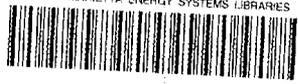


oml

**OAK RIDGE
NATIONAL
LABORATORY**

MARTIN MARIETTA

MARTIN MARIETTA ENERGY SYSTEMS LIBRARIES



3 4456 0268434 3

ORNL/TM-10616

An Investigation of Very High Level Languages and Their Implementation on a Forth Language Microprocessor

H. G. Arnold
W. B. Dress
R. S. Loffman

OAK RIDGE NATIONAL LABORATORY
CENTRAL RESEARCH LIBRARY
CIRCULATION SECTION
OAK RIDGE, TENN.

LIBRARY LOAN COPY

DO NOT TRANSFER TO ANOTHER PERSON
If you wish someone else to see this
report, send to name with report and
the library will arrange a loan.

OPERATED BY
MARTIN MARIETTA ENERGY SYSTEMS, INC.
FOR THE UNITED STATES
DEPARTMENT OF ENERGY

Printed in the United States of America. Available from
National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Road, Springfield, Virginia 22161
NTIS price codes—Printed Copy: A04 Microfiche A01

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

ORNL/TM-10616

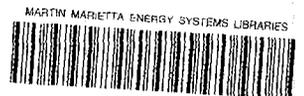
ENERGY DIVISION

AN INVESTIGATION OF VERY HIGH LEVEL LANGUAGES AND
THEIR IMPLEMENTATION ON A FORTH LANGUAGE MICROPROCESSOR

H. G. Arnold
W. B. Dress
R. S. Loffman

November 1987

Prepared by the
Oak Ridge National Laboratory
Oak Ridge, Tennessee 37831
operated by
Martin Marietta Energy Systems, Inc.
for the
U.S. Department of Energy
under Contract No. DE-AC05-84OR21400



3 4456 0268434 3

TABLE OF CONTENTS

	<u>Page</u>
ABSTRACT	v
1. INTRODUCTION	1
2. OVERVIEW OF THE APPROACH	2
3. THE PROCESSING POTENTIAL BY WAY OF BENCHMARKS	3
3.1 PROCESSING GOALS	3
3.2 BENCHMARKS	5
4. VERY HIGH LEVEL LANGUAGE	11
4.1 CHARACTERISTICS OF VERY HIGH LEVEL LANGUAGE	11
4.2 VHLL CONCLUSIONS	18
5. FORTH AS A MEANS FOR WRITING A VERY HIGH LEVEL LANGUAGE	19
5.1 WHAT IS OPS5	20
5.2 WRITING OPS5 IN FORTH	22
5.3 USING REAL-OPS	30
5.4 HIGH SPEED EXPERT SYSTEMS	34
6. OPS5 AND THE FORTH CHIP	35
7. VERY HIGH LEVEL LANGUAGE POTENTIAL OF THE FORTH CHIP	39
7.1 LANGUAGE TIERS AND REDUCED INSTRUCTION SETS	40
7.2 TIERING IN THE FORTH CHIP MICROPROCESSOR	43
7.3 VERY HIGH LEVEL LANGUAGES ON THE FORTH CHIP	44
7.4 VERY HIGH LEVEL LANGUAGE POTENTIAL FOR THE FORTH CHIP	47
8. OBSERVATION AND RECOMMENDATIONS	49
9. REFERENCES	52

LIST OF FIGURES

	<u>Page</u>
Figure 1. Comparison of Times to Perform One Million Iterations for Selected Microcomputers	6
Figure 2. Comparison of Times to Perform One Million Iterations for Selected Mainframes and Microcomputers	7
Figure 3. Time for 100,000 LISP Iterations (Seconds)	9
Figure 4. Time for Towers of Hanoi Solution in Seconds for Selected Mini- and Microcomputers Running OPS and Forth Inference Engine	9
Figure 5. Conceptual Diagram of REAL-OPS Showing Main Data and Control Flows	24
Figure 6. Data Flow to and from a Real-Time Expert System	29

LIST OF TABLES

	<u>Page</u>
Table 1. Glossary used by Data-Handling Routines for Communicating Asynchronous Events to the Expert System	32

ABSTRACT

The potential for using a Forth language microprocessor to implement very high level languages (VHLLs) in Artificial Intelligence research was investigated by surveying the current state-of-the-art of VHLLs, by benchmarking several computers and microcomputers against a customized Forth Language microprocessor, and by extrapolating the results to draw conclusions about implementing expert systems on the Forth language microprocessor.

1. INTRODUCTION

For the next five to ten years, the focus of the computer world will be on the area of Artificial Intelligence applications, such as symbolic processing, decision support systems and expert systems. Much work on the advancement of expertise in this area has already taken place at research institutions, universities, and in private industry. Many predictions about the future of computing have grown out of this work (concerning the number of rules per second, for example, or execution speed), but it is too early to say which prediction will prove most accurate. However, it is certain that there are two requirements for any of these predictions to become a reality. One is a machine architecture capable of providing the high processing speed which will allow computers to simulate human intelligence. The second requirement is communication interface between high execution speed computers and the experts using the machines.

Recent work at Oak Ridge National Laboratory (ORNL) has addressed both of these requirements by demonstrating, with impressive results, the feasibility of putting a high level language on a fast-architecture microcomputer. The success of this work demonstrates the potential of handling symbolic processing at high enough speeds to meet both the above mentioned requirements: high processing speed and human-machine interface.

2. OVERVIEW OF THE APPROACH

Two parallel efforts were undertaken to determine the appropriateness of the Forth language microcomputer chip to Artificial Intelligence, and to determine the possible future direction of research activity with the chip. The first involved performing benchmark tests of processing speeds on several different computers. The results of these benchmarks provide a basis for comparison between the Forth chip and other computer architectures. The second involved a survey of literature, and communications with researchers to determine the state-of-the-art of very high level languages, with an eye toward the need for such a language in data processing use of the chip.

Another effort undertaken in the project involved rewriting of the expert systems language OPS5 in Forth and determining if it could be ported to the Forth chip. The success of this effort, in combination with the high-processing speed as illustrated by the benchmark tests, provide sufficient evidence that the integration for Forth, Forth engines and expert system technology could provide the high performance needed for Artificial Intelligence applications in the next five to ten years.

3. THE PROCESSING POTENTIAL BY WAY OF BENCHMARKS

Symbolic processing is not number crunching. It is essentially the use of symbols, strings or typographical notation to accomplish data processing needs. The more common examples of symbolic processing are in the area of expert systems, and include the object-oriented programming, list processing and rule processing techniques that are the underlying themes of this research. While the subject of expert systems is important in itself, it must be recognized that symbolic processing may require a different reference point -- a different architecture, different goals, and consideration of a different end-user community. The users of symbolic processing are not necessarily the same accountants and scientists who specified and developed the current numerical data processing architectures and models. It is more likely that the new audience for these types of programs are in the front office, and are looking for what we call Decision Support Systems, which help in the accumulation and analysis of data for the purpose of making decisions more effectively.

3.1 PROCESSING GOALS

Several groups of researchers have looked at Artificial Intelligence (AI) in computer systems. If one accepts the premise that symbolic processing is the basis of writing AI programs, and that expert systems are the forerunners of AI programming, then these researchers have identified two needs that AI development tools must meet. First, there must be a Very High Level Language (VHLL) to simplify the communication between computer science experts and experts

in other areas. The next chapter describes the characteristics of VHLL. Second, expert systems must have high processing speeds. DARPA¹ has concluded that in order to meet the likely needs of expert systems, a computer program must be capable of handling a rule base of 30,000 rules, at a rate of 10,000 rules per second. This expert system will not be able to handle all types of decisions, but may be able to identify friend or foe in a real-time situation. Present expert systems on existing mainframes can process about 200 rules per second (depending on the definition of a rule). Therefore, these researchers estimate that the goal of 10,000 rules per second may be achievable by the early 1990s.

Preliminary investigations into using Forth as the programming language have shown that the potential for meeting this type of goal may not require the mainframe route. A version of OPS5 on a microcomputer² has run as fast as the same widely-accepted language runs on popular minicomputers. This microcomputer version of OPS was written in Forth for a 68000 processor desktop computer. Extrapolations of this initial OPS performance to other environments indicates that the DARPA goals may not only be achievable, but may even be possible today in shoebox-size computer systems. Benchmarks of a Forth chip (a microprocessor using Forth as its machine language) in October 1985 indicated that speeds of up to 20 times as fast as the 68000 were possible if the Forth chip could support symbolic processing as well as it supports integer arithmetic. This set of early benchmarks led to the investigation of the potential for symbolic processing speed through other benchmarks, which are reported here.

