONCATENATION: The concatenation operator allows concatenated items to appear on either side of an assignment. Two SIMPLE variables A and B, can be concatenated together and then logically used as if they existed as a single entity, i.e.,

```
A//B = A//B+2
```

The integer 2 is added to the logically formed variable A//B and the result stored back in A//B.

e) BIT SELECTION: Last of all an operator is available which can test a selected bit of a data item. This operator is limited to boolean expression only. It returns a value of true if the bit in question is a 1, otherwise it returns a false value:

```
(IR, 6)
```

This means that the sixth bit of IR is to be tested to see if it is true or false.

Since we have defined a number of diadic and monadic operators, precedence rules have to be established for expression evaluation. The primary rule for expression evaluation is:

All monadic operators are executed
before the diadic operator is executed

We present these rules in the form of a chart as shown in Fig. 2.11. The blank entries mean that the corresponding entries cannot occur simultaneously in an assignment statement or a boolean expression.

There is only one predefined variable in VMPL. This is the reserved name POP. Its value is the value of whatever gets popped off the stack, i.e., it is a way of getting data off the stack via the pop function. POP is treated as a simple variable and it can be used anywhere a simple

	=	[ ]	()	@	//	+	-	*	/	.AND.	.OR.	.XOR.	.NOT.	.GT.	.LT.	.EQ.	.SHTXX.	.ROTY.
=		>	>	>	>	>	>	>	>	>	>	>	>				>	>
[ ]			<	<		<	<	<	<	<	<	<	<	<	<	<	<	<
()				>		<	<	<	<	<	<	<	<				<	<
@						<	<	<	<	<	<	<	<	<	<	<	<	<
//						<	<	<	<	<	<	<	<				<	<
+													>					
-													>					
*													>					
/													>				>	>
.AND.													>				>	>
.OR.													>				>	>
.XOR.													>				>	>
.NOT.														<	<	<	>	>
.GT.																	>	>
.LT.																	>	>
.EQ.																	>	>
.SHTXX.*																		
.ROTY.**																		

*XX=L0
    L1
    R0
    R1

** Y=L
    R

$\delta < \beta$ MEANS δ IS DONE (EXECUTED) BEFORE β.

Fig. 2.11 PRECEDENCE RULES FOR VMPL OPERATORS

variable is used, i.e., an operand in an expression, index to the variable of class MEMORY, etc. However, it should be remembered that using this variable has a side effect on the PSTACK variable. The PSTACK variable gets incremented or decremented as defined by its descriptor.

In I3 and I5, we mentioned that the setting/clearing of VMPL's predefined flags can be directly indicated in an assignment statement. This is shown in Fig. 2.12. For this example we assume CARY and ZRO are user defined flags associated with VMPL defined carry and zero flags (via the descriptors C and Z). Variable B is added to C and the result is stored in A. The carry and zero conditions generated by this operation is reflected by the setting/clearing of the CARY and ZRO flags. The code to achieve this is generated by the compiler.

2.4.3 CONTROL CONSTRUCTS

In I5 we indicated the need for a simple and complete set of control constructs modified to the needs of emulator writing.

Conditional statements, which according to Elshoff's (50) study make up 17.8% of a program, are provided for in VMPL by the IF....THEN...ELSE....construct with certain modifications. Two IF statements are provided in VMPL; IFTRUE which causes the THEN portion to be executed if the boolean expression is true, and the IFFALSE, which avoids confusion over negative predicates by executing the THEN clause when a predicate is false. This will hopefully remove the semantic difficulty for a programmer, when deciding between:

IF NOT (Boolean expression)

or

IF (Boolean expression)

The FOR statement allows the repetition of a loop a fixed number of times. As such the FOR statement only allows integers to be used as it indexes--so it may be easily mapped into loop counter mechanisms available on some host machines. The WHILE statement provides a way for conditional looping, executing the loop as long as the condition is true.

As mentioned in I5, the case statement of Pascal (56) has been modified so that it can be directly mapped into a N-way branch

```
A    =  B  +  C          ⟨CARY, ZRO⟩   ;

REG1 = REG2 + REG8        ⟨ ZRO ⟩   ;
```

Fig. 2.12 ENABLING VMPL FLAGS ;

instruction found on many host machines. VMPL's case statement is the SELECT statement. An example is shown in Fig. 2.12. The selection in the example is done based on the value of OPCD (first operand). If its value is 1 (target operand), a jump is made to the procedure block LAB1 (target label 1); if its value is 2, a jump is made to LAB2, etc. The digit 4 (second operand) following the identifier OPCD is a count of the number of labels available to the SELECT statement. This provides two pieces of information to the compiler. One, it indicates to the compiler how many bits of the first operand (in our example OPCD) will be used for the jump selection. This quantity (the number of bits of the first operand used for jump selection) is Log_2 of the second operand (in the example $\text{Log}_2 4=2$). It is used by the compiler to mask any extra bits from the second operand (OPCD) if it is wider than it (in the example if OPCD is wider than 2 bits, it's significant bits are masked to make it only 2 bits wide). Secondly, it indicates to the compiler the number of jumps involved in the select operation. The host may have a larger or smaller n-way branch instruction and as such has to handle these cases appropriately. Again, a multipass compiler can handle this situation by counting the number of labels a jump has to be made to, and as such the presence of the second operand indicates redundancy. As mentioned earlier redundancy leads to reliability, one of the requirements put on VMPL by P2.

The CONDition statement is equivalent to the SELECT statement in some respects. Each boolean expression is evaluated and if true, the corresponding block of code is executed Fig. 2.12. However, unlike the SELECT statement, the CONDition statement after executing the block of code tests the next condition to see if it is true, and so on, until the end of the statement. Also, unlike the SELECT statement the CONDition statement requires each block of code to follow the statement rather than as a separate procedure block.

Two things are obvious from Fig. 2.12. One is the consistent use of 'BEGIN - END' brackets for all five control constructs. Since the 'END' recognizes the statement to which it belongs (ENDIF, ENDFOR, ENDWHILE, ENDSELECT, ENDCOND), it solves the dangling END problem which reduces errors by atleast 5-9% (52) and makes VMPL more reliable. The other is

```
IFTRUE  (A .EQ. B)  THEN   ;  BEGIN ; ----- END   ;  ELSE ;
  BEGIN ; ----- END ; ENDIF ;


WHILE (A .LT. B)   ; BEGIN   ; ----  END ; ENDWHILE   ;


FOR A = 3  TO  6  ;  BEGIN ; ----- END ;  ENDFOR;


SELECT   (OPCD,4)  FROM;                COND ;
    (0,LAB1)   ;                  (A.EQ.B)   ;  BEGIN ; ---- END ;
    (1,LAB2)   ;                  (P.EQ.Q)   ;  BEGIN ; ---- END ;
    (2,LAB3)   ;                  ENDCOND ;
    (3,LAB4)   ;
ENDSELECT      ;
```

Fig. 2.13 VMPL CONTROL CONSTRUCTS

the consistent use of the BEGIN-END bracket within four of the control constructs. This helps to contain and delimit the statements if more than one of them is present within the control structure.

2.4.4 MISCELLANEOUS STATEMENTS

There are a few more statements available in VMPL. Elshoff (50) and Knuth (51) determined that increment and decrement accounts for many of the operations in assignment statements. As such an INCrement and DECrement statement is provided. Since register clear (all zeroes) and set (all ones) is a common instruction found in most target instruction set processors, VMPL provides the CLEAR & SET instructions.

In order to be able to push data into the stack, the PUSH statement is available in VMPL. As mentioned earlier, data always pops out of the stack when POP is referenced.

EXECUTE and RETURN statements are provided to transfer control to and from a subprocedure.

VMPL does have a GOTO statement. However, we have limited the scope of the GOTO destination to its block, i.e., the label to which a GOTO transfer's control has to be in the same block as the GOTO statement. The LEAVE statement is used to leave one block of code and enter another. The only other statement which can transfer control from within one block to another is the SELECT statement since its labels are names of other blocks. Thus, the compiler knows the place within a block from where control passes from one block to another. The default place for this transfer is the last statement within the block.

A HALT statement is provided to map the targets halt instruction into the host's halt instruction.

Examples of these statements are given in Fig. 2.13.

2.4.5 PROGRAM STRUCTURE

I2, I7 and I8 indicate that the VMPL program is made up of two parts. First come the global declarations. These are followed by a set of

```
INC     A  //  B;

DEC   A[POP]      ;

SET   ADDR  ;

CLEAR  IR  ;

PUSH   DATA  ;

LEAVE  LABEL1  ;

HALT  ;

GOTO   IOEND;
```

Fig. 2.14 MISCELLANEOUS VMPL STATEMENTS

blocks each with their own set of declarations and executable code. The blocks are of two types: PROCedures and SubPROCedures, and each has a unique name. Each block declares all the variables it is going to use from the global variable set, all the subprocedures it is going to call and any LOCAL variables it requires. Local variables exist only for the execution life time of the block.

All identifier and labels in VMPL can have any number of characters. The first character has to be alphabet and the first seven should be a unique combination. A colon (:) must follow a label to indicate the termination of the label.

Comments can be placed anywhere in a VMPL program. They have to be enclosed by ((*) and (*)) brackets. The only restriction is that they should not split a basic token in the language, e.g., an identifier, a reserved word or an integer.

2.6 CONCLUSIONS

In this section we have presented the design and syntax of VMPL. The language design is based on a set of 'bases' which are used to derive a set of implications, each of which have been rigorously examined and justified. By providing a basis for the language and a set of implications, we provide a uniform way for making modifications and extensions to the language. This may be required for two reasons.

1) Once a sufficient number of programs have been written in VMPL, we should have enough information about its strong and weak points. We can then go back to our original bases and see which are the ones which require modification. Having done that we can go through the whole process of redesigning the language in the way we did originally. The point is that, since we tried to justify the constructs of the language when we designed them, it will be easier to change them based on the data we obtain about the language at a later time.

2) We may want to make extensions to the language by removing some of the constraints which we put on the problems we are trying to solve, e.g., we may want to include third generation computers with I/O processors in the list of target machines we can emulate using VMPL. This can be done by making the appropriate extensions to the original set of 'bases' and then design the constructs needed to fit this new requirement. Since we have the complete 'design plan' of the language in the form of an implication-justification 'chart', we can see which parts of the language are affected by the new extensions and which are not. Thus, the language can grow easily or be extended according to new demands without requiring the language designer to start from ground zero every time he has to make some changes in the language.

The only bias of the author which becomes obvious in this language design experiment is the usage of the instruction use and reliability data available about other programming languages. We feel that it is necessary to use this data to design programming languages. This helps the language designer in producing a language each of whose syntactical constructs have justified their inclusion. It also tells him all the constructs which should not be included. Already we see that this approach is being used in designing a major universal programming language (65) which may revolutionize the whole software industry.

3. MACHINE CONSIDERATIONS

3.1 INTRODUCTION

As indicated in chapter 1 problem 2, one of the reasons why earlier efforts have not effectively solved the high level microprogramming language problem is the lack of identification of primitives for writing code on m-computers. What are the characteristics of the host machine? Can we develop sufficient generalizations about the host? Can these generalizations be used to transport a higher-level language emulator which in turn effectively manipulates the host? Are there any common characteristics among the various machines which defines a class of host machines for which a portable VMPL system is suitable?

> If we want to design an efficient compiler for a high-level machine-independent language for a m-machines, we must classify the host machines before generating code

Eckhouse (13) was the first one to make an attempt in this direction. He took sixty-five architectural features and seven m-computers and made a table of features appearing in the various computers. However, his investigation has the following problems:

1) Four out of the seven m-computers investigated were microprogrammed computers (INTERDATA 3 & 4, IBM 2025, and the SPIRAS 65). Another one of the computers was Rosin's (16) hypothetical MPP. Another was the IBM 2050, host for the IBM System 360 Model 50 and as such not truly user microprogrammable. His analysis thus included only one computer, the Standard MLP-900, which is user microprogrammable. Although this m-computer was designed for general purpose emulation, it is not widely used. In fact, the only know MLP-900 is the PRIM project machine at the University of Southern California (17).

   The results which Eckhouse obtained were truly based on one machine, which is not widely used.

2) A closer look at the features which Eckhouse selected indicate that they are tailored for the seven machines chosen for the tabulation. As such some of the results which he obtained in terms of the specific categorizations or the general features of m-computer architecture, are biased and not general enough.

A valuable result obtained by Eckhouse related to Input/Output states:

> "I/O is generally integrated into the basic machine (m-machine) architecture. Therefore, I/O programming is a task which requires specific knowledge of timing constraints, device idiosyncracies, etc."

However, while designing MPL (his language) Eckhouse doesn't specifically deal with the problem of I/O and gets around it by saying:

> "...problem with I/O is the particular device-dependent structure of I/O at the microprogram level which tends to prevent a generalized, higher level language approach to the problem. However, if we consider the simpler machines such as MPP and the INTERDATA 3, we can easily discover several ways to treat I/O."

This further reinforces our earlier claim of the machine dependent nature of MPL. Furthermore, it indicates that the results about m-computer architecture which Eckhouse derives were subsequently not used in designing the language, or modified and narrowed down to specific cases. In order to recognize the architectural features of m-computers, we studied approximately 30 m-computers (18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37).

In the next section we will describe and analyze briefly the architectural features of m-machines with particular emphasis on those features that impact on generality. The notion of "generality" is included to make conclusions about m-machine architecture general enough to be machine-independent. This will then lead to a classification of machines acceptable to WMPL's compiler.

3.2 m-COMPUTER ARCHITECTURAL FEATURES

The features of m-computer architecture which we will discuss are:

1) Storage facilities.
2) Functional units.
3) Data widths.
4) Bus structure.
5) Microinstructions.
6) Input, Output & Interrupt Structure.

Although we discuss these features in general, we make specific references to emulation. Emulation is not only the most common application of microprogramming (5), The execution of a program in some language is sometimes performed as an emulation of a virtual machine (target machine) that interprets some high level language. Microprograms are used to implement high level language programs in a variety of ways as shown in Fig. 3.1.

All facilities where 'data' (program + data) can reside are known as storage facilities. In m-computers, we find a hierarchy of storage units quite different from other computers. The functional units are usually combinational circuits which operate on data in various ways. The data width along with the bus structure determines the computer's real usefulness in emulation. Microinstructions are the mechanism through which the programmer exercises control over the various facilities available on the machine. Input, Output & Interrupts are sources through which the m-computer communicates with the external world.

An overview of the various features of m-computers taken into account is shown in Fig. 3.2.

3.2.1 STORAGE FACILITIES

The storage facilities of an m-computer can be classified into the following four categories.

a) LOCAL STORAGE: Local storage consists of host machine registers usually used by the target machine as working registers. Efficient emulation requires atleast as many dedicated and general purpose registers as the target machine being emulated. A number of single bit

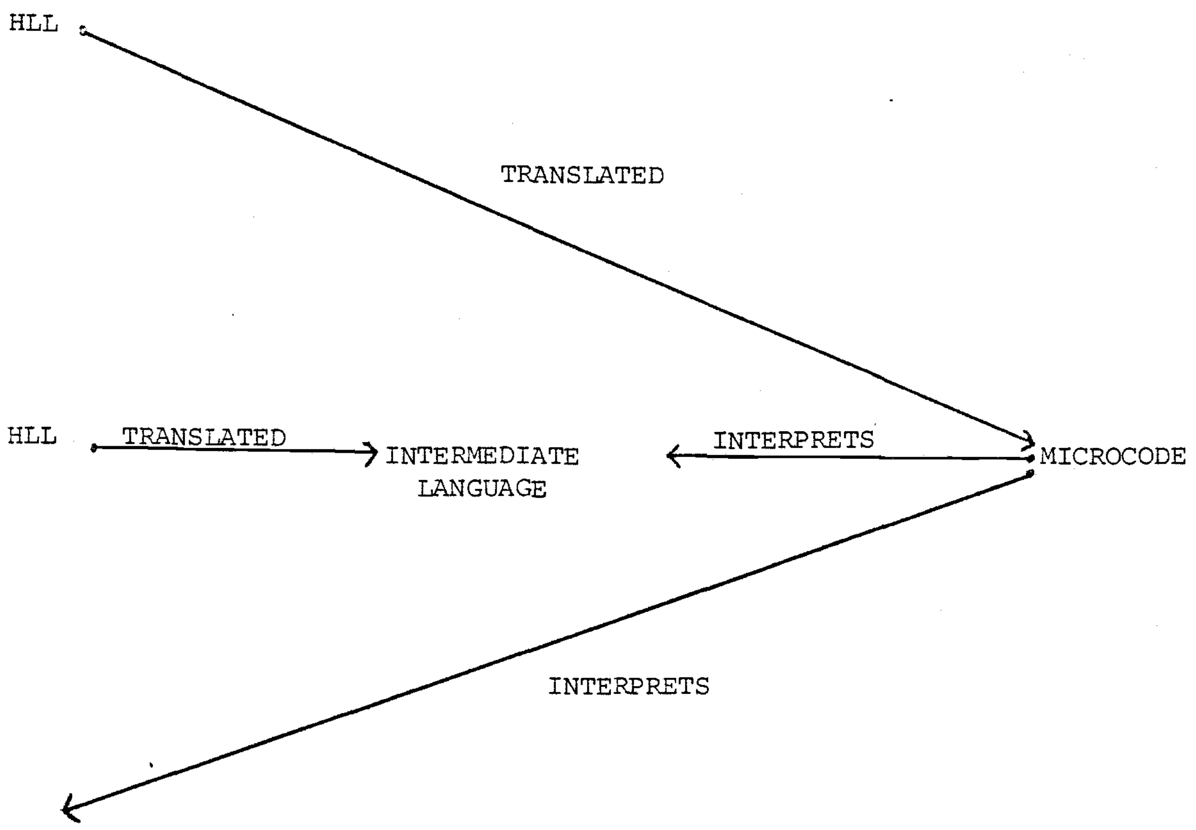


Fig. 3.1 MICROPROGRAMMED IMPLEMENTATION OF HIGH LEVEL LANGUAGES (HLL)

1- STORAGE

- LOCAL STORAGE
- MAIN MEMORY
- CONTROL STORE
- NANO STORE

2- FUNCTIONAL UNITS

- ALU
- SHIFTERS
- MASKERS
- EXTRACTERS
- OTHER FUNCTIONAL UNITS

3- DATA WIDTH

- FIXED
- ITERATIVE CAPABILITY
- FLOATING

4- BUSSES

- FIXED
- FLOATING

5- MICROINSTRUCTIONS

- VERTICAL
- DIAGONAL
- HORIZONTAL
- ENCODING
- RESIDUAL CONTROL
- SEQUENCING

6- INPUT, OUTPUT AND INTERRUPTS

- FIXED
- FLEXIBLE
- SOFT

Fig. 3.2 OVERVIEW OF m̃-COMPUTER ARCHITECTURAL FEATURES.

registers usable for storing condition codes and other status bits can also be of great utility. Special purpose registers or storage hardware frequently improve the efficiency of a machine. For example, the presence of a hardware stack has two advantages. First, it can be used to facilitate the emulation of target machines that have stacks. Second, it can be used for subroutine linkage at the microprogram level. Another type of local storage, present in the market these days, is the availability of intergrated circuits which form a 2, 4 or more bit wide local storage register. However, the physical units (IC's) containing this local storage can be concatenated together to form a local storage registers of any width.

b) MAIN MEMORY: Arbitrary target machine instructions (macro instructions) reside in main memory. Thus, word width, total number of words available and the addressing modes are important factors for general purpose emulation.

c) CONTROL STORE: Control store refers to the memory containing microinstructions of the m-computer. The structure, size and volatility of the control store is a large contributing factor in determining different classes of m-computers. Depending on volatility this memory can be read only (ROM) or read/write (RAM). In order to have concurrent fetch & execution of microinstructions, the control store support registers (e.g. the address register, data register, etc.) should be separate from other local storage registers. Further support units may be provided to address this memory in different ways, i.e., a separate ALU may be provided to do address calculations, etc.

d) NANO STORAGE: This is a read/write memory, one level below the control store. In the nano storage are stored nano-instructions which make up the nano program. Each microinstruction is associated with a nano-program. A microinstruction is thus executed by executing its corresponding nano program. The nano program is thus the microarchitecture emulator. This is equivalent to a macro-instruction being executed by its corresponding microprogram which is made up of a set of microinstructions. This level or hierarchy of instructions is shown in Fig. 3.3. The reason why some machines (38) are provided with a nano-storage is that the user can have a much more 'finer' control over the machine. With microinstructions he can define his own macro-instructions. With nano-instructions the user can define his own micro- instructions. This helps him in designing the microinstructions which are most suitable for designing the macro-instructions.

3.3.3 FUNCTIONAL UNITS

The most common and dominant of the functional units is the arithmetic logic unit (ALU). One or more ALU's provide the basic functional operations (ADD, SUBTRACT, AND, OR, etc.) done by the m-machine on data.

Generality of ALU's requires them to process data in different arithmetic conventions (one's and two's complement), process data of different arithmetic modes (binary, decimal, etc.) and process data of varying length (word, byte, etc.). The availability and generality of various kinds of status flags (CARRY, ZERO, etc.) associated with the ALU are also important characteristics of the machine. The presence of a separate set of flags for the target-virtual flags, is a definite help for target flag emulation.

At times secondary ALU's are provided for a fixed set of tasks like evaluating the effective address, operating in a certain mode, etc. Typically, a fixed size of ALU is available, i.e., it can operate on operands of a certain length. However, one or more of the ALU's can be physically cascaded to form an ALU of larger width, if it is required to operate on operands of a larger width. This effect can also be achieved in some m-computers which have an iterative or/and residual control mechanism for the ALU (20).

A shifter unit (or units may be associated with either the inputs or the outputs of the ALU (implicit shifting) or it may be a separate unit in the machine (explicit shifting). Shifters may be single or multiple bit shifters, arithmetic, logical or circular shifters and may shift single or double words.

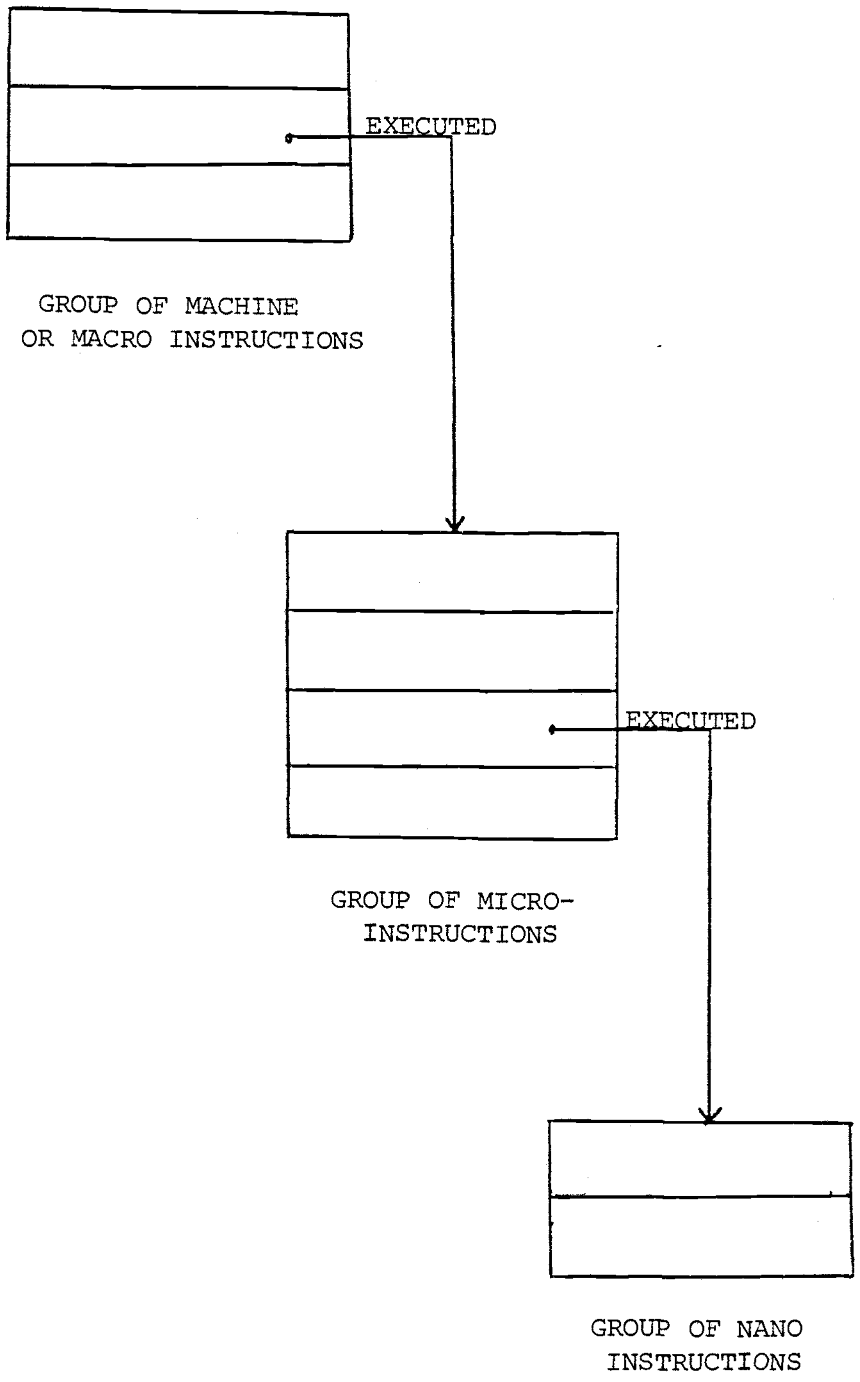

EXECUTED
GROUP OF MACHINE OR MACRO INSTRUCTIONS
EXECUTED
GROUP OF MICRO-INSTRUCTIONS
GROUP OF NANO INSTRUCTIONS

Fig. 3.3 HIERARCHY OF INSTRUCTIONS

Masking of data before sending it to the ALU or after receiving it from the ALU is achieved by units known as maskers. They can also be implicit or explicit units. As mentioned earlier, this facility of the machine improves the generality of the ALU functions. It can also, along with the shifter, be used to extract a field of arbitrary length and position from a word or a register. This 'extract' feature can also be provided in a variety of different ways including a special register designed to provide extract capabilities or have hardware assisted features which decode target machine instructions by extracting various fields. In the case of implicit shifters, the capability of pre or post shifting, along with pre or post masking of data to and from the ALU is useful for calculations involving variable bit arithmetic, bit testing, etc.

Fig. 3.4 illustrates some common ALU structures found in current m-computers (5).

Other types of functional units are also available on m-computers. A common set of units found on some machines is the multiply and divide unit. This may also be present implicitly in a machine by providing a microperation which does the basic multiply-step/divide-step used in a repeated addition/subtraction algorithm for multiplication/division.

3.2.3 DATA WIDTHS

Data widths characterize the machine according to the width of the previous two features and the busses connecting them. The target data width can be smaller, equal to or greater than the data width available on the host. This poses a problem in emulation which can be solved in various ways, e.g., by masking, shifting, iteration, etc. Alternately, physical concatenation of modular units can be used to achieve wider data widths. However, once configured, the host data width is permanent since it is hardwired, but the emulated target machine data width may be multiples of the host machine.

