A COMPARATIVE STUDY OF HIGH LEVEL MICROPROGRAMMING LANGUAGES

by

73
349 5839

Eugene Schreiner

B. S., Fort Hays Kansas State College, 1967

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1973

Approved by

Myron A. Calhoun
Major Professor

LD
266T
R4
1973
S36
C.2
Document

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I wish to acknowledge the assistance given me by Dr. Myron Calhoun, my major professor, Dr. Paul Fisher, and Dr. Virgil Wallentine in the preparation of this report. I especially want to thank Dr. Wallentine for suggesting the topic and the basic format and Dr. Calhoun for proof-reading the report.

Also, I wish to thank Sharon, my wife, for typing my report and many other student's papers so that I could attend graduate school.

1.0 INTRODUCTION

1.1 Basic Concepts of Microprogramming

The concept of microprogramming is normally attributed to Wilkes (25). He envisioned the control portion of a computer as a number of register transfers (micro-operations) which, when performed in the correct sequence, resulted in the execution of a machine instruction. A collection of these micro-operations which requires one basic time unit to execute is called a microinstruction. A sequence of microinstructions is called a microprogram. In recent years, microprograms, stored in high-speed, nondestructive read-only storage (ROS), have been used to control the sequences of the machine instructions of a number of digital computers.

The reader who may be unfamiliar with microprogramming may refer to Appendix A which contains the description of a simple hypothetical microprogrammed computer (microprocessor) or to one of the following references (9, 17, 20, 25).

To date, the main reason for the use of microprogram control has been in order to make a complete line of computers with different internal structures and performance ranges accept the same instruction set. An example of this is the IBM S/360 line of computers. Recently several

microprogrammable computers with writable control storage have been implemented--MPL-900 (formally IC-9000), Microdata 3200, Hewlett-Packard 2100S, and B1700. A microprogrammable computer differs from a microprogrammed computer by the support of a facility which allows the user to change the microprogram. These microprogrammable machines have the advantage of allowing the user to tailor the microprogram to match his specific application. The advantages and disadvantages have been discussed in a number of papers (9,pp.29-30; 17, pp. 117-125; 23, pp. 12-21).

As more microprogrammable computers become available, as the trend appears to be, it will become the user's responsibility to take full advantage of this new tool.

1.2 The Need for a High Level Microprogramming Language

Today if a microprogrammer wishes to alter the control storage of a microprogrammable computer, he has to write the new microprogram in an assembly-like language. A similar situation once prevailed in the programming of computers. The programmer had to write all programs in detailed assembly language. When higher level programming languages were developed, the programmer's job was simplified considerably. The many advantages of higher level programming languages should also be available to the microprogrammer. Therefore, the development of a high level microprogramming language seems to be an admirable goal.

A high level language would also reduce the burden

manufacturers have in producing the microprograms for their computers. Microprograms for manufacturer's computers are often more difficult to write than the ones a user would write for a microprogrammable computer. This increased difficulty arises from the use of long, complicated, special purpose microinstruction formats. At the present time, a flow-chart-like language is often used which is automatically assembled into microcode (24, p.236).

The argument against the development of a high level language for microprogramming usually stresses the point that microprogramming is not like programming, in that in microprogramming efficient microcode is more important than ease of programming. This is true; microcode that is used over and over cannot be inefficient code. But I feel the microprogrammer should have a high level language to use if he wishes. A situation where such a language would be useful to a microprogrammer is when he is writing micro-sequences that will not be used a high percent of the time, but will be swapped in and out of control storage as needed. The need for a standard notation to describe microprogramming would be satisfied by a high level microprogramming language.

The speed of computers has increased tremendously in the past few years and will probably continue to increase. It is costing more to write and debug programs than to execute them. Ease of programming has at times become a more important issue than the speed at which the program is

executed. This has caused a great increase in both special purpose and general purpose high level programming languages. This increased computer speed makes a high level micro-programming language attractive. If optimization techniques are used at compile time, the sometimes less efficient microcode produced for these routines may be outweighed by the savings in the time to write the microcode.

There have been a number of attempts to develop a high level microprogramming language. These have had varying degrees of success. This report will discuss various proposed languages and the attempts to implement them as microprogramming languages.

1.3 Purpose of this Report

The purposes of this report are as follows:

1. To indicate the characteristics that are required for a suitable high level microprogramming language.
2. To present a survey of the current literature in the area of defining and implementing such a microprogramming language.
3. To judge each of these languages on how well they satisfy these characteristics.
4. To determine if there is a suitable high level microprogramming language currently available.

2.0 REQUIREMENTS FOR A HIGH LEVEL MICROPROGRAMMING LANGUAGE

The following (2.1-2.8) is a list of the basic requirements for a high level language for microprogramming. The list is a composite of the characteristics and requirements given by a number of authors: Husson (17), Chu (3), Hill and Peterson (16), Eckhouse (10), Crockett (7), and Tsuchiya (23). The languages that are compared in this report were selected with these requirements in mind. Only languages which are designed to simulate microprogrammed computers and/or produce microcode are discussed. Therefore, this report does not include a number of currently implemented high level hardware design languages. Although these hardware design languages do satisfy many of the requirements listed below, they do not, in my opinion, appear to be adaptable to microprogramming.

The languages compared in this report are:

CDL--Computer Design Language,

CASD--Computer-Aided System Design,

MPL--Microprogramming Language,

SIMPL--A Single Identity Microprogramming Language,

APL--A Programming Language, and

AHPL--A Hardware Programming Language.

A general description and example programs written in each language is given in Appendix B. A chart comparing the features of these six languages is given in Appendix C.

2.1 Natural Programming Style

A microprogramming language will satisfy this requirement if it has syntax like a common high level programming language. In many cases the syntax can be simpler than the syntax of languages like PL/1, FORTRAN, and ALGOL, because it need only be designed for the special purpose of microprogramming. If the language is easily understood the microprogrammer will only need the source listing to communicate the details of his microprogram to others. The more natural the language, the easier users from various disciplines will master it.

In my opinion the PL/1 and ALGOL-like languages of CDL, CASD, and MPL satisfy this requirement, while APL and AHPL do not. A glance at the example programs in Appendix B should verify this opinion. Two reasons I feel APL doesn't satisfy this requirement are: (1) APL's vocabulary is quite large and descriptions are difficult to read initially; and (2) the notation contains Greek letters and other unfamiliar symbols which make the language difficult and cumbersome for the average programmer. AHPL solves these problems, to some extent, by eliminating many APL operators that are not pertinent to computer descriptions.

SIMPL's notation is not as easy to read as CDL, CASD, or MPL because unique variable names must be used to describe changes in the data in a hardware component. I would say the ease of comprehension of SIMPL's notation

lies somewhere between that of the PL/1 and ALGOL-like languages and APL.

2.2 Descriptive

The language must be descriptive so both structural properties and microprogram control of the systems can be described. The microprogrammer must be able to declare every computer component from the main memory to lights on the control panel. He must be able to describe the timing points which will effect the execution of the microprogram. He must have means to easily describe the action implied by every micro-operation. Both the declaration and execution statements must be free of all ambiguous notation. Unspecified notation may cause a compiler to take a default action which could lead to a catastrophe, such as generating a temporary register that does not exist in this computer's hardware. The language must have sufficient detail so gating of bit, word, and bit-array level operations can be specified. At the same time the description of irrelevant components and operations should not be required. Lastly, the language must have means available to allow the description of the microinstruction format and control fields.

In summary, the language must describe the computers internal structure, microinstructions, and micro-operations in an unambiguous and precise way.

Using CDL a microprogrammer can describe the internal

structure of a computer using register, subregister, cas-register, memory, clock, switch, light, and terminal declaration statements. The micro-operations are unambiguously described using labeled execution statements, where the label allows the programmer to explicitly indicate the clock phase and other conditions which must be present to evoke the execution of a micro-operation. The description of the microinstruction control fields is accomplished using decoder, terminal, and labeled execution statements.

CASD and MPL describe a computer using PL/1-like statements. Declaration statements are used to describe the internal structures of the computer. Micro-operations are described using PL/1-like execution statements. To my knowledge, CASD and MPL lack the facilities to describe the microinstruction control fields.

A microprogrammer using SIMPL would describe the internal structure of the computer and its microinstruction in a separate MODEL program. The MODEL program is written once and is then used as part of the input to the SIMPL compiler for each microprogram written for this computer. An ALGOL-like execution statement is used to indicate micro-operations. (See Appendix B for example programs)

Using APL a microprogrammer can describe micro-operations in a concise, precise, and unambiguous manner. APL does not have adequate means to declare internal computer structure and microinstruction control fields. APL has

been critized by Eckhouse (10, p. 8), Darringer (8, p. 9), and Gentry (13, p. 4) as lacking the facility to declare and thereby accurately describe the properties of computer components.

AHPL has register, bus logic, and bus load declaration statements. It does not have the means to declare other internal components. Micro-operations are described using a subset of APL notation. There are only four APL logic and arithmetic functions allowed in AHPL. These are "and," "or,""not," and "exclusive-or." Since AHPL was not designed for microprogramming, it lacks the means to describe micro-instruction formats.

2.3 Procedural

The microprogrammer must be able to specify the sequence in which a set of operations is to be performed. Unlike high level programming languages that may have complex control structures such as recursion, a microprogramming language requires only simple structures. Some method to write subroutines and perform single or multiple branches is all that is required.

Facilities to satisfy this requirement are provided by: procedure and computed go to statements in CASD, MPL, and SIMPL; special operator and labeled execution statement in CDL; combination logic subroutines and branch statements in AHPL. APL's function satisfies the requirement for sub-

routines and the programmer can write his own multiple branch statements, see (1, p. 107) or Example APL Program line 7 of FETCH in Appendix B.

2.4 Timing and Concurrent Operations

Some means to indicate timing signals must be available to allow the micro-operations to be executed at a particular time. The language will need some kind of timing signals so the programmer can have micro-operations performed in parallel or sequentially. One of the advantages of horizontal microinstruction formats is to allow parallel data transfer. The language must have a way of indicating these parallel data transfers if efficient microcode is to be generated by the compiler. Facilities to indicate synchronous and asynchronous micro-operation execution are also required.

The CDL's labeled execution statements can be used to indicate concurrent operations that are synchronous or asynchronously performed. CDL has a clock declaration statement that can be used to describe multiple phase clock cycles.

SIMPL has no explicit timing control statements. Implicit timing is accomplished through the use of the "single identity" type program. The compiler recognizes concurrent operations and parallel data paths. Asynchronous operations are programmed using status indicators that are checked by the "if STATUS then . . ." statement.