3.2 BENCHMARKS

One rough measure of symbolic processing speed is the rate at which integer operations can be performed, i.e. simple DO-LOOPS. Figure 1 shows the results of running a million iterations on several types of computers in empty loops for which only the loop instructions were executed and in loops for which a 16-bit or 32-bit number was stored into memory. While the results are not to be interpreted as the actual speed of symbolic processing, they do offer some feel for the relative speed at which a rule represented by some type of pointer might be processed (as opposed to a rule represented by a string compared to another string). The overall conclusion is that 32-bit micros running Forth are more than twice as fast as 16-bit micros, maybe even as much as three to four times as fast.

Figure 2 shows the same benchmarks with the fastest 32-bit micro compared to the Forth chip and to three large computers. Based on these results, the Forth chip should be expected to perform integer operations about 15 times as fast as a fast microcomputer. What is interesting to note is that the chip can keep up with a VAX 11/780 when running empty loops and there are conditions using a special FOR-NEXT feature of the chip under which it can almost keep up with one of the fastest mainframes made. This apparent anomaly is due to the inability to utilize the fast machine's architecture and illustrates the point that symbolic processing requires a different approach than floating point parallelism.

Time for One Million Iterations

(00:00.0 Min:Sec)

CONTENTS OF LOOP	Empty Loop	16-Bit Integer Store *	32-Bit Integer Store
COMPUTER			
valForth Atari 800	1:47.30	5:25.19	22:45.00
MVP-Forth IBM-XT	1:35.10	3:01.50	4:29.80
Forth-32 IBM-XT	1:08.55	4:04.55	4:13.25
MVP-Forth IBM-AT	0:35.85	1:07.25	1:46.10
Forth-32 IBM-AT	0:24.59	1:23.96	1:25.25
MacForth MacIntosh	0:19.10	1:06.46	1:06.31

• VARIABLE ITEST (16-BIT Integer Store)
 : THOUSAND 999 0 DO 1 ITEST ! (W!) LOOP ;
 : MILLION 999 0 DO THOUSAND LOOP ;

Figure 1. Comparison of times to perform one million iterations for selected microcomputers

Time for One Million Iterations

(00:00.0 Min:Sec)

CONTENTS OF LOOP	Empty Loop	16-Bit Integer Store *	32-Bit Integer Store
COMPUTER			
MacForth MacIntosh	0:19.10	1:06.46	1:06.31
Novix Forth	0:02.50 0:00.17 **	0:03.22 0:01.20	0:05.81 0:03.69
Cray-XMP (FORTRAN)	0:00.12		
IBM-3033 (FORTRAN)	0:00.41		
VAX-780 (FORTRAN)	0:02.112		

** FORTH algorithm to compare with mainframe times
(based on special customized loop similar to optimized compilers)

Figure 2. Comparison of times to perform one million iterations for selected mainframes and microcomputers

In an attempt to add the complexity of VHLL to the execution burden, a simple set of LISP instructions was written for the Forth chip and compared to the same LISP instructions running on a microVAX and LMI LISP machine. Figure 3 shows these results for 100,000 iterations of a LISP do-loop that performed list processing within the loop. While the comparisons are not actually on the same basis, since the Forth chip automatically did "garbage collection" and the LISP machine takes "forever" to do it, the conclusion is that the Forth chip can run LISP just about as fast as a LISP machine. This does not address the potential for optimization of the code for the chip nor the problem of the tested prototype chip in handling strings of bytes 40 times as slow as it should because of cell addressing.

A closer case to optimization was done with an inference engine for Forth called FORPS³. This expert system lacks the VHLL features by requiring Forth words in its rules, but clearly takes advantage of the Forth language in writing an expert system. The results of this system running on the chip are compared to other computers in Figure 4. The comparison is only incidental, however, since the results present an opportunity to calculate the speed of rule processing in an environment that may be recognizable, the classic "Towers of Hanoi" problem solution. In a goal-directed inference situation, the 68000 processor running Forth achieved speeds close to the 200 rules per second of mainframe machines (while minicomputers running a VHLL in the form of OPS could only do about 10 rules per second). The Forth chip, however, achieved processing times in the range of 4000 to 6000 rules per second in solving the Towers problem. It must be noted that there were only

Garbage Collection	LISP MACHINE LMI	MINI COMPUTER Micro-VAX	FORTH CHIP Novix
(NO)	22	60	NA
(YES)	(15 Min?)	95-167	179

Figure 3. Time for 100,000 LISP iterations (seconds)

No. of Disks					
	LISP MACHINE	VAX 780	68000 8 MHz	68000 10 MHz	NOVIX 6 MHz
5				0.29	0.01
7	22	60	26	1.15	0.06
10				9.25	0.42

(Seven disks required 256 rule firings for inference engine.)

Figure 4. Time for Towers of Hanoi solution in seconds for selected mini- and microcomputers running OPS and forth inference engine

four rules and that OPS would handle large rule bases more efficiently; but the fact remains that such processing speeds are possible in small computers when the architecture of the machine, the design of the solution, and the type of the problem are compatible.

4. VERY HIGH LEVEL LANGUAGE

An important consideration in this research is the concept of a Very High Level Language. It is through a language of this type that a human user must interface with the system of a computer -- especially if the system is an attempt to simulate the decision-making process that an "expert" goes through. The underlying theory behind expert systems is that the computer is programmed in such a way that an expert in one field need not also be a computer programming expert in order to use the system. Current thinking is that such a language should come very close to the language of the user expert, to facilitate the human/machine communication.

For this reason, a survey of the state of the art in VHLLs was undertaken. The survey was to determine if a VHLL should be developed for use on a high-speed computer using expert systems.

4.1 CHARACTERISTICS OF VERY HIGH LEVEL LANGUAGE

The field of VHLLs is a new research area, and new knowledge and understanding are being gained continually. However, because of their relatively recent inception, there is a diversity of ideas and concepts about VHLLs. The names or labels for them are equally diverse. A characterization of VHLLs is important, because it will provide a basis for common understanding, which in turn will facilitate communication. Improved communication will result in the sharing of research and ideas, thereby clarifying future directions for the field. A survey was done to determine some characteristics of VHLLs, as found in both current literature and research activity.

At present, there seems to be no consensus on what a VHLL is or what the criteria are by which one determines whether something (a programming language, presumably) is or is not a VHLL. Many similar ideas appear to have different variations or labels, and it is acknowledged that no one language or environment is capable of doing all the necessary or desired tasks. Therefore, VHLLs must be extremely flexible.

Programming languages have always been described by generic labels. For example, the "generation approach" ranges from first-generation languages (1GL) through fourth-generation languages (4GL), and is now entering the fifth generation. However, first-generation languages were a very primitive means of utilizing the first computers. In fact, "languages" may be a too-generous label, since they dealt with computers at the hardware level.

Second-generation languages were the first to provide for the stored program concept. They were of two types: machine-level languages, and their improved version, assembly languages. Machine languages consist of binary symbols (strings of 0s and 1s) which are difficult to deal with and are meaningless at face value. Assembly language instructions consist of mnemonics, with each instruction representing one machine instruction; these were an improvement upon machine languages because they introduced some readability and structure to programs. However, both of these languages require very skilled programmers who have extensive knowledge of the underlying hardware architecture.

Third-generation languages were a major improvement, because one program instruction represents multiple machine instructions. This eases the burden on the programmer by lessening the amount of code to be written, and the resulting code is more understandable and easier to maintain than assembly language code. Like assembly languages, 3GLs are procedural, with each program statement executed in the order it was written. Fortran, Cobol, and PL/1 are among the third-generation languages.

Fourth-generation languages cover a wide range of capabilities, including more English-like querying of data, report generating, and graphics. They are often labeled as productivity tools, because they require few programmer-written instructions, and applications can be developed in a relatively short amount of time. They are less procedural than previous generations and are more suited to use by nonprogrammer professionals. 4GLs are often associated with database management systems and include query languages, forms, and report writers.

Fifth-generation languages are the up-and-coming generation. It is difficult to make generalizations about 5GLs as yet; perhaps they are synonymous with expert systems or VHLLs. (The use of "fifth-generation" here should not be confused with the Japanese government's Fifth Generation Computer Systems (FGCS) project. The FGCS project integrates knowledge, engineering applications, very high level programming languages, decentralized computers, facilities for human-oriented input/output, and it exploits Very Large Scale Integration (VLSI) technology.⁴)

"High level language" is another programming language label. High level languages are those which translate one programming instruction of a low level language into several assembly-level instructions. They are comparable to the third and fourth generations mentioned above. This one-to-one translation increases programmer productivity and decreases programming error rates. Maintenance is also easier because of the more readable code produced by such languages.

More advanced levels of languages in this scheme of categories include VHLLs and expert system languages. Here however, distinctions become hard to discern, because there is less consensus on the true nature of these languages. A VHLL is probably comparable to a fifth-generation language; the rest of this discussion will be focused on characteristics of VHLLs. However, an underlying set of questions is whether expert systems are synonymous with VHLLs; whether VHLLs are required in writing expert systems; and whether VHLLs require the use of expert systems.