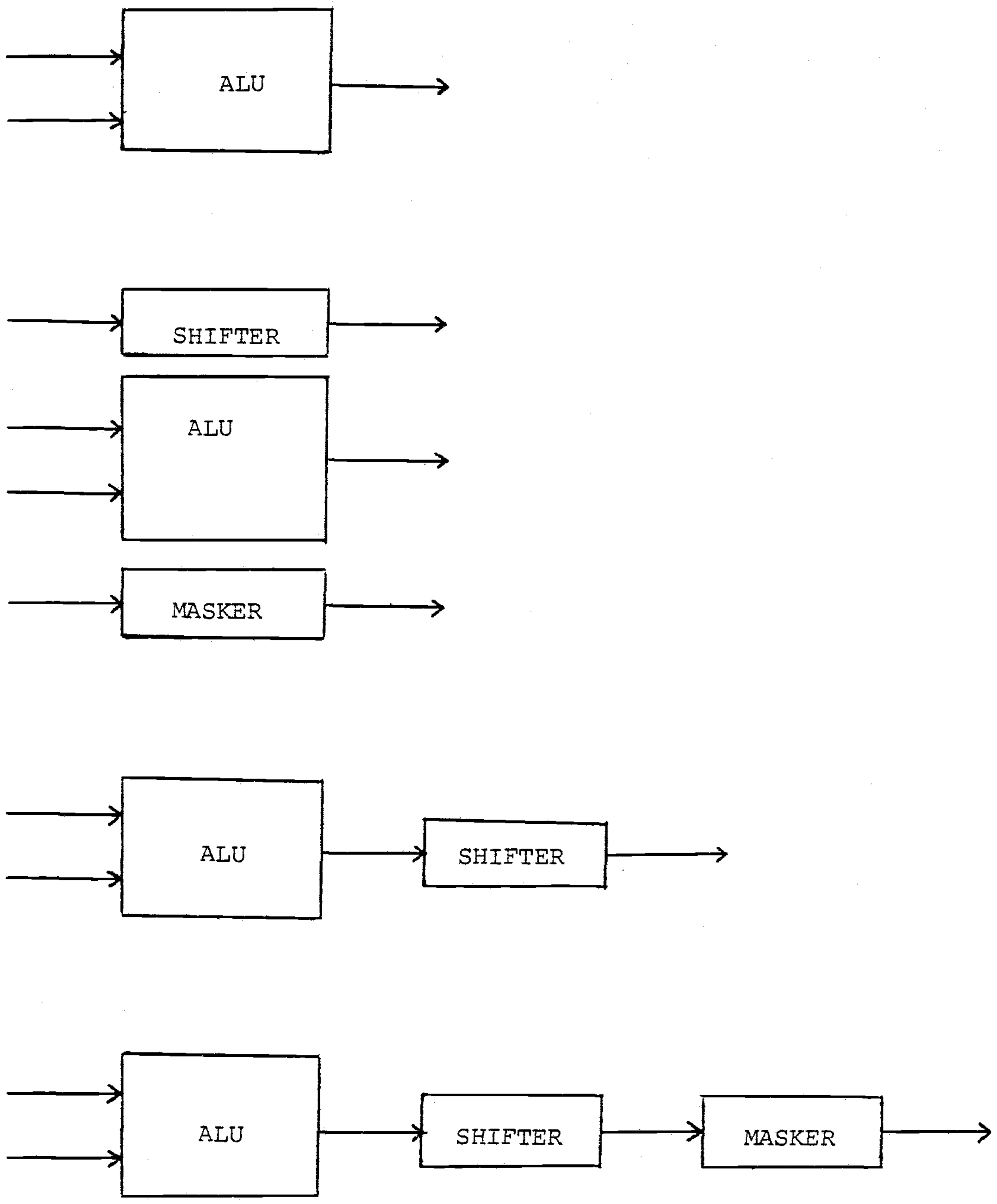


Fig. 3.4 SOME ALU STRUCTURES

3.2.4 BUSSES

Often, all the functional units and storage units are connected to each other by a number of data paths called busses. The bus structures, in terms of the number of busses and their width, affects the flexibility and ease of microprogramming the machine. By generality we mean that this aspect of host machines can ease the process of emulating targets with a wide variety of (macro) instruction formats and addressing modes. These two aspects of the bus structure also affect the addressing and microinstruction format which in turn leads to generality of the machine. At times, the bus structure is alterable in that connections can be set up and altered under microprogram control through special purpose local storage registers called residual control registers.

3.2.5 MICROINSTRUCTIONS

The mechanism through which the microprogrammer exercises control of the various facilities available on the machine, mentioned in the previous features, is obtained through microinstructions. Each facility in an m-computer is either directly or indirectly controlled by micro-operations which in turn are combined together in different ways to produce microinstructions. We can classify microinstructions depending on the fixed number of micro-operations per microinstruction.

1) Vertical microinstruction format represents at most one or two micro-operations and resembles classical machine language instructions.
2) Horizontal microinstruction format represents several micro-operations. Thus a single horizontal micro-instruction controls a number of hardware facilities at the same time.
3) Diagonal microinstruction format lies between the two extremes. It has a limited capability to perform a few micro-operations in parallel.

Micro-operations within microinstructions are represented either by single bits or multiple bits known as fields. The latter approach is known as encoding of micro-operations. In the simplest design, reminiscent of Wilkes' original proposal (1), there is no encoding at all and each bit represents one micro-operation (direct control), i.e., controls one resource or operation in the machine. Typically, single level or direct encoding of micro-operations is represented as a field within the microinstruction. Bits that control mutually exclusive resources, such as operations that an ALU can perform, are combined into fields as shown in Fig. 3.5.

The next step is to have two level or indirect encoding wherein mutually exclusive micro-operations may be combined to form fields. The meaning of a field depends on the value of another field. This scheme is known as bit steering. The various forms of microinstruction encoding are shown in Fig. 3.6.

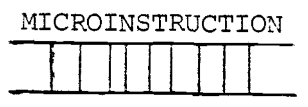
The most common technique for microinstruction sequencing is to include the address of the next microinstruction within the current microinstruction. A second technique is to provide an incrementing capability for the microinstruction address register. Conditional and unconditional branches are implemented in both sequencing techniques. By storing the next address in the microinstruction, multiple or n-way branch capability is obtained. This is a useful feature to have in m-machines used for general purpose emulation.

3.2.6 INPUT, OUTPUT & INTERRUPT STRUCTURE

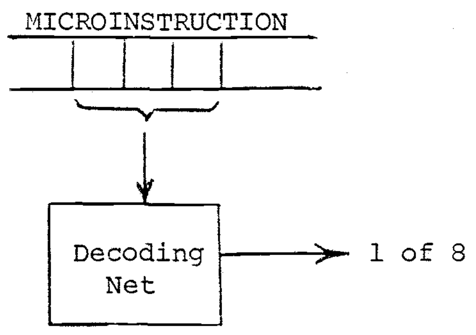
Requirements for input/output facilities and an interrupt structure depends on the environment in which the machine is to be used. For general purpose microprogramming any 'reasonable' I/O and interrupt structure provides the required support. However, if the machine is to be used for general purpose emulation the requirements for I/O and interrupt structure are very demanding. Mapping of target I/O facilities to host I/O facilities must be complete. This means that the host should allow the emulation of all target I/O facilities and I/O activities.

Interrupts should be soft (38) for the sake of generality. This means that interrupts set up bits for examination rather than forcing

NO ENCODING	ENCODED	ALU OPERATION
00000001	000	1- ADD
00000010	001	2- SUBTRACT
00000100	010	3- ADD WITH CARRY
00001000	011	4- SUBTRACT WITH BORROW
00010000	100	5- AND
00100000	101	6- OR
01000000	110	7- COMPLEMENT
10000000	111	8- COMPLEMENT + 1

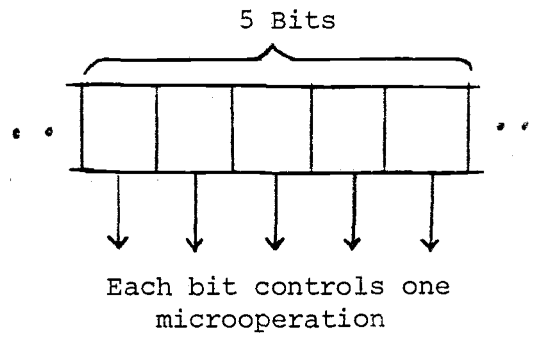


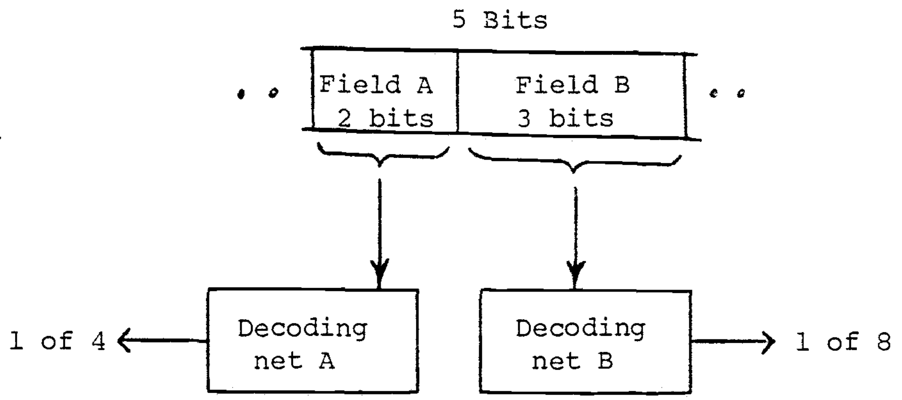
(a)



(b)

Fig. 3.5 NO ENCODING (a) VS SINGLE LEVEL ENCODING





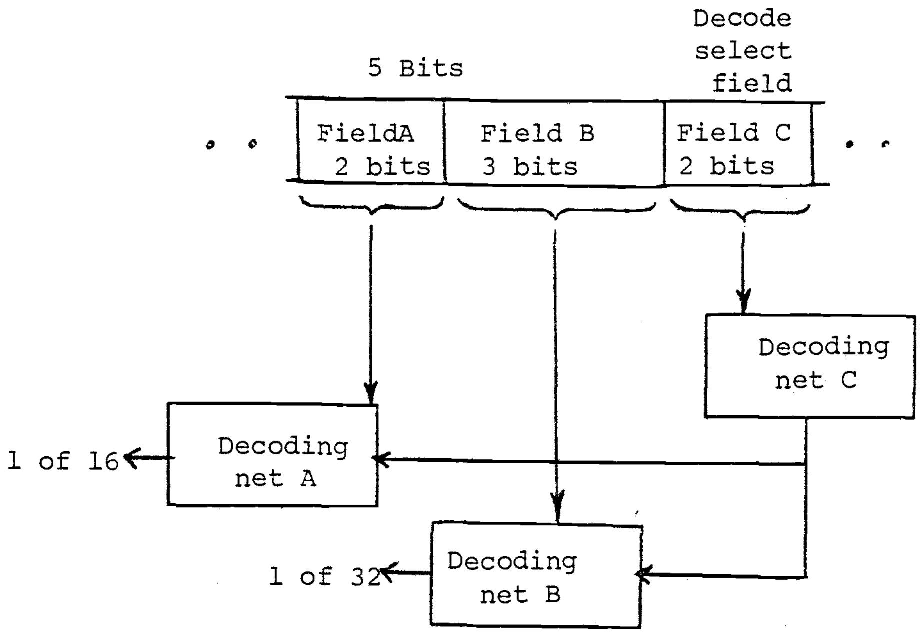


Fig. 3.6 MICROINSTRUCTION ENCODINGS:
a, NO ENCODING
b, SINGLE LEVEL ENCODING
c, BIT STEERING

predetermined control storage transfers or any other equivalent predetermined (built in) activity. Thus the microprogrammer has the flexibility in handling them according to his needs or to be more accurate, according to the needs of his emulated target.

3.3 m-COMPUTER PRIMITIVES

We can arrive at some interesting and more specific conclusions about common characteristics of m-computers, now that we have identified their various architectural features. These characteristics form or identify the primitives of the host for which we seek to generate efficient VMPL code. We will make use of them when we identify primitives to manipulate the hardware. Essentially, this latter design is a syntactical and semantic description of a language. Hence, the design of microprogramming language primitives is achieved by associating or mapping the code generation primitives that we derived in chapter 2 onto the 'hardware'primitives that exist for a class of m-machines. The basic characteristics of m-computers belonging to the class that VMPL is portable over are:

1) Word-oriented, Von-Neuman type fixed data width m-machines (however, there are large and small data width machines).
2) Local storage limited to a few (8-16) general purpose registers.
3) Input-Output and interrupt structure mostly hard and very much integrated into the machine architecture.
4) Microinstruction format (horizontal, diagonal or vertical) not limited to one type.
5) Microinstructions belong to four classes: 1) data modifications, 2) data transfer, 3) transfer of control and 4) machine control.
6) Subroutining and functional calls may not be present at the host level.
7) The most general ALU structure which completely covers all ALU structures is shown in Fig. 3.7. Most m-computer ALU's are a subset of this structure.

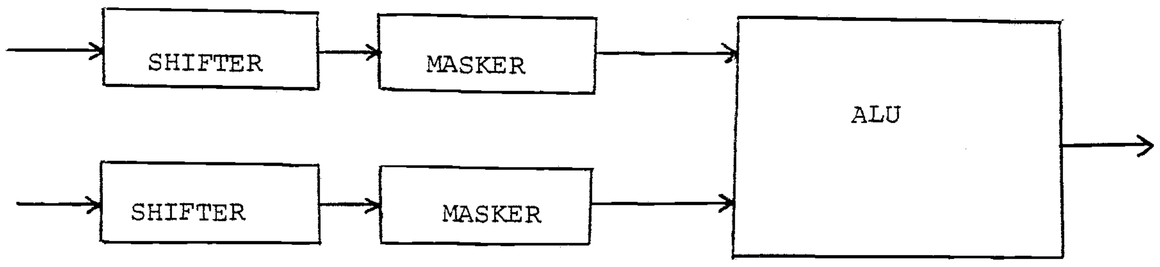


Fig. 3.7 A GENERAL ALU STRUCTURE

8) An n-way branching instruction may be included in the micro-instruction repertoire.

9) Iterative, shifting and masking capabilities, which might be present would make the effective data widths variable in nature.

10) Generally, only one set of status flags is present. Furthermore, flag generation for variable length operations is not available (implementation can be obtained because of 8 above).

The first characteristic means that the m-machines are not stack or parallel in nature. In general the machines are word oriented in that the operations on the data, and the data widths available on the machines are equal to the word size of the machine. As such, byte (or any other word size multiple) operations have to be explicitly executed by the microprogrammer by masking and shifting. What this characteristic actually indicates is that the experience gained from the design and implementation of compilers for higher level languages for conventional computers applies to the design of a compiler for VMPL too.

Data widths of m-computers reflect on the class of computers which can be considered as targets for the purpose of emulation. As an example, if all m-computers were eight bits wide with no capability of repetitive (or pseudo iterative obtained via masking and shifting) control then the only computers which could be emulated (serve as targets) would be eight bit computers.

Since local storage is generally limited to a few registers, emulation of computers with a large number of registers becomes difficult. However, there are exceptions to this case. MATHILDA (35), a dynamically microprogrammable m-computer, has large amounts of local storage organized as a hierarchy of registers. Special purpose registers are very limited in nature, like the stack control registers on the MICRODATA 3200, the floating point registers on the INTERDATA 8/32, the extract registers on the Burroughs 1700.

The problem with I/O is its particular machine dependent structure at the micro-level. A few m-computers (HP 21-MX, VARIAN 73, PDP 11/40E, INTERDATA 8/32, etc.) do their I/O via a set of busses (command, data, address, status, etc.) a few others (MICRODATA 3200, CAL DATA PROCESSOR, AMP, etc.) use memory mapped I/O, and yet another group (QM-1, META 4, INTERDATA 85, PRIME 300, MATHILDA, etc.) do their I/O via special purpose registers. All these variations suggest a lack of a primitive I/O structure associated with m-computers. In fact, the memory mapped case is very restrictive as far as emulation of general target machines is concerned.

Interrupt structures also vary a lot between different machines. Interrupts can be soft (MICRODATA 3200, MATHILDA, QM-1) in that they are under the control of the microprogrammer, partially soft (VARIAN 73, HP21MX) wherein software or hardware logic can be provided to mask out or conditionally jump on interrupts, or hard (PRIME 300, PDP 11/40-E) wherein hardware logic determines a fixed set of trap address in case of an interrupt. This last case is obviously the most restrictive for general purpose emulation.

With respect to the fourth characteristic, microinstruction format, we can assume that all m-computers have horizontal microinstructions, since this structure essentially covers the other two cases. That is, diagonal microinstructions are a degenerate case of horizontal microinstructions and vertical microinstructions are a degenerate case of diagonal microinstructions. Thus we can limit our treatment to m-computers with horizontal microinstructions.

The fifth characteristic, classes of microinstructions, indicates that the same type of microinstructions are found in most m-computers. Thus the task of designing a high level language processor is simplified if an intermediate language compatible with these microinstructions is designed. For example, the branch-on-decode opcode process of emulation can be efficiently realize via an appropriate intermediate language construct because of the presence of the n-way branch microinstruction found in most m-computers.

Iteration, shifting and masking help achieve emulation of a target machine on a host whose data widths do not match the target. This feature is a requirement for the so-called universal emulator. (39) Conversely, the sixth characteristic, subroutining, points to those features that are not available to the microprogrammer. A high level language creates a virtual machine with language constructs such as subroutine, function calls. Furthermore, most m-computers do not have all operations associated with programming languages, e.g., multiply and divide.

At what level should high level language constructs not present in the host machine be handled? Should the programmer (in the high level language) be concerned about these things or should the compiler handle these deficiencies? In order to keep the programmer away from the machine the compiler should handle all those situations which can be recognized and solved effectively and in an efficient manner.

Although a variety of ALU structures are present, we can basically recognize the presence of an ALU, shifter and masking units. The shifts generally are single bit and can be achieved separately from the ALU. The masking unit is usually associated with the output of the ALU but in general is implicitly provided via the AND microinstruction of the ALU.

Status flags create a problem for general purpose emulation. Not only are the ways in which the status flags get activated non-uniform across the different m-machines but no machine other than the MICRODATA 3200 actually provides two sets of flags which can be used separately by the host and the target machine. Furthermore, even if iterative control is available, status flag generation is limited to fixed length data. Thus variable precision operations--because of host target data width mismatches--at the microlevel are difficult to achieve.

All these statements illustrate the need for sophisticated software which can minimize the differences, supplement the deficiencies and recognize the variations inherent in m-computers.

3.4 m-COMPUTER CLASSES

We can classify m-computers by their architectural features according to the following overlapping classes:

1) CLASS 1 Computers that are microprogrammed.
2) CLASS 2 Computers that are virtual machine dependent.
3) CLASS 3 User microprogrammable machines.

4) CLASS 4 General purpose emulation computers.

5) CLASS 5 High level language support computers.

6) CLASS 6 Bit slice architecture based machines.

3.4.1 CLASS 1

Class 1 m-computers cannot be microprogrammed by their users. Computers belonging to this class have a control store made up of read-only-memory. The IBM 2050 'host' for the System 360 Model 50, and the PDP 11/40 are two computers that belong to this class.

3.4.2 CLASS 2

The next class of m-computers to be considered are those that have been designed with the primary purpose of implementing a particular target machine architecture. These machines often support limited user microprogramming. Thus this is the class of virtual machine dependent hosts.

Most of the features like data widths, word and byte operations, bus structure, input & output, interrupt structure, in these computers are reflections or direct mappings of the architecture of the target machine. In general these machines have more to offer than the original machine they emulate. They may have more hardware functional units and software, to implement extra user level machine instructions. Furthermore, additional read write control is provided for the user to implement his own instructions. These enhancements are the primary reason why such machines are available in the market and are in general a cost effective buy over the original machine. Although these machines are called 'general purpose' emulators by their manufacturers, they are not so because of their 'machine dependent' nature.

INTERDATA 8/32 and CALDATA 1/35 are two machines belonging to this class. INTERDATA 8/32 has the same basic system level architecture as the system 360 series. Data width, ALU operations, input & output and interrupt handling is the same as in the 360's. Special hardware features are provided for machine language instruction decoding and are tailored for the 360 instruction set. Provisions for more than a single set of registers useful in a multiprogramming environment and hardware multiply & divide are enhancements over the earlier models of the 360. The standard emulator for the 360 instruction set uses only part of the total available control store, so the rest is available for user microprogramming.

CALDATA 1/35 emulates a PDP 11/35 which is a low end version of the PDP 11/40. Keeping in line with Digital Equipment Corporation (manufacturers of PDP computers), all devices on the CALDATA are addressed via the MACROBUS (equivalent to DEC's UNIBUS). The word size, number of registers, etc., are the same. Input-output and interrupt handling is hardwired according to PDP 11/35 specifications. Of course, a writable control store is provided which makes the computer user microprogrammable.

3.4.3 CLASS 3

This is the class of user microprogrammable computers. There are two categories of these machines. To one category belong those computers which are general purpose machines. They are provided with a variety of microinstructions (horizontal or diagonal in nature), some general processing facilities, 8 to 16 general purpose registers, etc. Interrupt and I/O handling is partially soft. Usually an emulator for a machine language and other software development aids are provided for user microprogramming. It is important to note that the emulator is designed 'after' the machine has been constructed, i.e., the micro-architecture is quite general and not restricted to the requirements of the machine language.

The other category is of special purpose machines which are simple in structure (usually vertical microinstructions) and are used as peripheral processors, signal processors, etc. They offer the flexibility which a hardwired machine doesn't have and are cost effective alternatives to the general purpose m-computers discussed above.

HP 21-MX and CASH 8 belong to this class. HP 21-MX is a general purpose 16 bit computer with diagonal microinstructions. It does provide some facilities to handle machine language instructions. However, these facilities are very general and can be used effectively by a microprogrammer. Interrupt handling is partially soft and input & output is done via a main I/O bus.

CASH 8 is an inexpensive microprogrammable computer. It is used most frequently as a substitute for hardwired process controllers. It is extremely simple in architecture (doesn't have a main memory) with very few restrictive facilities.

3.4.4 CLASS 4

This is the class of computers which can be used as general purpose emulation machines. There are essentially two categories of architectures which fit this class.

In the first category, we have m-computers with a large and flexible local storage and enough processing facilities to efficiently emulate various user level architectures with ease and efficiency. Emulation of various word widths can be done easily by providing residually controlled masking and shifting capabilities. The input & output and the interrupt structure is soft to be tailored according to requirements.

In the m-machines of the second class the microarchitecture can be designed by the user. This is done by providing an architectural level below the micro level known as the nano level and correspondingly nano-architecture. Data widths, input & output and interrupt handling are determined by nano-instructions which make the machine at the micro-architecture look very soft. This two level structure permits flexibility in the design of upper level microinstructions which in turn make the design of machine language instructions simpler.

MATHILDA and QM-1 are two computers belonging to this class. MATHILDA is a general purpose dynamically user microprogrammable computer with an abundance of local storage and provision for masking and iterative control that make this computer suitable for general purpose emulation.

Input-output registers are provided for general purpose I/O transfers and a soft interrupt structure is available.

QM-1 is also a general purpose m-machine employing two levels of control. The microlevel architecture is defined by nanoprograms residing at the nano level. Data widths are fixed but iteration is available via automatic repeated execution of the nano instructions. I/O and interrupt handling are soft.

3.4.5 CLASS 5

This class of machines is the class of so-called high level language (hll) support machines. These machines have features built into them which support a particular style of language. Essentially, the various data types of a language require some kind of iterative or residual control for 'varying' the data width dynamically. A block structure language is easily implemented on a stack-like architecture, i.e., the hardware support of a stack at the microarchitecture level.

BURROUGHS 1700 and MICRODATA 3200 are high level support machines. The BURROUGHS 1700 was designed to support a variety of intermediate languages known as s-languages (34). Various high level lanaguages are compiled to different s-languages which are then executed on the computer. Main memory is bit addressable and the processing facilities can be used iteratively, providing any required effective data width. Residual control is provided to configure the machine into a structure which is most efficient for a particular s-language. All I/O and interrupts are soft.

MICRODATA 3200 is provided with an emulator for a stack oriented machine. The emulator is known as the 32/S. The architecture of the 32/S was designed in conjunction with MPL-Microdata Programming Language, a machine independent high level programming language. Since MPL is a block structured language, the stack on the 3200 is used extensively. This is an example of direct hardware support for a high level language.

3.6.6 CLASS 6

This class of machines is a set of families of integrated circuit 'chips' available in the market and microprogrammable by their nature. They are known as bit slice processors. The ALVs are essentially parts of registers, functional processing units, etc., 2 to 4 bits wide. Any number of these can be cascaded together to produce the desired word width. Also any support circuitry comprising extra registers, input & output and interrupt handling can be attached to the machine to form a variety of architectures. Thus tailoring is achieved using other support chips. Many architectures can be emulated as long as the chips are connected together as required. INTEL 3000 and AMD 2900 series of IC's belong to this class of machines.

3.5 CONCLUSIONS

In this chapter we have recognized some of the hardware primitives and constraints we are going to work with. These can be summarized as follows:

1) Word oriented, Von Neuman type machine architecture.
2) Fixed data widths.
3) Limited local storage.
4) Hard Input & Output and interrupt structure.
5) Conventional microinstructions.
6) No subroutining facilities.
7) Limited number of functional units.
8) N-way branching.
9) Iteration, shifting & masking capabilities available.
10) Only host status flags available.

We also categorized m-computers into 6 classes. In the rest of this thesis all references to m-computers will mean computers belonging to classes 2 to 6, i.e., those m-computers which are user microprogrammable in nature.

4. PORTABILITY

4.1 INTRODUCTION

This chapter is concerned with solving the third problem which is identified in Chapter 1. The statement of the problem is:

> A general tool (for microprogramming) has not been provided. Specifically, the problem of portability has not been addressed or solved effectively (in earlier attempts for high level microprogramming languages).

Portability has been defined in several ways. According to Waite (57), the portability of a program is a measure of the ease with which it can be implemented on a new machine. Another definition is that portability is the property of a system which permits it to be mapped from one environment to a different environment (58). Although these qualitative definitions sufficiently describe the problem, Brown (59) established a more quantitative measure:

> A program or programming system is called portable if the effort required to move it into a new environment is much less than the effort that would be required to reprogram it for the new environment.

4.2 CLASSIFICATION OF TECHNIQUES

Techniques for transferring software from one environment to another can be divided into two general categories. These two categories are:

1) Conversion techniques
2) Inherently portable techniques

4.2.1 CONVERSION TECHNIQUES

Within this category come all techniques which accept programs which were written for a particular system without any forethought of portability. This means that the requirement to transfer the programming system is not considered at system development time so as to provide inherent portability properties. Consequently, such systems are transferred by various conversion techniques. Some of these techniques produce an equivalent program for the new machine via decompilers, while others simply attempt to execute the original program (program written for machine A) via emulation or simulation (for machine B).

Although these techniques have been used for some programming systems, it seems obvious that inherently portable systems provide a more general and desirable solution to the portability problem.

4.2.2 INHERENTLY PORTABLE TECHNIQUES

From the discussion on conversion techniques, one comes to the conclusion that those techniques are not the real solutions to the problem of portability simply because the problem is not an initial design constraint. As indicated by Halstead (58), a procedure which avoids a problem instead of solving it is often the best solution after all. Inherently portable techniques as such attempt to facilitate the transfer problem by writing the original code in such a way that it will 'inherently' (by its very nature) be easy to transfer.

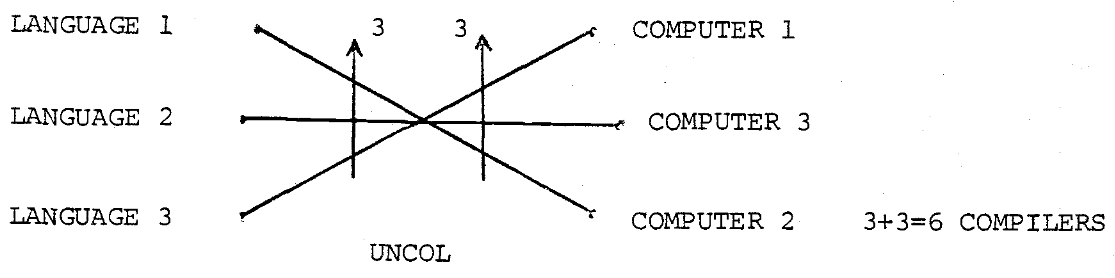
This is very much the way that 'machine independent high level' languages attempt to solve the portability problem. However, because of the various and varied implementations of these languages on different computer systems, the portability problem is only partially solved (60). Lack of universal standards, provisions of extensions by vendors which are usually machine dependent and efficiency considerations have made simple recompilation of software to facilitate the transfer from one system to another impractical. Thus, programs written in these languages cause invalid results when run on one system even though they ran correctly on the original system.

The problem of providing compilers for many high level languages on many machines is the basic goal behind the UNCOL (Universal Computer Oriented Language) concept (42). UNCOL was a language proposal intended to provide a common path between many high level languages and many machine languages. Each high level language could be translated into UNCOL once and in turn one UNCOL translator could be written for each machine. Thus, in order to implement M high level languages on N distinct machines, we would require M + N compilers. This implementation effort would be considerably less than that of conventional compilation techniques which would require M * N compilers (42) Fig. 4.1.