CASD and MPL also use implicit timing. The compiler recognizes mutually exclusive operations that may be executed concurrently. Each labeled statement begins a block of possible concurrent micro-operations. CASD has the additional features of "DO CONCURRENTLY" and "WAIT" statements that can be used to indicate asynchronous operations.

AHPL uses the letters "S" and "A" at the beginning of a statement to indicate synchronous and asynchronous operations, respectively. Also, in synchronous operations, if more than one statement is written on a line, the statements are executed concurrently. Through the use of English-word operators like "WAIT," "DELAY," "DIVERGE," and "CONVERGE," complicated parallel synchronous operations within asynchronous operations can be specified.

APL has no explicit means for a programmer to indicate timing. Bingham (2) said "(when they used APL to generate microcode) the machine had two overlapping phases so timing was not difficult." A programmer would have to write his own timing functions if he wished to use APL for microcode generation.

2.5 Hierarchical Descriptions

The microprogrammer must be able to describe the microsequence exactly or conceptually. Components defined in terms of primitive operations (OR, AND, NOT) should be named and referred to symbolically in higher levels of the hierarchy. For example this would mean that every time

addition is required the microprogrammer does not have to describe the add sequence in terms of AND-OR logic. What he does is write an addition routine and then uses it each time addition is performed.

CDL allows the programmer exact or conceptual microprogram descriptions through the use of operator and terminal statements. The example CDL program in Appendix B demonstrates the use of these statements to describe the AND-OR logic of the ADD function.

CASD, MPL, and SIMPL use PROCEDURE blocks, which may be contained in other PROCEDURE blocks, to build the complete program. The example programs in Appendix B use a PROCEDURE block to describe the addition of the contents of two registers.

APL uses FUNCTIONS to describe the microprogram at different hierarchical levels. The addition of two binary numbers is described by the functions ADDI and ADD1 in the example APL program in Appendix B.

AHPL uses the combined logic subroutine to achieve hierarchical descriptions of microprograms. An example combined logic subroutine that adds two numbers is given in the AHPL program in Appendix B.

2.6 Machine Independence

Machine independence actually means two things when applied to a high level microprogramming language. First, the language must be independent of the particular machines

on which it is implemented. Second, the language must be independent of the particular microprocessor it is used to microprogram.

2.6.1. To be independent of the machines on which it is implemented the language must produce identical "object" microcode for a program when the program is executed on the different machines. Assume translators for some high level microprogramming language were available for the IBM S/360 and Honeywell H4200 computers. A program run on each of these machines would produce identical outputs if the language was machine independent.

All the languages discussed in this paper are claimed to be machine independent. APL and CDL are since they have been implemented for a number of different computer systems. Although AHPL has only been implemented on the CDC 6400 there is no evidence that it is machine dependent. Since MPL, CASD, and SIMPL are proposed as high level languages, this implies they are machine independent.

2.6.2. To be machine independent in this second sense, the language must provide facilities to define a machine in which the microprograms are to be implemented. On a particular day a programmer may use the language to write a program for the Interdata 85 and the next day use the same language to write a program for the B1700.

Briefly this means the language must be descriptive. The language must allow the programmer to give a complete

description of any microprogram-controlled computer. This requirement was discussed in section 2.2. As mentioned in that section, APL is the only language which lacks means to completely describe a computer's internal structure.

In section 2.2 it was mentioned that the microinstruction format must be described. There are many different microinstruction formats, and there is no way for a translator to produce microcode for a particular one unless a complete description of it is given. This is discussed more in the following paragraphs.

There are two basic types of microinstruction formats. One, called the horizontal microinstruction format, consists of a number of subfields. A subfield may be from one to several bits long. Different combinations of bits in a subfield cause different control signals to be issued which in turn controls the flow of data.

The second type of microinstruction format is called a vertical microinstruction format. This format is similar to a machine instruction format. It usually has an op-code and some operands, which often means it is much shorter and less complicated than horizontal microinstructions. The vertical microinstruction is generally considered easier to program, but often programs written using it require more time to execute than programs written using the horizontal microinstruction (23, pp. 112-116).

The only languages that have a means to describe

these formats are CDL and SIMPL. In my opinion it would be impossible to have a machine independent microprogramming language that did not have a facility to allow descriptions of different microinstruction formats. Eckhouse (10) omitted a means to describe alternate microinstruction format in MPL. This is MPL's main deficiency. Eckhouse presents MPL programs that are supposed to produce microcode for the Interdata 3 and the IBM 2050 processor with little mention of how MPL adjusts for the completely different microinstruction formats. I assume he wrote a different translator for each microprocessor. This means MPL requires a different translator for each microprocessor it will be used to program.

Tsuchiya (23) admits that SIMPL is not completely independent of the computer for which the microcode is to be produced. The sequence of micro-operations that is specified by a particular SIMPL statement may be redefined as necessary. A microprogrammer may modify or redefine the semantics of microinstruction routines in different compilers. This flexibility may be a big advantage as more complicated and different microprogrammable computers become available.

In summary, no language discussed in this report can produce microcode for different microprocessors without some modification of its translator. Since SIMPL has a facility to describe the microinstruction less modification is required.

2.7 Simulator Available

The language should have a simulator available to provide the microprogrammer the facility to test and debug his microprogram before the source code is translated into microcode. A simulator will make it feasible for the microprogrammer to try alternate routines, microinstruction formats, and test microprogram efficiency. The simulator will increase the probability of the microprogram producing optimum microcode.

CDL has a number of simulators available. There are CDL simulators for IBM's 7094 and S/360; Univac 1108; and CDC 6600 (5, p. XIX).

Microprograms written in APL can be simulated on any of a number of APL time sharing systems. Raynaud, et al. (19) describes the simulation of microsequences for a computer system. The APL example program in Appendix B was simulated using APL/360.

A partial prototype simulator for MPL was developed as part of Eckhouse's research (10). The other high level microprogramming languages have not had a simulator implemented.

2.8 Translator Available

The language must have a translator that will produce the machine-dependent microcode, which can then be loaded into the control storage of the microprogrammed machine. Since efficiency is of prime importance in microprograms, the translation processor should include optimizing

techniques. A translator is, of course, the most important requirement of a high level microprogramming language. Without a translator the high level language will be no more than a microprogram notation that can be used for documentation or possibly as in the case of CDL and APL to simulate the microprogram.

Of the six languages discussed in this paper only MPL and APL have been used to produce microcode. MPL was used to produce a microprogram for the INTERDATA 3 as a test case (10). The final output of the translator was a notation very similar to the INTERDATA 3 micro-assembly language that is used for microprogramming. This MPL translator used optimizing techniques. Eckhouse states that a complete compiler (translator for MPL) has not been implemented.

APL has been used to prepare, translate, and check-out microcode (1). Bingham (2) said their APL programs produced 200 lines of microcode while rejecting 800 lines. The APL programs for this application required three man-weeks to prepare. This translator was for a particular machine and could not be used for general microprogramming; in addition Bingham's APL translator did not optimize the microcode.

3.0 COMPARISON OF THE HIGH LEVEL MICROPROGRAMMING LANGUAGES

The six languages surveyed in this report satisfy the requirements for a suitable high level microprogramming language in varying degrees. We can eliminate some of the languages as possible high level microprogramming languages.

CASD is only a proposed language and there are no plans to implement it. AHPL was not designed as a microprogramming language and lacks means to describe the microinstruction. CDL is an adequate language for simulating microprograms but requires the programmer to explicitly indicate the control bits. He actually has to write the microcode before he can simulate it. For this reason it does not appear feasible to take a language such as CDL and build a translator for it.

MPL satisfies most of the requirements for a high level microprogramming language. It does not have a completely implemented simulator or translator. A simulator is not as critical a requirement as a translator. If and when a complete translator for MPL is implemented the language will still lack the requirement of being machine independent, meaning a different translator will have to be written for each microprocessor.

SIMPL appears to be the most realistic and usable high level microprogramming language. The proposed SIMPL and MODEL languages possess all the facilities needed to

write microprograms for both horizontal and vertical microprogrammed computers. The property of allowing the microprogrammer to modify the semantics of a microsequence of the translator appear to make the SIMPL language the best high level microprogramming language. Unfortunately, as mentioned before, SIMPL is only a proposed language and no attempt has been made to develop a translator for it.

APL is the only language that has been used to produce usable microcode. The microprogrammer wishing to use APL must write his own APL routines to translate the APL description into microcode. Computer scientists have critized APL as being hard to read and understand by the novice programmer. This is no real problem because a microprogrammer is not a novice programmer. "(He) must be thoroughly familiar with the design philosophy, the architectural characteristics, and the technological capability that govern the design of (a microprogrammable computer)" (17, p. 16). I feel a specially designed microprogramming language with notations similar to AHPL would be a much better microprogramming language than any of the currently implemented versions of APL.

If I had to pick a currently available language for writing microprograms, it would have to be APL even though it lacks the aforementioned facilities.

What is needed is the implementation of a language like MPL which could be used to microprogram a few computers. If this language proved acceptable, then the implementation of a much more versatile language like SIMPL could be undertaken.

If SIMPL produces efficient microcode as its author claims it can, the use of high level microprogramming languages would become as popular with microprogrammers as high level programming languages are with programmers.

4.0 CONCLUSION

This report has covered the basic characteristics a suitable high level microprogramming language must possess. The language should be natural, descriptive, procedural, and machine independent. It should also have hierarchical structures, timing and concurrent operations, a simulator for the microsequences, and a translator.

Of the six languages compared APL is the only implemented language that has been used to produce microcode (1). The implementation of a translator for MPL would provide the microprogrammer with a language, but he would be limited to microprogramming for a particular machine. The implementation of a language like SIMPL would be a much more difficult task. If it can be shown that a language similar to SIMPL can produce microcode as efficient as is produced by conventional microprogramming methods, there would be more support and interest in implementing such a language.

In my opinion the need for a high level microprogramming language has not been great enough for manufacturers to support the development of such a language. As more microprogrammable computers become available the need may increase to a point where a manufacturer will develop a high level microprogramming language to support their computer. At the present the development of such a language appears to be a good research topic for students in computer science, but not a money-making proposition for manufacturers.

APPENDICES

APPENDIX A

A Hypothetical Microprogrammed Computer

This oversimplified microprogrammed computer will be used as the example computer for sequences written in the different high level microprogramming languages. The following description of the computer will be useful to the reader who is not familiar with microprogramming.