Three basic characteristics of VHLLs are consistent throughout current literature and research activity. These are that a VHLL should employ declarative statements, allow implicit referencing, and promote the communicability of knowledge⁵.

The first characteristic refers to the VHLL being nonprocedural, as opposed to traditional languages (Fortran, COBOL, etc.,) which are procedural. In procedural languages, the programmer details the processing to be done and the order in which it is to be executed; in other words, he or she not only instructs the computer what to do, but also how to do it. Nonprocedural languages, on the other hand, specify

what is to be done but not the steps needed to achieve the goal. This increases programmer productivity by allowing human thought to be devoted to the creative aspect of problem solving while the computer resources are devoted to the more mechanical aspects of executing the solution.

The second VHLL characteristic allows for implicit reference through the use of inheritance capability; that is, a particular characteristic is associated with a set (or class) of objects wherein all subsets inherit the same characteristics without them having to be repeated each time a subset is declared. This eases the specification burden on the programmer by allowing new classes to be built on existing ones. Inheritance also improves organization of information and programming flexibility through this building-block approach.

The third characteristic refers to the capability for the user and the computer to communicate. A database is of no value if it cannot be interpreted as knowledge either by the computer for processing or by the user in understanding the result. "User friendly" is a much-used term which partially describes the ease of human-computer interface.

After an extensive survey of the literature and several personal communications with researchers, it is evident that three additional features are desirable when considering VHLLs. These are that the language verify correctness, accommodate change, and deal with data in new ways.

The verification of correctness refers to the ability to check syntax -- the correct format and spelling of commands -- and semantics. (Forth does little syntax checking, but Brodie⁶ contends that syntax

checking, as it is commonly perceived, would limit the freedom and flexibility provided by Forth.) Semantics checking can occur at two different levels, internal and external. Internal semantics is concerned with whether what is being said adheres to the rules of the language. Syntax and internal semantics checking are present in most languages in varying degrees as a compiler function. The extent to which compilers supply meaningful messages to help correct these errors differs greatly among different languages. External semantic addresses whether the system is solving the right problem⁷. This type of checking requires a much broader understanding, because it requires vast knowledge of the problem domain and of the rules governing how it functions. This external semantics verification is not trivial, and perhaps implies the need for an Artificial Intelligence capability within the language.

The second desired capability, the ability to accommodate change, has always been important, and requires that the language be flexible. Forth, for example, embraces this capability by building new words based on existing words in its dictionary; and it is Forth's extensibility that makes possible the writing of languages such as OPS with it.

The third desired capability, the ability to deal with data in new ways, is similar to extensibility in that data requirements and ways to express data are unpredictable and dynamic. Traditional data types (integer, float, character, etc.) are no longer sufficient ways to express data needs.

These three desired capabilities, when combined with the aforementioned basic characteristics, produce the characteristics of a very high level language. These six characteristics are not all present in any one existing language, however. They are currently implemented by flexible data structures, abstract data types, knowledge engineering, Artificial Intelligence, expert systems, graphics and database management systems.

Newer programming techniques are focused on different aspects of these characteristics. "Access-oriented programming" enables procedures which are triggered by data activity to be invoked. More specifically, procedures are associated with data, so that when a particular piece of data is fetched or stored, another activity is initiated.

"Object-oriented programming" (e.g., Smalltalk) groups data into objects or abstract data types. Objects are characterized by a type of behavior which is inherited by subclasses. This type of behavior specifies how the data in the particular class can be manipulated. New classes can be built on top of existing classes utilizing the inheritance capabilities (mentioned previously) of object-oriented programming languages.

"Logic-oriented programming" (e.g., Prolog) is concerned with nonprocedural representation of knowledge and is used in inference situations.

"Function-oriented programming" (e.g., LISP) is concerned with transformations applied to data. These transformations are based on mathematics providing a sound basis. These techniques are primarily

used in the area of Artificial Intelligence and expert systems; however, as stated previously, it is not clear what the relationship is between VHLLs and expert systems/Artificial Intelligence.

4.2 VHLL CONCLUSIONS

Whatever direction VHLL research takes, the characterization of VHLLs is an important step because it will allow a framework for communication among researchers. Based on current understanding, six basic features of VHLLs are apparent. These are that the language (1) be nonprocedural, (2) allow implicit referencing, (3) provide a good interface between user and computer, (4) allow for verification, (5) be extensible, and (6) provide the means for better data representation.

Currently, no language satisfies every VHLL characteristic, and no common understanding of the requirements for a VHLL exists. Future languages and tools will either be designed for specific purposes or will be combinations of several tools. Both of these approaches have merit. The first will produce powerful tools; however, they will be limited in scope. The second will produce tools with broader application scope but less power for a particular task.

5. FORTH AS A MEANS FOR WRITING A VERY HIGH LEVEL LANGUAGE

As the success of Artificial Intelligence applications became evident in the area of expert systems for medical diagnostics⁸ and computer configuration⁹ problems, it was just a matter of time until extending the methodology to problems of real-time process control and data reduction was attempted. First attempts showed clearly that execution speed would be a limiting factor for any real-world control problem, so attention was given to making LISP machines run faster and to providing I/O channels with higher band width. One of the successes with a complex expert system digesting large amounts of incoming data and providing expert decisions in real time¹⁰ showed how effective such an approach could be.

However, the problems of efficient access to the inference engine and working data set by asynchronous external events were largely ignored, probably being left for hardware manufacturers to solve via the "bigger and faster" route. This seems an unsatisfactory state of affairs in that only those institutions able to afford the high-end, newly-developed LISP machines are able to consider applying real-time expert systems to their problems.

REAL-OPS (a version of OPS5 written in the Forth language) is an attempt to fill the gap between merely running a system faster (waiting until the necessary data are present at an I/O port) and the asynchronous, multi-tasking operation of a real-time, event-driven system. The idea is to start with an effective real-time, multi-tasking software base and build into it the necessary expert system capabilities in a manner recognizable to both the expert system and real-time control

communities. To this end, a multi-tasking version of Forth¹¹ containing the necessary interface to external devices (database machines, disk drives and other computers) was used. Thus, the problems (real-time, multi-tasking, and access to external events) were neatly solved, leaving the problem of integrating an established expert system language with an underlying Forth base.

OPS5, a widely used production-rule language, was chosen as the very high level language for embedding in Forth. The reasons for this choice were twofold: first, OPS5 is a powerful, forward-chaining, efficient language; second, it is conceptually and syntactically simple. The first property gives the means to produce a language for control applications since such problems are usually event-driven; that is, the expert controller must respond quickly and correctly to the instantaneous data stream that conveys the current state of the system being controlled. The second property allows easy extensions and modifications to the language, producing a more powerful result tailored to the needs of real-time, multi-tasking expert systems.

5.1 WHAT IS OPS5

OPS5 and the OPS class of languages are representative of the emerging very high level languages in that they go beyond an applications-oriented language such as a fourth-generation database language to provide transparent search and match algorithms as well as I/O and database maintenance features. Cosmetically, OPS5 is a scheme for constructing production rules (modules) in the form of IF...THEN... statements, where the IF part, or left-hand side (LHS), specifies a set

of data patterns which must be consistently matched by a portion of the actual data set residing in "working memory;" and the THEN part, or right-hand side (RHS) indicates the set of actions to be carried out when the LHS is satisfied. OPS5, developed at Carnegie-Mellon University¹² in the late 1970s, is one of the most popular and widely used production-rule languages, and the only one with a textbook¹³ devoted to it. Typically, OPS5 is used for writing expert systems; but, being conceptually and syntactically simple, it has much broader applications. The Brownston book¹³ devotes careful attention to deciding when an algorithmic approach should be used for a problem and when a production-system approach is more appropriate. Structure and complexity are the guide: unstructured, complex problems are more amenable to solution via a production paradigm whereas well structured, simple problems are better handled by specific algorithms. That is not to say an expert-system implementation cannot represent a well-structured problem. The often-cited "animals" expert system is usually expressed as a set of backward-chaining rules that form a static, almost algorithmic, decision-tree type system. A truly unstructured problem involving complex, dynamic data patterns, would be difficult to represent in such a fashion.

Each OPS5 rule is an independent module, loosely coupled to other rules by the data set in working memory. There are no global variables in standard OPS5; all variables refer to values of working memory elements, and the particular binding is valid only within the rule where it appears. Conceptually, the set of rules in an OPS5 system may be thought of as "peering" into working memory in parallel, each one

"looking" for a particular set of data patterns. When a rule finds a set of data patterns matching its own pattern prescriptions (condition elements), it is free to "fire."