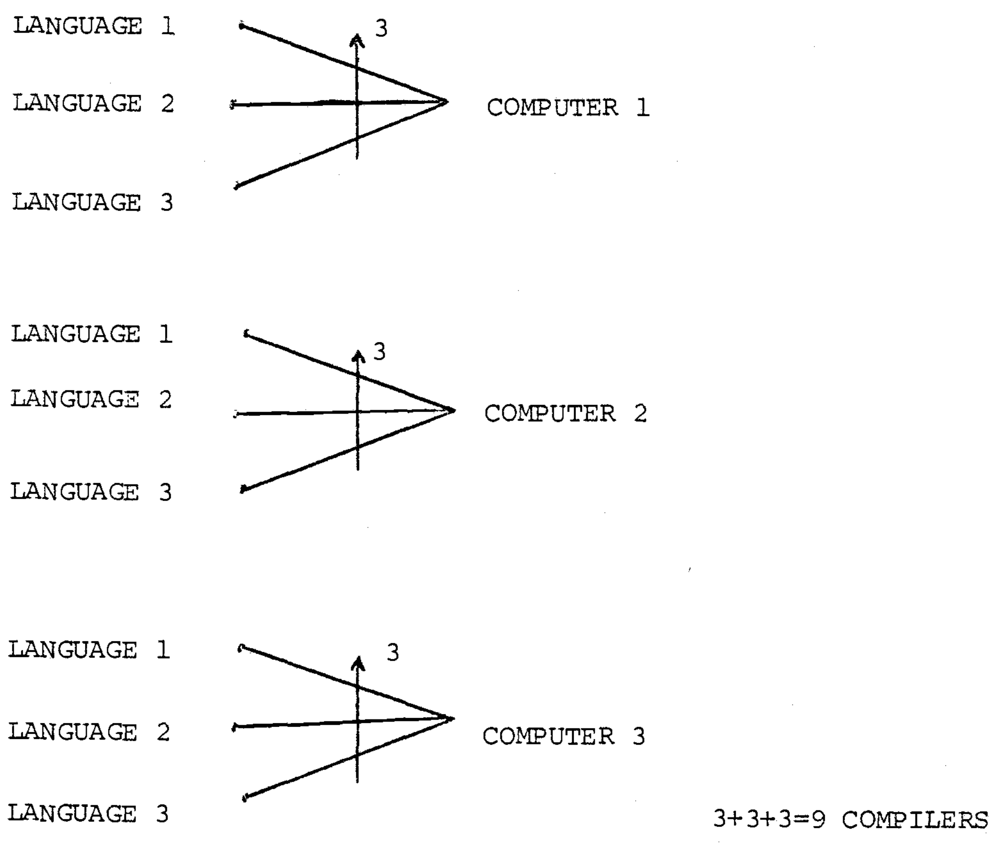
Although several attempts have been made in this direction, the need for UNCOL to be 'universal' is the main reason for its lack of success. The only viable example of this approach is the implementation of one high level language on various machines. PASCAL (56) is easily, i.e. with little effort, implemented on different systems. A compiler for PASCAL written in PASCAL is available in an abstract intermediate language called p-code. In order to implement PASCAL on a system, the implementor has to write a simulator for p-code. He can then transfer the available p-code program (actually the PASCAL compiler) to his system. At this point, he has a working PASCAL compiler on his system. The effort involved in writing the simulator (for p-code) is quite small as compared to writing a compiler for PASCAL.

In bootstrapping the compiler for a language, L is realized in a subset of that language. Then the compiler for this subset is realized in a low level language for the machine on which it is desired to implement the language L. The language is portable because the only effort required to implement L on any machine is the realization of the translator for the subset. The main drawback of this technique is that most high level languages (least of all subsets of high level languages) are not well suited for writing compilers for themselves. As such, a variety of modifications to the pure bootstrapping techniques have evolved (61).

One variation is the development of several special languages which are well suited for the first stage in the bootstrapping technique. Another variation makes use of several languages in a hierarchical structure. Only the lowest language need be mapped into a new machine in order to initiate the bootstrapping sequence.



(a) UNCOL USED AS INTERMEDIATE LANGUAGE



(b) DIRECT IMPLEMENTATION OF LANGUAGES

Fig. 4.1 HIGH LEVEL LANGUAGES AND COMPILATION

'Abstract machine modelling' is described by Newey (62) as a procedure based on the concept that the fundamental data types and operations required to solve a particular problem define a special purpose computer (the abstract model) which is ideally suited to that problem. The algorithm for obtaining the solution can be encoded as a program for this 'abstract machine model.' In order to obtain a running version, the abstract machine model is realized on an existing computer by implementing its basic data types and operations. The term 'descriptive language' instead of 'abstract machine' has been used by others (59). We will refer to these models as the Intermediate Language Models and the language describing the model as the intermediate language.

In some cases only one level of intermediate language is required to implement the compiler. This results in a very simple variation of the bootstrap technique described previously. However, an attractive alternative exists in the development of an entire hierarchy of intermediate languages to provide convenient steps from the high level language for describing the compiler to the lowest level to be bootstrapped.

The objective of the hierarchy is to ease the implementation of each intermediate language and to obtain a 'base' intermediate language which is easy to implement on the target computer. It is important that this base language be easy to implement because it is at this level that true portability is determined. This is so because after the hierarchy has been completed once, the only requirement for transferring the system to a new machine is the realization of the base intermediate language on the new machine (63).

4.3 APPLICATION TO MICROPROGRAM PORTABILITY

We have reviewed some of the techniques used to transfer computer software from one machine to another. These include methods of conversion and the production of inherently portable software. Unfortunately, none of these techniques provide a complete and general solution to the microprogram portability problem. Perhaps a combination of techniques will provide a reasonable solution to transfer programs; namely, a machine independent language using a multiphase compilation system. In such a combination, the compiler works downwards through a series of intermediate microprogramming languages for abstract machines. In the last phase, the compiler generates target machine dependent code. This last phase would have to be created for each target m-computer. This procedure is adopted by Mallett and Lewis (7) and is the one we use for producing inherently portable microprograms. In the next section we briefly describe their model for the compilation system.

4.3.1 THE MALLETT-LEWIS MODEL

This compilation system has three phases as shown in Fig. 4.2. The first phase is the machine independent part. This consists of a compiler that does the syntax and semantic analysis, translating the high level language through various levels of intermediate languages, as needed, to produce a sequential string of micro-operations for an abstract m-computer called the intermediate microprogramming language machine (IMLM).

Next, a machine (IMLM) dependent concurrency recognition and optimization algorithm composes the sequential micro-operations in the IMLM horizontal format. The last phase is the code generation part of the system and is machine dependent. For each host m-computer a machine dependent interface is needed. The interface programs will decompose (if necessary) the optimized IMLM microinstructions and translate them into host microinstructions. A final code optimization pass produces horizontal microcode.

The most important component of this system is the intermediate microprogramming language (IML). In the next chapter we describe the particular IML which we have used for our compilation system.

4.4 CONCLUSIONS

In this chapter we have discussed the portability problem in general and looked at the various solutions available. We have adopted the model of multiphase compilation proposed by Mallett and Lewis (7). Thus, the third problem in this chapter (and Chapter 1) as stated is solved via

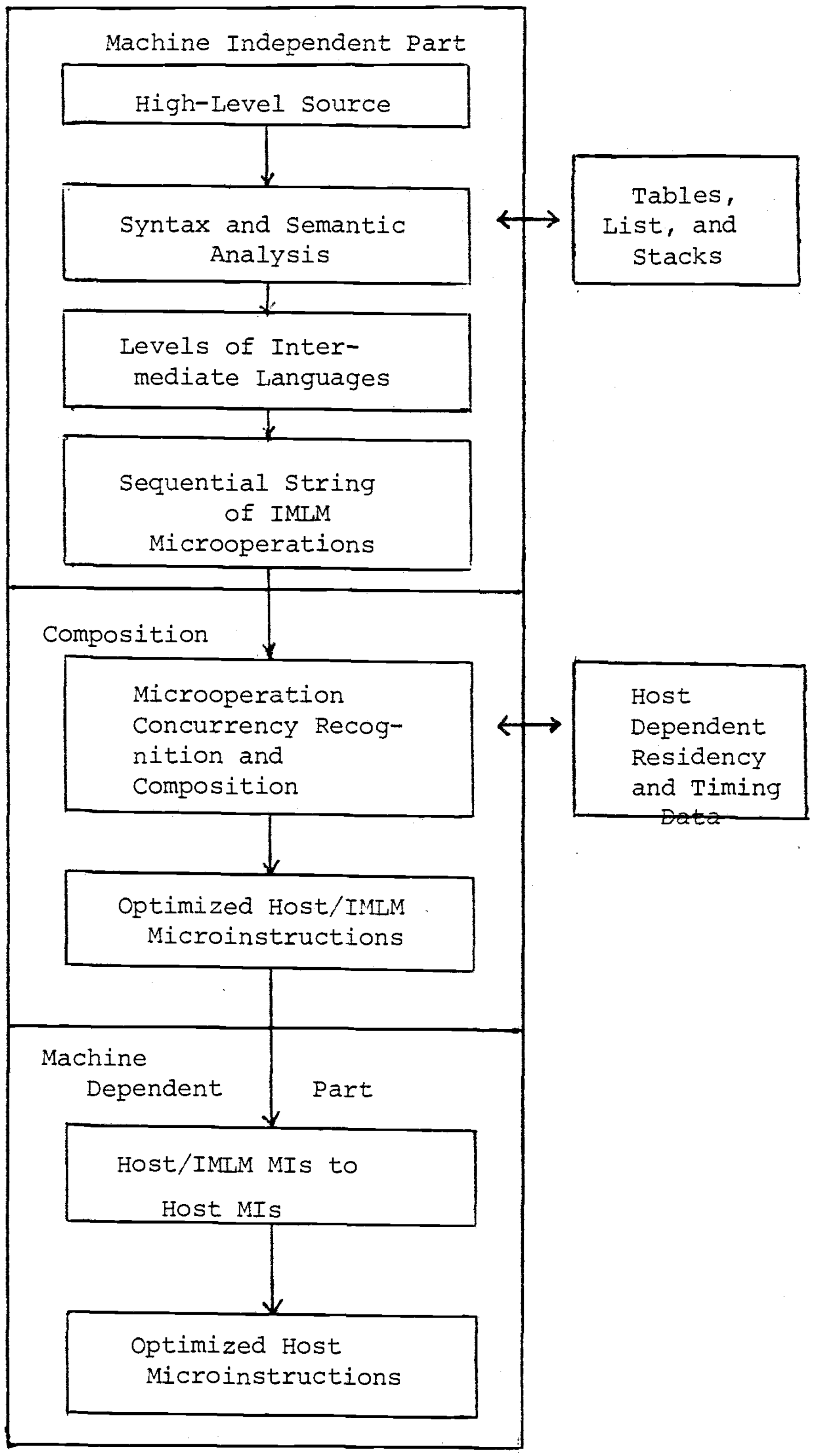

Machine Independent Part
High-Level Source
Syntax and Semantic Analysis
Tables, List, and Stacks
Levels of Intermediate Languages
Sequential String of IMLM Microoperations
Composition
Microoperation Concurrency Recognition and Composition
Host Dependent Residency and Timing Data
Optimized Host/IMLM Microinstructions
Machine Dependent Part
Host/IMLM MIs to Host MIs
Optimized Host Microinstructions

Fig. 4.2 THE MALLETT-LEWIS MODEL OF A TRANSLATION SYSTEM FOR PRODUCING INHERENTLY PORTABLE MICROCODE

the following implication:

IMPLICATION 9: Machine independency (portability) is obtained by compiling the language into an abstract intermediate machine language (IML). The intermediate language is then compiled into microcode.

5. INTERMEDIATE MACHINE LANGUAGE

5.1 INTRODUCTION

The use of an intermediate machine language (IML) in this research has greatly facilitated the translation process from a higher-level machine independent language into the final machine-dependent microcode. The Mallett-Lewis model (chapter 4) requires an intermediate language. The question now is what kind of intermediate language format should be used? In the next section we present a brief summary of the various intermediate language formats which are available and results obtained by various people as to their usage.

5.2 IML FORMATS & ATTRIBUTES

In (66) Lawson discusses the pragmatics of various intermediate language text representations (formats), along with their hardware implications. The five types of IML representations which he describes are:

a) QUADRUPLE OR THREE OPERAND FORMAT: A quadruple is composed of an operation code, and one to three operands. The basic structure for the quadruple intermediate text is:

```
        op-code      1st operand      2nd operand      3rd operand
```

The first two operands are essentially the sources and the third operand is the destination of the result.

b) TRIPLE OR TWO OPERAND FORMAT: A triple is composed of an operation code and one or two operands. The basic structure for a triple is:

```
        op-code      1st operand      2nd operand
```

In the case of the operation specified for the triple requires more than two operands, the third operand is on the top of the stack. Thus, this representation assumes the presence of a stack.

c) DUO OR ONE OPERAND FORMAT: A duo is composed of an operation code followed by one operand. Its basic structure is:

```
        op-code      operand
```

When the specified operation requires more than one operand, the other operands are assumed on top of the stack.

d) POLISH NOTATION: There is no rigid structure that determines a fixed relationship between operation codes and operands. Each position of the IML stream is examined to determine whether it is an operation or an operand. However, the representation does require the presence of an operand stack.

e) PROGRAM TREE: The elements of the IML stream (the IML instructions comprising the program) are not necessarily located in contiguous locations. Each node in the instruction stream must be examined to determine whether it is an operation or an operand. In this respect, it is similar to the polish notation. This notation also requires an operand stack.

Four measurement units are used to indicate the differences between the five IML text representation. These are:

1) Number of instruction elements: This is simply the number of unique units of information in the IML stream, i.e., an operation code, operand, etc.

2) Instruction increment size: This measurement indicates the number of units that must be skipped by the instruction fetch-decode unit of a computer to access the next element of the instruction stream.

3) Number of units directly examined: This measurement indicates the number of units that must be examined by the instruction fetch-decode unit of a computer to determine the composition of the instruction.

4) Stack requirements: This measurement indicates what stacks are required for each of the IML format and the type of stack usage.

Lawson's (66) results for the five IML formats are given in Table 5.1. These results are based on the IML representation of one assignment statement which is similar to those used in several high-level programming languages:

$$X = A * B/(C+D)$$

With these measurements in mind, we may now evaluate and contrast the attributes of the various IML formats.

TABLE 5.1 INSTRUCTION-STREAM COMPARISON FOR THE ASSIGNMENT STATEMENT

INSTRUCTION-STREAM FORM	NUMBER OF INSTRUCTION ELEMENTS	INSTRUCTION INCREMENT SIZE	NUMBER OF UNITS DIRECTLY EXAMINED	STACK REQUIREMENTS
THREE ADDRESS	16	4	4	NONE
TWO ADDRESS	12	3	12	1 (For Temporaries)
ONE ADDRESS	12	2	12	1 (For a Push-Down Accumu-lator)
POLISH	9	1	9	1 (Operand Stack)
PROGRAM-TREE	27	VARIABLE	9	2 (Operand & Operator Stack)

1) CONCISENESS

Conciseness is a measure of the number of IML elements required to represent the decomposed form of the high level language statement. The order of conciseness is as follows:

Polish; one address, two addresses, three addresses, program tree.

2) COMPLEXITY IN INTERPRETATION

This complexity deals with the work which the instruction fetch-decode unit of a computer has to perform in decoding the instruction and preparing the operands for the execution of the instruction. The three address format is the least complex since the instruction fetch-decode unit does not have to examine each unit (operation code, operand) to determine its class. The polish and program tree notation on the other hand, are the most complex since each unit of the IML stream has to be examined to determine the class of the element (i.e., operation-code or operand). The complexity ratings for the five formats are:

three addresses; one address, two address; Polish; program tree.

In another paper by Merwin (67) three IML formats are compared for execution speeds. The three formats selected are quadruples, triples and duos. Four programs written in FORTRAN were used to evaluate the performance of the three IML representations. It was found that the quadruple format is the most 'compact' when compactness is related to the execution time of the IML instruction stream of the four programs.

From the data which is collected in these two papers, we come to the following conclusions:

1) The quadruple (three address) IML format takes the least amount of execution time.
2) The polish IML format is the most concise. This means that when compiling a high level language statement into the IML statement, the polish notation produces the least number of IML units. In other words, the resulting polish statements are nearest to the original high level language statement. We will show later how this 'nearest' criteria is an

important requirement for the IML which we select.

3) Other than the quadruple format, all IML formats require a stack.

Eckhouse (12) was the first to use the concept of an IML for the compilation of a high level microprogramming language to microcode. He uses SML (13) as an intermediate language for the conversion of MPL into microcode. SML resembles the code of a single-address computer. There are only two types of instructions in SML--register loads and stores, and execute operations. Arguments needed for operations must first be loaded in argument or A-registers. Results of operations are left in result or R-registers, and may be stored in other registers or memory locations. Temporary or T-registers are available for intermediate results.

SML fails in solving the register allocation-deallocation problem. The compiler can use any number of A & T registers and no information is kept of the variables and the corresponding A & T registers in which they reside. This produces a lot of redundant load and store instructions at the SML level. These redundant load and store instructions then migrate to the microcode level producing non-optimum microcode. Furthermore, the SML code produced is not highly suited for two or three address m-computers.

From the initial discussion, it is obvious that we should not use the two address, one address, polish & program tree IML formats. The main reason for this decision is the requirement of a stack for their execution. Thus, the microcode which would be produced on further compilation of these formats would be well suited (efficient) for m-computers with stacks available at the microlevel. The availability of a stack at that level is not a primary characteristic of m-computer architecture (Chapter 3). MPL (18) provides us with a case where a one address IML format is used as the intermediate language. The intermediate language is executed on the MICRODATA 3200 by a set of microprograms. Central to execution of this one address IML (as indicated earlier) is the stack capability which is a hardware feature of the 3200 at the micro-architecture level. Thus, the hardware feature (stack) is used by the micro-programs in executing the IML instructions.

We did some tests on the two formats to further justify the usage of the three address IML format over the one used by Eckhouse.

A high level VMPL program (VM1-Appendix C) was taken and broken into three equal parts. These three parts were then coded or compiled down to a three address IML and a single address SML like language. Fig. 5.1 shows a piece of the VMPL codes and its corresponding three address and single address equivalents. The three address IML constructs shown are the actual three address IML constructs designed in the next section.

Analysis of the resulting code is based on the software science criteria developed by Halstead (68). The basic quantities which have to be measured are given in Table 5.2. Table 5.3 indicates these quantities for the programs shown in Fib. 5.1. The three measuring units are:

a) PROGRAM VOLUME (V): Whenever a given algorithm is translated from one language to another, its size changes. This size change is reflected by the volume that the algorithm occupies in the various languages. Volume V can be defined as:

$$V = N\log_2 n$$

b) PROGRAM LEVEL (L): The level of a program is a measure of the relative 'goodness' of a certain representation of a program as compared to another representation. The 'higher-level' language concept can be measured by this criteria through the equivalent program representation.

$$L = (2+n_2^*) \text{ Log}_2 \ (2+n_2^*)/V$$

c) INTELLIGENCE CONTENT (I): It seems fundamental to measure how much is said in a program in terms of its information content. This measure is called the intelligence content and is defined as:

$$I = L \times V$$

The results are tabulated in Table 5.4. Comparing the two IML format results, we come to the following conclusions:

1) Based on the volume criteria V, the three operand format has a lower volume than the single address format. This means that when VMPL is translated to the three address IML format, we

```
IFTRUE  (IR,7)  THEN ;
   BEGIN;
     PCTEMP=PC-1;
     PCTEMP=CRNTPG(PCTEMP);
     MAR=PCTEMP.OR.ADR;
     MDR=MEM[MAR];
   END;
ELSE;
   BEGIN;
     MAR=ADR;
   END;
ENDIF;
IFTRUE (IR,8) THEN;
   BEGIN;
     MART=MEM[MAR];
     IFTRUE (MAR.GT.7) THEN;
       BEGIN;
         IFTRUE(MAR.LT.16)THEN;
           BEGIN;
             INC MEM[MAR];
           END;
         ENDIF;
       END;
     ENDIF;
     MAR=MART;
   END;
ENDIF;
```

Fig. 5.1(a) A SMALL PART OF A VMPL PROGRAM

```
        CONDF     .IR.7      TL.001
        SUB       PC         C 1        PCTEMP
        EXTR      CRNTPG     PCTEMP     PCTEMP
        OR        PCTEMP     ADR        MAR
        RMOVE     MEM        MAR        MDR
        BRCH      E L.002
L.001
        MOVE      MAR        ADR
L.002
        CONDF     .IR.8      T L.003
        RMOVE     MEM        MAR        MART
        COMP      MAR        C 7
        CONDT     N          T L.004
        COMP      MAR        C 16
        CONDF     N          T L.004
        RMOVE     MEM        MAR        +T.001
        INC       +T.001
        WMOVE     MEM        MAR        -T.001
L.004   MOVE      MART       MAR
L.003
```

Fig. 5.1(b) THREE ADDRESS CODE FOR SOURCE PROGRAM OF Fig. 5.1(a)

```
         LOAD     R1,IR              L-002    LOAD     R1,IR
         LOAD     R9,*64                      LOAD     R2,*128
         AND                                  AND
         BNZ      L-001                       BNZ      L.003
         LOAD     P1,PC                       LOAD     R1,MAR
         LOAD     R9,*1                       READ     MEM
         SUB                                  STORE    R1,MART
         STORE    RO,PCTEMP                   LOAD     R1,MAR
         LOAD     R1,CRNTPG                   LOAD     R9,*7
         LOAD     R2,PCTEMP                   SUB
         AND                                  BIN      L.004
         STORE    RØ,PCTEMP                   LOAD     R1,MAR
         LOAD     R1,ADR                      LOAD     R2,*16
         LOAD     R2,PCTEMP                   SUB
         OR                                   BNN      L.004
         STORE    RO,MAR                      LOAD     R1,MAR
         LOAD     R1,MAR                      READ     MEM
         READ     MEM                         LOAD     R1,RØ
         STORE    RØ,MDR                      LOAD     R2,*1
         BRH      L.002                       ADD
L.001    LOAD     R1,MAR                      LOAD     R1,MAR
         STORE    R1,ADR                      WRITE    MEM
                                     L.004    LOAD     R1,MAR
                                              STORE    R1,MART
                                     L.003
```

Fig. 5.1(c) SINGLE ADDRESS CODE FOR SOURCE PROGRAM OF Fig. 5.2(a)

TABLE 5.2 BASIC QUANTITIES TO BE MEASURED FOR DEVELOPING SOFTWARE SCIENCE CRITERIA FOR PROGRAMS

n_1 =	NUMBER OF UNIQUE OR DISTINCT OPERATORS
n_2 =	NUMBER OF UNIQUE OR DISTINCT OPERANDS
N_1 =	TOTAL USAGE OF ALL OF THE OPERATORS
N_2 =	TOTAL USAGE OF ALL OF THE OPERANDS
n_2*=	NUMBER OF UNIQUE OR DISTANCE I/O PARAMETERS

$n = n_1 + n_2$

$N = N_1 + N_2$

TABLE 5.3 BASIC QUANTITIES FOR THE PROGRAMS OF Fig. 5.1

INSTRUCTION-STREAM FORM	n_1	n_2	N_1	N_2	n_2*	n	N
VMPL	14	10	59	26	7	24	85
THREE ADDRESS	11	12	17	31	7	23	48
SINGLE ADDRESS	12	13	46	31	7	25	77

TABLE 5.4(a) CALCULATED MEASURES FOR PROGRAMS OF Fig. 5.1

INSTRUCTION-STREAM FORM	V (VOLUME)	L (LEVEL)	I (INTELLIGENCE CONTENT)
VMPL	389.72	.0732	28.53
THREE ADDRESS	217.13	.1314	28.53
SINGLE ADDRESS	357.5	.0797	28.49

TABLE 5.4(b) CALCULATED MEASURES FOR VMPL PROGRAM AND ITS EQUIVALENT OF APPENDIX C

INSTRUCTION-STREAM FORM	V (VOLUME)	L (LEVEL)	I (INTELLIGENCE CONTENT)
VMPL	3039.82	.0741	225.25
THREE ADDRESS	1737.04	.1331	231.20
SINGLE ADDRESS	2806.38	.0811	227.60

produce the least number of statements. Thus, on a per statement basis, each three address IML statement is 'nearer' to the original VMPL statement than a single address IML statement.

2) The level criterion L indicated that the three address IML format is at a higher level than the single address IML format. This further reinforces the claim of the 'nearness' of the three operand IML format to VMPL. By keeping the same power in the instruction set of the three & single address IML's and changing only the addressability of the underlying machine, the level of the language changes, i.e., 'reducing addressability of a language reduces the level of the program written in the language.'

3) The intelligence content of the two formats indicates that this criteria holds invariant to within 10% under the translation process. This indicates an equivalence in the algorithmic power of the two IML formats. A third observation is based on the level figures of VMPL program segments compared to the corresponding IML figures. The VMPL level is found to be less than the three address IML notation! This is a strange result and can be explained in the following way.

Halstead's level calculations are based on the information-theoretic point of view of a program, i.e., essentially its alphabet. VMPL has a lot of redundancy for reliability which inflates the alphabet count. Thus, the level of the representative programs in VMPL is less than the level of the corresponding three address IML format. However, in spite of this redundancy, the VMPL representation has a higher level than the equivalent single address IML format. Thus, we reach an important result that:

'Redundancy in a language reduces the level of the programs written in the language.'

This result is also confirmed by Elshoff (69).

The general result from all this analysis is that the three operand IML format is 'nearer' to VMPL, than the single address VMPL format. By keeping 'nearer' to VMPL, IML statements reflect as much as possible the semantic content of VMPL statements. Thus, each IML statement, on a per statement basis, has a lot of information about the original VMPL statement for which it was produced. The three address IML statements bind the translated code to a high level register oriented machine.

Keeping the IML code high level helps in the later stages of compilation. The compilation model selected in Chapter 4, based on portability considerations, requires a high level IML which can easily be decomposed into a variety of low level microinstruction formats corresponding to various m-computers. Since most m-computers are register oriented, (Chapter 2) having an IML which is register oriented further helps us in achieving the goal of portability and efficiency.

By selecting the three address format, the IML achieves the following things:

1) Provides the most opportunities for subsequent economization of micro-code.
2) Resembles the code of a general class of register oriented m-machines.
3) Because of 1 & 2 tight micro-code can be produced for a variety of m-machines.

All this can be summarized by the following implication.

IMPLICATION 10: The IML format selected for the compilation model is the three address format or quadruples. Furthermore, the IML statements are high level in that they resemble VMPL statements as much as possible. Known characteristics of m-computer architecture and instruction repertoire can be used in designing various facets of IML statements.

5.3 INTERMEDIATE LANGUAGE

A program written in VMPL gets translated into an abstract quadruple like intermediate language. The various statements of the IML are discussed here. In discussing the IML, reference to VMPL statements has been made, since, IML is highly dependent on VMPL (I-10).

Basically there are two kinds of statements in IML. One group is associated with the various declaration statements of VMPL and is known as the Intermediate Information Statement Group (IISG). The other group is associated with the actual executable statements of VMPL and is known as the Intermediate Executable Statement Group (IESG).

5.3.1 IISG

An IISG statement is made up of five objects. The basic format of the statement is:

DECLARATIONTAG, IDENTIFIER, DIMENSION, LENGTH, OTHERINFORMATION

where

DECLARATIONTAG - is a unique number of the form NMA. N and M are numeric and A is an alphanumeric digit. This number uniquely identifies the corresponding VMPL declaration statement to which the IDENTIFIER belongs, i.e., identifies the type of the IDENTIFIER. A list is shown in Table 5.5.

IDENTIFIER - is the identifier (as defined in VMPL syntax) which is associated with the VMPL declaration.

DIMENSION - is an integer which gives the number of unique elements associated with the identifier. Its absence, a zero (0) or a one (1) all mean that the IDENTIFIER is dimensionless. Essentially, the only time this object is used is with the VMPL, MEMORY and STACK type variables.

LENGTH - is the length of the identifier, i.e., the number of bits. This object is used with most of the declarative statements of VMPL.

OTHERINFORMATION - is only associated with a few DECLARATIONTAG's. Its meaning and format varies with the tag and all the cases are discussed individually below:

1) When this object is associated with the tag belonging to the FIELD declaration statement of VMPL, it consists of three integers. These three integers are the three integers associated with the FIELD declaration which identify the bit positions and the shift amount (Fig. 5.2).

TABLE 5.5 DECLARATIONTAG VALUES FOR IISG

DECLARATIONTAG NMA

N	MEANING
0	NAME OF THE OTHERS
1	LOCAL
2	GLOBAL
4	SPROC

M	MEANING
0	NONE OF THE OTHERS
1	TEMPORARY
2	PERMANENT

A	MEANING
0	SIMPLE
1	MEMORY
2	STACK
3	PSTACK
4	FLAG
5	FIELD
6	USE
7	EXPECT
8	RETURN
9	EXTERNAL

A	MEANING
A	EMULATOR NAME
B	PROGRAM START
C	PROGRAM END
D	WORDSIZE
E	ARITHMETIC
F	PROC NAME
G	SPROC NAME
H	BLOCK CODE START
I	BLOCK CODE END
J	BEGIN BLOCK
K	END BLOCK

2) The other information object which gets associated with the stack pointer (PSTACK) VMPL declaration, indicates the push-pop sequence associated with the stack. It consists of four distinct symbols which were shown in Chapter 2 Table 2.3.
3) When this object gets associated with the VMPL, EXTERNAL variables it consists of a single alphabet. The single alphabet indicates whether the external IDENTIFIER is a simple variable, a flag variable or a procedure(Fig. 5.2.).
4) In the case of the global FLAG declaration statement of VMPL, this object (an integer) indicates which of the predefined four flags does the IDENTIFIER represent. In case the IDENTIFIER is a general flag, the value of this integer is 0(Fig. 5.2.).