Figure I shows the configuration and data flow of the simple microprogrammed computer. The abbreviations used in Figure 1 have the following meanings:

1. M--main memory; 8192 16-bit words
2. MAR--memory address register; 13 bits wide
3. MDR--memory data register; 16 bits wide
   IOC--instruction op-code; 3 bits
   IA--instruction address; 13 bits
4. IC--instruction counter; 13 bits wide
5. AC--accumulator; 16 bits wide
6. ALU--arithmetic logic unit; adds two 16 bit words
   note: ALU can do a number of arithmetic and logic operations, but for our example assume it can only add
7. CM--control memory; 8 16-bit words
   note: this is extremely small, but large enough to demonstrate the functions of a control memory
8. CMAR--control memory address register; 3 bits wide
9. MIR--microinstruction register; 16 bits wide
   CA--next microinstruction address; 3 bits
   CB--control bits; 13 bits

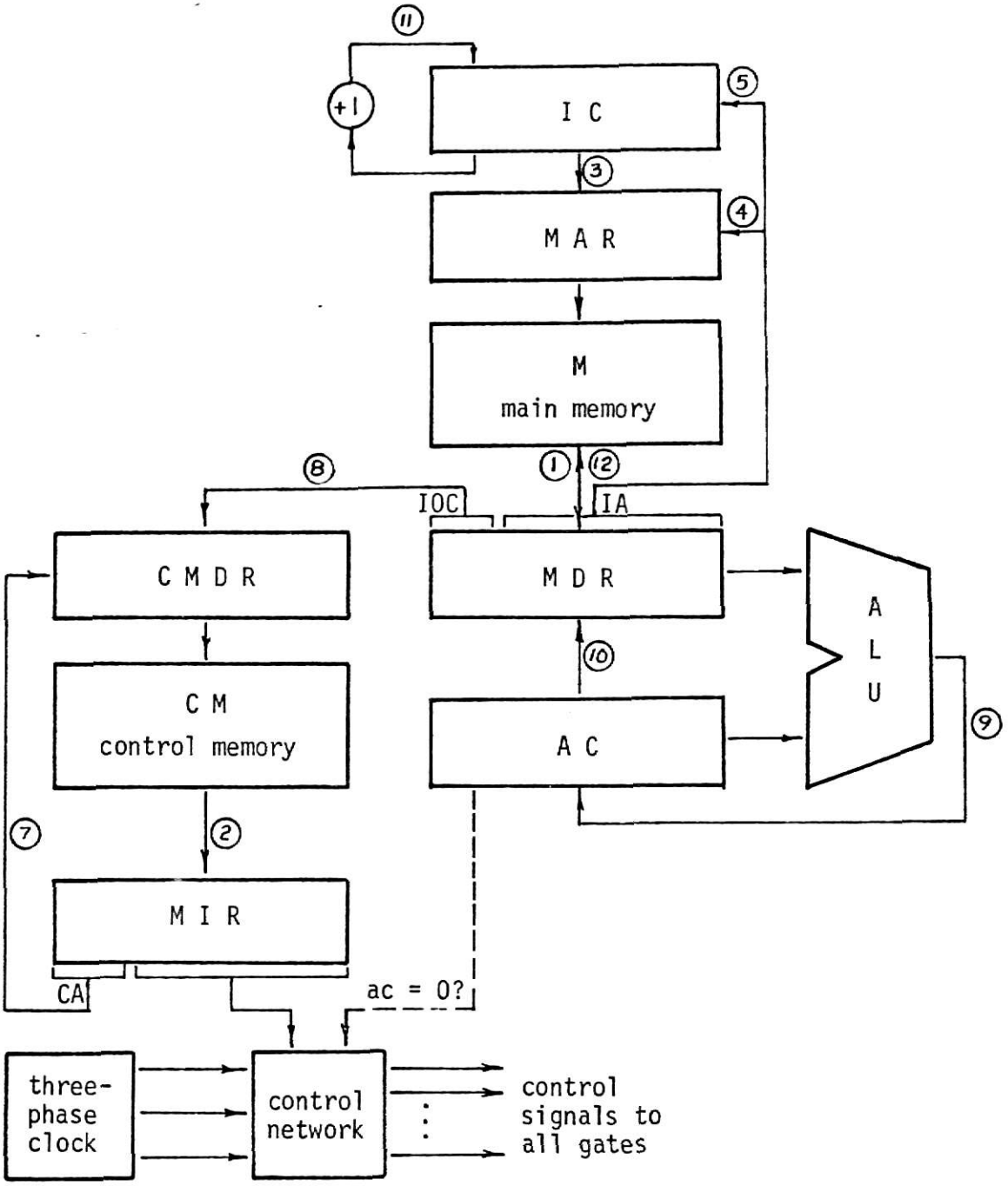

Figure 1

A Simple Microprogrammed Computer

The following ideas on the reading and writing of memory are based on the example given in Chu's book (5, pp. 92-93)

All timing of this computer is synchronous. The main memory cycle is equal to one clock cycle. All micro-operations can be completed in one of the three phases of the clock cycle except for the reading and writing of main memory and the reading of control memory.

To read main memory, the memory address must be transferred to the memory data register at clock phase three, P(3), of the preceeding clock cycle. This initiates the memory read operation. The word becomes available in the MDR by the end of clock phase one, P(1), of the current clock cycle.

To read from main memory the following clock phase and micro-operations are required:

Clock phase	Micro-operation
P(3)	IC → MAR or IA → MAR
P(1)	M → MDR
P(2)	⋮

To write a word into main memory, during P(3) the address is transferred to MAR. This initiates the write into memory operation. By the end of P(2) of the following clock cycle the word has been written MDR into main memory.

The sequence of micro-operations for writing into main memory is:

Clock phase	Micro-operations
P(3)	IC → MAR or IA → MAR
P(1)	any micro-operations(s)
P(2)	MDR → M
P(3)	⋮

For this simple computer, control memory is a read-only memory. To read a control word from control memory, the control word address is transferred to CMAR at P(2), and the read control memory operation is initialized. At the end of P(3) the word is available in MIR.

The sequence for the read from control memory is:

Clock phase	Micro-operations
P(2)	CA → CMAR or IOC → CMAR
P(3)	CM → MIR
P(1)	⋮

The following is a list of the possible micro-operations (register-to-register transfers) that can be performed by this simple microprogrammed computer.

1. M → MDR
2. CM → MIR
3. IC → MAR
4. IA → MAR
5. IA → IC
6. IF AC = 0 DO 5
7. CA → CMAR
8. IOC → CMAR
9. MDR + AC → AC
10. AC → MDR
11. IC + 1 → IC
12. MDR → M
13. 0 → AC

The microinstruction format for the computer is shown in Figure 2. The 16-bits of the microinstruction are divided into two fields: bits 0 through 2 are the next microinstruction address field, bits 3 through 15 are the control bits, of which bits 12-15 are not used in the example that follows. The numbers in the control bits field indicate the particular micro-operations or operations that will be executed when this control bit is "1" and the corresponding clock phase pulse is present. For example the 10 in bit position 4 indicates that if this bit is a "1" the micro-operations AC → MDR will be executed at clock phase P(1).

Clock phase	0 1 2	3	4	5	6	7	8	9	10	11	12	13	14	15
P(1)	next micro-instruction address	1	10	6					13					
P(2)					11 8	12	7							
P(3)					4		3	9		2				

Figure 2

Microinstruction Format

The type of microinstruction format shown in Figure 2 is an example of a format used for horizontal microprogramming. A horizontal microinstruction consists of a number of fields that control hardware gates.

Using this simple microprogrammed computer we can now show the micro-operations that are performed to fetch an instruction from main memory.

		Clock phases
FETCH	M → MDR	P(1)
	IOC → CMAR, IC + 1 → IC	P(2)
	CM → MIR, IA → MAR	P(3)

During clock phase P(1) a word (machine instruction) is read into MDR from main memory. The machine instruction op-code (IOC) is transferred to the CMAR and the instruction counter is incremented during the second clock phase, P(2). The microinstruction is read from control memory (CM) into MIR, and the machine instruction address (IA) is transferred to MAR in the third clock phase, P(3). The computer is now ready to execute the machine instruction under the control of the control bits that are in the microinstruction register (MIR).

A fetch sequence is executed for every machine instruction. After the fetch is completed the actual machine instruction is executed using the bits from the microinstruction, currently in MIR, to control its execution. Then the fetch

sequence is performed and then another instruction executed. This alternation of fetch-execute continues until the program reaches a halt instruction or some error condition stops the machine.

The micro-operations for the machine instructions for storing the accumulator, loading the accumulator from memory, branching if accumulator is zero, and adding a number in memory to the accumulator are shown below.

		Clock phases
STORE	AC → MDR	P(1)
	MDR → M, CA → CMAR	P(2)
	IC → MAR, CM → MIR	P(3)
LOAD	M → MDR, 0 → AC	P(1)
	CA → CMAR	P(2)
	MDR + AC → AC, IC → MAR, CM → MIR	P(3)
BRZ	IF AC = 0 IA → IC	P(1)
	CA → CMAR	P(2)
	IC → MAR, CM → MIR	P(3)
ADD	M → MDR	P(1)
	CA → CMAR	P(2)
	MDR + AC → AC, IC → MAR, CM → MIR	P(3)

The microprogram for these machine instructions is given in Figure 3. The microprogram has been loaded into

control memory. The fetch microinstruction is at location zero, store at location one, etc. The microprogrammer who must design and debug microprograms that may be hundreds of times more complex than the one shown in Figure 3 has a difficult task. For this reason a high level microprogramming language appears to be a necessary aid if the user is to be able to write meaningful microprograms.

Instruction	Location	Next Address bits 0-2	Control Bits bits 3-15														
FETCH	0	0	0	0	1	0	0	1	0	0	0	0	1	0	0	0	0
STORE	1	0	0	0	0	1	0	0	1	1	0	0	1	0	0	0	0
LOAD	2	0	0	0	1	0	0	0	0	1	1	1	1	0	0	0	0
BRZ	3	0	0	0	0	0	1	0	0	1	0	0	1	0	0	0	0
ADD	4	0	0	0	1	0	0	0	0	1	1	0	1	0	0	0	0

Figure 3

The Microprogram

APPENDIX B

General Description of the High Level Microprogramming Languages

CDL (Computer Design Language)

Chu (3) wrote that the logic designer of the digital computer has a difficult job trying to communicate design details by logic diagrams or Boolean equations. He proposed and described an ALGOL-like Computer Design Language (CDL) that would solve this problem. The purposes of the language were: (1) to define a standard language for describing the logic design of a digital system; (2) to simulate the logic and thereby reduce the design and debugging time while giving the designer a chance to experiment with different designs; (3) to evaluate the performance and cost of a system by simulation; and (4) to make it possible to design more complex machines.

CDL is composed of the following features:

1. Declaration Statements are used to describe each computer element, sub-element and combined-elements. Declaration statements can be used to describe register, memory, switch, light, decoder, terminal, or clock elements. A terminal statement is used to describe the details of a logic network such as an adder.

2. Micro-Statements describe the functional operations physically built in a computer. These consist of

(a) basic operators (see Table I), (b) micro-operations which are register transfers, (c) conditional statements, and (d) special operators which define operations not included in the basic operator group.