Internally, OPS5 is much more complicated than a set of if... then... procedures. To see that this must be so, consider an expert system with one thousand rules, each rule having ten conditions and each condition consisting of a ten-element pattern. The brute-force method would be to consider each of the hundred thousand possible pattern elements as a candidate for each data instance in working memory during each system cycle and then decide which rule to fire. Since there may be several thousand working memory elements, each with perhaps ten terms, any direct scheme would take too long to check all of the several billion possible matches to make a real-time expert system feasible. The Rete algorithm¹⁴ is responsible for OPS5's efficiency. This algorithm maintains lists of pointers from those actions potentially making working memory elements that could match particular condition elements to those same condition elements. The current state of the system is maintained and only differences from the current state are noted during each pass through the "recognize-act" cycle. This differential method obviates the need for matching each data-element term against each condition-element term during each system cycle.

5.2 WRITING OPS5 IN FORTH

The task of writing OPS5 in Forth is organized into several "chapters" as recommended by Brodie.¹⁵ The first few chapters deal

with the necessary tools and establish a hierarchical vocabulary structure. The tools chapter contains the words used throughout the entire application for such things as bit manipulation, memory management for dynamic data structure, and list bookkeeping. Since OPS5 requires a multitude of dynamic lists (for pointers, data values, stacks, etc.) as well as the ability to make and remove working memory elements, efficient memory management is essential to Forth implementation. Most LISP and C implementations of OPS5 practice the standard method of garbage collection¹⁶ by marking list nodes as unneeded and deferring memory reclamation until absolutely necessary. The entire expert system must then be put in standby mode while the memory is cleansed of unused data structures. An alternate method, adopted in Real Ops¹⁷, employs a synchronous mode of garbage collection, avoiding the unpredictable, deadly delays of the conventional method.

Figure 5 provides a conceptual overview of OPS5 embedded in Forth. Three entities are external to the system -- an expert system user, a Forth user, and any number of asynchronous external events. Since Forth provides what is usually referred to as the operating system, all communication actually takes place either via Forth's interpreter or by means of drivers written to handle external events. The memory partition common to OPS5 is shown with the major flow of control. Certain pieces are located in the heap memory partition, while other sections are found in the usual Forth vocabularies. The structures located in the heap have pointer references in the various system words, making everything accessible from Forth words.

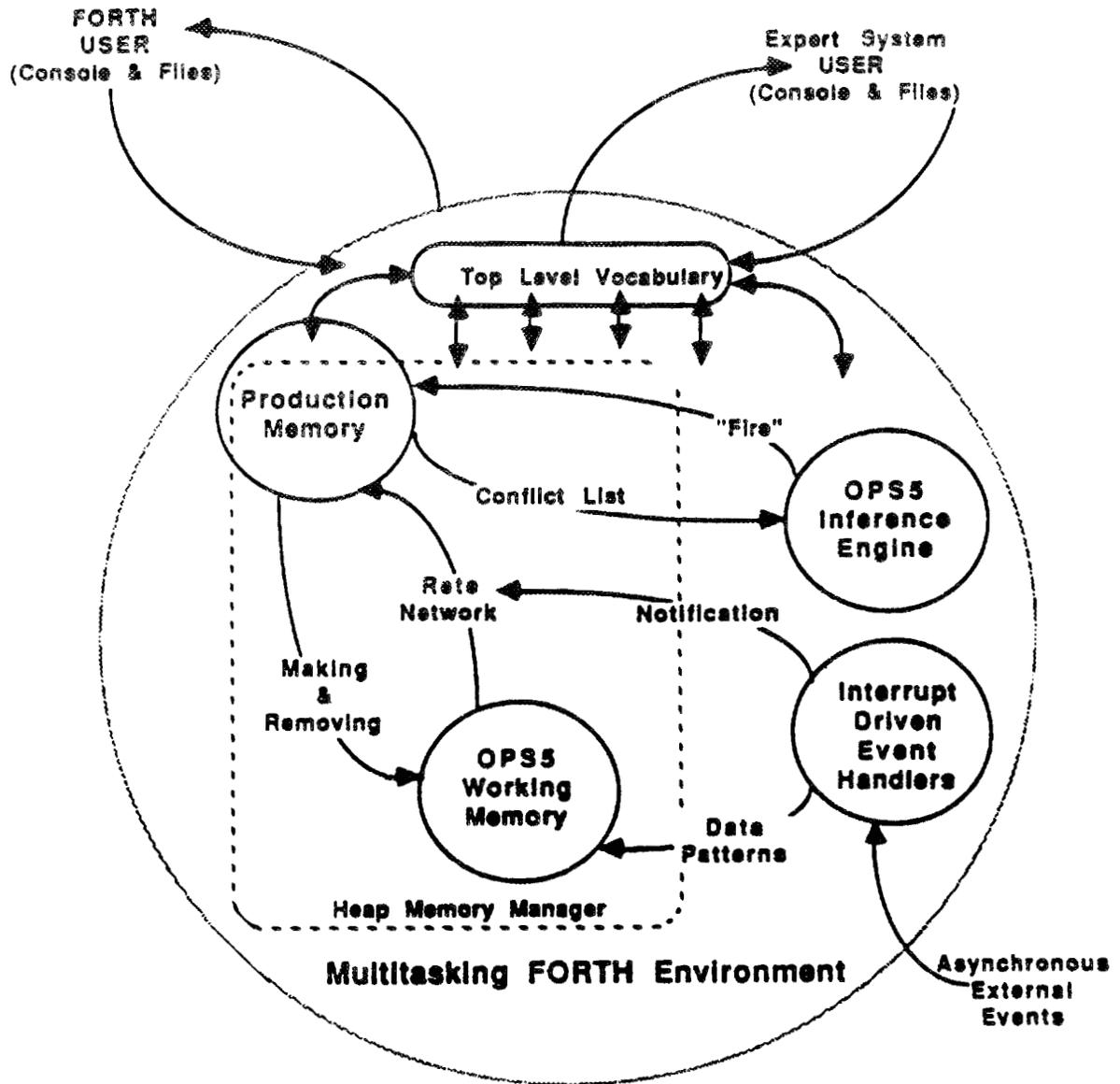


Figure 5. Conceptual diagram of REAL-OPS showing main data and control flows

The Parser. The next REAL-OPS chapter contains the parser, which is implemented as a transition network and makes extensive use of the DOER-MAKE construct¹⁵ for vectoring various words into the parser framework. A special state stack is maintained in the heap, allowing parser states to be nested.

LHS and RHS Compilers. The next implementation chapter contains the mechanism for building the data structures representing production rules and compiling the condition elements into data structures accessible to the rules. Similarly, the mechanisms for building the RHS actions into the rules are collected into a separate chapter. Pointers are maintained in the heap from each class object to the condition elements of that class, similar to establishing inheritance in Smalltalk object. The Rete network is built by matching each new action to all previous condition elements, and each new condition element to all previous actions as the rule set is being compiled. If an action could conceivably produce a working memory element satisfying a condition element, a pointer to that condition element is added to the network list for the action.

Bit maps for the satisfied condition elements in a rule and the terms present in a condition element are maintained while the system operates. Logical comparisons of bit maps speed the matching process and conflict resolution. Each rule also contains a pointer to a list specifying the current state of working memory as viewed from that rule.

RHS Actions. The heart of the system is the chapter containing all the words for effecting the RHS actions in running system. The simple

action of making or removing a working memory element may affect every rule in the system, so a means of matching the potentialities pointed to by the Rete network is needed. If a new working memory element actually matches a rule's pattern, a consistency check needs to be made. A rule waiting for a variable $\langle x \rangle$ and another variable $\langle y \rangle$ where the value bound to $\langle y \rangle$ is not to be greater than the value bound to $\langle x \rangle$ would be matched by all numerical instances of $\langle x \rangle$ and $\langle y \rangle$, but perhaps not consistently so (try $\langle x \rangle = 5$ and $\langle y \rangle = 11$).

Note that each LHS may consist of several condition elements (patterns) and that each condition element may have many working memory elements independently (or disjointly) matching it. The set of possible matches is the power set, or product of the sets of working memory elements associated with each condition element in the LHS. This can easily become a very large set; it is not uncommon for a rule to have 25 condition elements, each with as many as 50 working memory elements. The power set contains 50^{25} elements, an astronomical number indeed. It would be hopeless to attempt to examine each member of the power set for consistency, so, following Forgy,¹⁴ partial sets are constructed and checked for partial consistency. Consistency is checked starting with the first condition element and its most recent working memory element. Then the next condition element is examined, starting with its most recent working memory element, and so on. If the last condition element is reached and shows consistency for one of its matching working memory elements, the set is consistent, and the rule is placed in the conflict set. Should any of the sets of working memory elements become exhausted before the rule can be declared

consistent, the rule is not yet ready to fire and the process terminates. Pointers into the lists of matching working memory elements are kept with each rule so that the process does not need to be repeated in its entirety each time the rule needs checking. Thus, the problem is far less than indicated by the size of the power set.