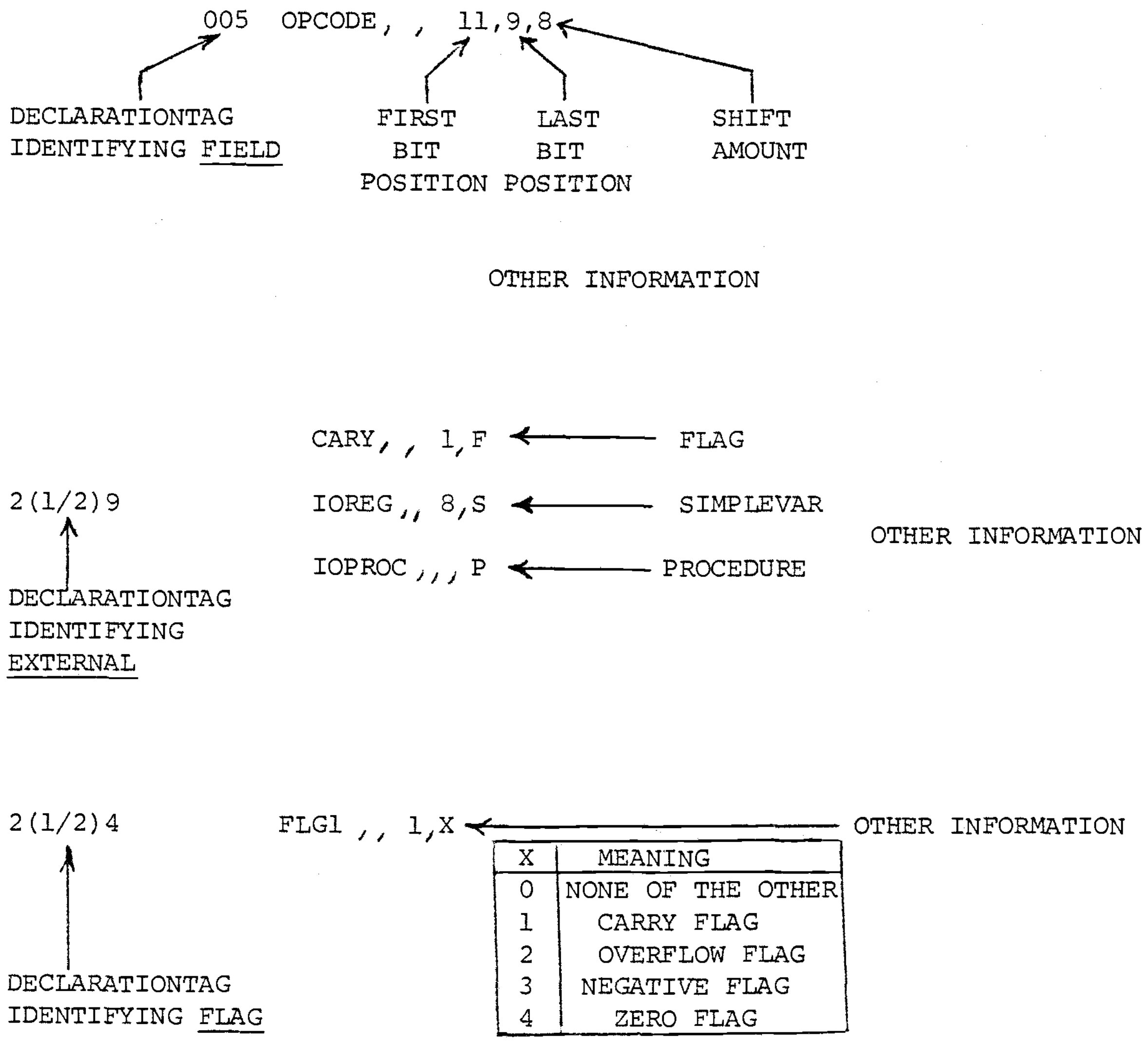
Examples of VMPL declaration statements and the corresponding IISG statements are shown in Fig. 5.3.

5.3.2 IESG

The IESG statements are based on quadruples with an operation and three operands. All three operands are optional in that some statements have none, some one, some two and some all three operands. First, the overall format is discussed and then the individual statements are discussed.

5.3.2.1 IESG FORMATS

The IESG statements are fixed formats with labels starting in the first column and existing by themselves in a line. A star (*) in the first column indicates a continuation of the previous statement. If the line with the star is empty, it indicates the end of the continuation.



X	MEANING
0	NONE OF THE OTHER
1	CARRY FLAG
2	OVERFLOW FLAG
3	NEGATIVE FLAG
4	ZERO FLAG

Fig. 5.2 OTHERINFORMATION VALUES FOR IISG

```
DCL  GLOBAL  PERMANENT MEMORY    MEM:[4096]:8;          (VMPL)

           221  MEM,4096,8                                (IML)

DCL GLOBAL  TEMPORARY   REG1:8, REG2:3;                 (VMPL)

           210  REG1, ,8                                  (IML)
           210  REG2, ,3

DCL LOCAL   PERMANENT   REG3:8;                          (VMPL)

          120  REG3, ,8                                   (IML)

DCL  FIELD  OPCODE (11,9,8);                             (VMPL)

           005  OPCODE, , ,11,9,8                         (IML)
```

Fig. 5.3 VMPL - IML EQUIVALENTS

All other statements start in column seven or eight. The various column designation are shown in Table 5.6. The general format of an IESG statement is:

OPERATION OPERAND1 OPERAND2 OPERAND3 FLAGS

The OPERATION object recognizes the action which the IESG statement has to take using the three operands which follow, i.e., OPERAND1, OPERAND2 and OPERAND3. The FLAGS object indicate the predefined VMPL flags which have to be enabled during this operation.

Another aspect of these IESG statements is the presence of modifiers with the OPERATION and OPERAND objects. This is done to retain the information contained in VMPL statements that will be useful in the production of compact, machine dependent microcode. A simple example of this are the compiler generated temporary variables. All such variables are tagged with modifiers (+) or (-). This helps the register allocation/deallocation phase of compilation know if the temporary is used later on in a sequence of IML statements (+), or not used (-). Since the compiler generates the temporaries, it knows their 'life-time' and can easily generate these modifiers.

Another group of tags are associated with the various labels associated with the GO TO, FOR, LEAVE & IF-THEN-ELSE statements. These along with some of the IISG statements recognize each sequential block of IML statements. The later phases of compilation uses these tags for microcode compaction by noting the beginning and end of every straight-line code segment. The heuristic underlying this approach is that there cannot be more parallelism at the microcode level than is available in the HLL (VMPL) program structure. The compiler is able to reveal the inherent concurrency of the high level algorithm by tagging the intermediate code with 'clues'. These 'clues' are subsequently used by the compiler. We believe that maximal code concurrency can be detected and used in producing a minimal number of horizontal microinstructions over straight line code segments. This aspect of code compaction is given in (72). The various modifiers are listed in Table 5.7.

TABLE 5.6 IML COLUMN DESIGNATIONS

COLUMNS	VALUES
8-14	OPERATION
17-23	OPERAND 1
20-32	OPERAND 2
35-41	OPERAND 3
43-46	FLAG SETTINGS
7	OPERATION MODIFIER
16,25,35	OPERAND MODIFIER

TABLE 5.7 OPERAND MODIFIERS

SYMBOL	MEANING
.	BIT OPERAND
/	CONCATENATED OPERAND
+	TEMPORARY NEEDED
-	TEMPORARY NOT NEEDED
C	CONSTANT
P	PARAMETER IDENTIFIER
T	LABEL FOR FIRST IF-THEN-ELSE BRANCH
E	LABEL FOR SECOND IF-THEN-ELSE BRANCH
G	LABEL FOR A GOTO
F	LABEL FOR A FOR
L	LABEL FOR A LEAVE
S	LABEL FOR A SELECT
A	ARGUMENT IDENTIFIER

5.3.2.2 IESG STATEMENTS

There are seven classes of IESG statements. Each class is treated separately.

CLASS 1 IESG STATEMENTS

The general format of statements belonging to Class 1 is:

```
        OPERATION     SRC1        SRC2         DEST
```

The OPERATION is either an arithmetic or a logical operation (one to one correspondence with VMPL arithmetic and logical operators Table 2.4) which require two sources, SRC1 & SRC2 and produce a result which is stored in DEST. The NOT logical operation only requires one source. As such, it does not have the SRC2 object. The various OPERATION's are listed in Table 5.8.

CLASS 2 IESG STATEMENTS

The shift and rotate instructions belong to Class 2 corresponding to the shift and rotate instructions of VMPL Table 2.5. The format is:

```
        OPERATION      SRC1      COUNT,VALUE      DEST
```

In case the operation is one of the shift instructions, SRC1 is shifted by COUNT places filling the empty places by VALUE and storing the result in DEST. If the operation is a rotate instruction, the VALUE part from the above format is missing Table 5.9.

CLASS 3 IESG STATEMENTS

Class 3 operations are for reading and writing into the MEMORY class variable of VMPL. The two operations are associated with reading from and writing into memory Table 5.10. The format is:

```
        OPERATION      MEMVAR       INDEX        VALUE
```

The object INDEX is used as an index into the memory array MEMVAR to point to a location. VALUE either receives the value or sends the value to this location.

TABLE 5.8 CLASS 1 IESG OPERATIONS

OPERATION	MEANING
ADD	ADDITION
SUB	SUBTRACTION
MPY	MULTIPLY
DVD	DIVIDE
AND	LOGICAL AND
OR	LOGICAL OR
XOR	LOGICAL XOR

TABLE 5.9 CLASS 2 IESG OPERATIONS

OPERATION	MEANING
SHIL	SHIFT LEFT
SHTR	SHIFT RIGHT
ROTL	ROTATE LEFT
ROTR	ROTATE RIGHT

CLASS 4 IESG STATEMENTS

Class 4 deals with the various branch operations. It can have one or two operands with the second (if two) or first (if one) always being a label. A list of operations is given in Table 5.11.

CLASS 5 IESG STATEMENTS

The seven statements which belong to Class 5 are there for the translation of some of the miscellaneous statements of VMPL (INC, DEC, POP, etc.). In all cases there is a one-to-one relation with the corresponding VMPL statement Table 5.12. They are one, two or implicit operand statements.

CLASS 6 IESG STATEMENTS

Class 6 contains two statements which are translated from the FOR and SELECT, VMPL statements. Their format is shown in Fig. 5.4. Note that for the SLCT statement we use the continuation aspect of IML statement format.

CLASS 7 IESG STATEMENTS

The three statements in Class 7 have again a one-to-one relation with corresponding VMPL statements and deal with the control aspect of program execution. They are shown in Fig. 5.5.

This finishes the design of the IML constructs.

5.4 CONCLUSIONS

Comparing the designed IML constructs and the various constraints and requirements on the IML from in Chapter 5 and the earlier sections of this chapter (I10), we come to the following conclusions:

1) The IML constructs are high level in that most of them have a one-to-one correspondence with VMPL statements.
2) The IML constructs are register oriented, i.e., they can be thought of as the machine language of an abstract register oriented machine.

TABLE 5.10 CLASS 3 IESG OPERATIONS

OPERATION	MEANING
RMOVE	READ MEMORY
WMOVE	WRITE MEMORY

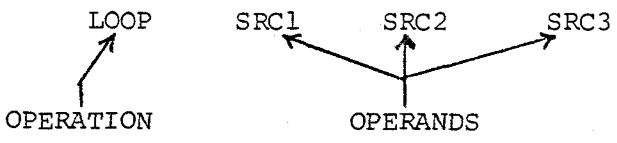
TABLE 5.11 CLASS 4 IESG OPERATIONS

OPERATION	MEANING
COMP	COMPARE
BRCH	BRANCH
CONDF	CONDITION FALSE
CONDT	CONDITION TRUE

TABLE 5.12 CLASS 5 IESG OPERATIONS

OPERATION	MEANING
INC	INCREMENT (ADD 1)
DEC	DECREMENT (SUB 1)
CLR	SET TO ALL ZEROES
SET	SET TO ALL ONES
MOVE	MOVE SRC1 TO SRC2
PUSH	PUSH SRC1 INTO STACK
POP	POP STACK INTO DEST

```
        FOR      SRC1  =  SRC2  TO  SRC3 ;              (VMPL)
```



(IML)

```
            SELECT (SRC1,SRC2) FROM
                (SRC3, LABEL1);
                (SRC4, LABEL2);
           ENDSELECT                                   (VMPL)


            SLCT      SRC1      SRC2
*                     SRC3      LABEL1
*                     SRC4      LABEL2
*                                                      (IML)
```

Fig. 5.4 CLASS 6 IESG STATEMENTS

OPERATION	MEANING
HALT	HALT
RET	RETURN

```
      EXECUTE          SRC1 (PAR1, PAR2)  ;              (VMPL)

      XEQ              SRC1       PAR1
  *                    PAR2
  *                                                      (IML)
```

Fig. 5.5 CLASS 7 IESG OPERATIONS

3) Furthermore, the IML code contains tags or 'clues' which recognize its various properties. These tags are used for the production of compact microcode.

From 1, 2 and 3 we see that the designed IML meets the requirements which were placed on it due to I9 and I10.

6. RESULTS

6.1 SIMULATOR

In Chapter 1 the objectives of this research were identified in terms of four problems. Problems 1, 2 and 3 were solved in the earlier chapters. According to the statement of problem 4, a testing methodology for microprogram correctness should be designed. We achieve this aspect of microprogram development via the provision of a simulator for the intermediate language, and testing as diagramed in Figure 6.1.

A program (P1) (emulator) written in VMPL is translated into the intermediate language (IML1). Say the program P1 emulates a machine M1 (target). Next we give the IML1 code to the simulator. We also provide the simulator with code which correspond to machine code programs for the machine M1. These programs (DP1) essentially correspond to a set of diagnostics written for the target machine and get stored in the MEMORY variable of the emulator. The simulator then executes (simulates) the IML1 code. This corresponds to executing the diagnostic programs DP1 on the machine M1. If the 'diagnostics' produce correct results, we suspect the correctness of the emulator (P1 or IML1). Now if this same IML1 code is translated into microcode of a host machine (H1), we suspect the correctness of this microcode.

Since we would like the simulator to run in an interactive fashion, simulator commands are added (embedded) to the original VMPL program, i.e. P1. This requires the addition of a simulator command statement to the VMPL and IML instruction repertoire. The format for a simulator command statement in VMPL is:

? COMMAND PARAMETERS

where the COMMAND object is a three letter mnemonic identifying a unique simulator command and the PARAMETERS object is a set of parameters which the COMMAND object requires. The simulator along with the commands, etc. is described in Appendix B. The IML representation of a command follows the general IML format, with the command mnemonic as the op-code and the parameters acting as the operands.

Thus, by the provision of a simulator for the IML, we have achieved

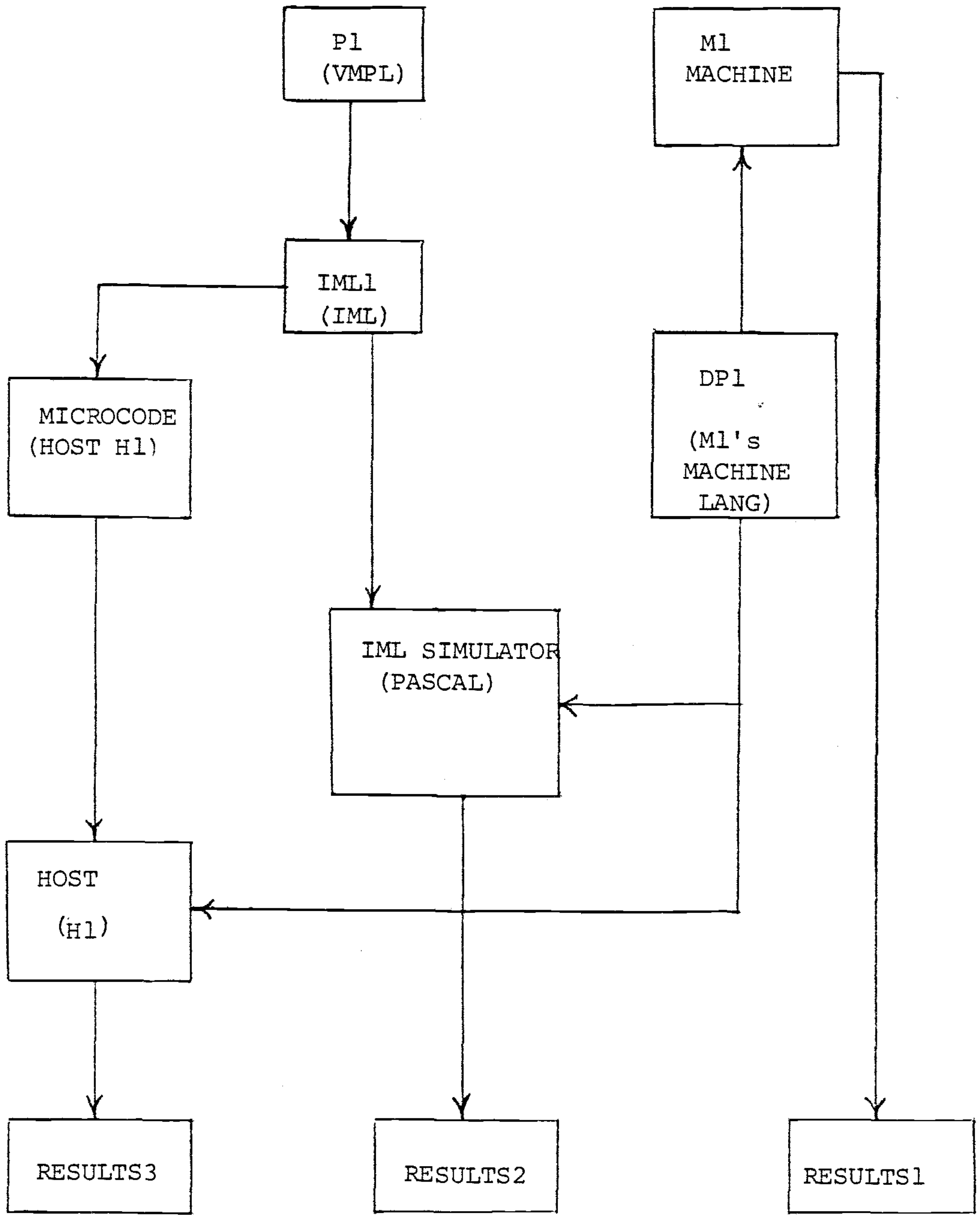


Fig. 6.1 MICROCODE CORRECTNESS MODEL

a solution to problem 4. This solution is an indirect approach in that we do not prove the correctness of the microcode directly. We try to prove the correctness of the code which is one level above the microcode; namely, the IML code which will ultimately be converted into microcode. Proving correctness at this level is found to be a simple task via simulation. To make the final assertion that the microcode is correct, we have to prove two things. The correctness of the simulator and the compiler which converts the IML code into microcode. This is a one time task and can be done by the compiler implementor.

6.2 VMPL COMPILER

A compiler for VMPL has been written using the OSU META system (71) (APPENDIX B) available on the OSU CYBER 73. The META system is based on Schorre's META II (70) and consists of three main components:

1) The META language
2) The META compiler
3) A package of support routines called METASUBS.

The META language is designed specifically for compiler-writing. Basically, the notation is BNF with slight modifications. To write a compiler, the user expresses the syntax of the language and the code to be produced in a series of META statements called rules. Each rule consists of items called 'recognizers' to be searched for in the input source language. The recognizers are the names of other rules, specific characters to be looked for, or basic recognizers suppled by META, i.e., .ID recognizes an identifier; .STRING, a string of characters enclosed by apostrophies; 'BEGIN', the word BEGIN, etc.

In addition to items to be recognized, each rule may also contain actions to be taken if the items are found. These actions may look up an item in the symbol table, put a string of characters on a stack maintained by META, or put something on the output buffer. Once a rule or set of rules has recognized a statement, these actions output the low level language equivalent for that statement.

Once the set of rules comprising the user's compiler is written in META language, the META compiler produces a sequence of calls to the METASUBS package. The sequence of calls which comprise the user's compiler can then be loaded into the machine (CYBER 73) along with METASUBS to process the user's language. METASUBS scans the source code in a top down, recursive-descent manner according to the user's rules and produces code for the associated actions. In addition, the package handles input and output, syntax error messages, a symbol table facility, listing control, etc.

The symbol table facility of META was found to be too primitive for successful compilation of VMPL programs. Thus, a symbol-table program (STP) was written using PASCAL (56). A program written in VMPL is translated into IML statements by a META-VMPL compiler. This output is then fed into STP which does all the symbol table cross checking of the IML code or indirectly the VMPL code. In doing this checking STP generates a series of symbol tables which can later on be used by the simulator described in the previous section.

According to the compilation model selected in Chapter 4, there is still another phase of compilation. This phase deals with the conversion of the IML code into microcode. This part of the compiler was written by P.Y. Ma and is discussed in (72).

In the next section, we discuss some of the results obtained on the compilation of various VMPL programs into IML programs.

6.3 VMPL-IML ANALYSIS

Three emulators were written using VMPL as source language. The three emulators are:

1) VM1 - PDP-8 minicomputer
2) VM2 - INTEL 8080 microprocessor
3) VM3 - An abstract register oriented machine.

These three emulator programs were translated into IML code. Two groups of experiments were then conducted on this resulting code.

6.3.1 COUNTING EXPERIMENTS (CE)

Several different counting experiments were performed. These experiments essentially consist of counting various objects of the IML code.

CE(1)

The first counting experiment consisted of counting the number of occurrences of the various variables used in the program. There are three types of variables:

1) Global variables
2) Local variables
3) Temporary variables

Temporary variables are the variables generated by the compiler. The results are shown in Table 6.1. We find that the percentage of usage of local plus temporary variables is about the same as the percentage of usage of global variables.

This locality of usage reflects on the block structured nature of emulator programs and can be used as an indicator of the locality of variable usage at the block level, i.e.:

'Local variables on a percentage basis are used as much (if not more) as global variables'

CE(2)

The next experiment we did was to measure the distance between successive references to a particular variable. IML statements were numbered in a sequential manner, say 1 to N. Then for each variable of all the three emulators we did the following: We counted the number of times the variable was used twice in the same IML statement (distance between usage of variable is 0), the number of times the variable was used in two consecutive IML statements (distance between usage of variable is 1), the number of times the variable was used in two IML statements, separated by one IML statement which did not use it (distance between usage of variable is 2) etc. This count was then averaged for all the variables of all three emulators to come up with a common characteristic of variable usage. This characteristic is plotted as a curve in Fig. 6.2.

TABLE 6.1 RESULTS OF COUNTING EXPERIMENT CE(1)

EMULATOR	GLOBAL VARIABLES		LOCAL VARIABLES		TEMPORARY VARIABLES		LOCAL plus TEMPORARY VARIABLES	
	TOTAL	%	TOTAL	%	TOTAL	%	TOTAL	%
VM1	121	52.4%	54	23.4%	56	24.2%	110	47.6%
VM2	183	49.6%	97	26.3%	89	24.1%	186	50.4%
VM3	159	55.4%	68	23.7%	60	20.9%	128	44.6%

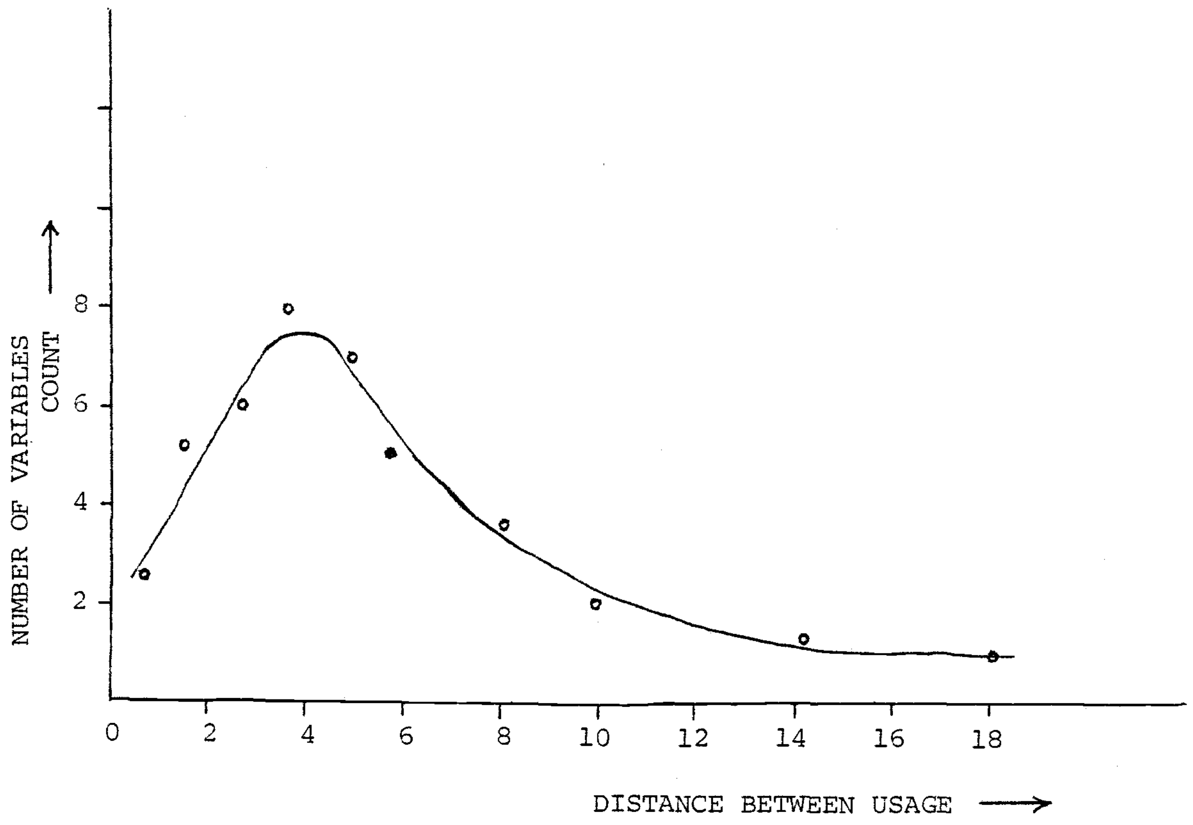


Fig. 6.2 RESULTS OF COUNTING EXPERIMENT CE(2)

Taking a point on this curve (for example) we see that on the average there is only 1 variable which is used every 18 IML statements. From the shape of this curve we see that the average distance between the usage of variables is between three and five. This curve does not include the figures for the temporary variables. The distance of usage for temporary variables was consistent and equal to 1, i.e., a temporary is always used immediately in the IML statement following the IML statement which generated it (a property of the way in which the compiler generates temporary variables). Thus, in general, we can conclude that:

'The average distance of usage for all variables is small'

This statement can then be interpreted as a locality of variable usage at the statement level.

CE(3)

Next, we did some counting experiments similar to the ones done by Elshoff (50) and Gannon (52) on the IML programs. It may be recalled that their data was used in the first place to design VMPL. Our results, Table 6.2, are more or less similar to their results. Some differences can be explained as follows:

1) The large number of branch statements is explained by the block nature of VMPL programs, the structured VMPL constructs and the overall structure of an emulator program. If we count CONDF, CONDT and BRCH as the only kind of branch statements, our figures match the Elshoff, Gannon data. This condition corresponds to a SELECT-less, non blocked program.

2) We get a low value for 'memory reference instructions.' This is primarily because the emulator (programs) are of register oriented machines. We suspect they will be higher for memory oriented target machines.

Thus, we see that programs written in VMPL produce the same kind of results for instruction usage as programs written in Fortran and PL/I.

In general, we can come to the following conclusion from this experiment:

'The usage of the Elshoff-Gannon data for the design of VMPL is valid'

TABLE 6.2 RESULTS OF COUNTING EXPERIMENT CE(3)

INSTRUCTION TYPE	VM1	VM2	VM3
3 OPERAND (ARITHMETIC 8 LOGICAL OPER.)	29.41%	31.2%	30.1%
2 OPERAND (MOVE etc.)	21%	20.8%	19.8%
1 OPERAND (VMPL MISC)	6%	8.7%	7.8%
BRANCH	37.6%	35%	39.9%
MEMORY REFERENCE	5.99%	4.3%	2.4%

6.3.2 VARIABLE ASSIGNMENT EXPERIMENT

In Chapter 3 we made the assertion that by having global-local variable declaration (variable priority) and a block structure for VMPL programs, we will produce tighter microcode. This was to happen because of a smaller number of load and store operation. The variable assignment experiment was conducted at the IML level, we believe that fewer IML statements produce fewer microinstruction statements (microcode).