TABLE I (5, p. 7)

Basic Operations in Computer Design Language

Operators	Symbols	Explanations
NOT	'	
OR	+	
AND	*	
EXOR	⊕	logical exclusive-or
Shift left	Shl	shift one or more bits inserting zeros at end where bits have been shifted away
Shift right	Shr	
Circulate left	Cil	shift one or more bits circular
Circulate right	Cir	
Count up	Countup	increment count by one
Count down	Countdn	decrement count by one
Addition	Add	
Substraction	Sub	

3. Sequencing is controlled by execution statements. The execution statement has a label and an operation or group of operations that are executed when the value of the label is true. The label is a Boolean function which usually

AND's a number of control registers and clock signals together. The sequencing can be synchronous or asynchronous with control coming from a clock, a control register, or a combination of the two.

Since Chu first proposed CDL in 1965 there have been CDL simulators implemented for the IBM 7094, IBM S/360, Univac 1108, and the CDC 6600 (5).

Chu uses CDL as the design language in two books (4, 5). The language has had some revision, but has always included the capability of describing microprogrammed computers. The language is easy to read and understand. It allows three forms of comment statements which aid in documentation. For examples of these comment statements see the CDL sample program.

As you can see from the example program that follows, the CDL programmer has to write the object microcode of the microprogram before he can write the CDL program. This is because the programmer must keep track of each timing pulse and control bit.

We must remember that it was not Chu's purpose to use CDL to produce microcode. It appears to me that CDL cannot readily be modified so it can be used to translate CDL source code into microcode. Even with this restriction CDL can benefit the microprogrammer. It provides very good documentation of the system designed and it provides a

simulator for the microcode after it has been generated by some means other than CDL.

In the sample CDL program that follows notice how completely the simple microprogrammed computer (described in Appendix A) is described. Not only does CDL show what is done but how it is done. The order of execution of CDL statements is not sequencially from one statement to the next, but a statement is executed when its label is true, otherwise nothing happens.

EXAMPLE CDL PROGRAM

```
COMMENT,CONFIGURATION OF THE SIMPLE MICROPROGRAMMED COMPUTER.
REGISTER,MDR(0-15),      $MEMORY DATA REGISTER
         AC(0-15),       $ACCUMULATOR
         MAR(0-12),      $MEMORY ADDRESS REGISTER
         IC(0-12),       $INSTRUCTION COUNTER
         MIR(0-15),      $MICROINSTRUCTION REGISTER
         CMAR(0-2),      $CONTROL MEMORY ADDRESS REGISTER
SUBREGISTERS,IOC=MDR(0-2),    $INSTRUCTION OP-CODE
             IA=MDR(3-15),    $INSTRUCTION ADDRESS
             CA=MIR(0-2),     $ADDRESS OF NEXT MICROINSTRUCTION
MEMORY,M(MAR)=M(0-8191,0-15),$MAIN MEMORY
       CM(CMAR)=CM(0-7,0-15),$CONTROL MEMORY
CLOCK,P(1-3),                $THREE-PHASE CLOCK

COMMENT,SEQUENCES OF THE SIMPLE MICROPROGRAMMED COMPUTER.
COMMENT,FETCH SEQUENCE WHEN CMAR=0.
/MIR(3)*P(1)/            MDR<-M(MAR),
/MIR(6)*P(2)/            CMAR<-IOC,IC<-COUNTUP IC,
/MIR(6)*MIR(11)*P(3)/    MIR<-CM(CMAR),MAR<-IA,

COMMENT,STORE SEQUENCE WHEN INSTRUCTION OP-CODE=1.
/MIR(4)*P(1)/            MDR<-AC,
/MIR(8)*MIR(7)*P(2)/     M(MAR)<-MDR,CMAR<-CA,
/MIR(8)*MIR(11)*P(3)/    MAR<-IC,MIR<-CM(CMAR),

COMMENT,LOAD SEQUENCE WHEN INSTRUCTION OP-CODE=2.
/MIR(3)*MIR(10)*P(1)/    MDR<-M(MAR),AC<-0,
/MIR(8)*P(2)/            CMAR<-CA,
/MIR(8)*MIR(9)*MIR(11)*P(3)/ AC<-MDR ADD AC,MAR<-IC,
                         MIR<-CM(CMAR),

COMMENT,BRZ SEQUENCE WHEN INSTRUCTION OP-CODE=3.
/MIR(5)*P(1)/            IF AC=0 THEN IC<-IA,
/MIR(8)*P(2)/            CMAR<-CA,
C/MIR(8)*MIR(11)*P(3)/   MAR<-IC,MIR<-CM(CMAR),$THIS IS A COMMENT

COMMENT,ADD SEQUENCE USING THE 'BUILT-IN' FUNCTION ADD.
C/MIR(3)*P(1)/           MDR<-M(MAR),
C/MIR(8)*P(2)/           CMAR<-CA,
C/MIR(8)*MIR(9)*MIR(11)*P(3)/ AC<-MDR ADD AC,MAR<-IC
                         MIR<-CM(CMAR),

COMMENT, THE ADD SEQUENCE CAN BE DESCRIBED AT SEVERAL LEVELS.
COMMENT, USER DEFINED ADDITION USING CDL'S 'OPERATOR' AND
         'TERMINAL' FUNCTIONS.
OPERATOR,R<-A ADD B,
TERMINAL,C(15)<-0,
         C(0-14)<-A(1-15)*B(1-15)+A(1-15)*C(1-15)
         +B(1-15)*C(1-15),
         R(0-15)<-A(0-15)⊕B(0-15)⊕C(0-15),

                         END
```

CASD (Computer-Aided System Design)

Computer-Aided System Design was described in a paper by Crocket, et al. (7). Its goal was to unify the design of digital computers. This proposed system has a high level input language which is used to describe the proposed digital computer and its control section. The system can then simulate the operations of a proposed computer. Several translators, including one that produces microcode, are part of the proposed system.

The CASD system consists of the following facilities:

1. high level source language,
2. high level simulator,
3. translator,
4. on-line conversational mode, and
5. file system.

The high level source code is input to an encoder which generates an intermediate code which can be used as input to the simulator or the translator. The on-line conversational mode allows the designer complete control over his design. He can enter, debug, modify, and test the design. The file system is used for automatic documentation and all record keeping on the designs in the system.

The high level language of the CASD system is an enriched subset of PL/1. The basic building blocks of the language are the PL/1 PROCEDURE's. Each procedure represents

a logical module of the design (shifter, adder, etc.). Combinations of these procedures represent the complete computer design. Each component is defined by the PL/1 DECLARE statement.

Two of the main differences between the CASD language and PL/1 are:

1. Concatenated items may appear on either side of an assignment statement. This makes it possible to combine two components such as registers. For example, two 16-bit registers Q and M could be assigned a 32-bit constant CONT using the statement, Q||M:=CONT;

2. No defaults are allowed and everything must be declared. This is to assure the CASD translator will not generate a temporary register, which may not be in the proposed design. For the same reason subscripted subscripts and subscripted parameters passed in subroutines are not allowed.

Unlike CDL the CASD language does not require explicit timing signals to be supplied by the programmer. Operations are written as sequential statements. The CASD system determines whether or not the execution of the operations will be parallel or sequential by applying a set of rules (7). The programmer has a number of ways to override these rules, with the most common being the use of a label statement to begin a new group of parallel-executed synchronous micro-operations.

The proposed simulator is interpretive, to allow the designer to experiment with alternate designs and "watch" how they perform.

The proposed translator, really a number of translators,

will automatically translate the proposed design into microcode or other useful logic specifications.

CASD would satisfy the basic requirements for a high level microprogramming language if it were implemented as it has been proposed, but CASD has not been implemented and no plans are presented for such an undertaking in the paper by Crockett (7).

The sample CASD program presented below is very similar to PL/1, but statements with the same label are executed in parallel if there is no logic conflict. A very good example indicating many of the capabilities of CASD is given on page 295 of Crockett's paper (7).

EXAMPLE CASD PROGRAM

```
SMC: PROCEDURE OPTIONS(MAIN);

DECLARE
   1 MDR,                          /* MEMORY DATA REGISTER */
     2 IOC BIT(3),                 /* INSTRUCTION OP-CODE */
     2 IA BIT(13),                 /* INSTRUCTION ADDRESS */
   AC BIT(16),                     /* ACCUMULATOR */
   (MAR,IC) BIT(13),   /* MEMORY ADDRESS REG. & INST. COUNTER */
   CA BIT(3),            /* ADDRESS OF NEXT MICROINSTRUCTION */
   CON_BITS BIT(13),               /* CONTROL BITS */
   MIR DEFINED CA||CON_BITS  /* MICROINSTRUCTION REGISTER */
   CMAR BIT(3),           /* CONTROL MEMORY ADDRESS REGISTER */
   M(0:8191) BIT(16),              /* MAIN MEMORY */
   CM(0:7) BIT(16);                /* CONTROL MEMORY */

   /*  SEQUENCES OF THE SIMPLE MICROPROGRAMMED COMPUTER */
   /* HERE THE ROUTINES TO START, STOP, AND RUN THE COMPUTER */
START: CALL RUN;
RUN: PROCEDURE; /* RUN FETCHES THE INSTRUCTION AND EXECUTES IT */
DECLARE OPLABEL(0:7) LABEL
                INITIAL(FETCH,STORE,LOAD,BRZ,ADD,(3)ILLEGAL);

              /* FETCH SEQUENCE WHEN CMAR=0 */
FETCH:  MDR:=M(MAR);
FETCH2: CMAR:=IOC; IC:=IC+1;
FETCH3: MIR:=CM(CMAR); MAR:=IA; GO TO OPLABEL(CMAR);

              /* EXECUTION ROUTINES */
              /* STORE INSTRUCTION */
STORE:  MDR:=AC;
STORE2: M(MAR):=MDR; CMAR:=CA;
STORE3: MAR:=IC; MIR:=CM(CMAR); GOTO OPLABEL(CMAR);

              /* LOAD INSTRUCTION USING THE CASD '+' FUNCTION */
LOAD:   MDR:=M(MAR); AC:=(16)'0';
LOAD2:  CMAR:=CA;
LOAD3:  AC:=AC+MDR; MAR:=IC; MIR:=CM(CMAR); GOTO FETCH;

              /* BRZ INSTRUCTION */
BRZ:    IF AC=(16)'0' THEN IC:=IA;
BRZ2:   CMAR:=CA;
BRZ3:   MAR:=IC; MIR:=CM(CMAR); GO TO OPLABEL(CMAR);

              /* ADD INSTRUCTION */
ADD:    MDR:=M(MAR);
ADD2:   CMAR:=CA;
ADD3:   AC:=ALU(AC,MDR); MAR:=IC; MIR:=CM(CMAR); GOTO FETCH;

   /* ADDITION SUBROUTINE USING USER DEFINED ADDITION */
ALU: PROCEDURE(A,B);
DECLARE(A,B,D,SUM) BIT(16); D:=B;
L:      SUM:=A#D; D:=A&D; A:=SUM; IF ¬D THEN RETURN;
        D:=SUBSTR(D,2,15)||'0'; GO TO L; END ALU;

        RETURN; END RUN;

        END SMC;
```

MPL (Microprogramming Language)

In his Ph. D. dissertation, Eckhouse (10, 11) informally defines a PL/1-like microprogramming language called MPL. He sets the background for defining MPL by surveying what has been done in proposing and implementing languages for system programming and hardware design. He also conducted a survey of a ". . .representative sample of microprogrammable machines" (10, p. 21) to determine similarities and differences which must be considered when specifying a high level microprogramming language.