Conflict Resolution. The recognize-act cycle mentioned above culminates in a set of satisfied rules. Since only one rule may fire each cycle, conflict resolution applies one of several strategies to pick the winner. Deciding on the winning rule involves sorting all rules in the conflict set (which is implemented as a list of rule parameter field addresses) by recency of the working memory element satisfying the rule's first condition element -- the rule with the most recent element wins (Means-End Analysis strategy). This sorting is done by a combination of sorting routines: a standard exchange sort for fewer than four rules, an insertion sort for four to 14 rules, and an iterative version quicksort above 14.¹⁸ (Consult reference 12 or reference 13 for a detailed description of the various strategies and the reasons for choosing one over the other. If more than one rule wins under the strategy chosen, a random choice is made to decide the one winner.)

Top Level User Interface. The final chapter contains the interface to the user for controlling the OPS5 system, defining the class objects and specifying the rules. The spirit of Forth is retained as much as possible in that REAL-OPS is a fully interactive, incrementally compiling version of OPS5. Rules and class objects may be added at any

time, the system may be run one or many cycles, and rules may be removed at will. Working memory elements may be made or removed from the top level, and the state of the system may be examined via a set of commands for displaying working memory, matches to condition elements, and the conflict set.

Multitasking and Real Time. In rewriting OPS5 in Forth, care was taken not to be inconsistent with multi-tasking needs. In order to address the needs of the real-time community, the necessary alterations must be made and the interface to external events provided. If an event is expected, it is a simple matter to enter a wait loop, periodically examining an I/O port or register for the presence of the event. This nice, calm situation succeeds only rarely in the real world of process control and autonomous vehicles, for example. If the event is not expected, how can the "expert" reason about it? Hopefully, the canned expert is not to remain eternally blind, so a way must be found for of getting asynchronous (unexpected) events recognized.

Figure 6 illustrates a typical block diagram for an interrupt-driven expert system. Sensors continuously provide data about the process or experiment being controlled. The sensor data should provide redundant and multi-variate information (one sensor measuring several different properties), perhaps covering different aspects of the same process variable. For example, temperature information could be obtained from thermo-couples, from radiation measurements in different parts of the spectrum and from a fiber-optic device. The expert system

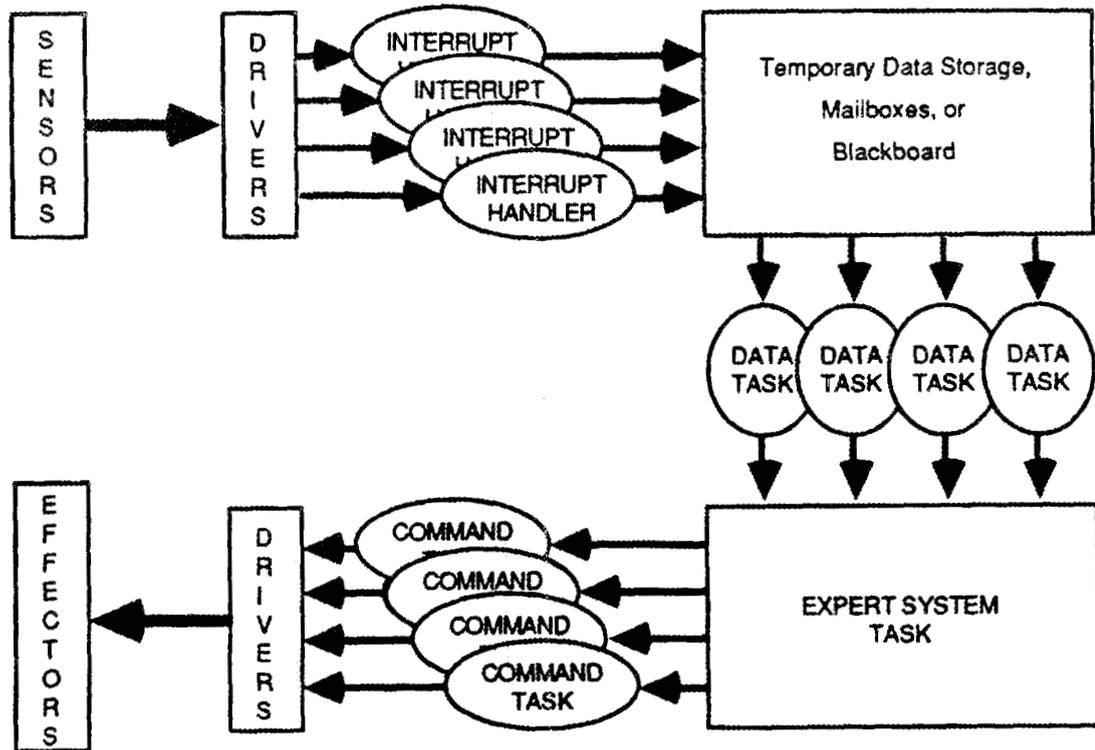


Figure 6. Data flow to and from a real-time expert system

would be required to "fuse" the various data into a consistent picture about the temperature of the object.

The expert system may be required to take actions (other than **MAKE** and **REMOVE**), changing the course of events in the external world. Sensor reading would presumably change, causing the expert system to respond to the new information. The traditional expert system "knows" where it is going at all times as it is receiving answers to questions it is asking by design. These answers fall into definite, planned categories ("does the animal have feathers?"). Since the system conceived here can neither limit nor know the range or patterns of data

that might be presented to it by the real world, external events will alter the course of the patterns of rule firings in ways not foreseen by the programmer; events are not synchronous with the recognize-act cycle. Rules should then be robust enough to allow for this eventuality. In a process control problem, the encoded expertise could consist mainly of rules for deciding which control strategies to apply and when the process is at critical points in the multidimensional phase space. A set of goal-oriented rules would provide an overall optimization much as probing is done in control theory.¹⁹

5.3 USING REAL-OPS

The problems of event access described above are recognized by the word "asynchronous," which means the expert system is not "consciously" looking for events, but it must respond to events nonetheless. The method of treating such data is to provide special pathways²⁰ into working memory. The method can be compared to a once-popular beach toy. The toy had a bucket mounted in a bistable configuration; filling the bucket with sand or water eventually caused the bucket to tip over, spilling its contents into a mechanism of wheels and other buckets. A bucket-defining word was written allowing the programmer to create access pathways into the working memory of REAL-OPS. A bucket looks much like an OPS condition element, but has an attached access-oriented procedure that dumps the contents into working memory when the bucket is filled. Also, a message is sent over the network notifying all interested rules when the working memory element has been made. The

bucket vocabulary, shown in Table 1, is typically used from within interrupt service routines or data-handling routines.

An integrated application using REAL-OPS should consist of three parts: the expert system, the real-world interfaces (drivers), and the operator interface for control and display. The combination of buckets and FORTH actions on the RHS of rules provides entry of data into the expert system's awareness as well as carrying out required real-world actions. The ability to call Forth from rules also allows the expert system warnings through both the graphics and sound mechanism, and display menus for operator data entry.

Desired explanations of the system's behavior choices can similarly be enhanced through the use of Forth words graphically displaying the interconnections between rules and condition elements.

Table 1. Glossary used by data-handling routines for communicating asynchronous events to the expert system

Word	Example	Description
BUCKET	BUCKET (METER ^VALUE ^ID ^TYPE)	Creates a new bucket structure called METER, allocating three slots identified as VALUE, ID, and TYPE
PUT	(see below)	Puts the data element into the indicated slot and marks that slot as being filled
INTEGER	INTEGER PUT METER ^ID , or INTEGER 101 PUT METER ^ID	Uses the top item on the parameter stack for a PUT operation, identifying it as an integer. The second method takes the data item from the in-line code for the PUT expression
SYMBOL	SYMBOL ValM PUT METER ^TYPE	Converts the in-line symbol to a token for the PUT operation. A parameter version operates as in the INTEGER case
STRING	STRING " Meter out of service" PUT METER ^TYPE	Converts each symbol in the string into a token, builds a vector in memory, and uses a reference to the vector for the PUT operation. Also has a parameter version as above
VECTOR	VECTOR PUT METER ^VALUE	A reference to a vector is used for the PUT operation. There is also an in-line version as above
DUMP	DUMP METER	Forces a dump of the METER contents into working memory. This is also the demon that dumps a full bucket
FLUSH	FLUSH METER ^VALUE , or FLUSH METER *	Empties the indicated slot (or the entire structure if *). Used to prevent DUMPing if critical data are expected

Results with REAL-OPS. Real-time benchmarks are difficult to find since each application is different. Falling back on the standard benchmarks of conventional expert systems written in OPS5, two come to mind as having broad appeal: "Towers of Hanoi" and "Monkey and Bananas." "Towers" is a recursive OPS5 program consisting of two rules

plus a rule for initializing the towers with rings and one for printing out the results. When printing is inhibited (but the print rule is left in the system), REAL-OPS running on a HP236 (8-MHz MC68000 cpu) runs a seven-tower problem in 25.7 s (as reported in previous sections), while the LISP-based OPS5 takes 2 min. on the MicroVAX II, 60 s on a VAX 11/780, and 18.9 s on Texas Instruments' Explorer.