There are two sets of variables in the experiment, the number of available 'host' registers and the register allocation scheme which is adopted. Four register allocation schemes were selected:

1) All program variables are treated as global variables. In case a register is needed for allocation, the first available register (registers are numbered in an arbitrary sequential form) is deallocated. (Type a)
2) The same as above except a count on the frequency of usage of a variable is kept. In case of deallocation, the variable with the smallest frequency count is deallocated. (Type b)
3) The VMPL designed priority structure is adopted. If two variables having the same priority have to be deallocated, the first one (based on an arbitrary number sequence) is deallocated. (Type c)
4) Same as above except a count on the frequency of usage of a variable is kept. In case of deallocation of two variables having the same priority, the variable with the smallest frequency count is deallocated. (Type d)

TYPE a allocation scheme is one adopted by most present day compilers. TYPE b is a step forward in variable allocation where some information about variable usage is used. TYPE c is the priority scheme outlined by us in the design of VMPL. TYPE d is a further enhancement of this technique.

The results for the three emulators are plotted in Fig. 6.3. The dependent variable is the number of load/store instructions required

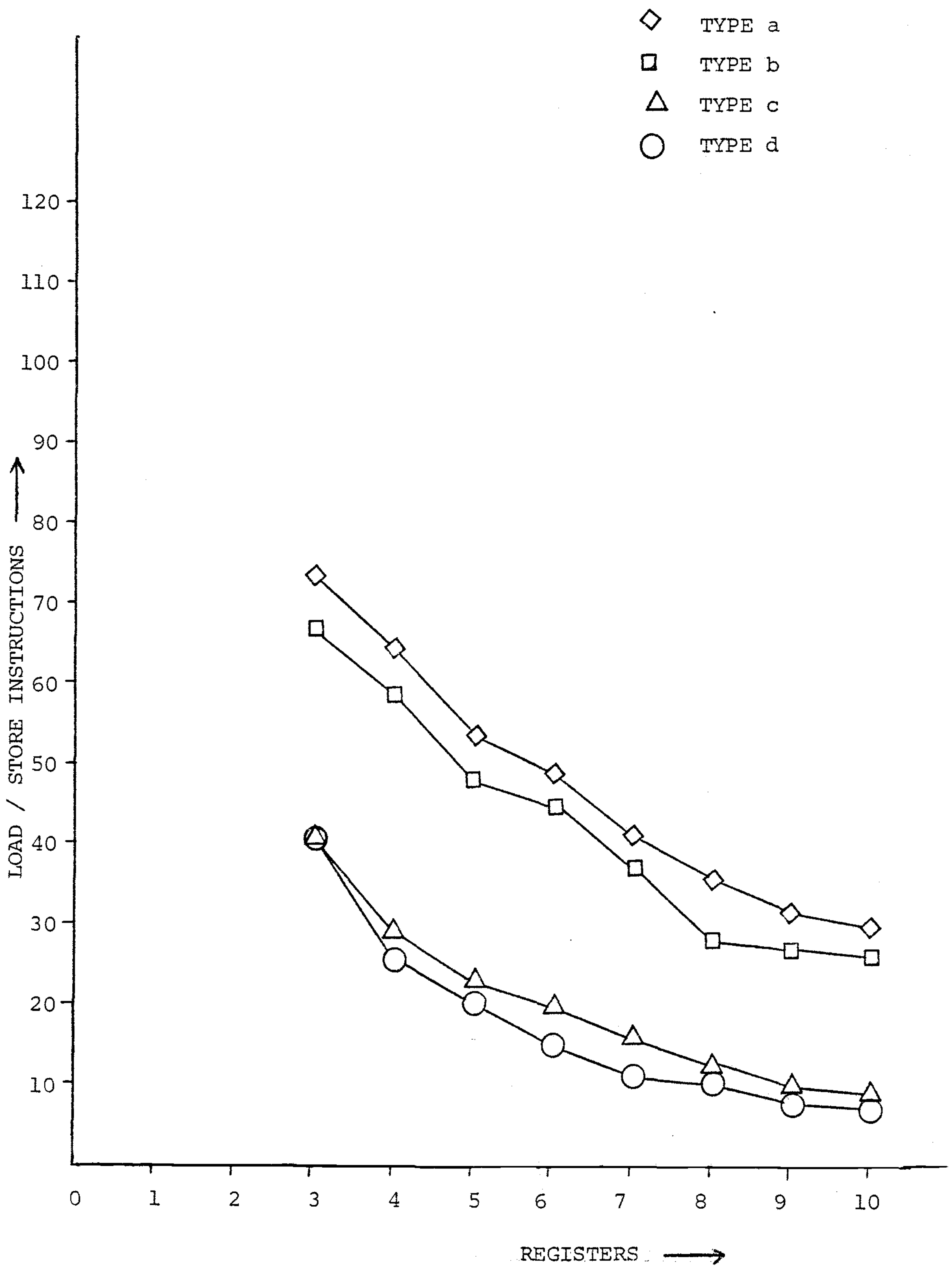


Fig. 6.3(a) RESULTS OF VARIABLE ASSIGNMENT EXPERIMENT ON VM3

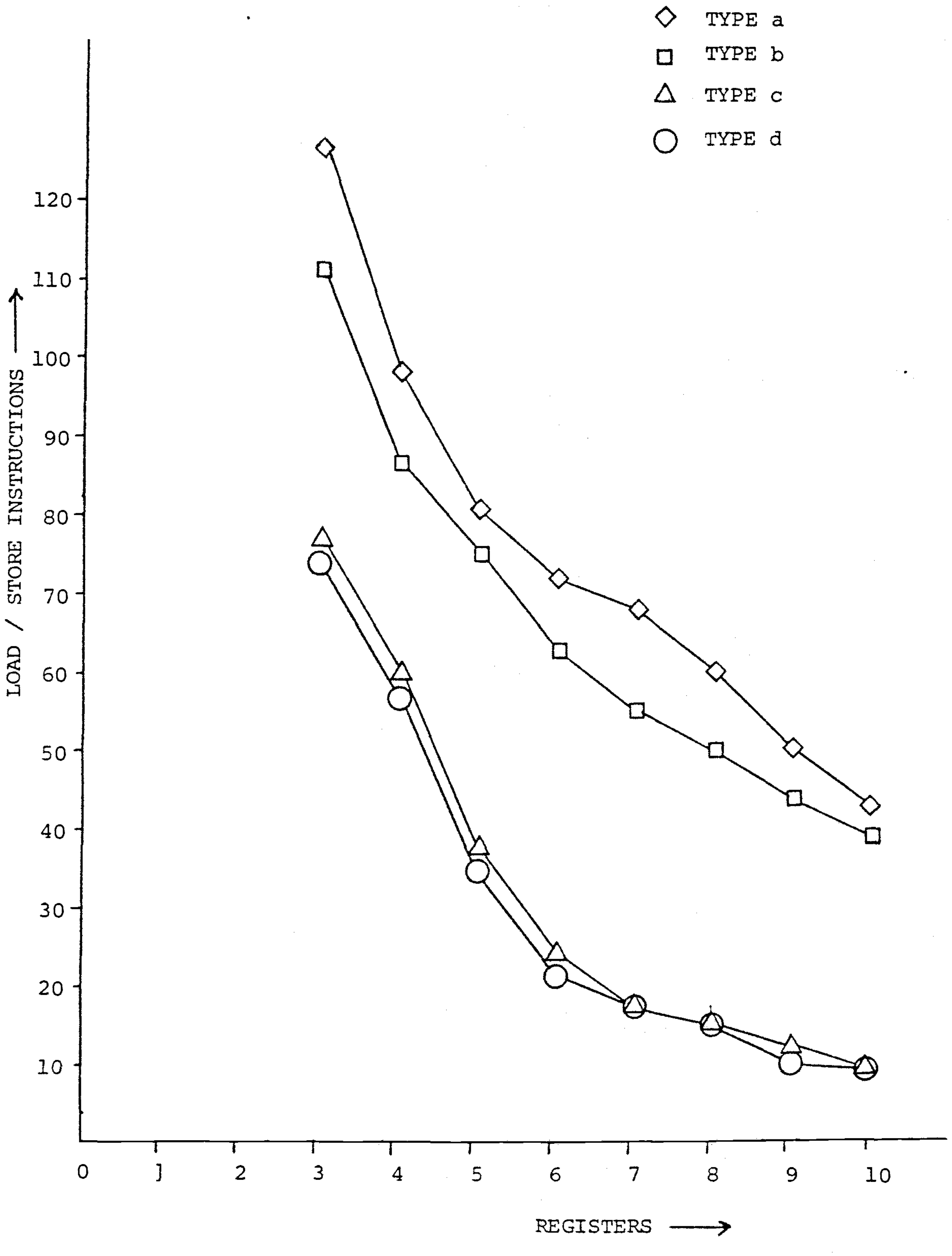


Fig. 6.3(b) *RESULTS OF VARIABLE ASSIGNMENT*
EXPERIMENT ON VM2

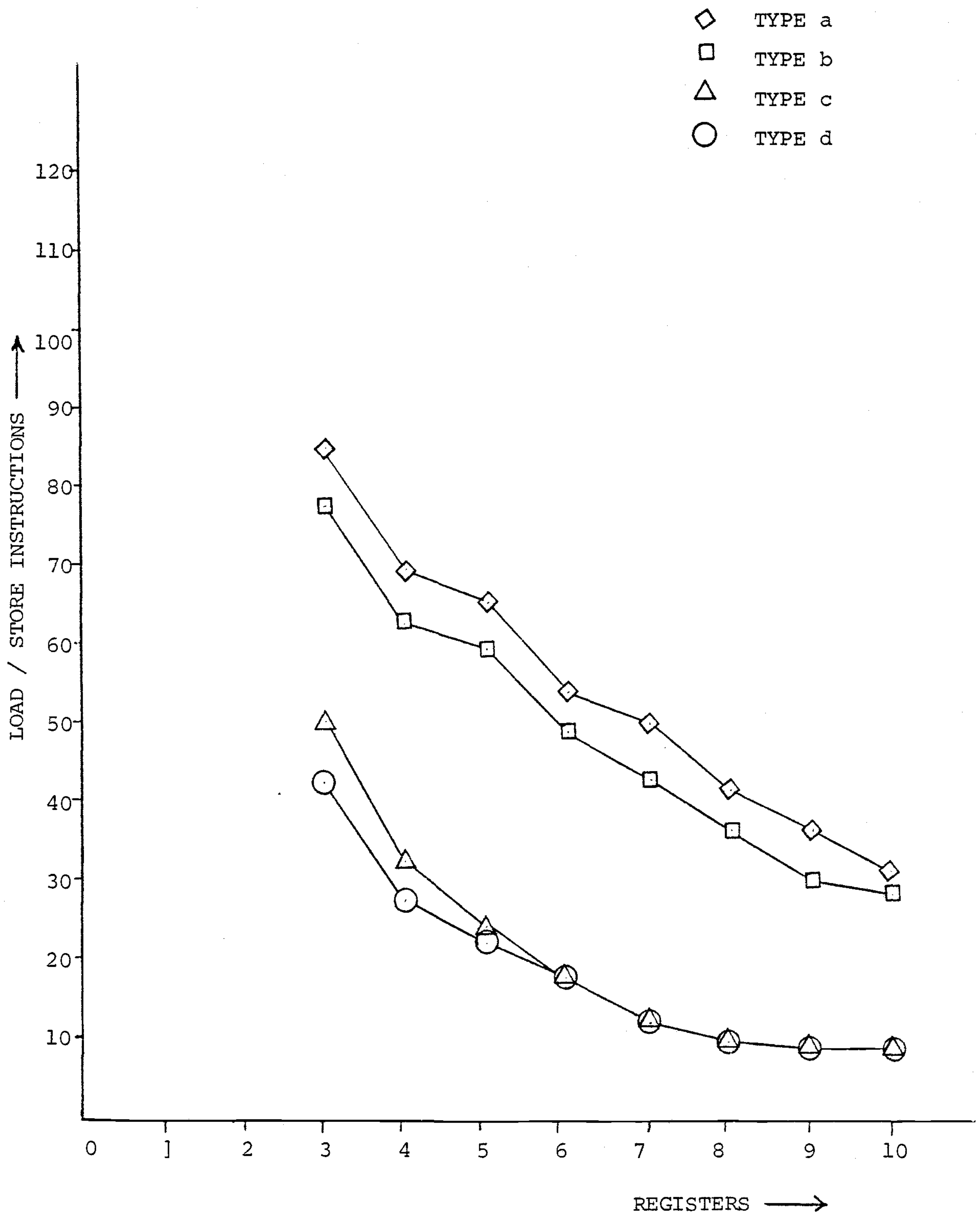


Fig. 6.3(c) RESULTS OF VARIABLE ASSIGNMENT EXPERIMENT ON VM1

for each emulator. For all three cases we can rate the register allocation schemes as follows:

TYPE d, TYPE c, TYPE b, TYPE a

with TYPE d producing the least number of load/store instructions and TYPE a producing the greatest number of load/store instructions. The difference between TYPE d and TYPE c is not large but both are significantly better than TYPE b or TYPE a.

Thus, we have proved (for the cases under consideration) that:

'Using a VMPL variable priority scheme produces fewer load/store instructions than other schemes'

From the Counting Experiments and the Variable Assignment Experiments done on the three emulator programs, we come to the general conclusion that VMPL has successfully met its design goals of producing tight microcode.

6.4 VMPL EXTENSIONS

While designing VMPL, we also pointed out the modular top down design approach we had followed. To test this aspect of VMPL, we do the following three experiments on VMPL definition and syntax. These experiments are done by changing PREMISE 5 (Chapter 3) to include target machine which have a) both word and byte operations, b) multiple stacks, and c) multiple memories.

BYTE OPERATIONS

The effects of including byte operations as a basic design goal on VMPL and IML would be:

1) Similar to the global size declaration, we have a global byte declaration which gives the size of a byte in terms of the number of bits. The number of bytes in a word would then be:

$$\text{number of bytes} = \left\lceil \frac{\text{size of word}}{\text{size of byte}} \right\rceil - 1$$

The least significant byte would be byte zero.

2) In order to recognize a byte of a certain variable (word), a digit would have to be appended to the variable identifier in VMPL statements. The exact format may be similar to a FORTRAN index, a PASCAL structured element, etc., i.e.,

   ACC (0)

   ACC.0

   referring to the first byte of the variable ACC. The second format would be preferred because we already use square brackets to refer to memory variables and a set of two brackets may be less reliable.

3) The modification of variables as pointed out in statement b above would have to be migrated down the the IML level. Thus, the integer identifying the byte could be used as one of the operand modifiers in the IML statement.

MULTIPLE STACKS

The effect of including multiple stacks on VMPL and IML syntax would be:

1) The STACK and PSTACK global declaration statements would have to be modified. This would have to be done to recognize which stack pointer is associated with which stack.
2) The PUSH and POP statements, both in VMPL and IML, would similarly have to be modified to recognize the stack associated with the operation.

MULTIPLE MEMORIES

To include this aspect of target machines, the only change required of VMPL is the modification of the global MEMORY declaration. More than one memory variable would be allowed in the declaration. The RMOVE & WMOVE IML statements already include the name of the memory variable as part of their syntax.

From these three experiments, we infer that making **extensions to** VMPL, via the redefinition of the original premises on which the language was designed, does not effect the parts of VMPL and IML already designed.

In fact, the designed parts make it easy to modify and extend VMPL and consequently IML. Thus, because the additions were done easily, we consider this another aspect of a successful language design experiment.

6.5 CONCLUSIONS AND FUTURE WORK

The original objectives in developing a high level machine independent microprogramming language have been met as described in the body and appendices of this thesis.

All through this language design experiment we have followed a top down structured approach using all the data available on language syntax and statements. The language must be used to produce a number of large-scale emulators before the practical benefits of this design approach can be used as feedback to improve upon some of the language implications and maybe even modify some of the constraints.

Questions which remain to be considered in further research in this area include:

1) How does one produce even more efficient and reliable microcode from a high level microprogramming language?
2) How can high level languages be used for m-machines with two or more levels of microinstruction interpretation? Should the nano programs directly interpret the IML statements or not?
3) The study of universal emulators or hosts which are suitable for the emulation of a variety of targets if needed.
4) Finally, a more detailed study of I/O is needed for making the whole emulation process successful and accurate.

REFERENCES

1) M. V. Wilkes, 'The Best Way to Design an Automatic Calculating Machine', Report of the Manchester University Computer Inaugral Conference, Manchester, England, July 1951.

2) M. V. Wilkes, 'The Growth of Interest in Microprogramming: A Literature Survey'. Computing Surveys, September 1969.

3) S. S. Husson, Microprogramming Principles and Practices, Prentice Hall, Englewood Cliffs, New Jersey, 1970.

4) S. G. Tucker, 'Microprogram Control for System 1360', IBM Systems Journal, October 1967.

5) A. K. Agrawala, T. G. Rausher, Foundations of Microprogramming: Architecture, Software and Applications, Academic Press, New York 1976.

6) J. E. Nicholls, The Structure and Design of Programming Languages, Addison-Wesley, Menlo Park, California, 1975.

7) P. W. Mallett, T. G. Lewis, 'Considerations for Implementing a High Level Microprogramming Language Translation System', Computer Magazine, August 1975.

8) Y. Chu, Computer Organization and Microprogramming, Prentice Hall, Englewood Cliffs, New Jersey 1972.

9) C.V. Ramamoorthy, T. Masahiro, 'A High Level Language for Horizontal Microprogramming', IEEE Transactions on Computers, August 1974.

10) G. R. Lloyd, 'PUMPKIN - (Another) Microprogramming Language', SIGMICRO Newsletter, April 1974.

11) G. R. Lloyd, A. Van Dam, 'Design Considerations for Microprogramming Languages', SIGMICRO Newsletter, April 1974.

12) R. Eckhouse, 'A High Level Microprogramming Language', Sprin Joint Computer Conference, AFIPS Press, Montvale, New Jersey, 1971.

13) R. Eckhouse, 'A High Level Microprogramming Language (MPL)', Ph.D. Thesis, State University of New York at Buffalo, June 1971.

14) D. J. DeWitt, 'A Machine Independent Approach to the Production of Horizontal Microcode', Ph.D. Thesis, University of Michigan, June 1976.

15) J. E. Cheatem, et. all., 'On the Basis of ELF--an Extensive Language Facility', Spring Joint Computer Conference, AFIPS Press, Montvale, New Jersey, 1969.

16) R. F. Rosen, 'MPP--A Tool for Teaching and Research in Microprogramming', Technical Report PN-3R, SUNY at Buffalo, 1970.

17) L. C. Richardson, 'PRIM--Overview', ISI/RR-76-19, University of Southern California, February 1974.

18) MICRODATA 3200 COMPUTER, MICRO 32/S COMPUTER REFERENCE MANUAL, Microdata Corporation, May 1974.

19) MODEL 8/32 MICRO-PROGRAM DESCRIPTION, no. 05-058A15, Interdata Incorporated, December 1974.

20) BURROUGHS B1700 SYSTEMS REFERENCE MANUAL, Burroughs Corporation 1972.

21) QM-1 HARDWARE LEVEL USER's MANUAL, Nano-data Corporation, March 1974.

22) CASH-8 REFERENCE MANUAL, Standard Logic Incorporated, June, 1973.

23) 21MX COMPUTER SERIES REFERENCE MANUAL, Manual Part No. 02108-90002, Interdata Incorporation.

26) H. W. Lawson, B. Magnhagen, "Advantages of Structured Hardware", Second Annual Symposium on Computer Architecture, IEE, January 1975, (DATASAAB FCPU).

27) D. R. Oestreicher, J. Goldberg, "MLP-900 Reference Manual", Information Sciences Institute, University of Southern California, March 1974.

28) CONTROL DATA 5000 SERIES OF MICROPROGRAMMABLE PROCESSORS REFERENCE MANUAL, Publication No. 14232000, Control Data Corporation, August 1972.

29) "Data General Corporation Introduces New Eclipse Line of Small Computers", SIGMICRO Newsletter, October 1974.

30) CAL DATA 1 COMPUTER FAMILY, California Data Products, June 1974.

31) PRIME 300 COMPUTER, Prime Computer Incorporated.

32) VARIAN 73 SYSTEM HANDBOOK, Varian Data Machines, June 1972.

33) R. G. Barr, et. all,. "A Research-Oriented Dynamic Microprocessor", IEEE Transactions on Computers, November 1973.

34) D-MACHINE USERS MANUAL, Burroughs Corporation, April 1971, (BURROUGHS INTERPRETER).

35) B.D. Shriver, P. Kornerup, "An Overview of the MATHILDA System", Department of Computer Science, University of Aarhus, Aarhus, Denmark, 1975.

36) WRITABLE CONTROL STORE FOR PDP-11/40, 3 Rivers Corporation, June 1977.

37) SCHOTTKY BIPOLAR LSI MICROCOMPUTER SET: 3001 MICROPROGRAM CONTROL UNIT AND 3002 CENTRAL PROCESSING ELEMENT, Intel Corporation, 1975.

38) A. B. Salisbury, Microprogrammable Computer Architectures, Elsevier Computer Science Library, New York, 1976.

39) R. F. Rosin, 'Contemporary Concepts of Microprogramming and Emulation', Computing Surveys, December 1969.

40) J. T. Golden, FORTRAN IV Programming and Computing, Prentice Hall, Englewood Cliffs, New Jersey, 1965.

41) A. Lysegard, Introduction to COBOL, Studentlitteratur, Lund, Sweden, 1908.

42) M. E. Conway, 'Proposal for an UNCOL', Communications of the ACM, October 1958.

43) F. Bates and M. L. Douglas, Programming Language/One, Prentice Hall, Englewood Cliffs, New Jersey, 1970.

44) E. G. Mallach, 'Emulation: A survey', Honewell Computer Journal, Volume 6 Number 4, 1972.

45) Intel 8080 Reference Manual, Intel Corporation, Santa Clara, California, 1976.

46) C. G. BEll and A. Newall, Computer Structures, Readings and Examples, McGraw Hill, New York, 1971.

47) CDC 6600 REFERENCE MANUAL, Control Data Corporation, June 1976.

48) PDP 11/40 REFERENCE MANUAL, Digital Equipment Corporation, Maynard, Massachusetts, 1972.

49) A. Lunde, 'Empirical Evaluation of Some Features of Instruction Set Processor Architecture', Communications of the ACM, March 1977.

50) J. L. Elshoff, 'An Analysis of Some Commercial PL/I Programs;, IEEE Transactions on Software Engineering, June 1976.

51) D. E. Knuth, 'An Empirical Study of FORTRAN Programs', Software Practice and Experience, 1971.

52) J. D. Gannon, and J. J. Horning, 'The Impact of Language Design on the Production of Reliable Software', Proceedings International Conference on Reliable Software, ACM SIGPLAN Notices, June 1975.

53) R. Kosaraju, 'An Analysis of Structured Programs', Journal of Computing and Systems Science, December 1974.

54) H. F. Ledgard, and M. Marcotty, 'A Genealogy of Control Structures', Communications of the ACM, November 1975.

55) R. M. Lewis, D. J. Rosenkrantz, R. E. Stearns, Compiler Design Theory, Addison-Wesley, Menlo Park, California, 1976.

56) K. Jensen, N. Wirth, PASCAL Users Manual and Report, Springer-Verlag, New York, 1974.

57) W. M. Waite, R. J. Orgas, 'A Base for a Mobile Programming System', Communications of the ACM, September 1969.

58) M. H. Halstead, 'Using the Computer for Computer Conversion', Datamation, May 1970.

59) P. J. Brown, 'Levels of Languages for Portable Software', Communications of the ACM, December 1972.

60) F. L. Alt, 'The Standardization of Programming Languages', Proceedings of the ACM 19th National Conference, 1964.

61) R. C. Smeder, 'An Investigation of the Bootstrapping Process as Applied to Compiler Generation', A.U.S. Government Research Report, AD-727673, 1971.

62) M. C. Newey, P. D. Poole and W. M. Waite, 'Abstract Machine Modelling to Produce Portable Software', Software-Practice and Experience, Vol. 2, April 1972.

63) P. C. Poole, 'Hierarchical Abstract Machines;, Proceedings of the Software Engineering Conference, Culham, England 1971.

64) P. C. Poole, W. M. Waite, 'Machine Independent Software', Proceedings of the ACM, 2nd Symposium on Operating System Principles, 1969.

65) A. D. Fisher, 'A Common Programming Language for the DOD-Technical Requirements;, DOD Report No. AD-A028 297/Owc.

66) H. W. Lawson, 'Programming-Language-Oriented Instruction Streams', IEEE Transactions on Computers, May 1963.

67) R. E. Merwin et. all., 'Direct-Micorprogrammed Execution of the Intermediate Text From a High-Level Language Compiler'.

68) M. H. Halstead, Elements of Software Science, Elsevier North-Holland, 1977.

69) J. L. Elshoff, 'An Investigation into the Effect of the Counting Methods used on Software Science Measurements', SIGPLAN Notices, February 1978.

70) Schorre, 'META-II. A Syntax Oriented Compiler Writing System', Proceedings ACM 19th National Conference 1964.

71) G. A. Bachelor, 'META/CYBER 73 Reference Manual, Department of Computer Science, Oregon State University, 1975.

72) P. Y. Ma, "Optimizing Microcode Produced from a High Level Language", Ph.D. Thesis, Oregon State University, August 1978.

73) K. Malik, IML Simulator Reference Manual, Oregon State University, 1979.

APPENDICES

APPENDIX A

VMPL SYNTAX

This appendix defines the syntax of Virtual Microprogramming Language (VMPL) in a slightly modified BNF form. The differences from the standard BNF are:

1) In order to save space and repetition, the following statement is used

   ⟨A⟩ , ⟨B⟩ , ⟨C⟩ ::= ⟨E⟩

   which means

   ⟨A⟩ ::= ⟨E⟩
   ⟨B⟩ ::= ⟨E⟩
   ⟨C⟩ ::= ⟨E⟩

2) In case one of the meta symbols has to be used as part of the syntax of VMPL, it is enclosed in apostrophes, i.e.

   '⟨'

   meaning that the angle bracket (⟨) is used as a VMPL symbol and not a meta symbol.