MPL is similar to CASD except for the following:

1. The IF statement can be used to test for an EVENT previously declared. For example, OVFL may be declared as an overflow toggle and tested by an "IF OVFL THEN. . ." statement.

2. Expressions in MPL do not allow operations such as multiplication, division, and exponentiation, but it has right and left shifts and exclusive-or.

3. MPL allows the use of four memory reference built-in functions to access main, control, local, or auxiliary memory.

The compiler for MPL has three phases (see Figure 4). The first phase translates the MPL source code into an intermediate language and builds a dictionary that is used to store the machine-dependent aspects from the data declarations. The intermediate language is like Conway's (6) simple machine language (SML) which has only two types of instruction--register loads and stores, and execute operations. The

second phase produces virtual code, and the third generates the machine dependent microcode using the dictionary produced in phase one. The last phase utilizes optimization techniques to produce efficient microcode. This last phase has to be reprogrammed for each microprocessor. The first two phases of the translator are machine independent while the third is machine dependent.

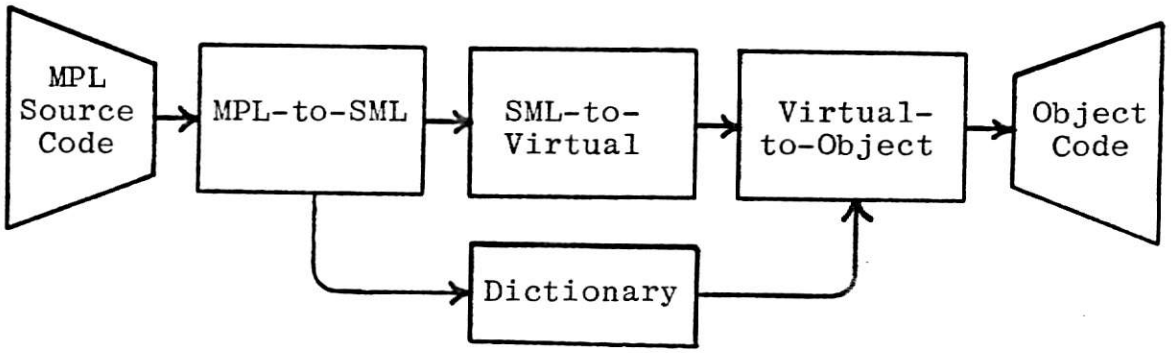

Figure 4

Organization of the Translator (10, p. 36)

To test his compiler, Eckhouse used the design of the INTERDATA 3 machine to emulate a subset of IBM S/360 instructions. The final output was INTERDATA code in micro-assembly language. The INTERDATA uses a vertical micro-instruction format which is programmed using an assembly-like language. Eckhouse also wrote in MPL the microsequence for the instruction fetch for the IBM 2050 processor. Parallel processing is taken care of by the same methods used in CASD. MPL is independent of the machine its compiler

runs on, but is not independent of the microprocessor it is producing code for.

In my opinion, Eckhouse demonstrated the feasibility of a microprogramming language for a microprocessor, but fell short in describing a language that could be used for all microprocessors. He did not completely implement his proposed compiler. Eckhouse states". . .research in this area calls for a full translator for some meaningful machine with a read/write control store. As yet, no such machine is available to the author." (10, p. 56) An August 1972 article by Rosin, Frieder, and Eckhouse (21) indicated that they plan to use MPL as the basis for preparing a compiler for generating microcode. I have no information on whether or not MPL is currently implemented.

The sample MPL program that follows demonstrates the many similarities between CASD and MPL input source code.

EXAMPLE MPL PROGRAM

```
SMC: PROCEDURE OPTIONS(MAIN);

   DECLARE MDR BIT(16),       /* MEMORY DATA REGISTER */
           IOC BIT(3) DEFINED MDR POSITION(1), /* OP-CODE */
           IA BIT(13) DEFINED MDR POSITION(4), /* ADDRESS */
        AC BIT(16),           /* ACCUMULATOR */
        MAR BIT(13),          /* MEMORY ADDRESS REGISTER */
        IC BIT(13),           /* INSTRUCTION COUNTER */
        MIR BIT(16),          /* MICROINSTRUCTION REGISTER */
        CA BIT(3) DEFINED MIR POSITION(1), /* ADDR. NEXT M.I. */
        CMAR BIT(3),  /* CONTROL MEMORY ADDRESS REGISTER */
        MS(0:8191) BIT(16),/* MAIN MEMORY-MUST USE MS ABBR. */
        CS(0:7), BIT(16),/* CONTROL MEMORY-MUST USE CS ABBR. */
        L(0:4) LABEL;

        /* SEQUENCES OF THE SIMPLE MICROPROGRAMMED COMPUTER */
             /* FETCH SEQUENCE WHEN CMAR=0 */
L(0):   MDR=MS(MAR);
FETCH2: CMAR=IOC; IC=IC+1;
FETCH3: MIR=CS(CMAR); MAR=IA; GO TO L(CMAR);

             /* STORE SEQUENCE WHEN INSTRUCTION OP-CODE=1 */
L(1):   MDR=AC;
STORE2: MS(MAR)=MDR; CMAR=CA;
STORE3: MAR=IC; MIR=CS(CMAR); GOTO L(CMAR);

             /* LOAD SEQUENCE WHEN INSTRUCTION OP-CODE=2 */
             /* HERE WE USE MPL'S '+' FUNCTION TO ADD */
L(2):   MDR=MS(MAR); AC=0B;
LOAD2:  CMAR=CA;
LOAD3:  AC=AC+MDR; MAR=IC; MIR=CS(CMAR); GOTO L(CMAR);

             /* BRZ SEQUENCES WHEN INSTRUCTION OP-CODE=3 */
L(3):   IF ¬AC THEN  IC=IA;
BRZ2:   CMAR=CA;
BRZ3:   MAR=IC; MIR=CS(CMAR); GOTO L(CMAR);

             /* ADD SEQUENCE WHEN INSTRUCTION OP-CODE=4 */
L(4):   MDR=MS(MAR);
ADD2:   CMAR=CA;
ADD3:   AC=ADD(AC,MDR); MAR=IC; MIR=CS(CMAR); GO TO L(CMAR);

             /* USER DEFINED ADDITION ROUTINE */
ADD: PROC(A,B);
  DECLARE(A,B,C,SUM) BIT(16);
        C=B;
L:      SUM=A.EXOR.C; C=A&C; A=SUM; IF ¬C THEN RETURN;
        C=C.LSH.1; GO TO L;
        END ADD;

        END SMC;
```

SIMPL (A Single Identity Microprogramming Language)

The most recent and ambitious attempt to define a suitable high level microprogramming language is given by Tsuchiya (23) in chapter 4 of his Ph. D. dissertation. Although Tsuchiya only proposed the language called SIMPL, his ideas on the features of a high level microprogramming language appear to be more realistic than the other languages reported in this paper.

As the name implies, A Single Identity Microprogramming Language has an input source language that requires a unique identifier for each data path and contents of the data path. The single identity language is discussed by Tesler and Enea (22). For example, if we wish to add the memory data register, MDR, to the accumulator, AC, we could write:

MDR.1 + AC.1 → AC.2;

where the digital part of the identifier must be unique in each SIMPL statement. Now if we want to add the new contents of AC to MDR again, we could use the statement:

MDR.1 + AC.2 → AC.5.

By making each data path and contents unique the compiler automatically recognizes parallel operations. Statements are executed when all the variables to the left of the assignment symbol "→" are defined, and not necessarily

in lexicographical order.

Tsuchiya's language has the potential of being used to write programs for multilevel microprogrammable machines which use either vertical or horizontal microinstruction formats. SIMPL has been defined to handle the most complex microinstruction formats, such as those that are used by IBM's S/360 and Honeywell's H4200 computers.

Some of the characteristics of SIMPL are: (1) high level ALGOL-like syntax, (2) descriptive and unambiguous, (3) permits implicit parallel micro-operations, (4) optimizes the microcode, (5) allows the microprogrammer to change the meaning of the semantics routines when microprogramming for different computers.

Because the proposed SIMPL compiler is very complex, it will require a substantial amount of computer time to compile the source SIMPL code and produce the microcode. This is somewhat offset by the fact that the microcode produced may be used many times over and the initial cost of computer time to produce the microcode may be less than the cost to produce microcode by conventional methods. Tsuchiya didn't propose a simulator for SIMPL, so a microprogrammer will not have the means to test and debug his microprogram. Of course, the biggest inadequacy is that Tsuchiya does not indicate any plans to develop a compiler for SIMPL.

SIMPL has no means for the microprogrammer to describe the internal hardware structure of the microprogrammed

computer for which he wishes to write a microprogram. This job is done by another proposed language called Machine Organization Definition Language (MODEL). MODEL is used to give the complete description of the microprocessor's organization. This description is then compiled and used as input to the SIMPL compiler. Once a machine's organization is described using MODEL it need not be redescirbed for each microprogram written for this particular machine.

The complete machine organization can be described in MODEL by defining the following facilities of the machine:

1. names and sizes of data paths,
2. status indicator names,
3. data buses,
4. microinstruction control fields, and
5. any peculiar element for a particular machine.

See the MODEL description of the simple microprogrammed computer for an example.

The input source language SIMPL is much like ALGOL. It contains the following reserved words: begin, end, procedure, equivalence, entry, exit, comment, do, for, step, until, while, if, then, else, goto, true, false.

The most noticable differences between ALGOL and SIMPL are given below:

1. SIMPL's statements are not executed in the order they appear, but are executed when all the variables (data paths) on the left of the assignment symbol "→" are defined. This

condition is overridden if there is a conflict in the use of data paths. AC.1 + MDR.2 → AC.2 and AC.1 - MDR.2 → AC.3 could not be executed concurrently if addition and subtraction are performed by the same hardware

2. All variable names in SIMPL must correspond to the names defined in the MODEL program.

3. All variables must be uniquely defined each time they are used (single identity idea).

4. Arithmetic and logical expressions must not contain more than two operands and one operator. The statement AC.1 + MDR.2 - X.3 → AC.2 is illegal.