So far, only the sections concerned with matching condition elements to working memory elements and with managing the heap memory pool have been rewritten in 68000 assembler code, leaving much room for speed improvement. The "Monkey" problem, an enhanced version of the one found in reference 13, has 30 rules. It runs slightly faster on the 68000 than VAX speeds and is comparable to an optimized C version of OPS5 on the IBM PC. This, of course, provides a hint of where to find additional candidates for code optimization.

A less flashy property of REAL-OPS is its adherence to the spirit of Forth: it is an interactive language. Rules are incrementally compiled as they are entered from the console or from external files. There is no restriction (as there is in OPS5) that all class objects must be defined before any rules may be entered. The only restriction (a very Forth-like one) is that a class must be defined before it can be used in a rule. As mentioned above, rules may be removed and re-entered. Working memory elements may be entered and removed at will, and the current state of the network and conflict set may be examined (all as in OPS5). In short, developing a program in REAL-OPS has much the same flavor of interactive development as in Forth, much to the user's delight and productivity.

5.4 HIGH SPEED EXPERT SYSTEMS

The main thrust of this work is to show one way of attaining the goal of very high execution speeds for Artificial Intelligence, particularly for expert systems, the most recent of AI's applied successes. It has been shown here how Forth can be used to rewrite a successful and popular expert systems language with a resulting improvement in performance and flexibility, as well as extension to handling real-time data. This final section will address a means for perhaps attaining improvements of great magnitude in execution speed by use of the Forth chip.

Ideal Forth Engine. From the early days, Forth assumed that the ideal stack-oriented, threaded-code engine was available, and it ran efficiently on this ideal machine. The only trouble was that this engine had to be emulated in the assembly language of the actual processor being used. Upon comparing instruction sets of various processors with those instructions actually used to implement the Forth engine, it was evident that Forth is an efficient utilizer of hardware resources (many machine instructions and registers are never used by the Forth emulator). This observation must have been made by a number of people because there are about a half-dozen architectures providing either an efficient emulation of the Forth engine or actually implementing a two-stack, threaded-code processor.

6. OPS5 AND THE FORTH CHIP

OPS5 is not a very high level language because it does not satisfy all of the criteria required of VHLLs; however, it is a development tool for producing very high level languages, specifically Expert Systems. Not all expert systems contain VHLLs but an expert system that a subject matter expert can use for instructing a computer in the expert's analysis methods must embody many of the VHLL characteristics. Compare, for example, the code of FORPS for solving the Towers of Hanoi problem with the statements, in OPS to solve the same problem. FORPS uses Forth programming statements, whereas OPS uses more natural language expressions. The ideal case would be to write a "language" in OPS that is even more natural to the subject matter expert. Both expert systems are goal-directed, and the expert systems change goals appropriately to achieve a solution to the problem. OPS can also assume results and backward chain to confirm rules that lead to the assumed results, as does Expert II discussed in Section 7.3. The combination of these inference capabilities makes OPS a powerful expert system tool. Until recently, only very limited versions of OPS have been implemented on microcomputers. The complexity of the inference engine and the scope of the language require large amounts of program memory when written in a language such as C, and the data requirements can become quite large for sets of rules large enough to be of any interest to decision makers. Because of the unique way in which Forth is suited to writing languages requiring minimal overhead in both memory and computation speed, ORNL began the development of OPS in Forth for microcomputer applications of expert systems in the area of

real-time control (References by Dress). In the interest of real time control, execution speed is an overriding concern, so it was only natural that interest in the transport of OPS to the Forth chip was high when the possibility arose. Concurrently, data systems researchers were interested in the ability to achieve more intelligence in data systems transactions by providing more execution speed to the transaction processor. Thus, the merging of the OPS-Forth work with the data systems research work was agreed to.

OPS5 (Real-OPS in the ORNL version) is too big for current versions of the Forth chip. Even though Real-OPS requires less than 100 bytes of program memory and the chip will probably achieve a 25% or more reduction by optimized compilation, there would be no room for the rule base except in virtual memory (whether in RAM or on hard disk). Because of this it is not known if the potential 20 times increase in execution speed could be achieved until a redesign of Real-OPS is accomplished. This redesigning requires that some capabilities of OPS be eliminated and that the architecture for saving and locating rules be changed to require less space. It has been demonstrated that the chip has sufficient capability to allow a custom design of OPS to fit on it for specific applications and there is at least one military sponsor that is willing to pay ORNL to explore this application approach.

The successful completion of that project (the redesign of Real-OPS) will serve to confirm the speed potential of OPS on the Forth chip; and at that time the need for a more general OPS on the chip can be evaluated. By then, the production version of the chip will have

been released and the enhanced programmability will make it easier to transport a general purpose OPS to the chip. Also, the decision to develop a 32-bit chip will be nearer to resolution, and its availability will remove most of the known translation difficulties. OPS on the chip has great potential interest. It can access databases, provide for multi-user interaction and be made to provide communication capabilities between computer systems and subject matter experts. The chips can be distributed throughout a computer system, serving as the main processor in many cases, certainly if the 32-bit version is implemented to provide database access heretofore not attempted. Therefore, the marriage of the chip with a tool as powerful as OPS is an important step and one that is entirely feasible from the standpoint of what is now known about OPS and the chip.

It might be added here that the reluctance to move from the 16-bit to the 32-bit architecture for the Forth chip is a philosophical and economic decision that has no real answer. While microcomputer users realize the advantages of the 32-bit architecture in conventional computers, there are at least two reasons why these advantages are not so clear with the Forth chip. First, the processing speed inherent in the 32-bit architecture has to do with the number of steps required to access large memory addresses. A 16-bit processor requires at least twice as many cycles to access memories greater than 64k than does a 32-bit processor for example; but the use of Forth as the machine language for a processor reduces the size of the program significantly, thereby decreasing the size of program space to be accessed. Second, the microprocessor that is now used more than any other in applications

(the Z-80) has an 8-bit architecture, indicating that microcomputers with keyboards are not the most likely place that a microprocessor may be applied, and raising the question of investment return for any developer of a chip. The 32-bit Forth chip will be a complex product to physically implement because of the number of pins it requires, but it is otherwise a straight-forward silicon project and an economic decision for the manufacturer to make regarding the best use of design resources.

7. VERY HIGH LEVEL LANGUAGE POTENTIAL OF THE FORTH CHIP

The research described in this report concentrated on two aspects of the very high level language problem: 1) whether the processing speed required for complex or real-time expert systems could be achieved and 2) whether sufficient complexity could be preserved in a language executing at high speed to qualify it as a very high level language. The previous sections report the results of symbolic processing speed benchmarks indicating that processing speeds approaching 5000 inferences per second may be possible under some conditions. One of those conditions was that language complexity be sacrificed. Such a sacrifice is not unrealistic in that many expert systems may be required to function as "black boxes" (smart black boxes) for data communication language with the computer for direct input of the expert's information processing methods. Both aspects are also of interest to decision systems and database management systems developers because complexity of data queries and speed of query processing are current limitations on the utility of decision support systems using large databases.

The Forth chip is of interest because it is small enough to fit in "black boxes" to make them run faster; but the main question is: Is it "big" enough to handle decision systems and database management transactions? To investigate this aspect of the problem, several prototype systems were examined using the Forth chip and other microprocessors. By a combination of direct test results and extrapolations of proven performance, several conclusions can be made regarding the very high level language potential of the Forth chip. Before discussing the

evaluations, however, it would be relevant to examine microcomputer languages and describe how the Forth chip achieves about twenty-times faster processing speed even though it does not run any faster than the more conventional microprocessors it was compared with.

7.1 LANGUAGE TIERS AND REDUCED INSTRUCTION SETS

Typically a microprocessor has a "microcode" of instructions to process data in certain ways. To aid the programmer, a machine language instruction set is usually provided that combines several microcode instructions into one mnemonic word. This is often referred to as assembly language because an assembler can be used to translate the mnemonics into the proper machine instructions. Most high level languages such as FORTRAN and C are written in the assembly language for each different microprocessor. The high level language therefore has instructions that are aggregates of assembly language instructions but can be more like natural human language than the assembler mnemonics. These languages are also more transportable among different computers than assembly instructions and are used by software developers to write application programs such as database management systems and expert systems. Since these applications also have a language for the manipulation of data or interpretation of instructions, the end result is a tier of up to four languages underlying anything currently approaching a very high level language. Most very high level languages will be written in a high level language such as LISP or OPS5 (OPS5 in

fact is usually written in LISP), resulting in a tier of five or six languages between the user and the instruction set that the micro-processor was wired to understand.