B.N.F.

```
PROGRAM::= ⟨PROGRAM HEADING⟩ { ⟨GLOBAL DECL⟩ } ⟨BLOCKS⟩
             PROGRAM ENDING
 ⟨PROGRAM HEADING⟩ ::=EMULATOR: ⟨ID⟩ ;
 ⟨PROGRAM ENDING⟩ ::=ENDEMULATOR;
 ⟨GLOBAL DECL⟩ ::=DCL ⟨GLOBAL INFO DECL⟩ ;/DCL ⟨GLOBAL VAR DECL⟩ ;
 ⟨GLOBAL INFO DECL⟩ ::= ⟨WORDSIZE DECL⟩ /⟨ARITHMETIC DECL⟩ / ⟨FIELD DCL⟩
 ⟨WORDSIZE DECL⟩ ::=WORDSIZE ⟨INTEGER⟩
 ⟨ARITHMETIC DECL⟩ ::=ARITHMETIC ⟨ARITHMETIC TYPE⟩
 ⟨FIELD DCL⟩ ::=FIELD ⟨FIELD⟩ {,FIELD}
 ⟨FIELD⟩ ::= ⟨ID⟩ ( ⟨FBP⟩ , ⟨LBP⟩ , ⟨SHIFT COUNT⟩ )/ ⟨ID⟩ ( ⟨FBP⟩ , ⟨LBP⟩)
 ⟨FBP⟩ , ⟨LBP⟩ ::= ⟨INTEGER⟩
 ⟨SHIFT COUNT⟩ ::= ⟨INTEGER⟩ / ⟨NEG INTEGER⟩
 ⟨GLOBAL VAR DECL⟩ ::=PERMANENT ⟨GLOBAL TYPE DECL⟩ /TEMPORARY
                          ⟨GLOBAL TYPE DECL⟩
 ⟨GLOBAL TYPE DECL⟩ ::= ⟨MEMORY DECL⟩ / ⟨STACK DECL⟩ / ⟨PSTACK DECL⟩ /
                          ⟨EXTERNAL DECL⟩ / ⟨FLAG DECL⟩ / ⟨SIMPLE DECL⟩
 ⟨MEMORY DECL⟩ ::=MEMORY ⟨MEMORY⟩ ⟨SIZE DECL⟩
 ⟨MEMORY⟩ ::= ⟨ID⟩ : { ⟨INTEGER⟩ }
 ⟨STACK DECL⟩ ::=STACK ⟨STACK⟩ ⟨SIZE DECL⟩
```

```
<STACK> ::= <ID> :[ <INTEGER> ]/ <ID>
<PSTACK DECL> ::=PSTACK <PSTACK> <SIZE DECL> <STACK OPS>
<PSTACK> ::= <ID>
<EXTERNAL DECL> ::=EXTERNAL <EXTERNAL> {,<EXTERNAL>}
<EXTERNAL> ::= <ID> <EXT TYPE>
<FLAG DECL>::=FLAG <FLAG> {,<FLAG>}
<FLAG> ::= <ID> <FLG TYPE>
<SIMPLE DECL> ::=<ID> <SIZE DECL> {,<ID SIZE DECL>}
<SIZE DECL> ::=: <INTEGER> / <EMPTY>
<STACK OPS> ::= ↑<SOP1> / ↓<SOP1> / +<SOP2>/-<SOP2>
<SOP1> ::= +/-
<SOP2> ::= ↑/↓
<EXT TYPE> ::= :P/:F/L <INTEGER> / <EMPTY>
<FLG TYPE> ::=:C/:O/:N/:Z/ <EMPTY>
<BLOCKS> ::= <BLOCK> {<BLOCK>}
<BLOCK> ::=PROC: <ID> : <PROC BLOCK>/ SPROC: <ID> <SPROC ARG>;<SPROC BLOCK>
<PROC BLOCK> ::={<PROC DECL>}; # <CODE> #
<SPROC BLOCK> ::= {<SPROC DECL> ;} # <CODE> #
<SPROC ARG> ::= <EMPTY> /( <ARG> {, <ARG> } )
<ARG> ::= <ID> <SIZE DECL>
<PROC DECL> ::=DCL <GLB USE DECL> / DCL <LOCAL DECL> / DCL<SPR USE DECL>
<GLB USE DECL> ::=GLOBAL USE <ID> { , <ID> }
<LOCAL DECL> ::=PERMANENT <SIMPLE DECL> / TEMPORARY
                <SIMPLE DECL>
<SPR USE DECL> ::=SPROC USE <ID>{, <ID>}
<SPROC DECL> ::=DCL  GLOBAL <ER DECL> /  DCL LOCAL <ER DECL> /
                 DCL <SPR USE DECL>/ <GLB USE DECL>
<ER DECL> ;;=EXPECT <ID> {, <ID> /RETURN <ID> {, <ID>}
<CODE> ::= <STATEMENT> ; { <STATEMENT  ;>}
<STATEMENT> ::= <UNLABELLED STMT> / <LABEL> : <UNLABELLED STMT>
<UNLABELLED STMT> ::=<SIMPLE STMT> / <STRUCTURED STMT>
<SIMPLE STMT> ::=  <ASSIGNMENT STMT> <SET FLAGS> / <SET STMT>
                   <SET FLAGS> / <CLR STMT>  <SET FLAGS> / <INC STMT>
                   <SET FLAGS> / <DEC STMT>  <SET FLAGS> / <INC STMT>
                   <LEAVE STMT> / <GO TO STMT> / <XEQ STMT> /
                   <PUSH STMT> / <RETURN STMT> /
                    COMMENT STMT
<ASSIGNMENT STMT> ::= <CONCAT ASSIGN> / <SIMPLE ASSIGN>
<CONCAT ASSIGN> ::= <CONCAT VAR>  <ASSIGNMENT OPERATOR>
                    <CONCAT EXPR>
<CONCAT EXPR> ::=  <CONCAT FACTOR> / <CONCAT FACTOR>  <PM OPERATOR>
                   <INTEGER>
<CONCAT FACTOR> ::= <VAR> / <CONCAT VAR>
<CONCAT VAR> ::= <ID> '//' <ID>
<SIMPLE ASSIGN> ::= <VAR>  <ASSIGNMENT OPERATOR>  <RIGHT HAND SIDE> /
                    <SIMPLE CONCAT ASSIGN>
<VAR> ::= <ID> / <ARRAY VAR>
<ARRAY VAR> ::= <ARRAY ID>  <ARRAY SUBSCRIPT>
<ARRAY ID> ::=@ <ID> / <ID>
<ARRAY SUBSCRIPT> ::= <ID> / <INTEGER> / <POP>
<RIGHT HAND SIDE> ::= <TERM>  <OPERATOR>  <TERM> / <TERM>
<TERM> ::= <VAR> / <INTEGER> / <UNARY> / <SHIFT VAR> / <FIELD VAR> /
           <POP> / <ROTVAR>
<UNARY> ::= .NOT. <VAR>
<SHIFT VAR> ::= <VAR> <SHIFT OPERATOR> <INTEGER>
<FIELD VAR> ::= <ID> ( <VAR> )
<ROT VAR> ::= <VAR> <ROT OPERATOR> <INTEGER>
<SET STMT> ::= SET <VAR>
<CLR STMT> ::= CLAR <VAR>
<INC STMT> ::= INC <VAR>
<DEC STMT> ::= DEC <VAR>
<HLT STMT> ::= HALT
<LEAVE STMT> ::= LEAVE <LABEL>
<GOTO STMT> ::= GOTO <LABEL>
<XEQ STMT> ::= EXECUTE <ID> / EXECUTE <ID> ( <ID> {, <ID>})
<PUSH STMT> ::= PUSH <ID>
<RETURN STMT> ::= RETURN
<COMMENT STMT> ::= [* {<ALPHABET>} {<DIGIT>} {<SYMBOL>} *]
<SET FLAGS> ::= EMPTY /'<' ID {, <ID> '>'}
<STRUCTURED STMT> ::= <IFTRUE STMS> / <IFFALSE STMT> / <SELECT STMT> /
                      <COND STMT> / <FOR STMT> / <WHILE STMT>
<IFTRUE STMT> ::= IFTRUE <IFSTMT>
<IFFALSE STMT> ::= IFFALSE <IF STMT>
<IF STMT> ::= ( <BOOLEAN EXPRESSION> ) THEN; <COMPOUND STMT>
              ELSE <STMT> ENDIF
<SELECT STMT> ::= SELECT ( <ID> , <INTEGER> ) FROM;
                  <SLCT ITEM> ; {<SLCT ITEM> ;} ENDSELECT
<COND STMT> ::= COND; <COND ITEM> ; {<COND ITEM> ;} END COND
<FOR STMT> ::= FOR <ID> = <INTEGER> TO <INTEGER> ;
               <COMPOUND STMT>
<WHILE STMT> ::= WHILE ( <BOOLEAN EXPRESSION> ); <COMPOUND STMT>
                 ENDWHILE
<COMPOUND STMT> ::= BEGIN; <CODE> END;
<ELSE STMT> ::= EMPTY / ELSE; <COMPOUND STMT>
<BOOLEAN EXPRESSION> ::= <BOOL FACTOR> / <BOOL FACTOR> .EQ. <BOOL FACTOR>/
                         <BOOL FACTOR> .LT. <BOOL FACTOR> /<BOOL FACTOR> .GT.
                         <BOOL FACTOR> / <BIT VAR>
<BOOL FACTOR> ::= <TERM>
<SLCT ITEM> ::= ( <INTEGER> , <LABEL> )
<COND ITEM> ::= ( <BOOLEAN EXPRESSION> ); <COMPOUND>
<LABEL> ::= <ID>
<PM OPERATOR> ::= +/-
<ARITHMETIC TYPE> ::= 1/2
<NEGATIVE INTEGER> ::= - <INTEGER>
<OPERATOR> ::= +/-/*/ // .AND./.OR./.XOR.
<SHIFT OPERATOR> ::= .SHTLO./.SHTLI./.SHTRO./.SHTRI.
<ROT OPERATOR> ::= .ROTL./.ROTR.
<ASSIGNMENT OPERATOR> ::= '='
<INTEGER> ::= <DIGIT> <DIGIT>
<ID> ::= <ALPHABET> {<ALPHABET> / <DIGIT>}
<EMPTY> :=
<ALPHABET> :=A/B/C/D/E/F/G/H/I/J/K/L/M/N/O/P/Q/R/S/T/U/V/W/X/Y/Z
<DIGIT> :=0/]/2/3/4/5/6/7/8/9
<SMBOL> ::= ./// (/) /[/]/↑/↓/{/}/ +/-/*/./?/ /&/
```

APPENDIX B

SIMULATOR DESCRIPTION

The simulator for the IML is written in PASCAL. It is logically broken up into two parts. The first part sets up the symbol table, checks to see if all the variables have been used according to the requirements of VMPL, and sets up a binary equivalent of the IML.

The second part then executes the binary code set up by the first part. As indicated in Chapter 6, VMPL is enhanced by the COMMAND statement. The modification which this statement makes to VMPL BNF is shown in Fig. B.1.

All commands are given by three unique alphabets. While translating commands to IML, the IML operation is the command itself. However, in column 1, we have a down arrow ($\downarrow$) symbol which indicates to the later part of compilation (which produces microcode) that this IML statement is essentially a comment.

The simulator commands can be broken down into types:

1) DISPLAY
2) INSERT
3) CONTROL

The DISPLAY commands display the value of the various variables declared in the VMPL program. The INSERT command inserts (sets) values into the various variables declared in the VMPL program.

The CONTROL command covers a number of commands. These are:

a) Set and clear break points. The VMPL variables can be tagged such that whenever they are used, control returns to the simulator user.

b) Single step. This is for single step control of the IML program.

c) Symbol table. Display symbol tables set up in part 1 of simulator.

d) Error history. Display error history. When error occurs, this command gives information about the original VMPL statement which generated the IML statement which generated the error.

```
SIMPLE STMT   ::=  ALL AS IN APPENDIX A  /  COMMAND STMT

COMMAND STMT   ::=  DELETE CMDS    PARAMETERS  /
                   INSERT CMDS    PARAMETERS  /
                   CONTROL CMDS   PARAMETERS  /


DELETE CMDS   ::=    SEE (73)

INSERT CMDS   ::=    SEE (73)
CONTROL CMDS   ::=   SEE (73)
PARAMETERS  ::=      SEE (73)
```

Fig. B.1 MODIFICATION TO VMPL BNF

e) Option control. A number of options associated with error control can be enabled or disabled.

f) Load. Activate the loader to load data into the emulator memory.

g) Execution Control. This starts and stops monitor and starts and stops the simulation.

A detailed description of these commands, their parameters is given in (73).

APPENDIX C

This Appendix contains the VMPL-META compiler and the source of the emulator programs written in VMPL and referenced in Chapter 6 as VM1, VM2 and VM3. The programs are in the following order:

1) VMPL-META compiler.
2) VM1.
3) VM2.
4) VM3.

```
.SYNTAX VMPL     ≠META COMPILER FOR  V M P L .≠

.VARS SIZE,STK1,STK2  .,
.VARS TMP,GL,PT,DUM,CLM1  .,
.VARS OP1,OP2,N1,N2,N3  .,
.VARS CM,ZM.NM,OM   .,
.FLAGS ART,C1,C2,C3,NOPR,POP1. CAT1,CAT2  .,

VMPL = .LENID 6 .COMFLAG ≠[*          ≠
        ≠EMULATOR≠ ≠:≠ .ID .OUT( .LB ≠00A ≠ * .COL 72) .ONERROR RECVR1 ≠:≠
         $(GLOBALDECL) .OUT( .LB ≠008 PROGRAMSTART≠ .COL 72)
         $(≠PROC≠ PROC / ≠SPROC≠ SPROC /≠ENDEMULATOR≠ .OUT( .LB
          ≠00C PROGRAMEND≠ .COL 72) ≠:≠ .TRETURN)  .,

RECVR1 = .EARROW .MESSAGE ≠SYNTAX ERROR. SCANNING RESUMES AT NEXT <:>≠
           .SCAN ≠:≠  .,

[* GLOBAL DECLARATIONS *]


GLOBALDECL = ≠DCL≠ .SELECT
                   (≠WORDSIZE≠ WORDEF                                    ≠:≠ /
                    ≠ARITHMETIC≠ ARTDEF                                  ≠:≠ /
                    ≠GLOBAL≠ GLBDEF                                      ≠:≠ /
                    ≠IPROC≠ IPRDEF                                       ≠:≠ /
                    ≠FIELD≠ FLDDEF                                       ≠:≠ )
              .,

WORDEF = .INTEGER .OUT(.LB ≠00D ,,,≠ * .COL 72) .SET(SIZE, *) .,

ARTDEF = .PUT(.LB ≠00E≠) (≠1≠ .OUT(≠ONE≠ .COL 72) /
              ≠2≠ .OUT(≠TWO≠ .COL 72) / ≠ONE≠ .OUT(≠ONE≠ .COL 72) /
              ≠TWO≠ .OUT(≠TWO≠ .COL 72))  .,

IPRDEF = IPRD1 $(≠,≠ IPRD1)  .,

IPRD1 = .PUT(.LB ≠306 ≠) .SELECT
                          (≠C≠ .OUT(≠C≠ .COL 72)  /
                           ≠O≠ .OUT(≠O≠ .COL 72)  /
                           ≠N≠ .OUT(≠N≠ .COL 72)  /
                           ≠Z≠ .OUT(≠Z≠ .COL 72)  )  .,

FLDDEF = FLDOUT $(≠,≠ FLDOUT)  .,

FLDOUT = .ID .PUT(.LB ≠005 ≠ * ≠,,,≠) ≠(≠ .INTEGER .PUT(* ≠,≠) ≠,≠ .INTEG
         .PUT(* ≠,≠) (≠,≠(.INTEGER .PUT(*) ≠)≠ / ≠-≠ .INTEGER .PUT(≠-≠ *)
          ) / ≠)≠ .PUT(≠0≠) ) .OUT(.COL 72)   .,

GLBDEF = .SET(GL,≠2≠) PTDEF GLBD1  .,

PTDEF = (≠PERMANENT≠ .SET(PT,≠2≠) / ≠TEMPORARY≠ .SET(PT,≠1≠))  .,

GLBD1 = (≠MEMORY≠ .SET(TMP,≠1 ≠) MEMDEF                                /
          ≠STACK≠ .SET(TMP,≠2 ≠) STKDEF                                 /
           ≠PSTACK≠ .SET(TMP,≠3 ≠)PSTKDEF                               /
           ≠EXTERNAL≠ .SET(TMP,≠9 ≠) EXTDEF $(≠,≠ EXTDEF)               /
           ≠FLAG≠ .SET(TMP,≠4 ≠) FLAGDEF $(≠,≠ FLAGDEF)                 /
           ≠SIMPLE≠ .SET(TMP,≠0 ≠) SIMDEF $(≠,≠ SIMDEF)                 /
           .EMPTY .SET(TMP,≠0 ≠) SIMDEF $(≠,≠ SIMDEF)                     ) .,
MEMDEF = .ID TAGPUT .PUT(* ≠,≠) ≠:≠ ≠[≠ .INTEGER .PUT(* ≠,≠) ≠]≠ SIZEDECL
            .OUT(.COL 72)  .,

STKDEF = .ID TAGPUT .PUT(*≠,≠) ≠:≠ ≠[≠ .INTEGER .PUT(*≠,≠) ≠]≠ SIZEDECL
            .OUT(.COL 72)  .,

PSTKDEF = .ID TAGPUT .PUT(* ≠,,≠) ≠:≠ (.INTEGER .PUT(* ≠,≠) ≠:≠ ≠(≠ /
              ≠(≠ .PUT(SIZE ≠,≠)) STACKOPS ≠)≠ .OUT(.COL 72)  .,

STACKOPS = .SELECT
              (≠↑≠ .PUT(≠↑,≠) .SET(STK1,≠↓≠) STOP1 /
               ≠↓≠ .PUT(≠↓,≠) .SET(STK1,≠↑≠) STOP1 /
               ≠+≠ .PUT(≠+,≠) .SET(STK1,≠-≠) STOP2 /
               ≠-≠ .PUT(≠-,≠) .SET(STK1,≠+≠) STOP2 )  .,

FLAGDEF = TAGPUT .ID .SET(DUM,*) .PUT(* ≠,,1,≠)
             (≠:≠ ( ≠C≠ .SET(CM,DUM) .PUT(≠1≠)  /
                    ≠O≠ .SET(OM,DUM) .PUT(≠2≠)  /
                    ≠N≠ .SET(NM,DUM) .PUT(≠3≠)  /
                    ≠Z≠ .SET(ZM,DUM) .PUT(≠4≠)  )  /
              .EMPTY (.IFEQUAL(DUM,≠C≠) .SET(CM,≠C≠) .PUT(≠1≠)  /
                      .IFEQUAL(DUM,≠O≠) .SET(OM,≠O≠) .PUT(≠2≠)  /
                      .IFEQUAL(DUM,≠N≠) .SET(NM,≠N≠) .PUT(≠3≠)  /
                      .IFEQUAL(DUM,≠Z≠) .SET(ZM,≠Z≠) .PUT(≠4≠)  /
                      .EMPTY .PUT(≠0≠)               )) .OUT(.COL 72)  ..

EXTDEF = .ID TAGPUT .PUT(* ≠,,≠)(≠:≠ (≠P≠ .PUT(≠,P≠)/ ≠F≠ .PUT(≠1.F≠) /
            .INTEGER .PUT(*) ) / .EMPTY .PUT(SIZE))  .OUT(.COL 72)  .,

SIMDEF = .ID TAGPUT .PUT(* ≠,,≠)SIZEDECL .OUT(.COL 72)  .,

TAGPUT = .PUT(.LB GL PT TMP)  .,

STOP1 = (≠+≠ .PUT(≠+≠) .SET(STK2 ,≠-≠) /
         ≠-≠ .PUT(≠-≠) .SET(STK2,≠+≠) ) .PUT(≠,≠ STK2 ≠,≠ STK1)  ..

STOP2 = (≠↑≠ .PUT(≠↑≠) .SET(STK2,≠↓≠) /
         ≠↓≠ .PUT(≠↓≠) .SET(STK2,≠↑≠) ) .PUT(≠,≠ STK2 ≠,≠ STK1)  .,

SIZEDECL = (≠:≠ .INTEGER .PUT(*) / .EMPTY .PUT(SIZE))  .,


[* MAIN BLOCKS *]


PROC = ≠:≠ .ID .OUT(.LB ≠00F ≠* .COL 72) ≠:≠ $(≠DCL≠ PROCDECL ) CODE  .,

SPROC = ≠:≠ .ID .PUT(.LB ≠00G ≠*) ARG ≠:≠ $(≠DCL≠ SPROCDECL ) CODE  ..

ARG = ( ≠(≠ ARGCD .OUT(.COL 1 ≠*≠.COL 72) / .EMPTY .OUT(.COL 72))  .,
ARGCD = $(.ID .PUT(.COL 25 ≠A≠ *) (≠,≠ .ID .OUT(.COL 34 ≠A≠ * .COL 72)
              / ≠)≠ .OUT(.COL 72) .TRETURN)
           (≠,≠ .PUT(.COL 1 ≠*≠)/≠)≠ .TRETURN ))  .,
CODE = ≠E≠ .OUT(.LB ≠00H≠ .COL 72) EXECODE ≠E≠ .OUT(.LB ≠00I≠ .COL 72)  ..

PROCDECL = .SELECT
              (≠GLOBAL≠ ≠USE≠ GLUDEF ≠:≠       /
               ≠LOCAL≠ LCLDEF ≠:≠              /
               ≠SPROC≠ ≠USE≠ SPRDEF ≠:≠        )  .,

GLUDEF = GLUD1 $(≠,≠ GLUD1)  .,

GLUD1 = .ID .OUT(.LB ≠206 ≠ * .COL 72)  .,

LCLDEF = .SET(GL,≠1≠) PTDEF (≠SIMPLE≠/.EMPTY) .SET(TMP,≠0 ≠) SIMDEF
          $(≠,≠ SIMDEF)  .,

SPRDEF = SPR01 $(≠,≠ SPR01)  .,

SPR01 = .ID .OUT(.LB ≠406 ≠ * .COL 72)  .,

SPROCDECL = (≠SPROC≠ ≠USE≠ SPRDEF ≠:≠        /
              ≠GLOBAL≠ GLERD ≠:≠              /
              ≠LOCAL≠ LCLSERD ≠:≠             )  .,

GLERD = .SET(GL,≠2≠) .SET(PT,≠0≠) (≠EXPECT≠ .SET(TMP,≠7 ≠) /
        / ≠RETURN≠ .SET(TMP,≠8 ≠)) GLERD1 $(≠,≠ GLERD1)  .,

GLERD1 = .ID TAGPUT .OUT(* .COL 72)  .,

LCLSERD = .SET(GL.≠1≠) .SET(PT,≠0≠) (≠EXPECT≠ .SET(TMP.≠7 ≠) LCLER /
           ≠RETURN≠ .SET(TMP,≠8 ≠) LCLER / .EMPTY PTDEF (≠SIMPLE≠ /
           .EMPTY) .SET(TMP,≠0 ≠) LCLS )  .,

LCLER = GLERD1 $(≠,≠ GLERD1)  .,

LCLS = SIMDEF $(≠,≠ SIMDEF)  .,

EXECODE = .ONERROR RECVR1 $(.ID + ≠:≠ .OUT(.COL 1 * .COL 72)
           / STMT)  .,
STMT = (NOTASSIGN ≠:≠ / ASSIGN ≠:≠)  .,

NOTASSIGN = .SELECT
             (≠SET≠              SETSTMT   /
              ≠CLEAR≠            CLRSTMT   /
              ≠INC≠              INCSTMT   /
              ≠DEC≠              DECSTMT   /
              ≠HALT≠             HALTSTMT  /
              ≠RETURN≠           RETSTMT   /
              ≠LEAVE≠            LEAVSTMT  /
              ≠IFTRUE≠           IFTRSTMT  /
              ≠IFFALSE≠          IFFLSTMT  /
              ≠COND≠             CONDSTMT  /
              ≠BEGIN≠            COMPOUND  /
              ≠SELECT≠           SELTSTMT  /
              ≠GOTO≠             GOTOSTMT  /
              ≠EXECUTE≠          EXECSTMT  /
              ≠PUSH≠             PUSHSTMT  /
              ≠FOR≠              FORSTMT   /
           ≠↓≠                 CMDSTMT  /
              ≠WHILE≠            WILESTMT  )  .,
COMPOUND = ≠:≠ .OUT(.COL 1 ≠00J≠ .COL 72) .ONERROR RECVR1 $(≠END≠ .EXIT
            / .ID+≠:≠ .OUT(.COL 1 * .COL 72) / STMT) .OUT(.COL 1
             ≠00K≠ .COL 72)  .,

SETSTMT = .SET(OP1,≠SET≠) SCOUT1  .,
CLRSTMT = .SET(OP1,≠CLR≠) SCOUT1  .,
SCOUT2 = (≠POP+≠ .T+ .OUT(.COL 8 ≠POP≠ .COL 16 ≠+≠ *T .COL 72) .SET(N2.*T)
           / .ID .SET(N2,*) / .INTEGER .SET(N2,*) .SETFLAG C1 )  .,

SCOUT3 = .T+ .OUT(.COL 8 OP1 .COL 16 ≠+≠ *T .COL 72) SETFLAGS .PUT( .COL 8
           ≠WMOVE≠ .COL 17 N1 .COL 26 N2 .COL 34 ≠-≠ *T) (
           .IFTEMP N2 .PUT(.COL 25 ≠-≠)    / .IFFLAG C1 .PUT(.COL 25
           ≠C≠) / .EMPTY)  .,

SCOUT1 = .CLRFLAG C1 (.ID+≠[≠ .SET(N1,*) SCOUT2 ≠]≠ SCOUT3 /
               ≠≤≠ .ID+≠[≠ .SET(N1,*) SCOUT2 ≠]≠ SCOUT4 SCOUT3 / .ID SETFLAGS
               .PUT(.COL 8 OP1 .COL 17 *) ) .OUT(.COL 72)  .,

SCOUT4 = .T+ .PUT(.COL 8 ≠RMOVE≠ .COL 17 N1 .COL 26 N2 .COL 34 ≠+≠ *T
               )
            (.IFTEMP N2 .PUT(.COL 25 ≠-≠)/ .EMPTY)( .IFFLAG C1 .PUT(.COL 25
                 ≠C≠)/ .EMPTY) .OUT(.COL 72)  .SET(N2,*T)  .,

INCSTMT = .SET(OP1,≠ADDC≠) .SET(OP2,≠INC≠) IDOUT1  .,

DECSTMT = .SET(OP1,≠SUBC≠) .SET(OP2,≠DEC≠) IDOUT1  .,

IDOUT1 = .CLRFLAG C1 (.ID+≠[≠ .SET(N1,*) SCOUT2 ≠]≠ IDOUT2 /
                 ≠≤≠ .ID+≠[≠ .SET(N1,*) SCOUT2 ≠]≠ SCOUT4 IDOUT2 /
                .ID+≠//≠ .SET(N1,*) .ID .OUT(.COL 7 ≠+≠ OP2 .COL 17 *
         .COL 43 ≠C≠ .COL 72) SETFLAGS .OUT(.COL 8 OP1 .COL 17 N1 .COL 25
                ≠C0≠ .COL 35 N1  .COL 72 ) / .ID SETFLAGS .PUT(.COL 8 OP2
                .COL 17 *) ) .OUT(.COL 72)  .,
IDOUT2 = .T+ .PUT(.COL 8 ≠RMOVE≠ .COL 17 N1 .COL 26 N2 .COL 34 ≠+≠
           *T) (.IFTEMP N2 .PUT( .COL 25 ≠+≠) / .IFFLAG C1
       .PUT(.COL 25 ≠C≠) / .EMPTY ) .OUT(.COL 72) SETFLAGS .OUT(.COL 8 OP2
             .COL 16 ≠+≠ *T .COL 72)  .PUT(.COL 8
                 ≠WMOVE≠ .COL 17 N1 .COL 26 N2 .COL 34 ≠-≠ *T)
             (.IFTEMP N2 .PUT(.COL 25 ≠-≠)   / .IFFLAG C1 .PUT(.COL 25
              ≠C≠) / .EMPTY )  .,
HALTSTMT = .OUT(.COL 8 ≠HALT≠ .COL 72 )  .,

RETSTMT = .OUT(.COL 8 ≠RET≠ .COL 72 )  .,

LEAVSTMT = .ID .OUT(.COL 8 ≠BRCH≠ .COL 16 ≠B≠ * .COL 72 )  .,

IFTRSTMT = .CLRFLAG C2 ≠(≠IFTR≠)≠ IFOUT  .,

IFTR = TERM1 .STACK N2 (.IFFLAG C2 .SETFLAG C3 .CLRFLAG C2
            / .EMPTY) (≠,≠ .PUT(.COL 12 ≠F≠) BITOUT .TRETURN
            / ≠.≠ .SELECT
             (≠EQ≠.SET(OP1,≠Z≠) ≠.≠ TERM1 OPR2 .PUT(.COL 12 ≠F≠) IFOUT1 /
              ≠LT≠.SET(OP1,≠N≠) ≠.≠ TERM1 OPR2 .PUT(.COL 12 ≠F≠) IFOUT1 /
      ≠GE≠ .SET(OP1,≠Z,N≠) ≠.≠ TERM1 .UNSTACK DUM .STACK N2
         .SET(N2,DUM) CHFLAG OPR2 .PUT(.COL 12 ≠F≠) IFOUT1 /
       ≠LE≠ .SET(OP1,≠Z,N≠) ≠.≠ TERM1 OPR2 .PUT(.COL 12 ≠F≠) IFOUT1 /
              ≠NE≠ .SET(OP1,≠Z≠) ≠.≠ TERM1 OPR2 .PUT(.COL 12 ≠T≠) IFOUT1 /
              ≠GT≠.SET(OP1,≠N≠) ≠.≠ TERM1 OPR2 .PUT(.COL 12 ≠T≠) IFOUT1 )
              )  .,

IFFLSTMT =  .CLRFLAG C2 ≠(≠ IFFL ≠)≠ IFOUT  .,
CHFLAG = .CLRFLAG C1 (.IFFLAG C2 .SETFLAG C1 / .EMPTY)
              (.IFFLAG C3 .SETFLAG C2 /.CLRFLAG C2)
              (.IFFLAG C1 .SETFLAG C3 /.CLRFLAG C3)  .,

IFFL = TERM1 .STACK N2 (.IFFLAG C2 .SETFLAG C3 .CLRFLAG C2
            / .EMPTY) (≠,≠ .PUT(.COL 12 ≠F≠) BITOUT .TRETURN /
           ≠.≠ .SELECT
           (≠EQ≠ .SET(OP1,≠Z≠) ≠.≠ TERM1 OPR2 .PUT(.COL 12 ≠T≠) IFOUT1 /
            ≠LT≠ .SET(OP1,≠N≠) ≠.≠ TERM1 OPR2 .PUT(.COL 12 ≠T≠) IFOUT1 /
    ≠NE≠ .SET(OP1,≠Z≠) ≠.≠ TERM1 OPR2 .PUT(.COL 12 ≠F≠) IFOUT1 /
    ≠LE≠ .SET(OP1,≠Z,N≠) ≠.≠ TERM1 OPR2 .PUT(.COL 12 ≠T≠) IFOUT1 /
    ≠GE≠ .SET(OP1,≠Z,N≠) ≠.≠ TERM1 .UNSTACK DUM .STACK N2
    .SET(N2,DUM) CHFLAG OPR2 .PUT(.COL 12 ≠T≠) IFOUT1 /
            ≠GT≠ .SET(OP1,≠N≠) ≠.≠ TERM1 OPR2 .PUT(.COL 12 ≠F≠) IFOUT1 )
              )  .,

IFOUT1 = .PUT(.COL 8 ≠COND≠ .COL 17 OP1)  .,

IFOUT = .OUT(.COL 25 ≠T≠ *1 .COL 72) ≠THEN≠ ≠:≠
      (.ID+≠:≠ .OUT(.COL 1 * .COL 72) / .EMPTY) NOTASSIGN ≠:≠ (≠ELSE≠
       ≠:≠ .OUT(.COL 8 ≠BRCH≠
          .COL 16 ≠E≠ .COL 17 *2 .COL 72 ) .OUT(.COL 1 *1 .COL 72 )
         (.ID+≠*≠ .OUT(.COL 1 * .COL 72) / .EMPTY) NOTASSIGN ≠;≠ .OUT(
           .COL 1 *2 .COL 72) /
          .EMPTY .OUT(.COL 1 *1 .COL 72)) ≠ENDIF≠  .,

OPR2 = .UNSTACK N1 .PUT(.COL 7 ≠*COMP≠ .COL 17 N1 .COL 26 N2
          .COL 43 OP1)
          (.IFTEMP N1 .PUT(.COL 16 ≠-≠)   / .EMPTY )
          (.IFTEMP N2 .PUT(.COL 25 ≠-≠)   / .EMPTY )
          (.IFFLAG C3 .PUT(.COL 16 ≠C≠) .CLRFLAG C3 / .EMPTY)
          (.IFFLAG C2 .PUT(.COL 25 ≠C≠) .CLRFLAG C2 / .EMPTY)
          .OUT(.COL 72)  .,

BITOUT = .INTEGER .PUT(.COL 8 ≠COND≠ .COL 16 ≠.≠ N2 ≠,≠ *)
         (.IFTEMP N2 .PUT(.COL 15 ≠-≠) / .EMPTY)  .,

CONDSTMT = ≠:≠ $( ≠(≠ CONDITEM / ≠ENDCOND≠ .TRETURN )  .,

CONDITEM = IFTR .OUT(.COL 25 ≠L≠ *1 .COL 72) ≠)≠ ≠:≠
        (.ID+≠:≠ .OUT(.COL 1 * .COL 72) / .EMPTY) NOTASSIGN ≠:≠ .OUT(
          .COL 1 *1 .COL 72)  .,

SELTSTMT =.PUT(.COL 9 ≠SLCT≠) ≠(≠ .ID .PUT( .COL 17 *) ≠,≠ .INTEGER ≠)≠
         .OUT(.COL 25 ≠C≠ * .COL 72) ≠FROM≠ ≠:≠ $(≠(≠ SLITEM ≠)≠ ≠:≠ /
            ≠ENDSELECT≠ .OUT(.COL 1 ≠*≠ .COL 72 ) .TRETURN )  .,

SLITEM = .INTEGER .PUT(.COL 1 ≠*≠ .COL 7 ≠C≠ *) ≠,≠ .ID .OUT(.COL 16 ≠S≠
         * .COL 72)  .,

GOTOSTMT = .ID .OUT(.COL 8 ≠BRCH≠ .COL 16 ≠G≠ * .COL 72)  .,

EXECSTMT = .ID .PUT(.COL 8 ≠XEQ≠ .COL 17 *) ( ≠(≠ EXECPAR .OUT(.COL 1
         ≠*≠ .COL 72) / .EMPTY .OUT(.COL 72))  .,


EXECPAR =$(.ID .PUT(.COL 25 ≠P≠ *)(≠,≠ .ID .OUT(.COL 34 ≠P≠ * .COL 72)
        / ≠)≠ .OUT(.COL 72) .TRETURN) (≠,≠ .PUT(.COL 1 ≠*≠) / ≠)≠
        .TRETURN ))  .,

PUSHSTMT = .CLRFLAG C2 (.INTEGER .SET(N2, *) .SETFLAG C3 / .EMPTY TERM2)
            .PUT(.COL 8  ≠PSH≠ .COL 17 N2)
             (.IFFLAG C2 .PUT(.COL 15 ≠C≠) / .EMPTY)
             (.IFTEMP N2 .PUT(.COL 16 ≠-≠) / .EMPTY)
             SETFLAGS .OUT(.COL 72)  .,

FORSTMT = .OUT(.COL 1 *1 .COL 72) .ID .PUT(.COL 9 ≠LOOP≠ .COL17 *) ≠=≠
  .INTEGER .PUT(.COL 25 ≠C≠ *) ≠TO≠ .INTEGER .OUT(.COL 34 ≠C≠ * .COL 72
     ) ≠;≠ (.ID+≠:≠ .OUT(.COL 1 * .COL 72) / .EMPTY) NOTASSIGN ≠:≠
          ≠ENDFOR≠ .OUT(.COL 8 ≠BRCH≠ .COL 16 ≠F≠ *1 .COL 72)  .,

WILESTMT = ≠(≠ .OUT(.COL 1 *1 .COL 72) IFTR .OUT(.COL 25 ≠L≠ *2 .COL 72
         ) ≠)≠ ≠:≠
        (.ID+≠:≠  .OUT(.COL 1 * .COL 72) / .EMPTY) NOTASSIGN ≠:≠
           .OUT(.COL 8 ≠BRCH≠ .COL 16 ≠L≠ *1 .COL 72 / .COL 1 *2
          .COL 72) ≠ENDWHILE≠  .,

CMOSTMT = .ID .OUT(.COL 1 ≠↓   ≠ * .COL 72) ≠↓≠  .,
ASSIGN = .CLRFLAG C2 .CLRFLAG C3 .CLRFLAG POP1 .CLRFLAG NOPR
          .CLRFLAG ART .CLRFLAG C1 .CLRFLAG CAT1 .CLRFLAG CAT2
```

```
                ( .ID+≠//≠ CATASSG / .EMPTY SIMASSG )   ..

       SIMASSG =(.ID+≠[≠ .STACK * SBVAR1 ≠]≠ (≠=*≠ .SETFLAG ART/≠=≠
                 .EMPTY) EXPR2 (.IFFLAG NOPR .EMPTY / .EMPTY EXPR1) ASS1 /
                 .ID .STACK * (≠=*≠ .SETFLAG ART / ≠=≠ .EMPTY) EXPR2
                 (.IFFLAG NOPR EXMV / .EMPTY EXPR1) ASS2 /
                 ≠≤≠ .ID+≠[≠ .STACK * SBVAR1 ≠]≠ (≠=*≠ .SETFLAG ART / ≠=≠
                 .EMPTY) EXPR2 (.IFFLAG NOPR .EMPTY / .EMPTY EXPR1) ASS3)   ..


       SBVAR1 = (≠POP+≠ .SETFLAG POP1 /.ID .STACK * / .INTEGER .STACK *
                 .SETFLAG C1 )   ..

       EXPR1 = (.IFFLAG NOPR .EMPTY / .EMPTY .UNSTACK N3 .PUT(.COL 8 OP1
                .COL 17 N3 .COL 26 N2)
                (.IFFLAG C3 .PUT(.COL 16 ≠C≠) / .EMPTY)
                (.IFFLAG C2 .PUT(.COL 25 ≠C≠) / .EMPTY)
               (.IFTEMP N3 .PUT(.COL 16 ≠-≠) / .EMPTY)
                (.IFTEMP N2 .PUT(.COL 25 ≠-≠) / .EMPTY)
             (.IFFLAG CAT1 .PUT(.COL 25 ≠/≠) / .EMPTY)
              (.IFFLAG CAT2 .PUT(.COL 16 ≠/≠) / .EMPTY)
                SETFLAGS (.IFFLAG ART .PUT(.COL 7 ≠%≠) / .EMPTY)) ..



       EXPR2 = TERM1 .STACK N2 (.IFFLAG C2 .SETFLAG C3 .CLRFLAG C2 / .EMPTY )
            (.IFFLAG CAT1 .SETFLAG CAT2 .CLRFLAG CAT1 / .EMPTY)
               ( ≠+≠ TERM1 .SET(OP1,≠ADD≠)                              /
                ≠-≠ TERM1 .SET(OP1,≠SUB≠)                               /
                ≠*≠ TERM1 .SET(OP1,≠MPY≠)                               /
                ≠/≠ TERM1 .SET(OP1,≠DVD≠)                               /
                ≠.AND.≠ TERM1 .SET(OP1,≠AND≠)                           /
                ≠.OR.≠ TERM1 .SET(OP1,≠OR≠)                             /
                ≠.XOR.≠ TERM1 .SET(OP1,≠XOR≠)                           /
                .EMPTY .SETFLAG NOPR                                     )   ..

       TERM1 = (≠.NOT.≠ TERM2 .T+ .PUT(.COL 8 ≠NOT≠ .COL 17 N2 .COL 25 ≠+≠ *T )
                (.IFTEMP N2 .PUT(.COL 16 ≠-≠) / .EMPTY) .OUT(.COL 72)
             .SET(N2,*T) .TRETURN /
                .INTEGER .SET(N2,*) .SETFLAG C2 .TRETURN / .EMPTY TERM2 )
                ( ≠.SHTL0.≠ .INTEGER .SET(OP1,≠L≠) .SET(OP2,≠0≠) SHOUT        /
                  ≠.SHTR0.≠ .INTEGER .SET(OP1,≠R≠) .SET(OP2,≠0≠) SHOUT       /
                  ≠.SHTL1.≠ .INTEGER .SET(OP1,≠L≠) .SET(OP2,≠1≠) SHOUT       /
                  ≠.SHTR1.≠ .INTEGER .SET(OP1,≠R≠) .SET(OP2,≠1≠) SHOUT       /
                  ≠.ROTR.≠ .INTEGER .SET(OP1,≠R≠) ROUT /
                  ≠.ROTL.≠ .INTEGER .SET(OP1,≠L≠) ROUT /
                  .EMPTY )   ..

       ROUT = .T+ .PUT(.COL 8 ≠RO≠ OP1 .COL 17 N2 .COL 26 *
               .COL 34 ≠+≠ *T) (.IFTEMP N2 .PUT(.COL 16 ≠-≠) / .EMPTY)
                 (.IFFLAG CAT1 .PUT(.COL 16 ≠/≠) / .EMPTY)
               .OUT(.COL 72) .SET(N2,*T)   ..
       SHOUT = .T+ .PUT(.COL 8 ≠SH≠OP1 .COL 17 N2 .COL 26 * ≠,≠ OP2 .COL 34 ≠+≠
                *T ) ( .IFTEMP N2 .PUT(.COL 16 ≠-≠) / .EMPTY)
             (.IFFLAG CAT1 .PUT(.COL 16 ≠/≠) / .EMPTY) .OUT(.COL 72)
               .SET(N2,*T)   ..

       TERM2 = (.ID+≠[≠ SBVAR3 .SET(N2,*T)                                /
                .ID+≠(≠ FLOVAR .SET(N2,*T)                                /
                .ID+≠//≠ .SET(DUM1,*) .ID .SET(N2,DUM1 ≠,≠ *) .SETFLAG CAT1 /
                ≠POP+≠ POP2 .SET(N2,*T)                   /
                .ID .SET(N2,*)                                            /
                ≠≤≠ .ID+≠[≠ SBVAR3 SBVAR5 .SET(N2,*T)                     )   ..
```

```

FLOVAR = .STACK * (.ID+≠[≠ SBVAR3 FLD1          /
           ≠POP+≠ POP2 .SET(N2,*T) FLD1       /
           .ID .SET(N2,*) FLD1               /
           ≠S≠ .ID+≠[≠ SBVAR3 SBVAR5 FLD1   ) ≠)≠   .,
FLD1 = .T+ .PUT(.COL 8 ≠EXTR≠) .UNSTACK DUM1 .PUT(.COL 17 DUM1
         .COL 34 ≠+≠ *T) (.IFTEMP N2 .PUT(.COL 25 ≠-≠) / .EMPTY) .OUT(
         .COL 26 N2 .COL 72)  .,
POP2 = .T+ .OUT(.COL 8 ≠POP≠ .COL 16 ≠+≠ *T.COL 72 )   .,
SBVAR3 = .SET(N1,*) SBVAR4 ≠]≠    .,

SBVAR4 =.T+ (≠POP*≠.OUT(.COL 8 ≠POP≠ .COL 16 ≠+≠ *T .COL 72).PUT(.COL 8
  ≠RMOVE≠ .COL 17 N1 .COL 25 ≠-≠ *T .COL 34 ≠+≠) .T+ .OUT(*T .COL 72)
/ .ID .OUT(.COL 8 ≠RMOVE≠ .COL 17 N1 .COL 26 * .COL 34 ≠+≠ *T .COL 72 )
        / .INTEGER .OUT(.COL 8 ≠RMOVE≠ .COL 17 N1 .COL 25 ≠C≠ * .COL 34
          ≠+≠ *T .COL 72 ))   .,

SBVAR5 = .PUT(.COL 8 ≠RMOVE≠ .COL 17 N1 .COL 25 ≠-≠ *T) .T+
           .OUT(.COL 34 ≠+≠ *T .COL 72)   .,

EXMV =.UNSTACK DUM .UNSTACK DUM .PUT(.COL 8 ≠MOVE≠.COL 17 N2 .COL 26
          DUM) (.IFFLAG C3 .PUT(.COL 16 ≠C≠ ) / .EMPTY) SETFLAGS
          (.IFTEMP N2 .PUT(.COL 16 ≠-≠) / .EMPTY)
          (.IFFLAG ART .PUT(.COL 7 ≠%≠ )
           (.IFFLAG CAT1 .PUT(.COL 16 ≠/≠) / .EMPTY)
           / .EMPTY)   .,
ASS1 = (.IFFLAG NOPR .UNSTACK N1 / .T+ .OUT(.COL 34 ≠+≠ *T .COL 72)
           .SET(N1,*T)) (.IFFLAG POP1 .T+ .OUT(.COL 8 ≠POP≠ .COL 16 ≠+≠
    .COL 72 *T) .SET(N2,*T) / .EMPTY .UNSTACK N2) .UNSTACK N3 .PUT(
           .COL 8 ≠WMOVE≠ .COL 17 N3 .COL 26 N2 .COL 35 N1)
          (.IFTEMP N1 .PUT(.COL 34 ≠-≠) / .EMPTY)
          (.IFFLAG NOPR SETFLAGS (.IFFLAG C3 .PUT(.COL 34 ≠C≠) /
        .EMPTY) (.IFFLAG CAT2 .PUT(.COL 34 ≠/≠) / .EMPTY) / .EMPTY)
          (.IFTEMP N2 .PUT(.COL 25 ≠-≠) / .EMPTY)
          (.IFFLAG C1 .PUT(.COL 25 ≠C≠) / .EMPTY) .OUT(.COL 72)   .,
ASS2 = (.IFFLAG NOPR .EMPTY / .UNSTACK DUM .PUT(.COL 35 DUM))
         .OUT(.COL 72)   .,
ASS3 =(.IFFLAG NOPR .UNSTACK N1 / .T+ .OUT(.COL 34 ≠+≠ *T .COL 72)
          .SET(N1,*T))
          (.IFFLAG POP1 .T+ .OUT(.COL 8 ≠POP≠ .COL 16 ≠+≠
         *T .COL 72) .SET(N2,*T) / .EMPTY .UNSTACK N2) .UNSTACK N3 .T+
                                          .PUT(.COL 8 ≠RMOVE≠
   .COL 17 N3 .COL 25 N2 .COL 34 ≠+≠ *T) (.IFTEMP N2 .PUT(.COL 25 ≠-≠)
         / .EMPTY) (.IFFLAG C1 .PUT(.COL 25 ≠C≠)/ .EMPTY) .OUT(.COL 72)
         .PUT(.COL 8 ≠WMOVE≠ .COL 17 N3 .COL 25 ≠-≠ *T .COL 35 N1)
         (.IFTEMP N1 .PUT(.COL 34 ≠-≠) / .EMPTY) (.IFFLAG NOPR SETFLAGS
         (.IFFLAG C3 .PUT(.COL 34 ≠C≠) / .EMPTY)
         (.IFFLAG CAT2  .PUT(.COL 34 ≠/≠) / .EMPTY) / .EMPTY
          ) .OUT(.COL 72)   .,

CATASSG = .SET(STK1,*) .ID .SET(STK2,*)
                (≠=*≠ .SETFLAG ART / ≠=≠ .EMPTY) EXPR2
              (.IFFLAG NOPR .SET(OP1,≠MOVE≠) .SET(N2.≠        ≠) /
              .EMPTY) .UNSTACK DUM1 .PUT(.COL 8 OP1
                .COL 17 DUM1 .COL 26 N2)
              (.IFFLAG NOPR .PUT(.COL 25 ≠/≠ STK1 ≠,≠ STK2) / .EMPTY
              .PUT(.COL 34 ≠/≠ STK1 ≠,≠ STK2))
              (.IFFLAG CAT2 .PUT(.COL 16 ≠/≠) / .EMPTY)
              (.IFFLAG C3 .PUT(.COL 16 ≠C≠) / .EMPTY)
              (.IFFLAG CAT1 .PUT(.COL 25 ≠/≠) / .EMPTY)
              (.IFFLAG C2 .PUT(.COL 25 ≠C≠) / .EMPTY)
             (.IFTEMP DUM1 .PUT(.COL 16 ≠-≠) / .EMPTY)
               (.IFTEMP N2 .PUT(.COL 16 ≠-≠) / .EMPTY) SETFLAGS
```

```
                 (.IFFLAG ART .PUT(.COL 7 ≠%≠) / .EMPTY) .OUT(.COL 72)  .,

SETFLAGS = .PUT(.COL 43) (.IF ≠:≠ .TRETURN / ≠<≠ .EMPTY)  $(.ID
                SYMTEST /≠>≠ .TRETURN / ≠,≠ .EMPTY)  .,

SYMTEST = (.IFEQUAL(CM,*) .PUT(≠C≠)  UPARW /
           .IFEQUAL(ZM,*) .PUT(≠Z≠)  UPARW /
           .IFEQUAL(NM,*) .PUT(≠N≠)  UPARW /
           .IFEQUAL(OM,*) .PUT(≠O≠) UPARW )  .,
UPARW = .PUT(.COL 7 ≠*≠)  .,

.END
 .EOF
```

```
EMULATOR:PDP8 :
[*
  THIS IS AN EMULATOR FOR THE PDP-8 MINICOMPUTER
*]

[*
  MAIN DECLARATIONS
*]
DCL WORDSIZE 12 :
DCL ARITHMETIC 2 :
DCL GLOBAL PERMANENT MEMORY MEM:[4096] :
DCL GLOBAL PERMANENT SIMPLE ACCM,PC,MAR :
DCL GLOBAL TEMPORARY IR,MDR,OPCD:3 :
DCL GLOBAL TEMPORARY FLAG LINK:C :
DCL GLOBAL PERMAN NT EXTERNAL IOINST:P,DATADWH:12 :
DCL FIELD OPCODE(9,11,-9),CRNTPG(6,11),
          PGADR(0,6)      ,ROTFLD(1,3,-1),
          OSC(3,8,-3)     ,OSB(0,2) :
DCL IPROC 0 :
[*
  START OF FIRST PROCEDURE - INSTRUCTION FETCH
*]
PROC:INF :
[*
  PROCEDURE DECLARATIONS
*]
 DCL GLOBAL USE MEM,IR,PC :
[*
  EXECUTABLE CODE
*]
=
  IR=MEM[PC] :
  INC PC :
=
[*
  INSTRUCTION DECODE
*]
PROC:INSTDCD :
 DCL GLOBAL USE IR,OPCD :
=
  OPCD=OPCODE(IR) :
  SELECT (OPCD,3) FROM :
   (0,MRI) :
   (1,MRI) :
   (2,MRI) :
   (3,DCA) :
   (4,JMS) :
   (5,JMP) :
   (6,IO)  :
   (7,OPT) :
  ENDSELECT :
=
[*
  THIS PROCEDURE CALCULATES THE EFFECTIVE ADDRESS FOR
  MEMORY REFERENCE INSTRUCTIONS
*]
SPROC:EFTADR :
 DCL GLOBAL EXPECT MEM,IR,PC :
 DCL GLOBAL RETURN MAR :
 DCL LOCAL PERMANENT ADR:7,PCTEMP,MART :
```

```
≡
[*
  IS CURRENT PAGE, PAGE ZERO
*]
  ADR=PGEADR(IR) :
  IFTRUE (IR,7) THEN :
   BEGIN :
    PCTEMP= PC-1 :
    PCTEMP=CRNTPG(PCTEMP) :
    MAR=PCTEMP.OR.ADR :
   END :
  ELSE :
   BEGIN :
    MAR=ADR :
   END :
  ENDIF :
[*
  DIRECT-INDIRECT ADDRESSING
*]
  IFTRUE (IR,8) THEN :
   BEGIN :
    MART=MEM[MAR] :
    IFTRUE (MAR.GT.7) THEN :
     BEGIN :
      IFTRUE (MAR.LT.16) THEN :
       BEGIN :
        MEM[MAR]= MEM[MAR]+1 :
       END :
      ENDIF :
     END :
    ENDIF :
    MAR=MART :
   END :
  ENDIF :
  RETURN :
[*
  MAR NOW CONTAINS THE EFFECTIVE ADDRESS
*]
≡
[*
  MEMORY REFERENCE INSTRUCTIONS
*]
PROC:MRI :
 DCL GLOBAL USE MAR,MEM,MDR :
 DCL SPROC USE EFTADR :
≡
  EXECUTE EFTADR :
  MDR=MEM[MAR] :
  SELECT (OPCD,2) FROM :
   (0,AND) :
   (1,TAD) :
   (2,ISZ) :
  ENDSELECT :
≡
PROC:AND :
 DCL GLOBAL USE ACCM,MDR,LINK :
≡
  ACCM=ACCM.AND.MDR :
  LEAVE INF :
≡
PROC:TAD :
 DCL GLOBAL USE ACCM,MDR,LINK :
≡
```

```
  ACCM= ACCM+MDR <LINK> :
  LEAVE INF :
≡
PROC:ISZ :
 DCL GLOBAL USE ACCM,MDR,MAR,PC :
≡
  MEM[MAR]= MEM[MAR]+1 :
  IFTRUE (MEM[MAR].EQ.0) THEN :
   BEGIN :
    INC PC :
   END :
  ENDIF :
  LEAVE INF :
≡
PROC:DCA :
 DCL GLOBAL USE MEM,ACCM :
≡
  EXECUTE EFTADR :
  MEM[MAR]=ACCM :
  CLEAR ACCM :
  LEAVE INF :
≡
PROC:JMS :
 DCL GLOBAL USE MEM,MAR,PC :
≡
  EXECUTE EFTADR :
  MEM[MAR]=PC :
  INC MAR :
  PC=MAR :
  LEAVE INF :
≡
PROC:JMP :
 DCL GLOBAL USE PC,MAR :
≡
  EXECUTE EFTADR :
  PC=MAR :
  LEAVE INF :
≡
PROC:IO :
 DCL GLOBAL USE IR :
 DCL SPROC USE IOINST :
 DCL LOCAL PERMANENT DS:6,OS:3 :
≡
  DS=DSC(IR) :
  OS=OSB(IR) :
  EXECUTE IOINST(DS.OS) :
≡
PROC:OPT :
 DCL GLOBAL USE IR :
≡
  IFTRUE (IR,8) THEN :
   BEGIN :
    LEAVE OPR1 :
   END :
  ELSE :
   BEGIN :
    LEAVE OPR2 :
   END :
  ENDIF :
≡
PROC:OPR1 :
 DCL GLOBAL USE IR,ACCM,LINK :
 DCL LOCAL PERMAN NT ROTACT:3 :
```

```
=
  COND :
   (IR,7) :
    BEGIN :
     CLEAR ACCM :
    END :
   (IR,6) :
    BEGIN :
     CLEAR LINK :
    END :
   (IR,5) :
    BEGIN :
     ACCM=.NOT.ACCM :
    END :
   (IR,4) :
    BEGIN :
     LINK=.NOT.LINK :
    END :
   (IR,0) :
    BEGIN :
     ACCM= ACCM+1 :
    END :
  ENDCOND :
  ROTACT=ROTFLD(IR) :
  SELECT(ROTACT,7) FROM :
   (0,INF) :
   (1,INF) :
   (2,RAL) :
   (3,RTL) :
   (4,RAR) :
   (5,RTR) :
   (6,INF) :
   (7,INF) :
  ENDSELECT :
=
PROC:RAL :
 DCL GLOBAL USE LINK,ACCM :
=
  LINK//ACCM=LINK//ACCM.ROTL.1 :
=
PROC:RTL :
 DCL GLOBAL USE LINK,ACCM :
=
  LINK//ACCM=LINK//ACCM.ROTL.2 :
=
PROC:RAR :
 DCL GLOBAL USE ACCM.LINK :
=
  LINK//ACCM=LINK//ACCM.ROTR.1 :
=
PROC:RTR :
 DCL GLOBAL USE LINK,ACCM ;
=
  LINK//ACCM=LINK//ACCM.ROTR.2 :
=
PROC:OPR2 :
 DCL GLOBAL USE IR,ACCM,PC,LINK,DATASWH :
 DCL LOCAL PERMAN NT COUNT:2,CHECK:2 :
=
  CLEAR COUNT :
  CLEAR CHECK :
  COND :
   (IR,6) :
    BEGIN :
     COUNT=1 :
     IFTRUE (ACCM.LT.0) THEN :
      BEGIN :
       CHECK=1 :
      END :
     ENDIF :
    END :
   (IR,5) :
    BEGIN :
     COUNT=COUNT+1 :
     IFTRUE(ACCM.EQ.0) THEN :
      BEGIN :
       CHECK=CHECK+1 :
      END :
     ENDIF :
    END :
   (IR,4) :
    BEGIN :
     COUNT=COUNT+1 :
     IFTRUE (LINK.EQ.1) THEN :
      BEGIN :
       CHECK=CHECK+1 :
      END :
     ENDIF :
    END :
  ENDCOND :
  IFFALSE (IR,3) THEN :
   BEGIN :
    IFTRUE (CHECK.NE.0) THEN :
     BEGIN :
      INC PC :
     END :
    ENDIF :
   END :
  ELSE :
   BEGIN :
    IFTRUE (CHECK.EQ.COUNT) THEN :
     BEGIN :
      INC PC :
     END :
    ENDIF :
   END :
  ENDIF :
  COND :
   (IR,7) :
    BEGIN :
     CLEAR ACCM :
    END :
   (IR,2) :
    BEGIN :
     ACCM=ACCM.OR.DATASWH :
    END :
   (IR,1) :
    BEGIN :
     HALT :
    END :
 ENDCOND :
  LEAVE INF :
E
ENDEMULATOR :
```

```
EMULATOR:I8080 ;
[* THIS IS AN EMULATOR FOR THE INTEL 8080 MICROPROCESSOR *]
[*
   G L O B A L   D E C L A R T I O N S  *]
DCL WORDSIZE 8 ;
DCL ARITHMETIC 2 ;
DCL GLOBAL PERMANENT MEMORY MEM,[65536] ;
DCL GLOBAL PERMANENT SIMPLE A,B,C,D,E,H,L,PC:16,DT:3,SR:3 ;
DCL GLOBAL TEMPORARY IR ;
DCL GLOBAL PERMANENT CA,C,Z,N ;
DCL FIELD MTYPE(6,7,-6)   , BITS34(3,4,-3)   , BITS45(4,5,-4) ,
            LOBYT(7,0)      , HIBYT(15,8,-8)   ;
DCL GLOBAL TEMPORARY PSTACK SP:16:(A,+) ;
DCL GLOBAL EXTERNAL OKAY:P ;
PROC:INF ;
  DCL GLOBAL USE  MEM,PC,IR ;
  DCL SPROC USE OKAY ;
# EXECUTE OKAY ;
 [* THIS EXTERNAL PROCEDURE TESTS FOR INTEUPTS ETC ., TO SEE
     IF ITS IS 'OKAY' TO FETCH THE NEXT INSTRUCTION *]
 IR=MEM[PC] ;
 INC PC ;
#
PROC:INSDCD ;
  DCL GLOBAL USE  IR,DT,SR ;
  DCL LOCAL TEMPORARY OPCODE:2 ;
# OPCODE=MTYPE(IR) ;
  DT=DEST(IR) ;
  SR=SOURCE(IR) ;
  SELECT(OPCODE,4) FROM ;
      (0,TYPE1) ;
      (1,INC) ;
      (2,INC) ;
      (3,INC) ;
  ENDSELECT ; #
SPROC:ADRFT(ADDR:16) ;
  DCL GLOBAL EXPECT MEM,PC ;
  DCL GLOBAL RETURN PC ;
  DCL LOCAL RETURN ADDR ;
  DCL LOCAL TEMPORARY LBYTE,HBYTE ;
# LBYTE=MEM[PC] ; INC PC ;
  HBYTE=MEM[PC] ; INC PC ;
  ADDR=HBYTE//LBYTE ; #
SPROC:STORE(RGSTR) ;
  DCL GLOBAL EXPECT H,L ;
  DCL GLOBAL RETURN MEM ;
  DCL LOCAL EXPECT RGSTR ;
  DCL LOCAL TEMPORAY TEMP:16 ;
# TEMP=H//L ; MEM[TEMP]=RGSTR #
PROC:TYPE1 ;
  DCL GLOBAL USE  IR,SR ;
# SELECT(SR,8) FROM ;
      (0,INC) ;
      (1,TYPE11) ;
      (2,TYPE12) ;
      (3,TYPE13) ;
```

```
          (4,TYPE14) ;
          (5,TYPE15) ;
          (6,TYPE16) ;
          (7,TYPE17) ;
      ENDSELECT ; #
    PROC:TYPE11 ;
      DCL GLOBAL USE  IR,B,C,D,E,H,L,SP,CA ;
      DCL LOCAL TEMPORARY OPCODE:2,ADDR:16 ;
      DCL SPROC USE ADDRET ;
    # OPCODE=BITSUB(IR) ;
      IFTRUE(IR.3) THEN ;
        BEGIN ;
          COND ;
            (OPCODE.EQ.0) ;
              BEGIN ;
                H//L=H//L+B//C  <CA> ; LEAVE INF ;
              END ;
            (OPCODE.EQ.1) ;
               BEGIN ;
                 H//L=H//L+D//E <CA> ; LEAVE INF ;
               END ;
              (OPCODE.EQ.2) ;
               BEGIN ;
                 H//L=H//L+H//L  <CA> ; LEAVE INF ;
               END ;
              (OPCODE.EQ.3) ;
               BEGIN ;
                 H//L=H//L+SP <CA> ; LEAVE INF ;
               END ;
              ENDCOND ;
        END ;
      ELSE ;
        BEGIN ;
          EXECUTE ADDRET(ADDR) ;
          COND ;
            (OPCODE.EQ.0) ; BEGIN ; B//C=ADDR ; LEAVE INF ; END ;
            (OPCODE.EQ.1) ; BEGIN ;D//E=ADDR ; LEAVE INF ; END ;
            (OPCODE.EQ.2) ; BEGIN ;H//L =ADDR ; LEAVE INF ;,END ;
            (OPCODE.EQ.3) ;BEGIN ;SP=ADDR ; LEAVE INF ; END ;
          ENDCOND ;
        END ; ENDIF ; #
    PROC:TYPE13 ;
      DCL GLOBAL USE  IR,B,C,D,E,H,L,SP ;
      DCL LOCAL TEMPORARY OPCODE:2 ;
    # OPCODE=BITSUB(IR) ;
      IFTRUE((IR,3)THEN ;
        BEGIN ;
          COND ;
            (OPCODE.EQ.0) ; BEGIN ; B//C=B//C-1 ; LEAVE INF ; END ;
            (OPCODE.EQ.1) ; BEGIN ; D//E=D//E-1 ; LEAVE INF ; END ;
            (OPCODE.EQ.2) ; BEGIN ; H//L=H//L-1 ; LEAVE INF ; END ;
            (OPCODE.EQ.3) ; BEGIN ; DEC SP      ; LEAVE INF ; END ;
         ENDCOND
        END ;
      ELSE ;
        BEGIN ;
          COND ;
```

```
                (OPCODE.EQ.0) ; BEGIN ; B//C=B//C+1 ; LEAVE INE ; END ;
                (OPCODE.EQ.1) ; BEGIN ; D//E=D//E+1 ; LEAVE INE ; END ;
                (OPCODE.EQ.2) ; BEGIN ; H//L=H//L+1 ; LEAVE INE ; END ;
                (OPCODE.EQ.3) ; BEGIN ; INC PC      ; LEAVE INE ; END ;
             ENDCOND ;
          END ; ENDIF ; #
PROC:TYPE14 ;
   DCL GLOBAL USE  B.C.D.E.H.L.A.
        MEM.DT.CA.Z.N ;
   DCL LOCAL TEMPORARY TEMP.ADDR.16 ;
# COND ;
      (DT.EQ.0) ; BEGIN ; INC B <CA.Z.N > ; LEAVE INE ; END ;
      (DT.EQ.1) ; BEGIN ; INC C <CA.Z.N > ; LEAVE INE ; END ;
      (DT.EQ.2) ; BEGIN ; INC D <CA.Z.N > ; LEAVE INE ; END ;
      (DT.EQ.3) ; BEGIN ; INC E <CA.Z.N > ; LEAVE INE ; END ;
      (DT.EQ.4) ; BEGIN ; INC H <CA.Z.N > ; LEAVE INE ; END ;
      (DT.EQ.5) ; BEGIN ; INC L <CA.Z.N > ; LEAVE INE ; END ;
      (DT.EQ.6) ; BEGIN ;
                       ADDR=H//L ; MEM[ADDR]=MEM[ADDR]+1 <CA.Z.N> ;
                       LEAVE INE ; END ;
      (DT.EQ.7) ; BEGIN ; INC A <CA.Z.N > ; LEAVE INE ; END ;
   ENDCOND ; #
PROC:TYPE15 ;
   DCL GLOBAL USE  B.C.D.E.H.L.A.
       MEM.DT.CA.Z.N ;
   DCL LOCAL TEMPORARY TEMP.ADDR.16 ;
# COND ;
      (DT.EQ.0) ; BEGIN ; DEC B <CA.Z.N > ; LEAVE INE ; END ;
      (DT.EQ.1) ; BEGIN ; DEC C <CA.Z.N > ; LEAVE INE ; END ;
      (DT.EQ.2) ; BEGIN ; DEC D <CA.Z.N > ; LEAVE INE ; END ;
      (DT.EQ.3) ; BEGIN ; DEC E <CA.Z.N > ; LEAVE INE ; END ;
      (DT.EQ.4) ; BEGIN ; DEC H <CA.Z.N > ; LEAVE INE ; END ;
      (DT.EQ.5) ; BEGIN ; DEC L <CA.Z.N > ; LEAVE INE ; END ;
      (DT.EQ.6) ; BEGIN ;
                       ADDR=H//L ; MEM[ADDR]=MEM[ADDR]-1 <CA.Z.N> ;
                       LEAVE INE ; END ;
      (DT.EQ.7) ; BEGIN ; DEC A <CA.Z.N > ; LEAVE INE ; END ;
   ENDCOND ; #
PROC:TYPE12 ;
   DCL GLOBAL USE  IR.MEM.B.C.D.E.H.L.A ;
   DCL LOCAL PERMANENT OC:2.ADDR:16 ;
   DCL SPROC USE ADDRET ;
# OC = BITS34(IR) ;
   IFTRUE((IR.5) THEN ;
      BEGIN ;
         EXECUTE ADDRET(ADDR) ;
            COND ;
(OC.EQ.0) ; BEGIN ; MEM[ADDR]=L ; INC ADDR ; H=MEM[ADDR] ;
                       LEAVE INE ; END ;
(OC.EQ.1) ; BEGIN ; L=MEM[ADDR] ; INC ADDR ; H=MEM[ADDR] ;
                       LEAVE INE ; END ;
(OC.EQ.2) ; BEGIN ; MEM[ADDR]=A ; LEAVE INE ; END ;
(OC.EQ.3) ; BEGIN ; A=MEM[ADDR] ; LEAVE INE ; END ;
            ENDCOND ;
      END ;
   ELSE ;
      BEGIN ;
```

```
          COND ;
(OC.EQ.0) ;BEGIN ;ADDR=B//C ;MEM[ADDR]=A ;LEAVE INF: END ;
(OC.EQ.1) ;BEGIN ;ADDR=B//C ;A=MEM[ADDR];LEAVE INF: END ;
(OC.EQ.1) ;BEGIN ;ADDR=D//E ;MEM[ADDR]=A ;LEAVE INF: END ;
(OC.EQ.3) ;BEGIN ;ADDR=D//E; A=MEM[ADDR] ;LEAVE INF: END ;
          ENDCOND ;
        END ; ENDIF ; #
PROC:16 ;
  DCL GLOBAL USE B,C,D,E,H,L,DT.MEM ;
  DCL LOCAL TEMPORAY OPCODE:3,DATA ;
  DCL SPROC USE STORE ;
# DATA=MEM[PC] ; INC PC ;
  COND ;
     (DT.EQ.0) ; BEGIN ; B=DATA ; LEAVE INF ; END ;
     (DT.EQ.1) ; BEGIN ; C=DATA ; LEAVE INF ; END ;
     (DT.EQ.2) ; BEGIN ; D=DATA ; LEAVE INF ; END ;
     (DT.EQ.3) ; BEGIN ; E=DATA ; LEAVE INF ; END ;
     (DT.EQ.4) ; BEGIN ; H=DATA ; LEAVE INF ; END ;
     (DT.EQ.5) ; BEGIN ; L=DATA ; LEAVE INF ; END ;
     (DT.EQ.6) ; BEGIN ; EXECUTE STORE(DATA) ; LEAVE INF ; END ;
     (DT.EQ.7) ; BEGIN ; A=DATA , LEAVE INF ; END ;
 ENDCOND ; #
PROC:TYPE17 ;
  DCL GLOBAL USE  A,CA,DT ;
  DCL LOCAL PERMANENT TEMP ;
# COND ;
     (DT.EQ.0) ; BEGIN:TEMP=A.7 ; A=A.SHTLT.1 ; CA=TEMP ;
                           A=TEMP.OR.A ; LEAVE INF ; END ;
     (DT.EQ.1) ; BEGIN:TEMP=A.0 ; A=A.SHTR.1 ; CA=TEMP ;
                           A=TEMP.OR.A ; LEAVE INF ; END ;
     (DT.EQ.2) ; BEGIN:TEMP=CA ; CA//A=CA//A.SHTL.1 ;
                           A=A.OR.TEMP ; LEAVE INF ; END ;
    .(DT.EQ.3) ; BEGIN:TEMP=A.0 ; CA//A=CA//A.SHTR.1 ;
                           CA-TEMP ; LEAVE INF ; END ;
    (DT.EQ.4) ; BEGIN:LEAVE INF :END ;
    (DT.EQ.5) ; BEGIN: A=.NOT.A ; LEAVE INF ; END ;
    (DT.EQ.6) ; BEGIN: CA=.NOT.CA ; LEAVE INF ; END ;
    (DT.EQ.7) ; BEGIN: SET CA ; END ;
 ENDCOND ; #
ENDEMULATOR ;
```

```
EMULATOR:XCMPTR ;
[*
   THIS IS AN EMULATOR FOR A HYPOTHETICAL 12 BIT , 8 REGISTER
COMPUTER.EACH INSTRUCTION (EXCEPT A FEW) HAS BASICALLY THREE
FIELDS:
          1- OPCODE FIELD: 4 BITS <BIT 11 TO BIT 8>
          2- FIRST SOURCE FIELD: 4 BITS <BIT 7 TO BIT 4>
          3- SECOND SOURCE / DESTINATION FIELD: 4 BITS
                                       <BIT 3 TO BIT 0>
   THE 4 BIT SOURCE/DESTINATION FIELD IS ACTUALLY MADE UP OF
TWO FIELDS. THE MOST SIGNIFICANT BIT OF THIS FIELD IS A
DIRECT/INDIRECT BIT , WHILE THE 3 LEAST SIGNIFICANT BITS
POINT TO ONE OF THE EIGHT MACHINE REGISTES. IF THE DIRECT/
INDIRECT BIT IS A '1' THEN THE OPERAND (SOURCE/DESTINATION)
IS IN MEMORY AND THE ADDRESS IS IN THE REGISTER . IF IT IS
A '0' THE OPERAND IS IN THE REGISTER.
   THE FOLLOWING INSTUCTIONS HAVE THE ABOVE MENTIONED FORMAT
          A- 'ADD'    ADD SOURCE1 TO SOURCE2 AND STORE RESULT IN SOURCE2
          B- 'SUB'    SUBTACT   .......
          C- 'ADDCR'  ADD WITH CARRY   ......
          D- 'SUBCR'  SUBTRACT WITH CARRY  ......
          E- 'AND'    AND                    ......
          F- 'OR'     OR                        .......
          G- 'NOT'    COMPLEMENT SOURCE1 AND PUT IT IN DESTINATION
          H- 'MOV'    MOVE SOURCE1 TO DESTINATION
   THREE INSTRUCTIONS HAVE THE FOLLOWING FORMAT :
          1- OPCODE <BIT 11 TO BIT 8>
          2- ADDRESS <BIT 7 TO BIT 0>
   THE EFFECTIVE ADDRESS IS CALCULATED BY ADING THE ADDRESS
FIELD TO THE PROGRAM COUNTER. TWO'S COMPLEMENT SIGN EXTENDED
ADDITION IS DONE. THE INSTRUCTIONS ARE :
          A- 'CALL'   GO TO THE SUBROUTINE AT THE EFFECTIVE ADDRESS.
                      THE PC IS PUSHED IN THE STACK.
          B- 'JMP'    JUMP TO THE ADDRESS.
          C- 'IO'     EXECUTE AN IO OPERATION AS SPECIFIED BY THE ADDRESS
                      FIELD. THIS FIELD HASNT BEEN DEFINED FOR THIS PROGRAM.
   THREE INSTRUCTIONS ARE TWO WORDS LONG AND THE FIELDS ARE:
     WORD 1: 1- OPCODE <BIT 11 TO BIT 8>
             2- 0 <BIT 7 TO BIT 4>
             3- DEST FIELD <BIT 3 TO BIT 0>
     WORD 2: 1- DATA / ADDRESS FIELD <BIT 11 TO BIT 0>
   THE THREE INSTRUCTIONS ARE:
          A- 'MOVEM'   MOVE THE CONTENTS OF THE MEMORY LOCATION
                       POINTED BY THE SECOND WORD INTO THE DESTINATION
                      CORRESPONDIG TO THE DEST FIELD
          B- 'MOVI'    MOVE THE SECOND WORD INTO THE DEST FIELD
          C- 'JOC'    JUMP ON CONDITION. THE REGISTERS POINTED BY THE
                      REGISTER FIELD ARE COMPARED AND THE TWO DIRECT/
                       INDIRECT FIELD SPECIFY ONE OF FOUR CONDITIONS. THE
                       CONDITIONS ARE 1) LESS THAN 2)GREATER THAN 3) EQUAL
                       TO AND 4) NOT EQUAL TO. THE JUMP ADDRESS IS THE
                       SECOND WORD .
   THE LAST TWO INSTRUCTIONS ARE
          A- 'RET'    WHICH DOES A RETURN FROM A SUBROUTINE BY POPING
                       THE STACK INTO THE PC.
          B- 'NOP'    NO OPERATION
```

```
    THE STACK IS RESIDENT IN THE MAIN MEMORY AND REGISTER 7
SERVES AS THE STACK POINTER *]
[*
    G L O B A L   D E C L A R T I O N S
*]
DCL  WORDSIZE 12 ;
DCL  ARITHMETIC TWO ;
DCL  MEMORY MEM:[4096] ;
DCL  GLOBAL PERMANENT R0,R1,R2,R3,R4,R5,R6,IR ;
DCL  GLOBAL TEMPORARY PC ;
DCL  GLOBAL PERMANENT FLAG C,C,7:7 ;
DCL  GLOBAL PERMANENT PSTACK,R7:(A,+) ;
DCL  EXTERNAL TOPROC:P ;
DCL  FIELD SRC1(7,4,-4),SRC2(3,0),OC(10,8,-8),
               ADDRES(7,0),BIT98(9,8,-8),SRC10(6,4,-4),
               SRC20(2,0),BITS7(7,7,-7),BITS3(3,3,-3),
               BITS02(2,0) ;
PROC:NEXT ;
[* THIS PROCEDURE FETCHES THE NEXT INSTRUCTTON AND
     DOES SOME PRIMARY DECODE *]
 DCL GLOBAL USE MEM,PC,IR ;
 DCL LOCAL TEMPORARY OPCODE,4 ;
# IR=MEM[PC] ; INC PC ;
  IFTRUE (IR,11) THEN ; BEGIN LEAVE TYPE1 ; END;
  ELSE ; BEGIN ; LEAVE TYPE2 ; END; ENDIF ; #
SPROC:FETCH(ADRFLD:4,VALUE) ;
 DCL GLOBAL USE MEM,R0,R1,R2,R3,R4,R5,R6,R7 ;
 DCL LOCAL EXPECT ADRFLD ;
 DCL RETURN VALUE ;
 DCL LOCAL PERMANENT BIT3:1,BIT02:2 ;
# BIT3=BITS3(ADRFLD) ;
  BIT02=BITS02(ADRFLD) ;
  COND ;
     (BIT02.EQ.0) ; BEGIN ; VALUE = R0 ; GO TO INDRCT ; END ;
     (BIT02.EQ.1) ; BEGIN ; VALUE = R1 ; GO TO INDRCT ; END ;
     (BIT02.EQ.2) ; BEGIN ; VALUE = R2 ; GO TO INDRCT ; END ;
     (BIT02.EQ.3) ; BEGIN ; VALUE = R3 ; GO TO INDRCT ; END ;
     (BIT02.EQ.4) ; BEGIN ; VALUE = R4 ; GO TO INDRCT ; END ;
     (BIT02.EQ.5) ; BEGIN ; VALUE = R5 ; GO TO INDRCT ; END ;
     (BIT02.EQ.6) ; BEGIN ; VALUE = R6 ; GO TO INDRCT ; END ;
     (BIT02.EQ.7) ; BEGIN ; VALUE = R7 ;      END ;
 ENDCOND ;
INDRCT : IFTRUE (BIT3,1) THEN ; BEGIN ;
           VALUE=MEM[VALUE] ;
            END ; ENDIF ; #
SPROC:STORE(ADRFLD:4,VALUE) ;
 DCL GLOBAL USE MEM,R0,R1,R2,R3,R4,R5,R6,R7 ;
 DCL LOCAL EXPECT ADRFLD ,VALUE ;
 DCL LOCAL PERMANENT BIT3:1,BIT02:2,TEMP,VALUE1 ;
 DCL SPROC USE FETCH ;
# BIT3=BITS3(ADRFLD) ;
  BIT02=BITS02(ADRFLD) ;
  IFTRUE(BIT3.EQ.0)THEN ;
   BEGIN ;
     COND ;
       (BIT02.EQ.0) ; BEGIN ; R0=VALUE ; LEAVE NEXT ; END ;
       (BIT02.EQ.1) ; BEGIN ; R1=VALUE ; LEAVE NEXT ; END ;
```

```
            (BITO2.EQ.2) : BEGIN ; R2=VALUE ; LEAVE NEXT ; END ;
            (BITO2.EQ.3) : BEGIN ; R3=VALUE ; LEAVE NEXT ; END ;
            (BITO2.EQ.4) : BEGIN ; R4=VALUE ; LEAVE NEXT ; END ;
            (BITO2.EQ.5) : BEGIN ; R5=VALUE ; LEAVE NEXT ; END ;
            (BITO2.EQ.6) : BEGIN ; R6=VALUE ; LEAVE NEXT ; END ;
            (BITO2.EQ.7) : BEGIN ; R7=VALUE ; LEAVE NEXT ; END ;
        ENDCOND ;
      END ;
     ELSE ;
      BEGIN ;
        EXECUTE FETCH(ADDFLD,VALUE1) ;
        MEM[VALUE1]=VALUE ; END ; ENDIF ; #
     PROC:TYPE1 ;
      DCL GLOBAL USE IR ;
      DCL LOCAL PERMANENT ADDFD1:4,ADRFD2:4,OPRND1,
            OPRND2,OP:2,OD:3 ;
      DCL SPROC USE FETCH,STORE ;
     # ADRFD1=SRC1(IR) ;
       ADRFD2=SRC2(IR) ;
       OD=OC(IR) ;
       EXECUTE FETCH(ADRFD1,OPRND1) ;
       IFTRUE(OD.EQ.7) THEN ; BEGIN ;
      [* NOT *]  OPRND1=.NOT.OPRND1 <C,Z> ; END ;
       ENDIF ;
       IFTRUE(OD.GT.5) THEN ; BEGIN ;
         [* MOV . NOT *]
        STORE(ADRFD2,OPRND1) ;
        END ;
       ELSE ;
        BEGIN ;
          [* ADD,SUB,ADDCR,SUBCR,AND,OR *]
          EXECUTE FETCH(ADRFD2,OPRND2) ;
          COND ;
             (OD.EQ.0) ;BEGIN ;
                   OPRND2=OPRND2+OPRND1 <C,Z> ; GO TO STRSLT ; END ;
             (OD.EQ.1) ; BEGIN ;
                   OPRND2=OPRND2+OPRND1 ;
                   OPRND2=OPRND2+C <C,Z> ; GO TO STRSLT ; END ;
             (OD.EQ.2) ; BEGIN ;
                   OPRND2=OPRND2-OPRND1 <C,Z> ; GO TO STRSLT ; END ;
             (OD.EQ.3) ; BEGIN ;
                   OPRND2=OPRND2-OPRND1 ;
                   OPRND2=OPRND2-C <C,Z> ; GO TO STRSLT ; END ;
             (OD.EQ.4) ; BEGIN ;
                   OPRND2=OPRND2.AND.OPRND1 <C,Z> ; GO TO STRSLT ; END ;
             (OD.EQ.5) ; BEGIN ;
                   OPRND2=OPRND2.OR.OPRND1 <C,Z> ; GO TO STRSLT ; END ;
         ENDCOND ;
       STRSLT : [* STORE RESULT *]
               EXECUTE STORE(ADRFD2,OPRND2)
      END ; ENDIF ; #
     PROC:TYPE2 ;
      DCL GLOBAL USER IR,PC,MEM,R7 ;
      DCL LOCAL PERMANENT OC:2,ADDR:8 ;
      DCL SPROC USE IOPROC ;
     # IFFALSE(IR,10) THEN ;
       BEGIN ;
```

```
          [* CALL. JMP,TO,NOP *]
          OC=BIT10C(IR) ; ADDR=ADDRES(IR) ;
          COND ;
          (OC.EQ.0) : BEGIN ;
               PUSH(PC) ; PC=PC+ADDR ; LEAVE NEXT ; END ;
          (OC.EQ.1) : BEGIN ;
             PC=PC+ADDR ; LEAVE NEXT : END ;
          (OC.EQ.2) : BEGIN ;
            EXECUTE TOPROC(ADDR) : LEAVE NEXT ; END ;
          (OC.EQ.3) ; BEGIN ; LEAVE NEXT ; END ;
          ENDCOND ;
         END ; ENDIF ; #
      PROC:TYPE2A ;
       DCL GLOBAL USE MEM,PC,IR ;
       DCL LOCAL PERMANENT BYTE2,ADREDI:4,ADRED2:4 OC:2,
            BIT3:1,BIT7:1,TEMP,TEMP1 ,ADRES ;
       DCL EPROC USE FETCH STORE ;
      # OC=BIT98(IR) ;
        IF(OC.EQ.0) THEN ; BEGIN ;
         [* RET *] PC=POP ; LEAVE NEXT ; END ;
        ELSE ;
         BEGIN ;
           [* MOVEM , MOVI , JOC *]
           BYTE2=MEM[PC] ; INC PC ;
           COND ;
             (OC.EQ.1) ; BEGIN ; TEMP=MEM[BYTE2] ;
                ADRED2=SRC2(IR) ; EXECUTE STORE(ADRED2,TEMP)
                LEAVE NEXT ; END ;
             (OC.EQ.2) ; BEGIN ; ADRED2=SRC2(IR) ;
                EXECUTE STORE(ADRED2,BYTE2) ; LEAVE NEXT ; END ;
             (OC.EQ.3) ; BEGIN ; ADRES=PC+BYTE2 ;
               ADRED1=SRC10(IR) ;
               ADRED2=SRC20(IR) ;
               EXECUTE FETCH(ADRED1,TEMP) ;
               EXECUTE FETCH(ADRED2,TEMP1) ;
               BIT7=BITS7(IR) ; BIT3=BITS3(IR) ;
               OC=BITS7//BITS3 ;
               COND ;
                (OC.EQ.0) : BEGIN ; IFTRUE(TEMP.GT.TEMP1) THEN ;
                                BEGIN ; PC=ADRES ; END ; ENDIF ;
                               LEAVE NEXT ; END ;
                (OC.EQ.1) : BEGIN ; IFTRUE(TEMP.LT.TEMP1) THEN ;
                                BEGIN ; PC=ADRES ; END ; ENDIF ;
                               LEAVE NEXT ; END ;
                (OC.EQ.2) : BEGIN ; IFTRUE(TEMP.EQ.TEMP1) THEN ;
                                BEGIN ; PC=ADRES ; END ; ENDIF ;
                               LEAVE NEXT ; END ;
                (OC.EQ. 3) : BEGIN ; IFFALSE(TEMP.EQ.TEMP1) THEN ;
                                BEGIN ; PC=ADRES ; END ; ENDIF ;
                               LEAVE NEXT ; END ;
               ENDCOND ;
           END ;
          ENDCOND ;
         END ;
        ENDIF ; #
      ENDEMULATOR ;
```