5. All variables are global and no new variables may be introduced in SIMPL that were not previously defined in the MODEL program.

6. To read or write memory the expressions read (A, R) and write (A, R) are used. A is the memory address register and R is the register the word is read into or written from.

7. The semantics microinstruction routines may be redefined or modified in different SIMPL compilers as the need might be.

The production of microcode using MODEL/SIMPL is quite involved, but microprogramming is quite involved, especially when a large horizontal microinstruction format is used.

The basic steps required to generate a microprogram using MODEL/SIMPL are:

1. The microprogrammer describes the computer's organization in MODEL. He defines the data paths, buses, status indicators, and micro-instruction formats.

2. The MODEL program is compiled and the generated code is used as input to the SIMPL compiler.

3. The microprogrammer writes the microprogram in SIMPL source code using the variable names defined in MODEL.

4. The SIMPL program is compiled in four phases; (a) the syntax is checked and a variable table is build; (b) semantics is identified and appropriate micrprogrammed semantic routines are generated (at this point the microcode can be produced if the microprogram is written for a machine that uses vertical microinstruction); (c) analysis of the statements are made to determine concurrent micro-operations; and (d) micro-operations are optimized and microcode is produced for machines that use horizontal microinstructions.

The following are the MODEL and SIMPL programs for the simple microprogrammed computer.

EXAMPLE MODEL PROGRAM

```
COMMENT DESCRIPTION OF THE SIMPLE MICROPROGRAMMED COMPUTER;

DEFINE
   COMMENT NAMES AND SIZE OF ALL STORAGE ELEMENTS;
   M(16)↑13;  MAR(13);  MDR(16);  AC(16);  IC(13);
   CM(16)↑3;  CMAR(3);  MIR(16);

   COMMENT ALL DATA PATHS;
   M<->MDR;  CM->MIR;  MDR[3,15]->MAR;  IC->MAR;  MDR[3,15]->IC;
   MIR[0,2]->CMAR;  MDR[0,2]->CMAR;  IC<->IC;  0->AC;
   MDR+AC->AC;

   COMMENT A STATUS INDICATOR NAMED 'ZERO'-- TO TEST IF AC=0?;
   ZERO;

   COMENT DESCRIPTION OF THE MICROINSTRUCTION CONTROL FIELDS;
   MIR[3]= M->MDR;                  MIR[4]= AC->MDR;
   MIR[5]= IF ZERO THEN MDR[3,15]->IC;
   MIR[6]= MDR[0,2]->CMAR;          MIR[6]= MDR[3,15]->MAR;
   MIR[6]= IC+1->IC;                MIR[7]= MDR->M;
   MIR[8]= IC->MAR;                 MIR[8]= MIR[0,2]->CMAR;
   MIR[9]= MDR+AC->AC;              MIR[10]= 0->AC;
   MIR[11]= CM->MIR;

END;
```

EXAMPLE SIMPL PROGRAM

```
BEGIN
   COMMENT ALL ELEMENTS OF THE SIMPLE MICROPROGRAMMED COMPUTER
           HAVE BEEN DEFINED IN THE MODEL PROGRAM,
           DEFINE THE FOLLOWING EQUIVALENCES;
     EQUIVALENCE
        IOC=MDR[0,2],   IA=MDR[3,15],   CA=MIR[0,2];

PROCEDURE SMC;

   COMMENT SEQUENCES OF THE SIMPLE MICROPROGRAMMED COMPUTER,
           FETCH SEQUENCE WHEN CMAR=0;
FETCH:  READ(MAR.0,MDR.0);
        IOC.0->CMAR.0;  IC.0+1->IC.1;
        READ(CMAR.0,MIR.0);    IA.0->MAR.1;
        GOTO CMAR.0(FETCH,STORE,LOAD,BRZ,ADD,(3)ILLEGAL);

   COMMENT STORE SEQUENCE WHEN INSTRUCTION OP-CODE=0;
STORE:  WRITE(MAR.1,AC.0);
        FETCHMI;  GOTO FETCH;

   COMMENT LOAD SEQUENCE WHEN INSTRUCTION OP-CODE=2;
LOAD:   0->AC.0; GOTO ADD;

   COMMENT BRZ SEQUENCE WHEN INSTRUCTION OP-CODE=3;
BRZ:    IF ZERO THEN IA.0->IC.2;
        FETCHMI;  GOTO FETCH;

   COMMENT THE ADD SEQUENCE WHEN INSTRUCTION OP-CODE=4;
ADD:    READ(MAR.1,MDR.1);  MDR.1+AC.0->AC.1;
        FETCHMI;  GOTO FETCH;


   COMMENT PROCEDURE TO FETCH MICROINSTRUCTION;
PROCEDURE FETCHMI;
        CA.0->CMAR.1;  IC.1->MAR.2;
        READ(CMAR.1,MIR.1);
END FETCHMI;

END SMC;
```

APL (A Programming Language)

APL notation was introduced in the book titled A Programming Language by K. E. Iverson (18) in 1962. In this book Iverson describes the APL notation and demonstrates its power and versatility by a large number of examples including "microprogramming" sequences.

APL is now primarily used for interactive programming, and a modified version, called APL/360, is implemented for IBM computers. APL has been applied to a large class of algorithms. It has been used to give a detailed description of the IBM S/360 (12), to formulate computer problems, to develop interactive programmed instruction in mathematics, and to design computer systems (19).

One of the disadvantages of APL is the large number of operators that constitute the core of APL notation. Many of the symbols for these operators are unique to APL, making APL notation quite hard to read for the beginning programmer. However, working with APL/360 for a few hours makes the notation become somewhat easier to read and understand.

Since much literature on APL is available and the language has many operators (the current version of APL/360 has about 70 different operators), only a few properties of APL are described here. The properties discussed will explain the meaning of the operators used in the example APL program

which follows. The reader who wishes to become more familiar with APL may look at Iverson's book (18) or chapter 2 of Hellerman's book (15) on computer systems. A good introduction to APL/360 is given in APL/360 An Interactive Approach (14).

APL has two types of functions: (1) primitive functions or operators and, (2) defined functions. The user can use these functions to describe the microprogram with much detail or at a very abstract level. Normally the more compact the function the less documentational value it possesses. In the example program ADD1, eight statements are used to describe the addition of two binary fixed integers. ADDI adds the two binary fixed integers by first converting them to decimal numbers, adding, and then converting them back to binary integers all in one statement.

The APL operators used in the example program are given in Table II.

TABLE II

Some Basic APL Operators

Operator	Name	Example Expression*	Result
⍴	shape	⍴A	4
↑	take	3↑A	0 0 1
↓	drop	3↓A	1
⊤	encode	(2 2 2)⊤5	1 0 1
⊥	decode	2⊥1 0 1	5
⍳	index generator	⍳3	1 2 3
⌽	rotation	1⌽A	0 1 1 0
≠	logical exclusive-or	A≠B	0 1 1 0
∧	logical and	A∧B	0 0 0 1
∧/	compress & and	∧/A	0

*A = 0 0 1 1, B = 0 1 0 1

The operators ↑, ↓, and ⍳ are used to extract certain fields from the machine words. ⊤ and ⊥ are used to change the binary numbers into decimal numbers and vice versa. This is required for addition and indexing. ⍴ is used to find the length of a word. ⌽, ≠, and ∧ are used for "shifting," "exclusive-oring," and "anding." ∧/ "and" the bits of a word together.

The main advantage APL has over the other languages discussed in this report is that it has been implemented

on a number of computer systems. APL has been used for microcode preparation, translation, and checkout (1).

In the modeling of a machine described by Bingham (2), an assembly-like language was used as input to the APL programs that were basically decoders and executers. The microinstruction for the machine had 20 concurrent parallel fields for execution. Bingham states: ". . .APL is an adequate host for creating higher level languages for microprogram development." (2) Even though the microcode produced was not optimum, it was only used as a model and therefore the inefficiences were tolerable. Using APL it took six man-weeks to develop 200 lines of microcode. This time included one man-week to prepare an assembler and two man-weeks to model the machine. The model was run on a S/360-50, a S/370-145, and a B6700. Once the actual hardware for the model was produced, the microcode that had been produced was used as the basis for bootstrapping onto the actual machine.

The example APL program which follows was used to simulate the simple microprogrammed computer that was described in Appendix A. Data was stored in M (Main Memory) and, when executed, produced the correct results. This APL program only simulated the computer and no attempt was made to have it generate microcode.

EXAMPLE APL PROGRAM

```
      ∇SMC[□]∇
    ∇ SMC
[1]   ⍝ EXAMPLE APL PROGRAM
[2]   ⍝ COMPUTER CONFIGURATION
[3]   MDR←16⍴0
[4]   MAR←13⍴0
[5]   AC←16⍴0
[6]   MIR←16⍴0
[7]   CMAR←3⍴0
[8]   IC←13⍴0
[9]   M← 8192 16 ⍴0
[10]  CM← 8 16 ⍴0
[11]  FETCH
    ∇
```

```
      ∇FETCH[□]∇
    ∇ FETCH
[1]   ⍝ SEQUENCES OF THE SIMPLE MICROPROGRAMMED COMPUTER
[2]  START:MDR←M[(13⍴2)⊥MAR;]
[3]   CMAR←3↑MDR
[4]   IC←INC IC
[5]   MIR←CM[2 2 2 ⊥CMAR;]
[6]   MAR←3↓MDR
[7]   →5+3×(2 2 2 ⊥CMAR)
[8]  STORE:MDR←AC
[9]   M[(13⍴2)⊥MAR;]←MDR
[10]  →READMI
[11] LOAD:AC←(⍴AC)⍴0
[12]  NOP←0
[13]  →ADD
[14] BRZ:→(AC≠0)/READMI
[15]  IC←MDR[3+⍳13]
[16]  →READMI
[17] ADD:MDR←M[(13⍴2)⊥MAR;]
[18]  AC←AC ADDI MDR
[19] READMI:FETCHMI
[20]  →START
    ∇
```

```
      ∇ADDI[□]∇
    ∇ R←A ADDI B
[1]   ⍝ ADDITION ROUTINE MAY BE SPECIFIED
[2]   ⍝ AT DIFFERENT LEVELS OF DETAIL
[3]   ⍝ HERE WE SHOW NO DETAIL
[4]   R←(((⍴A)⍴2)⊤(((⍴A)⍴2)⊥A)+(((⍴B)⍴2)⊥B))
    ∇
```

```
      ∇ADD1[□]∇
    ∇ R←A ADD1 B
[1]   ⍝ HERE WE SHOW
[2]   ⍝ 'EXOR-AND' LOGIC
[3]   C←A
[4]   D←B
[5]  L:R←C≠D
[6]   D←C∧D
[7]   C←R
[8]   D[0]←0
[9]   D←1⌽D
[10]  →(D≠0)/L
    ∇
```

```
      ∇FETCHMI[□]∇
    ∇ FETCHMI
[1]   CMAR←MIR[⍳3]
[2]   MAR←IC
[3]   MIR←CM[(2 2 2)⊥CMAR;]
    ∇
```

```
      ∇INC[□]∇
    ∇ R←INC DU
[1]   R←13⍴0
[2]   I←12
[3]  L:J←¯12+I
[4]   R[I]←DU[I]≠(∧/J↑DU)
[5]   I←I-1
[6]   →(I≥0)/L
    ∇
```

AHPL (A Hardware Programming Language)

A Hardware Programming Language was introduced by Hill and Peterson in their book Digital Systems: Hardware Organization and Design (16). The AHPL notation was developed in order to describe precisely the sequences of operations and the flow of data from point to point in a digital computer. AHPL is a subset of APL, and is directly translatable into hardware operations. The authors had to add a few English word operators which are used to describe timing sequences.