The computer user does not have to know more than the language currently in use for instructing the computer, so the tiering is transparent. However, with each successive compilation of a language into another language, the efficiency of communication with the computer decreases, resulting in large memory requirements and slow execution of the computer code. For example, if a typical microcode instruction takes three computer clock cycles and an assembler mnemonic aggregates three microcode instructions, nine clock cycles are required for the assembly instruction. If three assembly instructions are required for a high level language word, then 27 clock cycles are needed to execute the high level instruction. A simple "store to memory" that may take less than ten clock cycles of microcode time can take 20 to 30 clock cycles if accomplished with assembly language, and up to 100 clock cycles when called for by a high level language.

To reduce this problem, many high level language compilers have been optimized to accomplish frequently-used instructions in minimal time. This in turn has led to the concept of the Reduced Instruction Set Computer (RISC), in which the most-used assembly instructions are hardwired into the microprocessor as microcode and designed to execute in one clock cycle if possible. While this results in a smaller set of assembly mnemonics (the reduced instruction set) because more of the

silicon chip is used for each instruction, the faster execution speed of the frequently-used instructions results in overall faster program execution.

Forth is a high level computer programming language. It is not quite the same as other high level languages and is not usually taught in computer science departments (only about 10 universities currently offer it as a separate course, and it is usually in the Electrical Engineering department curriculum). A Forth compiler is built in tiers as are other high level languages, but generally results in more compact code requiring less computer memory. Because Forth can be used to write Forth (it is extensible), it is a natural choice to be used in writing other languages such as expert systems and database management systems. The net result for a language such as OPS5 written in Forth is that it is really Forth words that look and behave the same as OPS5 words written in other languages. Therefore, OPS5 will execute faster when written in Forth because there is one less tier (it is really extended Forth) than if it were written in C or LISP (where the OPS words are aggregates of the host language). The reason that Forth is not more commonly used to write other languages is that it is not in the toolbox of many computer science professionals who prefer to use familiar procedures and languages. (C is extensible to some extent and has become the preferred language for writing most database management systems because it will execute faster than equivalent COBOL, FORTRAN, or LISP programs. However, C requires large amounts of computer memory to compile since it is not completely extensible, and its compiler must allow for more possible combinations of instructions than a fully extensible compiler as does Forth.)

7.2 TIERING IN THE FORTH CHIP MICROPROCESSOR

The developer of the Forth language, Charles Moore, long ago recognized the potential to wire (i.e. layout the silicon chip for) a microprocessor that would use Forth as its microcode language. (It is debatable that such a microprocessor has a microcode since Forth is a high level language.) Other developers had succeeded in microcoding existing microprocessors that could interpret Forth as assembly instructions and, in effect, produced a reduced instruction set of the Forth Language with impressive speed improvement. The concept of a Forth chip was watched closely by Forth users however, because the usual application of Forth was in process and control situations. In these situations, compactness is a decided benefit and the microcoded Forth machines are as large as conventional microcomputers, since most of the processing is done in memory rather than in the microprocessor itself. In addition to being smaller, the chip offered the potential benefits of even faster execution speeds by establishing a one-to-one correspondence between the Forth words and the microprocessor instruction set, and of smaller memory requirements resulting from the ability to combine frequently used Forth words (as opposed to assembly instructions) into single machine instructions. Both of these benefits would be the result of extending the concept of a RISC executing a machine instruction per clock cycle to the execution of a high level Forth instruction per cycle.

In July of 1985, Novix Corporation succeeded in producing less than 100 Forth chips based on Mr. Moore's concept. There was immediate interest by potential control application users, but ORNL obtained four

of these beta (not fully debugged) versions because of Novix' interest in exploring the expert system potential of the chip along with the instrumentation and Controls and the Energy Divisions of ORNL.

Because the Forth chip executes Forth directly rather than as the third tier language of a host microprocessor, a language for expert systems or database management would be only the first or second tier language in a computer using the chip as a microprocessor. Therefore, even though the microprocessor's clock would not be faster than that of other computers, each clock cycle would accomplish more resulting in a higher effective processing speed. This effective speed improvement was demonstrated as reported herein. Another advantage of having ORNL evaluate the expert system capabilities of the chip was the extensive amount of expert system programming that had been done in Forth by ORNL for other microprocessors. The Forth language is highly transportable among different kinds of computers, so, theoretically a large portion of the work of writing expert systems was already done, and all that remained was to determine the differences the Forth chip presented and if any were limiting for expert systems and other very high level language applications.

7.3 VERY HIGH LEVEL LANGUAGES ON THE FORTH CHIP

Previous work at ORNL in the Instrumentation and Controls Division had resulted in a prototype version of OPS5 running on a microcomputer at mainframes speeds (REAL-OPS as discussed earlier). Since OPS is a very complex language with a large instruction set, it was written in a

version of Forth that allowed for addressing large amounts of memory (about 100,000 bytes) and that saved the definition of OPS phrases and words very precisely. The Forth chip did not come equipped with these capabilities, addressing only 32,000 bytes of program memory and throwing away the exact names of words it compiled. While this did not appear to be an insurmountable set of obstacles, it was decided to explore the VHLL potential in other ways before deciding to proceed with the transport of OPS5 to the chip, since it would require an extended amount of reprogramming to convert REAL-OPS to the chip.

As was mentioned in the section on processing speed benchmarks, some LISP words were written in Forth and executed on the Forth chip. These words only demonstrate the ease with which a high level language such as LISP can be implemented on the chip and do not address the problem of memory size limitations; nor do they attempt to optimize for execution speed. In one case, a loop that simply iterates is executed. In the second case, the loop processes lists internally. While this is a very simplistic case, the Forth implementation performs what is referred to as garbage collection; that is, the program recovers unused memory when lists are moved or erased. Early versions of LISP attempted garbage collection when there was no memory left to use, resulting in a perceptible halt in execution at unpredictable stages in program execution (corrected to some extent in later versions). The LISP words implemented in Forth perform continuous garbage collection in a manner that makes the comparisons of execution speed with LISP machines of little meaning. While this essentially nullifies any conclusions that may be drawn about relative processing speed, it

implies that there is a potential advantage to be gained from the Forth implementation of LISP that would be beneficial if the Forth chip is used. This advantage is that memory management is such that large amounts of memory are not required for equivalent LISP programs on the chip.

To evaluate the implementation of an expert system on the Forth chip, three alternatives were considered. The first, to implement a couple of words as was discussed previously, was discarded. The second choice was to attempt to fully implement OPS5. This was also discarded. The selected alternative was to convert an existing expert system to the chip.

The expert system that was converted to the chip was Expert II which was written by Jack Park and is published by Mountain View Press. Expert II employs a simple "backward chaining" inference engine and has been widely used to experiment with expert systems on small microcomputers. It does not provide a full range of expert system capabilities, but other users have modified it to improve its efficiency and Park has improved on it until the development of Expert IV which he uses in private consulting to produce bona fide expert systems (such as a Pediatrics diagnosis system). This conversion of an expert system to the Forth chip revealed many of the problems and solutions involved in the concept of implementing complex language interpretation on the chip. For example, Expert II compiles the text of each rule in the addressable program memory which is in short supply on the chip. It was a simple matter to compile a pointer into the program memory and locate the text of the rules in the 32,000 bytes of data memory thereby

reserving the full 32,000 bytes of program memory for executable statements. For large expert systems with many rules, the pointer could be to a rule location on hard disk. Thus, for this type of expert system, the inference engine could be installed many times over (it only takes about 5,000 bytes).

A further refinement would be to use an inference engine such as FORPS (see previous sections) in conjunction with a rule compiler such as Expert II to additionally improve the management of memory as well as to retain the execution speed witnessed in the benchmarks for the Tower of Hanoi. (The Towers problem in FORPS is pure inference with no user input to solve the problem once stated; Expert II assumes successive answers then queries the user for each rule needed to prove the answer until a required set of rules is satisfied.)