GLOSSARY

ARCHITECTURE: Those facilities of a computer that are visible to a programmer.

COMPILER: A program that converts a high level representation of a program into a low level representation.

CONTROL STORE: The memory of a m-computer which holds the micro-instructions.

DYNAMICALLY MICROPROGRAMMABLE COMPUTER: A microprogrammable machine with the capability of swapping microprograms in and out of a writable control store at a speed which matches the basic processor clock speed. This capability can be looked upon as the dynamic redesign of the architecture of the machine to meet the needs of the immediate job to be done.

EMULATOR: A collection of microprograms which when stored in control store, define a computer, i.e., its machine instruction set, is known as an emulator. The machine doing (supporting) the emulation is known as the HOST and the machine which is emulated is known as the TARGET or the VIRTUAL machine.

FIRMWARE: Firmware is described as microprograms which are resident in the control memory of a computer.

ITERATIVE CONTROL: Some machines have the capability of repeatedly doing an operation (corresponding to some microoperation) until some condition (loop count termination, flag generation etc.) makes them stop. This is a useful feature for multiple precision operations.

INTERPRETER: A program that performs the instructions of another program. It differs from a compiler in that it produces the results directly while a compiler produces a representation of a program that must be interpreted by a program or directly executed by a machine to produce results.

m-COMPUTER: A microprogrammed or a microprogrammable computer.

MACRO-INSTRUCTIONS: These are the instructions which reside in main memory i.e., the conventional machine instructions. The highest level of control over the CPU is exercised by these instructions. Thus they are one level above microinstructions just as nanoinstructions are one level below microinstructions.

MICRO ARCHITECTURE: Those facilities of a microprogrammable computer that are visible to a microprogrammer.

<u>MICRO CONTROL PROCESSOR</u>: A microprocessor with a control store as part of its control unit.

<u>MICROINSTRUCTION</u>: A word contained in the control store of a microprogrammed control unit. It consists of a number of fields; some of which are microoperations and some are literal data like a constant, an address, etc.

<u>MICROOPERATIONS</u>: The most primitive or elementary operations which a machine can execute. According to Wilke's original model, these are the signals which go over a single wire to a well defined destination.

<u>MICROPROCESSOR</u>: A large scale integrated (LSI) circuit processor on a single chip or a couple of chips. The term 'micro' in microprocessor refers to the physical size of the unit involved.

<u>MICROPROGRAMMABLE COMPUTER</u>: When the control store is made up of read-write memory and facilities are provided for changing the contents of the control store, the computer is said to be microprogrammable. The extent of these facilities on a computer (both hardware and software) determines <u>Microprogrammability</u> of the computer.

<u>MICROPROGRAMMED COMPUTER</u>: A computer is microprogrammed if the microinstructions which the computer executes are stored in a read only memory.

<u>MICROPROGRAMMING</u>: A technique for designing and implementing the control function of a computer, as a sequence of control signals to interpret fixed or dynamically changeable instruction set of the computer. In more general terms it is the activity of programming using microinstructions.

<u>NANO ARCHITECTURE</u>: A level below microarchitecture. The nanoprograms residing in <u>NANO STORE</u> define the microarchitecture of a computer. This means a sequence of <u>NANOINSTRUCTIONS</u> are executed to emulate a microinstruction. This is similar to the execution of a sequence of microinstructions to emulate a machine (macro) instruction.

<u>NANOPROGRAMMED COMPUTER</u>: A computer is nanoprogrammed if the nanoinstructions which the computer executes are stored in a read only memory.

<u>NANOPROGRAMMING</u>: The activity of programming using nanoinstructions.

<u>NANO STORE</u>: The memory where the nanoinstructions reside.

RESIDUAL CONTROL: In m-computers the microoperations are converted into control signals which directly and immediately control machine resources. This is known as IMMEDIATE CONTROL. An alternative is the residual control scheme where microoperations do not control resources directly, but rather use several SETUP REGISTERS to control hardware resources. The value of a setup register may indicate the microoperation a functional unit has to perform or the address of a register etc. Microinstructions are used to control the values in these setup registers.

SIMULATOR: An interpreter in which the interpreted instructions are machine language instructions for some machine (real or abstract).

UNIVERSAL HOST: A computer that can be microprogrammed to emulate any desired target machine.

WRITABLE CONTROL STORE: The control store of a microprogrammable computer.