AHPL was not developed to be a high level microprogramming language that would produce microcode. It was developed to describe the sequencing of operations and data flow in a computer, and have this description automatically translated into logic block diagrams. A compiler that translates AHPL descriptions into Boolean equations of an equivalent hardware machine has been implemented on the CDC 6400 by Gentry (13).

If AHPL is not a high level microprogramming language, why has it been included in this report? The answer is that the implementation of an AHPL translator indicates to me the feasibility of using a subset of APL as the basis for a high level microprogramming language. By decreasing the number of APL primitive operators, as was done in AHPL, a microcode translator might be written. It would be much more difficult to produce a translator that handled a large number of APL's primitive operators. AHPL has been used to

describe microprogramming sequences and microprogrammed machines (16, ch. 8).

A general description of AHPL follows. A more complete, but informal description of AHPL is given in chapter 5 of Hill and Peterson (16). An outline of AHPL is given in Figure 5.

I. Control sequence routine
   - A. Branch statements
   - B. Transfer statements
     1. Left side: fixed or variable destination
     2. Right side: as a function of register, flip-flop, and buses
   - C. Timing
     1. Synchronous
     2. Asynchronous
     3. Parallel synchronous and/or asynchronous
   - D. Declarations
     1. Register
     2. Bus logic
     3. Bus load

II. Combinational logic subroutines
   - A. Subroutines
   - B. Functions

Figure 5

Outline of AHPL

The *control sequence* routine is the main AHPL program. The branch statement is in the form:

→(EXP)

where EXP is a statement number or logical expression that

when computed gives a statement number.

Transfer statements are register-to-register transfers, where the right side may be a constant or a logical function operating on data. The only operators allowed on the right side of a transfer statement are: or, ∨; and, ∧; not, ~; and sometimes exclusive-or, ⊕. Transfer statements written on the same line and separated by a semicolon are executed simultaneously.

Timing is indicated explicitly by the programmer. Parallel synchronous and asynchronous operations are allowed. AHPL uses "Sn" to indicate synchronous transfers, where n equals the number of basic time units for the transfer. "A" is used for asynchronous transfers. "T" indicates a terminal statement. The symbol "B" indicates a bookkeeping statement, which requires no time to execute and the variables used in the statement have no hardware counterpart (see the example AHPL program). The English word timing operators are shown in Figure 6.

Register, bus logic, and bus load declarations are written with a "D" preceeding each declaration. AHPL has no means of declaring or describing an array storage like main memory. This feature should not be difficult to add.

A combinational logic subroutine consists of AHPL program steps which define a logic network. Combinational logic subroutines are written much like control sequence routines.

Operator	Meaning
NO DELAY	Transfer without delaying the rest of the control sequence
n DELAY	Transfer after n units of time
WAIT	Wait until asynchronous transfer is complete
DIVERGE (X, Y)	Statements X and Y begin parallel asynchronous operations
CONVERGE (R, Q)	Statements R and Q converge to end parallel asynchonous operations

Figure 6

AHPL Timing Operators

Some non-logic operators are allowed on both sides of transfer and branching statements. These operators are used to index and identify certain fields or subfields in a data string. For example if A = 1 0 1 1 1, then $\alpha[2]/A$ = 1 0 and $\omega[4]/A$ = 0 1 1 1.

The example AHPL program that follows is a combination of the notation used by Hill and Peterson (16) in their book and the implemented language Gentry (12) describes in his Ph. D. dissertation.

EXAMPLE AHPL PROGRAM

```
* CONFIGURATION OF THE SIMPLE MICROPROGRAMMED COMPUTER.
D           MDR=16
D           AC=16
D           MAR=13
D           IC=13
D           MIR=16
D           CMAR=3
* REGISTERS ARE DECLARED, MAIN MEMORY-M AND
* CONTROL MEMORY-CM ARE NOT DECLARED IN AHPL

* SEQUENCES OF THE SIMPLE MICROPROGRAMMED COMPUTER.

* FETCH SEQUENCE WHEN CMAR=0.
[1]  S1 MDR←M[⊥MAR]
[2]  S1 CMAR←α[2]/MDR;IC←INC(IC)
[3]  S1 MIR←CM[⊥CMAR];MAR←IA
[4]  B  →(OPCODE(CMAR))

* STORE SEQUENCE WHEN INSTRUCTION OP-CODE=1.
[5]  S1 MDR←AC
[6]  S1 M[⊥MAR]←MDR;CMAR←α[2]/MIR
[7]  S1 MAR←IC;MIR←CM[⊥CMAR]
[8]  B  →(1)

* LOAD SEQUENCE WHEN INSTRUCTION OP-CODE=2.
[9]  S1 MDR←M[⊥MAR];AC←0
[10] S1 CMAR←α[2]/MIR
[11] S1 AC←ADD(AC,MDR);MAR←IC;MIR←CM[⊥CMAR]
[12] B  →(1)

* BRZ SEQUENCE WHEN INSTRUCTION OP-CODE=3
[13] B  ⊥AC:0 (=,≠)→(14,15)
[14] S1 IC←ω[3]/MDR
[15] S1 CMAR←α[2]/MIR
[16] S1 MAR←IC;MIR←CM[⊥CMAR]
[17] B  →(1)

* ADD SEQUENCE WHEN INSTRUCTION OP-CODE=4.
[18] S1 MDR←M[⊥MAR]
[19] B  →(10)
[20] T  END
```

```
* OPCODE SUBROUTINE--CALCULATE THE LOCATION
* TO BRANCH TO DEPENDING ON OP-CODE
SUBR          OPCODE(CMAR)
[1]   B   B←6ρ0;A←CMAR
[2]   S   B←((~A[0]∧~A[1]∧A[2])×5)+
+           ((~A[0]∧A[1]∧~A[2])×9)+((~A[0]∧A[1]∧A[2])×13)+
+           ((A[0]∧~A[1]∧~A[2])×18)
[3]   T   OPCODE(CMAR)←B
[4]   T   RETURN


* ADD SUBROUTINE--ADDS TWO 16 BIT NUMBERS.
SUBR          ADD(AC,MDR)
[1]   B   C,S←17ρ0,16ρ0
[2]   B   I←16
[3]   S   C[I],S[I]←FULLADDF(AC[I],MDR[I],C[I+1])
[4]   B   I←I-1
[5]   B   I:1 (<,>)→(6,3)
[6]   T   ADD(AC,MDR)←S
[7]   T   RETURN


* FULLADDER FUNCTION--ADDS THREE BITS TOGETHER.
FUNC          FULLADDF(A,B,C)
[1]   S   SO←(~A∧~B∧C)
[2]   S   CO←(A∧B)∨(A∧C)∨(B∧C)
[3]   T   FULLADDF(A,B,C)←CO,SO
[4]   T   RETURN


* INC SUBROUTINE--INCREMENTS IC BY 1
SUBR          INC(IC)
[1]   B   ICNEW←13ρ0
[2]   B   I←12
[3]   S   ICNEW[I]←IC[I]⊕(∧/ω[12-I](13)/IC))
[4]   B   I←I-1
[5]   B   I:0 (≥,<)→(3,6)
[6]   T   INC(IC)←ICNEW
[7]   T   RETURN
```

APPENDIX C

Language Features

This two-page chart indicates the major desirable features of a high level microprogramming language. For each language, the method used to support or implement the feature is given. A blank indicates that a particular feature is not supported in any way by the language.

Features \ Language	CDL	CASD
Ease of comprehension	ALGOL-like	PL/1-like
Defines internal computer structures	declares all components	declares most components
Describes microinstruction control fields	labeled micro statements	
Indicates timing	labeled micro statements	labels and implicit timing
Indicates concurrent operations	labeled micro statements	key words or implicit concurrent operations
Indicates what happens	micro statements	execution statements
Indicates how it happens	special operators; terminal statements	procedures
Simulates microprogram	Simulator available for S/360, IBM 7094, Univac 1108, CDC 6600	
Produces "object" microcode		
Microcode optimization		

MPL	SIMPL	APL	AHPL
PL/1-like	ALGOL-like	APL notation	APL-like
declares most components	declares all components in MODEL program		declares some components
	described in Model program		
labels and implicit timing	implicit timing		labels and key words
implicit concurrent operations	implicit concurrent operations		key words
execution statements	execution statements	assignment statements	transfer statements
procedures	procedures	functions	combined logic subroutines
		most APL versions	
one partially built "compiler"		one special application run on S/360, S/370, B6700	
third phase of proposed "compiler"	in fourth phase of proposed translator		

BIBLIOGRAPHY

BIBLIOGRAPHY

1. Bingham, H. W. "Use of APL in Microprogrammable Machine Modeling," ACM SIGPLAN NOTICES Special Interest Group on Programming Language, Vol. 6, No. 9, Oct. 1971, pp. 105-109.

A brief description of a microprogrammable machine model which allows microprogrammer interaction in developing the machine design is given. Bingham gives a number of advantages of APL for this process, such as: less core used for object code than ALGOL, quick checkout of alternate algorithms. He also found APL adequate for: source file construction and editing, translator writing, microcode preparation, translation, and checkout.

2. Bingham, H. W. Personal communication, Aug. 27, 1973.

Answer to my request for information on the use of APL for creating higher level languages for microprogramming. APL has been used for microcode preparation, translation, and checkout.