7.4 VERY HIGH LEVEL LANGUAGE POTENTIAL FOR THE FORTH CHIP

The research for this report indicates that VHLLs could be incorporated into the programming architecture of the Forth chip in much the same way that they have been implemented in Forth on other microprocessors. OPS5 could be transported to the chip with a major architectural overhaul to allow for the smaller program address space of the chip. While the extrapolations of processing speed from the benchmarks to OPS indicate a factor of 20 improvement over the current OPS speed in microcomputer Forth, it will not be known just how fast it executes until the new architecture is designed. Much depends on the ratio of hard disk to fast memory storage space required for the conversion. However, experience has shown that Forth programs on other

computers compress considerably when converted to the chip because of its optimizing compiler. Thus, it may be possible to design an OPS that can be compiled into the fast memory of the chip with only text strings residing on disk. Since text strings are only needed by the human user, and the human is the slowest component in the system, no apparent loss of speed would result if this is possible.

8. OBSERVATION AND RECOMMENDATIONS

The research started out as an evaluation of the VHLL potential of the Forth chip. At that time it was envisioned that a VHLL might be designed at some subsequent time to serve decision makers in the access of data, and that the speed of the chip might in some way be applied to the management of very large databases. The results of the survey of VHLLs indicates a not surprising amount of inconsistency among researchers and developers regarding just what a VHLL is and what it should do. However, it appears that yet another language will be of no immediate benefit to the data management and decision support communities. What would be of use is machine intelligence to eliminate the need for a specific language, and it was this underlying hope that started the research along the path of VHLLs in the first place.

The results of this research tend to support this observation because in the course of surveying and evaluating VHLL potential, methods for addressing the more basic problem of machine intelligence were encountered. These methods center on data storage and retrieval schema that are entirely consistent with language interpretation, pattern recognition, and smart data locations, and that require smart small microprocessors working with large data systems. Machine learning and pattern recognition would appear to be at the heart of the solution to any problem for which a VHLL (natural language?) appears to be the answer.

The developers of applications for the Forth chip have already succeeded in putting it in disk controllers, data busses, and database machines. These developments were not unexpected and the anticipation

of such developments was one of the reasons that this research did not concentrate on specific applications of the chip. For the forgoing reasons it is recommended that the emphasis of future work shift from the VHLL and expert systems arena to that of pattern recognition and data retrieval consistent with user interfaces for decision support systems. Such methodologies have been demonstrated and reported in the literature at various times, and have been awaiting the advent of inexpensive, fast computers for further development. In this context, the Forth chip becomes just another tool to integrate into the total decision support system at its appropriate place, possibly distributed throughout an intelligent system, but certainly to be used with other tools that will benefit the large data system user.

The success of others in incorporating the Forth chip into various components of the computer system opens the entire computer system to the development of machine intelligence. The research reported here demonstrates that expert systems and very high level languages can be made an integral part of such a system of distributed smart components where a central computer is no longer the sole basis for data systems or decision systems development. With intelligence at the user interface, at the data storage location, and at several points in the system to direct the flow of information, truly intelligent networks become feasible. Research is still necessary to accomplish the development of an intelligent system with or without Forth chips, but the very high processing speeds possible in small packages that include very high level language capabilities are a significant step forward in the pursuit of machine intelligence.

The implementation of OPS on the Forth chip should no longer be viewed as a research task, but rather the extension of research previously done into specific areas for which there is now a sponsor. On the other hand the power of OPS in such a small package as the chip is a needed tool for machine intelligence research and for the distribution of decision systems throughout computer networks. For this latter reason it is advantageous to continue the development of OPS on the chip along with research on networking systems that have the capability of learning from subject matter experts as well as for providing them with an effective means of communicating with a computer. Other tools must also be developed - in fact, the development of tools that can develop applications appear to be essential to the effective use of computers in public organizations, considering the uncertain mix of hardware, software, and user expertise that is likely to arise out of current competitive regulations. OPS, Forth, and very high speed microprocessors are central to such development, making OPS on the chip also central to the effective implementation of smart systems on microcomputers and in computer systems.

9. REFERENCES

1. Defense Advanced Research Projects Agency, "STRATEGICS COMPUTING - New Generation Computing Technology: A Strategic Plan for its Development and Application to Critical Problems in Defense," Oct. 28, 1983.
2. W. B. Dress, "REAL-OPS - A Real-Time Engineering Applications Language for Writing Expert Systems," 1986 Rochester Forth Conference, University of Rochester, June 1986.
3. C. J. Matheus, "The Internals of FORPS (A FORth-based Production System)," In Publication, The Journal of Forth, 1986.
4. P. C. Treleaven, "Computer Architectures for the Fifth Generation," book chapter in Fifth Generation Computer Project, State of the Art Report 123-34, Pergamon Infotech, Maidenhead, Berks., England, 1983.
5. J. Carbonell, Carnegie-Mellon University, communication to Oak Ridge National Laboratory, March 3, 1986.
6. L. Brodie, Thinking Forth, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1984.
7. J. Martin, Fourth-Generation Languages, Volume I, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1985.
8. Bruce G. Buchanan, "Expert Systems: Working Systems and the Research Literature," Expert Systems, Vol. 3, No. 1, pp. 32-51, January 1986.
9. Judith Bachant and John McDermott, "R1 Revisited: Four Years in the Trenches," AI Magazine, Vol. 5, No. 3, pp. 21-32, 1984.
10. Peter Hager, "NASA Says AI Systems Just Getting Off the Ground," Government Computer News, p. 72, April 11, 1986.
11. Creative Solutions, Inc., Multi-FORTH Version 2.00 User's Manual, Rockville, Maryland, 1984.
12. Charles L. Forgy, "OPS5 User's Manual," Technical Report, Carnegie-Mellon University, Department of Computer Science, 1981.
13. Brownston et al., Programming Expert Systems in OPS5, Addison-Wesley, Reading, Maryland, 1985.

14. Charles L. Forgy, "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem," Artificial Intelligence, Vol. 19, No. 1, 1982.
15. Leo Brodie, Thinking Forth, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1984.
16. Donald E. Knuth, The Art of Computer Programming, 2nd Ed., Vol. 1, Fundamental Algorithms, pp. 406-20, Addison-Wesley, Reading, Maryland, 1983.
17. W. B. Dress, "A Forth Implementation of the Heap Data Structure for Memory Management," The Journal of FORTH Application and Research, Lawrence P. Forsley, Ed., Vol. 3, No. 3, pp. 39-49, 1986.
18. Donald E. Knuth, op. cit., Vol. 3, Sorting and Searching, Chapter 5.
19. O. L. R. Jacobs, "Introduction to adaptive control," Self-Tuning and Adaptive Control: Theory and Applications, C. J. Harris and S. A. Billings, Eds, IEE Control Engineering Series 15, Peter Peregrinus Ltd., London and New York, 1981.
20. W. B. Dress, "Communicating Asynchronous External Data to an Expert System," Proceedings of Eighteenth Southeastern Symposium on System Theory, IEEE Computer Society, pp. 294-96, April 7-8, 1986.

INTERNAL DISTRIBUTION

- | | |
|----------------------|--|
| 1-5. H. G. Arnold | 25. F. C. Maienschien |
| 6. J. E. Christian | 26. F. R. Mynatt |
| 7-11 W. B. Dress | 27. M. S. Phifer |
| 12. W. Fulkerson | 28. G. O. Rogers |
| 13. W. B. Gettings | 29. M. W. Rosenthal |
| 14. R. K. Gryder | 30. F. L. Sexton |
| 15. G. R. Hadder | 31. E. W. Whitfield |
| 16. K. A. Hake | 32. T. J. Wilbanks |
| 17. R. B. Honea | 33. Document Reference Section |
| 18. H. L. Hwang | 34-36. Central Research Laboratory |
| 19. J. O. Kolb | 37. Laboratory Records Department |
| 20-24. R. S. Loffman | 38. Laboratory Records Department - RC |

EXTERNAL DISTRIBUTION

39. Jaime G. Carbonell, Associate Professor of Computer Science, Carnegie-Mellon University, Pittsburg, PA 15213
40. Charles R. Fenton, Deputy Director of Information Management, ASNI-CPC, Room 8N65, 200 Stovall Street, Alexandria, VA 22332-0300
41. Fritz R. Kalhammer, Vice President, Electric Power Research Institute, P. O. Box 10412, Palo Alto, CA 94303
42. Roger E. Kasperson, Professor, Government and Geography, Graduate School of Geography, Clark University, Worcester, MA 01610
43. Jesse Lipscomb, ASNI-CPC, Hoffman 2, Room 8N65, 200 Stovall Street, Alexandria, VA 22332-0300
44. Lawrence Lorton, ASNI-CPC, Room 8N65, 200 Stovall Street, Alexandria, VA 22332-0300
45. R. L. Perrine, Professor, Engineering and Applied Sciences, Civil Engineering Department, Engineering I, Room 2066, University of California, Los Angeles, CA 90024
46. Office of the Assistant Manager for Energy Research and Development DOE-ORO
- 47-76. Technical Information Center, DOE, P. O. Box 62, Oak Ridge, TN 37831
- 76-86. Extra copies to M. S. Phifer, 4500N, MS 207