3. Chu, Y. "An ALGOL-like Computer Design Language," Communications on the ACM, Vol. 8, No. 10, 1965, pp. 607-615.

Chu writes that an ALGOL-like Computer Design Language (CDL) will make the logic design of a computer easier to write, understand, document, simulate, and debug. He lists the purpose and requirements for this design language. The language is informally defined and five example sequences (fetch cycle, addition, subtraction, multiplication, and division) are written in CDL.

4. Chu, Y. Introduction to Computer Organization. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1970. 376 pp.

The construction, operation, and programming of the Bi-Tran Six computer is described using Computer Design Language (CDL). The possible syntax statements in CDL are given and CDL is used as the language to write algorighms for parallel addition and subtraction, multiplication, division, and their controlling sequences. CDL is used to describe microprogramming sequences.

5. Chu, Y. Computer Organization and Microprogramming. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1972. 533 pp.

This is the author's second book that uses Computer Design Language (CDL) to describe the organization and functions of digital computers. Chu gives a large number of detailed examples, both simple and complex, of computer organization. His general approach is to describe the computer organization first by using English statements and block diagrams, then using a flow chart, and lastly CDL. CDL is used in describing microprogramming, virtual memory, synchronous and asynchronous organization, I/O Control units, and the complete organization of the IBM's 360/40 computer.

6. Conway, M. E. "Proposal for an UNCOL," Communications of the ACM, Vol. 1, No. 10, Oct. 1958, pp. 5-8.

Conway's proposal is a very simple language for UNCOL. The language, called simple machine language, has two instructions: 1) a memory-to-memory transfer, and 2) an execute instruction. He states that this UNCOL translation is most efficient when the machine language is single-address and has few registers and instructions.

7. Crockett, E. D., Copp, D. H., Frandeen, J. W., Isberg, C. A., Bryant, P., Dickison, W. E., and Paige, M. R. "Computer-Aided System Design," Proc. of Fall Joint Computer Conference 1970, pp. 287-296.

This article is a report on a study in which the authors participated for IBM which defined the proposed Computer-Aided System Design (CASD) system. CASD is an on-line system which utilizes a high level PL/1-like language to describe digital devices. The proposed system attempts to unite all phases of computer design, including a microcode translator. The system was never implemented.

8. Darringer, J. A. "The Description, Simulation, and Automatic Implementation of Digital Computer Processors," Ph. D. dissertation, Electrical Engineering Dept., Carnegie-Mellon University, Pittsburgh, Pa., 1969.

Darringer describes the implementation of a high level ALGOL-like language that is used to simulate and produce Boolean equation specification

of a described processor. The language is called Algorithmic Processor Description Language or APDL. A 25 page summary of related work is given. The description, simulation, and translation of the Bi-Tran computer at three levels of detail are used as examples of the capabilities of APDL.

9. Davis, P. M. "Reading in Microprogramming," IBM Systems Journal. Vol. 11, No. 1, 1972, pp. 16-40.

This paper is divided into two parts: an expository section for the reader unfamiliar with microprogramming and an annotated bibliography. The bibliography has 53 references. Davis discusses such basic concepts as the control section, microinstruction formats, microprogramming, and design considerations. He supports writable control storage, but feels user microprogramming is unwise. He feels microprogramming is often a one shot process and therefore high level microprogramming languages are not required.

10. Eckhouse, R. H. "A High-level Microprogramming Language (MPL)," Technical Report, 1-71-mu. The State University of New York at Buffalo, June, 1971.

This Ph. D. dissertation defines a Pl/1-like microprogramming language (MPL). A three phase translator, of which the 1st phase is independent of the described machine, is described. The paper includes an extensive literature survey of high-level system programming and hardware design languages.

11. Eckhouse, R. H. "A High-level Microprogramming Language (MPL)," Proc. of Spring Joint Computer Conference, Vol. 38, 1971, pp. 169-177.

This is a report on the first part of Eckhouse's Ph. D. dissertation, which defines MPL. This report covers the same material as his dissertation except the application of MPL to microprogramming the IBM 2050.

12. Falkoff, A. D., Iverson, K. E., and Sussenguth, E. H. "A Formal Description of System/360," IBM System Journal, Vo. 3, No. 3, 1964, pp. 193-262.

This paper presents a precise formal description of the IBM S/360 using APL notation. The APL systems programs, of which there are 14, describe the behavior of the machine as seen by the programmer. The systems

programs include programs to describe the central processing unit, instruction execution, and I/O interruption.

13. Gentry, M. "A Compiler for Computer Hardware Expressed in Modified APL," Ph. D. dissertation, University of Arizona, June 1971, Department of Computer Science.

A logic design translator program is developed which automatically 'compiles' high level descriptions (AHPL) of computer architecture into boolean equations of equivalent hardware machine. Complex structures of purely combinational logic may be described separately and compiled as subroutines to the main program. A virtual wire list is generated and printed. The compiler is written in the SNOBOL4 language and run on a CDC 6400 computer.

14. Gilman L. and Rose, A. J. APL/360 An Interactive Approach. New York: John Wiley & Sons, 1970. 335 pp.

This book is a "programmer's guide" to the use of APL/360. It covers all the basic features of APL/360 using a modified programmed instruction method of presentation.

15. Hellerman, H. Digital Computer System Principles. New York: McGraw-Hill, 1967. 424 pp.

This book uses APL notation to describe digital computer systems. Chapter 2 gives a detailed treatment of several APL programming examples.

16. Hill, F. J. and Peterson, G. R. Digital Systems: Hardware Organization and Design. New York: John Wiley & Sons, 1973.

The authors use control sequences written in AHPL (a version of APL) to describe the hardware organization of computers. AHPL can be used to show all data flow and control operations. AHPL includes notation for timing, synchronous and asynchronous operations, combinational logic subroutines, and microprogramming control. A Small Instructional Computer is completely described in AHPL.

17. Husson, S. S. Microprogramming: Principles and Practices. Englewood Cliffs, New Jersey: Prentice-Hall, 1970. 614 pp.

The first four chapters of this book cover the general concepts of microprogramming and include lists of advantages and disadvantages. The author gives requirements for a high level microprogramming language and describes a compiler which uses intermediate macros to compile his high level microprogramming language. The last five chapters give detail descriptions of the microprogramming for IBM S/360 Models 40 and 50, RCA Spectra 70 Model 45, and the Honeywell H4200. The bibliography lists over 350 references on microprogramming and control storage technology.

18. Iverson, K. E. A Programming Language. New York: Wiley, 1962. 286 pp.

Iverson introduces his language (notation), which is called APL or "Iverson Notation." The language is presented in Chapter 1. Chapter 2 gives examples of "microprogramming" using APL notation. A large part of the book consists of examples used to demonstrate the power and versatility of APL.

19. Raynaud, Y., Robichaud, L., and Simian, G. "APL in the Description and Simulation of Computer Systems," SIG micro NEWSLETTER, A Quarterly Publication of the Special Interest Group on Microprogramming ACM, Vol. 3, No. 4, Jan. 1973, pp. 5-24.

This short article demonstrates how APL can be used to describe a computer at the general architecture level and at the microprogram control level. Simulation programs written in APL for both levels are given as examples.

20. Rosin, R. F. "Contemporary Concepts of Microprogramming and Emulation," Computer Surveys, Vol. 1, No. 4, Dec. 1969, pp. 197-212.

This paper covers the history and basic ideas of microprogramming. A simple machine is described and microsequences are written for it. Current microprogrammable machines are discussed. Areas of research in microprogramming are indicated.

21. Rosin, R. F., Frieder, G., and Eckhouse, R. H. "An Environment for Research in Microprogramming and Emulation," Communications of the ACM, Vol. 15, No. 8, August 1972, pp. 748-760.

This is a report on the development and the research project in microprogramming and emulation at State University of New York at Buffalo. It covers the method they used to evaluate and select a suitable machine for microprogramming research. They indicate the need for a high level language that can be used to produce microcode and nanocode. They plan to use the research reported in Eckhouses' Ph. D. dissertation as the basics for their task of preparing a high level microprogramming language.

22. Tesler, L. G. and Enea, H. J. "A Language Design for Concurrent Processes," 1968 Proc. of Spring Joint Computer Conference, Vol. 32, pp. 403-408.

The authors propose a single assignement language, called COMPEL, which can be used to program concurrent processes for multiprocessing systems. The single assignment rule is: "No variable is assigned values by more than one statement." The statements are executed when all the variables on the expression side are defined and not in the order the statements appear in the program. Advantages and examples of COMPEL programs are given.

23. Tsuchiya, M. "Design of a Multilevel Microprogrammable Computer and a High-level Microprogramming Language," Ph. D. dissertation, Department of Computer Science, University of Texas at Austin, 1973.

This paper is a report on a number of studies conducted by Tsuchiya and Ramamoorthy of the University of Texas that they hope will promote user microprogramming. A high level microprogramming language called SIMPL is proposed. A simple language for describing computer organization called MODEL is described. A multilevel microprogramming scheme is discussed. This scheme and the language SIMPL are used to show how microprogrammed computers can enhance performance over comparable conventional computers.

24. Tucker, S. G. "Microprogram Control," IBM System Journal, Vol. 6, No. 4, 1967, pp. 222-241.

Microprogramming the S/360 line of computers made possible a large fixed instruction set that is acceptable to all S/360 models. The S/360 uses a horizontal microinstruction format, which is coded using a somewhat higher level microprogramming language. The language is written on special flow chart like paper. S/360 microprogramming is

automated in a number of ways. The programming of the S/360 models 30 and 65 to emulate IBM 1401 and 7000 series is discussed.

25. Wilkes, M. V. "The Growth of Interest in Microprogramming--A Literature Survey," Computer Surveys, Vol. 1, No. 3, Sept. 1969, pp. 139-145.

Wilkes, who is considered the father of microprogramming, traces the evolution of microprogramming from his original paper on the subject in 1951 up to 1969. The bibliography includes 55 papers.

The "Editor's Preview" by W. S. Dorn in the same issue, includes a brief introduction to microprogramming. (pp. 135-6)

A COMPARATIVE STUDY OF HIGH LEVEL MICROPROGRAMMING LANGUAGES

by

EUGENE SCHREINER

B. S., Fort Hays Kansas State College, 1967

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1973

Since 1964 a number of computers with microprogrammed control have been built. As more microprogrammed and micro-programmable machines become available, it is apparent that the availability of a high level microprogramming language would enhance the utilization of microprogram-controlled computers.

This report is a critical survey of the current high level microprogramming languages. The requirements for a suitable language are given, and these are the criteria by which the available languages are judged.

The languages surveyed are: APL, AHPL, CASD, CDL, MPL, and SIMPL. The conclusions that are drawn in this report are:

1. APL is the only language that has been used to produce usable microcode for a particular application.
2. There is no currently implemented high level microprogramming language.
3. If and when MPL and SIMPL are implemented they should prove to be suitable languages for microprogramming.